
Welcome to On Time RTOS-32

On Time RTOS-32 offers a complete solution for high performance embedded systems using x86 compatible CPUs. However, it is important to understand how the various components (RTTarget-32, RTKernel-32, RTFiles-32, RTPEG-32, and RTIP-32) of On Time RTOS-32 fit together as parts of a scalable architecture.

RTTarget-32 is the foundation of On Time RTOS-32. It includes all development tools required to run 32-bit applications on an embedded system. RTTarget-32 can process a single 32-bit application built with a Win32 compiler to run on the embedded target. It supplies boot code to initialize the target's hardware and to subsequently invoke the application automatically. RTTarget-32 also supplies an extensible subset of the Win32 API, allowing it to run programs developed for Windows NT in an embedded environment under real-time conditions. RTTarget-32's Win32 emulation is comprehensive enough to support the startup code and run-time systems of most Win32 compilers unmodified. Both source level and binary compatibility between Windows NT and RTTarget-32 are possible. Many programs or libraries and DLLs for Windows NT can be ported without or with only minor changes to RTTarget-32. RTTarget-32 can be regarded as a very small Windows NT kernel for embedded systems with a minimum memory footprint of a mere 16k RAM/ROM.

Another advantage of RTTarget-32's Windows NT compatibility is its support for NT development tools. You can develop programs using Borland C++, Borland Delphi, Microsoft Visual C++, or Watcom C++. The respective compilers' command line tools as well as the IDEs such as Microsoft's Visual Studio are supported. RTTarget-32's Win32 compatibility allows its compiler support to be extended to any tool which generates standard Win32 PE files (Windows' executable file format).

RTKernel-32 is a real-time multitasking kernel for RTTarget-32. It is a high-performance real-time scheduler supplied as a set of linkable libraries which can extend the functionality available to RTTarget-32 programs. It adds most Win32 thread API functions for thread creation and management, semaphores, critical sections, etc., to RTTarget-32's Win32 emulation library. It also defines its own API and functionality beyond the capabilities available under Windows NT to meet the needs of real-time systems (e.g., deterministic scheduling, priority inheritance, a comprehensive interrupt handling API, debugging functions, etc.).

RTFiles-32 is the file I/O component of On Time RTOS-32. It provides its functionality through the Win32 API emulation of RTTarget-32, too. Therefore, all file I/O related functions of the C/C++ or Pascal run-time system can be used. RTFiles-32 supports the DOS/Windows FAT file system structure on floppy disks, IDE, and flash disks and adds enhancements for real-time file I/O. Just like RTKernel-32, RTFiles-32 must be used with RTTarget-32. RTFiles-32 supports RTKernel-32, but does not require it.

RTPEG-32 is an event-driven, object-oriented C++ GUI library for embedded systems. It supports implementing professional Windows 95 or custom look-and-feel user interfaces with mouse and keyboard support. Device drivers for VGA and SVGA/VESA graphics hardware are included.

RTIP-32 is an add-on to provide networking capabilities for On Time RTOS-32. It implements the core TCP/IP protocols for Ethernet and serial communications. RTIP-32's functionality is provided to the application using the popular Unix sockets API. RTIP-32 requires RTTarget-32 and can be combined with RTKernel-32 and/or RTFiles-32. For applications requiring several simultaneous connections, RTKernel-32 is recommended.

This manual describes RTTarget-32, RTKernel-32, RTFiles-32, and RTPEG-32. The RTIP-32 documentation is provided in a separate set of manuals.

Hardware and Software Requirements

The On Time RTOS-32 host tools run under the following operating systems:

- Windows 95/98/ME
- Windows NT/2000

In addition, one or more of the following compilers are required:

- Microsoft Visual C++ 2.0 or higher
- Borland C++ 4.5 or higher
- Borland C++ Builder 1.0 or higher
- Borland Delphi 2.0 or higher
- Watcom C/C++ 10.5 or higher

The On Time RTOS-32 run-time environment has the following minimum requirements on the target computer:

- The target computer must be able to execute Intel x86 32-bit protected mode instructions. This means that an i386 compatible CPU must be used.
- At least 8k of RAM and 16k of total memory is required. For development systems with source level debug support, at least 32k of RAM is required. Depending on the size of the application and optional components used, RAM and ROM requirements may be higher.

This Manual

This manual describes how programs developed with any of the compilers listed above can be executed on an embedded system. Familiarity with at least one of the supported compilers and the Win32 API is assumed. Furthermore, some knowledge of the Intel 386 CPU architecture (in particular, address translation, privilege levels, descriptor tables, etc.) is recommended.

Part I covers RTTarget-32, the core component of On Time RTOS-32. It describes how the development environment is used. Parts II, III, and IV describe RTKernel-32, RTFiles-32, and RTPEG-32.

After installation, it is important to read the accompanying Readme.html file. This file may contain last-minute changes and corrections to the User's Manual.

Please refer to section *Installation* in this chapter for installation instructions.

Technical Support

If a program running under On Time RTOS-32 does not function as expected, please perform these steps:

- Analyze the LOC file created by the locator RTLoc. If there are any error or warning messages, check Part I, Appendix C for a problem description and a possible solution.
- If you are using RTKernel-32, be sure to link the Debug Version of RTKernel-32.
- Refer to Part I, Appendix A to verify whether any special considerations apply to your compiler/linker configuration.
- Please refer to the demo programs that come with On Time RTOS-32 for tasks similar to those in your application. Compare your code and configuration files to those of the demo programs.
- Try to isolate the problem in a small test program which is much easier to test and debug.

If the problem persists, you may contact On Time for technical support. In this case, please supply the following information:

- On Time RTOS-32 version and components.
- Software development tools being used (compiler, assembler, linker, debugger, third-party libraries, etc.), including the exact version numbers as well as the exact version of your operating system.

- If you want to contact On Time by email, do **not** attach .EXE/.DLL/.RTB/.RTA files or any other large binary data to your message.
- If the problem is related to an RTTarget-32 configuration file, do **not** include the configuration file in your support request. Instead, please attach the .LOC file produced by RTLoc.

All registered users of On Time RTOS-32 are entitled to free technical support for the current or the previous major version. Support for older versions is not guaranteed.

Support Web Page and Mailing Lists

On Time offers extensive technical support facilities through its World Wide Web site at

<http://www.on-time.com>

The Web site provides information, examples, patches, maintenance releases, etc., for all On Time products, which can be viewed online or downloaded. In addition, email addresses and mailing list addresses are provided for private or public technical support.

Detailed information about our Internet technical support is available on our Web site.

Installation

On Time RTOS-32 comes as one or more self-extracting installation programs. To install On Time RTOS-32 on your hard disk, simply execute the installation program under Windows. You will be prompted to select an installation directory and a program group in your Start Menu. If you are installing a full release of an On Time RTOS-32 component, you will also be prompted to enter an installation key. The correct installation key is given on your license certificate.

Please be sure to read the Readme.html file after the installation program has completed. It may contain important corrections or additions to this manual.

The following directories are created under the On Time RTOS-32 installation directory:

Directory	Contents
Bin	Executable programs and DLLs delivered with On Time RTOS-32.
Include	On Time RTOS-32 include files.
Libbc	Libraries and various object files for Borland C++.
Libdel	Interface unit source files for Borland Delphi.
Libmsvc	Libraries and various object files for Microsoft Visual C++.
Libwat	Libraries and various object files for Watcom C++.
Demobc	Example programs for Borland C++.
Demodel	Example programs for Borland Delphi.
Demomsvc	Example programs for Microsoft Visual C++.
Demomsdev	Example programs for Microsoft Visual Studio 6.0 (IDE Projects).
Demowat	Example programs for Watcom C++.
Boot	Source code of RTTarget-32's boot code.
Monitor ¹	Source code of RTTarget-32's Debug Monitor.

¹ This directory is only created if you have purchased the RTTarget-32 source code.

Driver\Rtk32 ²	Source code of all RTKernel-32 drivers.
Driver\Rtf32 ³	Source code of all RTFiles-32 drivers.
Driver\Peg ⁴	Source code of all RTPEG-32 drivers.
Source\Rtt32 ¹	Source code of the RTTarget-32 run-time library RTT32.LIB.
Source\Emu	Source code of the 387 FPU emulator.
Source\Metaw	Source code of the MetaWINDOW support library Rtmetak.lib.
Source\Rtk32 ⁵	Source code of the RTKernel-32 scheduler.
Source\Rtk32sup ²	Source code of the RTKernel-32 supplemental modules.
Source\Rtf32 ⁶	Source code of the file system core of RTFiles-32.
Source\Peg ⁷	Source code of the GUI library RTPEG-32.
PegDoc ⁴	RTPEG-32 HTML online reference manual.

Here are the steps required to run one of the demo programs to test the On Time RTOS-32 installation.

- Connect your host PC with the target PC using a NULL modem cable. If a port other than COM1 is used on the host, click "Edit Settings" and record the COM port used in file Rttarget.ini. If a different port is used on the target, edit the COMPort command in configuration file Demopc.cfg included with each demo.
- Click on one of the demo program shortcuts created in the On Time RTOS-32 Start Menu folder. For a command line demo, this will open a command prompt window with all required environment variables defined and the current directory set to the directory of the selected demo. For Visual Studio demos, the respective workspace and project files for the demo are loaded in Msdev.exe.
- Insert an empty formatted diskette in drive A:.
- For command line demos, build the demo using the MAKE utility of your compiler (MAKE for Borland/Inprise, NMAKE for Microsoft, or WMAKE for Watcom). In Visual Studio, select "Target - Win32 Debug" as the active configuration and build the project. Visual Studio will create the Debug Monitor Boot diskette in this step.
- For command line demos, create a Debug Monitor boot diskette with command:

```
BootDisk Monitor A:
```

- Run the program under control of the debugger. For the command line demos, start the RTTarget-32 debugger with:

```
RTD32 Hello
```

In Visual Studio, any command which starts the Visual Studio debugger (e.g., Build, Start Debug, Step Into) will download the program and run it under the debugger. If you are prompted for the name of the executable to run, enter Debug\<project name>.exe.

Note that GUI demo programs or demos not running on PC compatible targets may need customer monitors.

To only download the program for execution, program RTRun can be used:

```
RTRun Hello
```

² This directory is only created if you have purchased RTKernel-32.

³ This directory is only created if you have purchased RTFiles-32.

⁴ This directory is only created if you have purchased RTPEG-32.

⁵ This directory is only created if you have purchased the RTKernel-32 source code.

⁶ This directory is only created if you have purchased the RTFiles-32 source code.

⁷ This directory is only created if you have purchased the RTPEG-32 source code.

Licensing Terms and Liability

The software products RTTarget-32, RTKernel-32, and RTFiles-32 (programs, libraries, object code, source code, and User's Manual) are the proprietary property of On Time Informatik GmbH, Hamburg, Germany.

By purchasing an RTTarget-32, RTKernel-32, or RTFiles-32 license, the licensee acquires the rights to develop and distribute products linked with libraries, object code, or boot code delivered with RTTarget-32, RTKernel-32, or RTFiles-32 under the following conditions:

1. The product does not contain files delivered with RTTarget-32 other than linked into the application, with the exception of files marked as redistributable in this manual.
2. The product does not compete with any product produced by On Time Informatik to any degree.
3. The product is not a software development product or a library.
4. The product is shipped as a binary image. It must not be shipped as a Win32 PE (Portable Executable, Win32 .EXEs and .DLLs) file if it has been linked with any library shipped with RTTarget-32, RTKernel-32, or RTFiles-32. It is also not permitted to ship linkable object files or library files containing any software components of RTTarget-32, RTKernel-32, or RTFiles-32.

Software products containing any RTTarget-32, RTKernel-32, or RTFiles-32 libraries may only be distributed as binary images such as RTB files (RTTarget-32 binaries), RTA files (RTTarget-32 disk boot images), BIN files (binary image files), HEX files (Intel HEX file format), DLM files (RTTarget-32 dynamically loadable modules), or such images burned into non-volatile memory such as ROM, EPROM, EEPROM, or Flash. If you plan to distribute products in a different form, please contact On Time for further information.

Proof of a legitimate license is a unique License Certificate issued by On Time. The license certificate bears On Time's company logo and lists all license numbers and their installation keys. License serial numbers are located on adhesive labels each of which also bears On Time's colored company logo.

A single RTTarget-32, RTKernel-32, or RTFiles-32 development license must at no time be used by more than one person. A license may be transferred to another person, if this person accepts these licensing terms. When a license is transferred to another person, the complete product contents including the License Certificate must be given to the new licensee; copies of RTTarget-32, RTKernel-32, or RTFiles-32 materials must be permanently destroyed.

This software and documentation is sold "as is" without warranty as to their performance, merchantability or fitness for any particular purpose. The entire risk as to the quality and performance of the software is assumed by the user. On Time Informatik GmbH warrants that the CD-ROM or diskettes on which the program is furnished will be free from defects in materials and workmanship under normal usage for a period of ninety days from the date of original purchase. Your sole and exclusive remedy in the event of a defect is expressly limited to the replacement of the CD-ROM or diskettes.

In no event shall On Time Informatik GmbH or anyone else who has been involved in the creation, production, or delivery of this software be liable for any direct, incidental or consequential damages, such as, but not limited to, loss of anticipated profits, benefits, use, or data resulting from the use of this software, or arising out of any breach of any warranty.

Part I

RTTarget-32

RTTarget-32 is a cross development system for Intel 80386 and higher CPUs. Win32 console mode applications can run without Windows on virtually any hardware with a 32-bit Intel 386 compatible CPU. Cross debugging is available using a Win32 debugger.

Programs to run with RTTarget-32 are developed using standard 32-bit compilers that can produce native Win32 console mode applications (such as Borland C++, Borland Delphi, Microsoft Visual C++, Watcom C/C++). Executable files built with these compilers are processed by RTTarget-32 to run on the target system.

RTTarget-32 consists of the following components:

- **Target Boot Code**
The boot code initializes the CPU and, possibly, other hardware. It locates and initializes the application program. Applications can be loaded from floppy disk, hard disk, EPROM disk, flash disk, or can run directly from ROM.
- **BootDisk Utility**
This program can be used to create bootable RTTarget-32 diskettes. All common disk formats (5.25", 3.5", hard disks, EPROM disks, flash disks) with FAT-12, FAT-16, or FAT-32 file structure are supported.
- **Locator**
RTTarget-32's locator RTLoc assigns fixed addresses to the application (this process is called *fixing up*). RTLoc can place a program anywhere in the target computer's address space. Read/write and read-only regions of the program can be separated into different memory areas.
- **Debug Monitor Resident on the Target**
RTTarget-32's Debug Monitor can be installed on the target computer to handle program downloads and communication with the source-level debugger on the host.
- **Source Level Cross Debugger**
RTTarget-32's cross debugger RTD32 is an extended version of Borland's Win32 debugger. It supports Borland, Microsoft, and Watcom debug symbol tables. In addition, RTTarget-32 programs can be debugged using Microsoft Visual Studio 6.0's debugger.
- **Run Utility**
As an alternative to starting programs from diskette or EPROM, RTTarget-32's RTRun utility can be used to download and run programs over a serial link.
- **Win32 Emulation Library**
RTTarget-32 includes a library providing a subset of the Win32 API. This subset supports the standard run-time systems of the supported compilers, making available functions like malloc() or printf() on the target computer.
- **Serial I/O Library**
RTTarget-32 comes with a powerful serial I/O library. It supports simultaneous interrupt-driven communication on up to four ports.

Features of RTTarget-32

The main features of RTTarget-32 are:

- **Target Booting**
RTTarget-32 can boot the computer directly from floppy disk, hard disk, EPROM disk, flash disk, ROM, or via MS-DOS. The boot code requires only about 6k of memory. In addition, the boot code maintains system data structures such as the GDT, IDT, and the page table.
- **Supports Borland C/C++, Borland Delphi, Microsoft C/C++, Watcom C/C++**
Popular 32-bit development systems can be used to develop powerful 32-bit embedded systems applications.
- **Supports Cross Debugging**
Source-level debugging is available for RTTarget-32 programs using RTTarget-32's debugger RTD32 and Microsoft's Visual Studio 6.0.
- **Supports C/C++ and Pascal Run-Time Systems**
Run-time system routines like printf, malloc, WriteLn, etc., are available. Porting existing applications is simplified considerably. Alternatively, programs can run without run-time system for even less memory overhead.
- **Low Resource Requirements**
32-bit programs developed with RTTarget-32 can run in as little as 16k of memory.
- **Supports Privilege Levels**
Programs can run at different privilege levels to optimize either for maximum protection or for best performance.
- **Supports Paging**
Page-level protection is fully supported. The memory protection features of the Intel 386 and higher CPUs are used to guarantee that programs cannot destroy protected data, code, or critical system tables.
- **RAM Remapping**
In addition to using paging for protection, RTTarget-32 can rearrange memory pages to create larger consecutive regions of RAM. RTTarget-32 can also create virtual regions of memory by combining several different areas of physical memory. However, physical addresses remain fixed in the virtual address space, making absolute memory addressing easy.
- **Supports DLLs**
Applications can consist of one main program plus up to 31 DLLs. Even DLLs not specifically designed for RTTarget-32 may be used. DLLs can be statically linked into a program image or loaded dynamically through a file system such as On Time's RTFiles-32.
- **Data compression**
Program code and data can be compressed to save EPROM or boot disk space and to accelerate downloads.
- **RAM Files**
Although RTTarget-32 does not include a file system, file I/O can be simulated with file images located in reserved RAM on the target. This feature is useful to support code that performs file I/O and cannot be changed or to easily generate different configurations of a program that reads information from a file at run-time. If a true FAT file system is required, On Time's file system RTFiles-32 is fully supported by RTTarget-32.

Terms and Definitions

The following terms will be used throughout this manual:

Host	The computer used for software development. The host must run under 32-bit Windows and have the RTTarget-32 tools and one or more of the supported compilers installed.
Embedded Systems	Computers that are typically embedded into some larger system (e.g., machine). Embedded systems usually have peripheral devices different from desktop PCs. Frequently, embedded systems do not have a user interface (and no screen and/or keyboard).
Target	The computer used to run applications developed with RTTarget-32. It must have an Intel 386 compatible or higher CPU. For cross debugging or downloading, it must be connected with the host by an RS232 serial link.
Cross Development	Software development on a host computer for a different target computer.
Cross Debugging	Debugging a program on the target with the debugger running on the host.
Fixup	A fixup is a location in a program image depending on an absolute address. Since Win32 programs must be able to run at any address, the linker writes a fixup table to the PE file containing a list of all such locations in the program. Windows will process the fixup table when the program is loaded. RTTarget-32 performs this function in the locate process.

Here is an example of a program sequence requiring a fixup:

```
int x;
int main(void)
{
    x = 3;
    ...
}
```

The compiler might translate this to:

```
mov [12345678], 3
```

where 12345678 is the address of global variable x. However, if the program is not loaded at the address assumed by the linker, the code will fail because an incorrect address is used for x.

The fixup table will contain an entry for this code sequence. The fixup is simply a pointer to the absolute address in the code (a pointer to the '12345678' value contained in the 'mov' instruction above). Thus, the program loader (or RTTarget-32's locator) will know this address needs to be adjusted in the code, depending on where the program is located.

The address of the absolute address in the code is known as the *fixup location*. The referenced address (address of variable x in this case) is the *fixup target* or *fixup value*.

Locate	The process of converting a relocatable executable file to an absolute image. The absolute image can only run at the addresses assigned in the locate process. A desktop operating system relocates a program when it is loaded, while RTTarget-32 locates a program before it is loaded on the target.
NT Program	32-bit PE-file program as originally defined for Windows NT. Some environments that can run such programs (with varying restrictions) are: Windows NT (of course), Windows 3.1 with Win32s, Windows 95, Borland's DOS extender 32RTM, and RTTarget-32.
Windows	This manual uses the general term Windows to refer to the Microsoft Windows operating system family.
PE File	Portable Executable file format used by Win32 for 32-bit applications. Both .EXE and .DLL files use this PE file format.

Program Entity	Any part of a program that requires memory on the target. Examples of program entities are: program code or data, stack, heap, boot code, page table, etc.
Discardable Entity	Special kind of Program Entity which is needed only for booting and program initialization. The memory area occupied by discardable entities can be reused by other program entities such as the stack or heap. Discardable entities in RAM are allocated top-down (as opposed to bottom-up for other entities) by RTTarget-32.
Module	A program module according to Win32. A module can be an .EXE or .DLL file, both of which must be in the PE file format. RTTarget-32 supports one EXE and up to 31 DLLs.
386	The term 386 is used throughout this manual to refer to any processor compatible with the Intel 80386.
TLS Data Segments	Win32 defines a special kind of data segment called TLS (Thread Local Storage). Data segments of this type are duplicated automatically for each thread. Declaring variables in such a segment is compiler dependent. For example, Borland C/C++ uses the <code>__thread</code> keyword while Microsoft uses <code>_declspec(thread)</code> . RTTarget-32 alone does not support multiple threads, but nonetheless supports TLS data for the program (which is a single thread). TLS data segments are supported for a multitasking system running under RTTarget-32 which might also support them (e.g., RTKernel-32). For further information about TLS or thread variables, please consult your compiler's documentation.
Uncommitted Memory	Uncommitted memory is a range of linear address space which does not have any associated physical memory. In systems which use paging, uncommitted memory can be committed by mapping physical pages to that address range. More information about uncommitted memory is available in the Win32 SDK documentation.
Image	The portion of a program entity which has initialized data associated with it (e.g., all of a code section, parts of the data section, none of the program stack).

Chapter 1

Running Win32 Programs without Win32

Windows NT/2000 and Windows 95/98/ME are very powerful and complex operating systems offering a comprehensive set of API calls. They fulfil widely differing application program requirements, but their drawback is a substantial overhead on system resources like main memory and disk space, even at times when not all system services are actually required. Moreover, the security mechanisms provided by these systems can further increase the run-time overhead. For example, hardware access and interrupt processing must be placed in device drivers. However, device drivers are difficult to program and to debug, and access from application programs is slow. Most importantly, high interrupt latencies and the non-deterministic time behavior of Windows' tasking makes Windows unsuitable for real-time systems.

Embedded systems typically will not need all services offered by Windows, but will often require low interrupt response times. Also, the high cost of RAM, disk space, and run-time royalties of systems running Windows may be prohibitive for systems built in large quantities.

RTTarget-32 allows running Win32 programs without Windows. Basically, three steps are required to achieve this goal:

- Absolute addresses must be supplied for the program. Windows can usually load a program at any address. The executable file contains a fixup table specifying the locations in the program image that need to be adjusted. This is called *fixing-up* or *locating*.
- Substitutes for commonly used Win32 API calls must be supplied. Most Win32 programs will contain calls to the Win32 API library (Win32 supplies about 4000 functions). RTTarget-32 supports a subset of the Win32 API large enough to support the standard C/C++ and Pascal run-time libraries and most programs using character-mode user interfaces.
- The target computer must be booted. This process includes initialization of the hardware and activating the application.

As an additional aid for software development, RTTarget-32 also supports debugging programs while they are running on the target.

Benefits of Running without Windows

The most important advantages of not using Windows are:

- Low resource requirements. RTTarget-32's boot code needs only about 6k of memory. Programs not using the run-time system can run in as little as 16k total memory. If the run-time system is used, about 64k is required for the heap and run-time system code.
- Low interrupt latencies. Using RTTarget-32, the interrupt latency can be as low as about 5 microseconds on a 16Mhz-386SX or EX. Under Windows, the same CPU can yield several milliseconds interrupt latency.
- Interrupt handlers in application code. RTTarget-32 allows the application to install interrupt handlers. This greatly simplifies the interaction with interrupt-generating hardware and is much faster than Windows device drivers.
- Hardware access. RTTarget-32 allows the application to directly access hardware through I/O ports. No overhead is incurred by emulation or complicated device drivers. Windows does not permit applications to access ports directly.
- Physical memory access. RTTarget-32 initializes the CPU such that physical addresses are equal to virtual addresses used by the application, even if paging is used. This facilitates direct hardware access through memory mapped devices and the use of DMA. Windows does not allow access to physical addresses.
- Access to privileged instructions. RTTarget-32 can run applications at privilege level 3 or 0. Windows supports only privilege level 3 for applications.

- Speed. Windows needs a substantial amount of CPU time for system management. With RTTarget-32, this time is freed and made available to your application. Thus, most applications will run faster under RTTarget-32 than under Windows. In addition, RTTarget-32's deterministic behavior makes it suitable for real-time systems.

Benefits of Running with Windows

The following operating system services are not (or only in part) supported by RTTarget-32:

- Running several processes concurrently. However, the real-time multitasking system *RTKernel-32* from On Time is compatible with RTTarget-32 and can be used for real-time multithreading.
- File system. RTTarget-32 can load programs from disk at boot time, but it supports file I/O only for the console, parallel ports, and for files located into the program (see Chapter 3, *File*). However, the full-featured file system *RTFiles-32* from On Time is compatible with RTTarget-32.
- Virtual memory. RTTarget-32 can remap unused RAM pages to create larger consecutive regions of RAM and it can create virtual regions consisting of several different physical regions. However, swapping pages to disk in order to enlarge the amount of available memory is not supported.
- Graphical user interface. RTTarget-32 supports character I/O only (e.g., `printf`, `puts`, etc.). However, On Time RTOS-32 component *RTPEG-32* is available for professional graphics user interfaces.
- Complete Win32 API. RTTarget-32 supports only a subset of the Win32 API. However, this subset can be extended by the application. The subset supplied is sufficient to support most parts of the standard C/C++ or Pascal run-time systems.

Preparing a Program for RTTarget-32

Preparing a program for RTTarget-32 is very similar to preparing Win32 console mode programs. The command line compiler or IDE compiles the program as if it were intended to run as a 32-bit console mode application. The only difference is that RTTarget-32's library *RTT32.LIB* is linked in. *RTT32.LIB* contains special versions of some Win32 API functions. This library makes the program independent of *KERNEL32.DLL* (Window's main API DLL).

Locating a Program

Locating an application is accomplished using program *RTLloc*. It requires two sources of input: An *.EXE* file (and possibly one or more DLLs) produced by a 32-bit compiler/linker and a configuration file. The output is an absolute binary file (extension *.RTB*, RTTarget-32 Binary) and a listing file (extension *.LOC*) which contains information similar to a linker map file. The configuration file specifies how to map the program onto the given hardware.

Here is a short example configuration file for a simple program compiled with Borland C++:

```
Region MyROM F0000h 64k ROM
Region MyRAM 0h 256k RAM

Locate Header Header MyROM
Locate Section CODE MyROM
Locate Section DATA MyRAM
Locate Stack Stack MyRAM 16k
Locate Heap Heap MyRAM
```

First, two regions of memory are defined. The first is named *MyROM*, is located at address *F0000h*, and is 64k bytes long. Its type is ROM. The second section is RAM and called *MyRAM* at address 0, size 256k. The *Locate* commands instruct *RTLloc* which parts of the program to map to which regions. The Header is required by the boot code. It contains information about where the application is located, its entrypoint, etc. *Sections* are data read from the *.EXE* file. *RTLloc* will determine their sizes from information in the *.EXE* file. The stack and heap are only allocated; initially, there is no data associated with them. The stack size is fixed at 16k in this example. Since we did not specify a size for the heap, *RTLloc* will assign all remaining space in region *MyRAM* to the heap.

After running RTLoc, the .LOC file will contain the following *Relocation Report* (and several other reports):

```
[Relocation Report]
```

Name	Address	Size	Image	Access

MyROM	000F0000	00010000	00006000	
Header	000F0000	00000096	00000096	ReadOnly
CODE	000F1000	00005000	00004A00	ReadOnly
MyRAM	00000000	00040000	00040000	
DATA	00000000	00002000	00001200	ReadWrite
Stack	00002000	00004000	00000000	ReadWrite
Heap	00006000	0003A000	00000000	NoAccess

All regions are listed with their respective locations, sizes, and space used. Indented below each region, all entities allocated to the region are listed with their locations, sizes, and data image sizes.

This example may seem trivial; real configurations can be more complex. For example, you could have several disjoint RAM and ROM regions as well as DEVICE regions, virtual regions, DLLs, etc. Chapter 3 covers configuration files in detail.

Cross Debugging a Program

The *Monitor* program delivered with RTTarget-32 can be loaded on the target. The Monitor will communicate with the debugger as soon as the latter has been started on the host. Subsequently, the debugger will operate exactly the same as it would for local debugging (except for some RTTarget-32 extensions).

Chapter 5 describes how cross debugging with RTTarget-32's RTD32 works in detail.

A Complete Example

Assume you have two PCs: a host running DOS or Windows and a target. We want to run a test program compiled with Borland C++.

Let's create the following test program in file HELLO.C:

```
#include <stdio.h>

int main(void)
{
    printf("Hello, RTTarget-32!\n");
    return 0;
}
```

Now we can compile and link the program like this⁸:

```
bcc32 hello.c rtt32.lib
```

To be able to run the program on the target, we must locate the program. For this purpose, a small configuration file must be created (HELLO.CFG):

```
Region NullPage 0 4k RAM
Region LowMem 4k 636k RAM
Region HighMem 1M 1M RAM

Locate BootCode BIOSBOOT.EXE LowMem
Locate BootData SystemData LowMem
Locate DiskBuffer DiskBuffer LowMem
Locate Header Hello LowMem
```

⁸ Depending on your directory structure, you may have to use the -L and -I options to enable BCC32 to find all required header and library files.

```
Locate NTSection CODE HighMem
Locate NTSection DATA HighMem
Locate Stack Stack HighMem 16k
Locate Heap Heap HighMem
```

Now we can locate using command⁹:

```
RTLloc Hello
```

which will produce files HELLO.RTB (the relocated program image) and HELLO.LOC (a detailed map file).

Now a bootable disk with our program is created. Insert an empty, formatted disk in drive A: and type:

```
BootDisk Hello A:
```

Place the disk in the drive of the target computer and reboot it. RTTarget-32's boot code will then initialize the PC, read your program from the diskette, switch to 32-bit protected mode, and execute the program.

⁹ RTTarget-32's BIN directory must be included in your system's path to execute programs RTLloc and BootDisk successfully.

Chapter 2

The i386 Microprocessor

This chapter introduces some important properties of the Intel 386 CPU. The information presented here also applies to all higher CPUs compatible with the 386 (such as the 486, Pentium, etc.) and compatible CPUs from other vendors (such as National Semiconductor, AMD, etc.). RTTarget-32 requires only 32-bit protected mode. Thus, even CPUs that do not support all the features described here can be supported by RTTarget-32 (for example, the NS486SXF from National Semiconductor).

This chapter only covers some general concepts of interest to most programmers, mainly to introduce terms used throughout the rest of this manual. Many subjects are not covered (e.g., hardware multi-tasking, gates, etc.). The i386 is a very complex processor; for a complete description, please refer to the 386TM DX Microprocessor Programmer's Reference Manual from Intel (order number 230985).

The 386 can operate in one of three different modes described below; in addition, the basic mechanisms of address translation, memory protection, and interrupt handling are discussed.

Real-Address Mode

In this mode (also referred to as *Real Mode*), the 386 behaves like a very fast Intel 8086 CPU with a few new instructions and wider registers. The accessible address space is officially limited to 1 megabyte, just like on the 8086 (although undocumented features of the 386 allow addressing 4GB even in real mode). For compatibility with earlier CPUs, the 386 starts operating in this mode after power on or reset.

In real mode, the processor calculates the physical address of a memory reference by shifting the value of a segment register to the left by 4 binary digits and then adding the offset address to this value. Thus, two 16-bit values (segment and offset) are combined to form a single 20-bit physical address. There are no linear addresses in real mode.

Under RTTarget-32, this mode is only used in the boot code. The boot code starts executing in real mode and carries out most of the initialization. Subsequently, it switches to 32-bit protected mode and never switches back.

Virtual 8086 Mode

Virtual 8086 mode emulates the real-address mode. However, some protected mode features of the 386 are in effect. For example, paging is enabled to allow the virtual 8086 machine to run anywhere in the physical address space. The IDT is also in effect to route all interrupts to native protected mode. The GDT and LDT are not used. Address calculation works just as in real mode; however, instead of physical addresses, linear addresses subject to paging are generated by combining segment and offset values.

Protected Mode

This mode was first introduced with the 80286 processor. The differences between this mode and real-address mode are *Protection* and a larger address space. Except for some very minor differences, the instruction set available in protected mode is the same as in real mode.

Segment registers are interpreted differently than in real mode. They hold *Selectors*, which are indices into one of two tables, the *Global Descriptor Table* (GDT) or the *Local Descriptor Table* (LDT). These tables contain *Segment Descriptors*. Each descriptor contains information about a *Segment* in memory. This information includes the start address (*Base*), size (*Limit*), access rights, etc., of the segment. Each time a segment register is loaded by the processor, the information about the corresponding segment is loaded from one of the two tables. Every memory reference to the segment is executed by retrieving the base of the segment and adding the offset to the base.

Memory protection is also implemented using the segment descriptors. First, the processor checks whether a value loaded in a segment register references a valid descriptor. Then it checks that every linear address calculated actually lies within the segment. Also, the type of access (read, write, or execute) is checked against the information in the segment descriptor. Whenever one of these checks fails, exception (interrupt) 13 (hex 0D) is raised. This exception is called a *General Protection Fault* (GPF).

16-Bit Protected Mode

This is the only protected mode available on 80286 processors. Segments can have any length between 1 and $2^{16} = 64$ kilobytes. A segment base has 24 bits on an 80286 CPU, limiting the available address space to 16 megabytes. On 386 and higher CPUs, a segment base can have 32 bits. Thus, even in 16-bit protected mode, the complete 32-bit address space of 4 gigabytes is available (although many segments are required to use the complete address space).

Near pointers are 16-bit offsets interpreted relative to a segment register. Far pointers consist of a 16-bit selector and a 16-bit offset.

32-Bit Protected Mode

The 386 introduced 32-bit protected mode, the only mode supported by RTTarget-32. The difference from 16-bit protected mode is that the size of segments is no longer limited to 64k; rather, a segment can be up to 4 gigabytes in size. Thus, a single segment can be used to address the complete address space.

Near pointers are 32-bit offsets which are also interpreted relative to a segment register. Far pointers consist of a 16-bit selector and a 32-bit offset for a total of 48 bits.

In 32-bit protected mode, the address space is still segmented. However, since all available memory can be addressed with a single segment, the CPU can be set up such that segmentation can (almost) be ignored by the programmer. All segment descriptors to be used are initialized to refer to a segment starting at linear address 0 and extend over the complete address space or at least to the highest address that the system needs to access. Such an environment is referred to as a *flat memory model*. RTTarget-32 (and Win32) use this flat memory model.

Descriptors and Descriptor Tables

The 386 maintains three different descriptor tables: the *Global Descriptor Table* (GDT), the *Local Descriptor Table* (LDT), and the *Interrupt Descriptor Table* (IDT). The IDT can hold up to 256 interrupt, trap, or task gates; GDT and LDT can hold up to 8191 descriptors. Most GDT and LDT entries hold segment descriptors, although other descriptors (gates, TSS, etc.) are possible. Among other things, a segment descriptor contains the following information:

- the linear start address of the segment (the *Base*)
- the size of the segment (the *Limit*)
- the descriptor privilege level (the *DPL*)
- execute, read only, or read/write access permission

The main purpose of the IDT is to hold *Interrupt* or *Trap Gates*. Basically, these gates simply point to the entrypoint of an interrupt service routine.

RTTarget-32 manages a GDT with a total of 16 descriptors.

The following table summarizes the properties of RTTarget-32's GDT descriptors:

Index	Selector	DPL	Description
0h	0h	-	Reserved by Intel (NULL selector)
1h	8h	0	Ring 0 code segment
2h	10h	0	Ring 0 data segment
3h	1Bh	3	Ring 3 code segment
4h	23h	3	Ring 3 data segment
5h	28h	0	RTTarget-32 Boot Code TSS
6h	30h	0	Reserved for RTKernel-32 (LDT)
7h	3Bh	3	Callgate to call Ring 0 from Ring 3
8h	40h	3	BIOS Data Segment (Base == 400h)
9h	4Bh	3	Win32 main thread TEB segment
Ah	50h CPL ¹⁰	CPL	16-bit call stub segment ¹¹
Bh	5Bh	3	16-bit data stub segment ¹¹
Ch	60h CPL ¹⁰	CPL	16-bit PnP BIOS code segment ¹¹
Dh	6Bh	3	16-bit PnP BIOS data segment ¹¹
Eh	-	-	Reserved
Fh	7Bh	3	Reserved for MetaWINDOW

RTTarget-32's IDT has 128 entries as follows:

Index	Description
0h - 1Fh	CPU Exceptions
20h	Reserved
21h	DOS Emulation
22h - 30h	Reserved
31h	DPML Emulation
32h - 3Fh	Reserved
40h - 4Fh	IRQ 0..15
50h - 5Fh	Reserved for IRQ 16..31
60h	RTTarget-32 Boot Code API
61h	RTTarget-32 Program Terminate
62h - 6Ch	Reserved by RTTarget-32
6Dh - 6Eh	Reserved by RTTarget-32 Debug Monitor
6Fh	Reserved by RTTarget-32 (Boot Code Data Base)
70h - 7Fh	Free for Application Use

By default, RTTarget-32 does not maintain an LDT.

¹⁰ Current Privilege Level. The privilege level the program is executing at.

¹¹ Needed by RTTarget-32 to call 16-bit PnP BIOS services. Applications which do not call any PnP BIOS functions can use these selectors for other purposes.

Privilege Levels

To provide a higher degree of control for protection, protected mode defines privilege levels: *Descriptor Privilege Levels* (DPL), *Current Privilege Levels* (CPL), and *Input/Output Privilege Levels* (IOPL). Four different levels (0 to 3) are defined. A higher numerical value implies a lower privilege level. The DPL has already been introduced, it is stored in each segment's descriptor. The CPL is the privilege level at which the CPU is currently running (also frequently referred to as the *Ring* in which a program is running). It corresponds to the DPL of the code segment being executed. The low 2 bits of the CS register hold the CPL.

In the flat memory model, the application will usually not reload segment registers and is consequently not concerned with segment level protection. However, the CPL also controls access to some privileged instructions. Some instructions can only be executed at CPL 0. Also, the CPL affects how page-level protection functions.

The IOPL defines the minimum CPL required to directly access I/O ports and to execute *I/O Sensitive Instructions* (IN, INS, OUT, OUTS, CLI, STI). In addition, the POPF instruction behaves differently, depending on CPL and IOPL. The IOPL is maintained by the CPU in the EFLAGS register.

RTTarget-32 can run programs at CPL 0 or 3. IOPL is always initialized to 3, allowing the program to use I/O ports and I/O sensitive instructions without restrictions at any CPL.

Paging

Paging allows regions of memory to be mapped to different locations in the physical address space. In addition, the memory access rights can be controlled, much as they can for segments. However, since the flat memory model uses only one segment, page level protection is better suited here.

The paging mechanism uses *pages* (4096 bytes) of memory as the smallest mappable unit. Each page can have one of the following access rights: none, system read only, system read/write, user read only, or user read/write. System access means that only software running at CPL 0 can access the memory. Write access checking is not performed at CPL 0. This is important to note, since it implies that memory cannot be protected if applications run at CPL 0.

The following table shows the effective page access privileges enforced by the CPU at run time for code executing at CPL 0 or 3:

	NoAccess	SysRead	System	ReadOnly	ReadWrite
CPL 0	NoAccess	ReadWrite	ReadWrite	ReadWrite	ReadWrite
CPL 3	NoAccess	NoAccess	NoAccess	ReadOnly	ReadWrite

Each column represents a particular access privilege value for a page set in the page table. The rows show which privilege actually applies at a particular CPL. Any violation will trigger exception 14 (page fault) at run-time.

Unlike segmentation, paging is optional and must be enabled at boot time to become effective. RTTarget-32 supports running either with or without paging.

Virtual, Linear, and Physical Addresses

The 386 memory management can become quite confusing. Here is a summary of the different types of addresses and how one type is translated to another:

Virtual addresses are used by an application program. They consist of a 16-bit selector and a 32-bit offset. In the flat memory model, the selectors are preloaded into segment registers CS, DS, SS, and ES, which all refer to the same linear address. They need not be considered by the application. Addresses are simply 32-bit near pointers.

Linear addresses are calculated from virtual addresses by segment translation. The base of the segment referred to by the selector is added to the virtual offset, giving a 32-bit linear address. Under RTTarget-32, virtual offsets are equal to linear addresses since the base of all code and data segments is 0.

Physical addresses are calculated from linear addresses through paging. The linear address is used as an index into the *Page Table* where the CPU locates the corresponding physical address. If paging is not enabled, linear addresses are always equal to physical addresses. Under RTTarget-32, linear addresses are equal to physical addresses except for remapped RAM regions (see Chapter 3, *Virtual Command* and *FillRAM Command* for details).

Consequently, addresses used by the application are equal to physical addresses under RTTarget-32. Specifically, this will be true for any device access (such as video RAM, dual-ported RAM, some I/O boards, etc.) The only exception is remapped RAM pages if they have been requested in the locate process.

Exceptions and Interrupts

The 386 supports *Exceptions*, *Software Interrupts*, and *Hardware Interrupts*, which are summarized by the term *Interrupt*. Interrupts are numbered 0 to 255. They are events that transfer control to an *Interrupt Handler* (or *Interrupt Service Routine, ISR*) which must handle the event. The interrupt handler's address is read from the IDT. Each descriptor in the IDT contains a pointer to the corresponding handler (along with some supplemental information).

Exceptions are triggered by the CPU in case of an error. For example, if a program attempts to write to a memory location for which it only has read access, the CPU will trigger an exception. The exception could be handled by an operating system to abort the misbehaved program or activate a debugger to allow further investigation of the problem. RTTarget-32's boot code will initialize all exception vectors to point to a routine that simply displays a register dump and then stops. If a program wants to handle such exceptions, it can instruct RTTarget-32 to map CPU exceptions to Win32 exceptions. The Debug Monitor, RTTarget-32's debugger interface, will handle all exceptions and allow debugging the cause of the exception.

Software interrupts are explicitly triggered by a program using the INT instruction. Actually, this is similar to a procedure call. However, the address of the procedure to be called is found in the IDT and a change of privilege level can occur in the call. RTTarget-32 defines several interrupts for its API and for emulating subsets of some other APIs such as DOS and DPML. The use of interrupts allows the application to run at CPL 0 or 3 while the RTTarget-32 boot code (which handles some API requests) always runs at CPL 0.

Hardware interrupts are triggered by some hardware external to the CPU. The most significant difference from exceptions and software interrupts is that hardware interrupts can occur at any time and at any place in the code. Since this may cause problems, the processing of hardware interrupts can be suspended temporarily using the CLI/STI instructions. RTTarget-32's boot code installs dummy interrupt handlers and displays a warning message if an interrupt occurs (exception: timer and keyboard interrupts on IRQ 0 and 1 are ignored). The application should install interrupt handlers before any external hardware will generate interrupts. Interrupt handlers should not chain to the RTTarget-32 boot code handlers.

Chapter 3

RTLoc: Locating a Program

This chapter describes the command line utility RTLoc. Its purpose is to locate a Win32 executable program to run on the target.

Invoking RTLoc

To invoke RTLoc, use the following command line:

```
RTLoc [Options] Application [ConfigFile...]
```

Application is the name of the program to locate. If no path is specified, file *Application.EXE* must be present in the default directory or in the directory RTLoc was loaded from.

Following *Application*, any number of configuration files may be specified. They are searched for in the default directory, and, if not found there, in the directory from which RTLoc was loaded (usually RTTarget-32's BIN directory). If no configuration file is specified, *Application.CFG* is assumed.

RTLoc will produce two files: *Application.RTB* (the relocated binary image of the program) and *Application.LOC* (a text file containing information about the relocation process). Optionally, an Intel HEX or binary file can also be generated.

Example:

```
RTLoc -Rp Hello myconf.cfg
```

RTLoc Options

The general syntax for an option is:

```
-X[+|-]
```

where X is one of the options listed below. The option may be followed by a minus sign to disable it or a plus sign to enable the option. If neither sign is supplied, RTLoc defaults to a plus (enable). Any number of options may be specified on the command line.

The options are:

- o ROMable, default: enabled if a *HexFile* or *BinFile* command is present, disabled otherwise. If this option is enabled, RTLoc will issue a warning message if the application is not ROMable. If it is disabled, a warning is produced if any data is placed in ROM.
- b Binary, default: disabled if a *HexFile* or *BinFile* command is present, enabled otherwise. Controls the generation of an RTTarget-32 Binary File (.RTB file). .RTB files are required for the *Reserve* command, cross debugging, RTRun, and BootDisk.
- g Debug symbol conversion, default: disabled if a *HexFile* or *BinFile* command is present, enabled otherwise. Controls the generation of debug symbol tables for the RTTarget-32 debugger RTD32. Symbol table conversion is only required for Microsoft and Watcom compilers. Disabling this option can speed up RTLoc significantly. Use this option if you are using Microsoft or Watcom C/C++ and you do not need to debug the program with RTD32.
- c Compression, default: enabled if compression code is located, disabled otherwise. Controls whether copied sections should be compressed.
- d Discard discardable entities, default: enabled. Discardable entities are only needed for program initialization. If they are discarded, their address space can be reused by the program's heap and stack. Disabling this option prevents RTLoc from allocating the same address space to a discardable entity and the heap or stack. Discardable entities are: copied sections, decompression code and data, boot vectors, and the disk buffer. Discardable entities in RAM are allocated top-down (as opposed to bottom-up for all other entities).

h	Hex files based, default: disabled. Intel hex files contain address information to inform the reading program or device (usually an EPROM programmer) of the addresses to receive data. Usually, an EPROM programmer will expect all addresses to be relative to the start of the EPROM chip. However, other software (such as download utilities) or devices may interpret these addresses as relative to the target system's physical address space. Adding option -h to RTLoc's command line or configuration file will produce hex files with addresses of the target's address space. If the option is disabled (default), addresses relative to the start address given in the HEXFILE directive are generated.
t	Truncate bin files, default: disabled. If enabled, RTLoc will truncate bin files to the size actually used instead of writing out a file of the size specified in the BinFile command(s).
s	Start address record in hex files. When this option is enabled, the execution start address (the address of the boot vector) is written into hex files for each hex file which contains a boot vector (<i>Locate BootVector...</i> command). If option -s is not specified, RTLoc defaults to -s+ if such a record can be generated and to -s- otherwise. A valid start address record cannot be generated for targets booting in real mode with a boot vector outside the real mode address space. However, even in this case, a start address record can be forced by specifying -s+ on the RTLoc command line.
q	Quiet, default: disabled. If enabled, RTLoc will write no messages to standard output; however, messages are still written to the LOC file.
w	Warnings, default: enabled. If disabled, no warning messages are issued. This option also affects the LOC file.
i	Information messages, default: enabled. If disabled, no information messages are issued. This option also affects the LOC file.
m	Max messages, default: disabled. Usually, RTLoc will stop when more than 20 warning or error messages have been encountered. With this option enabled, there is no upper limit for the number of warnings.
DSym[=Val]	Defines a preprocessor symbol. Such symbols can be used in configuration files for symbol substitution or for conditional processing with #ifdef.
FPath	Search for data files in <i>Path</i> . The path given in this option is used by RTLoc to search files given in <i>Locate File...</i> commands.
+Cmd	Process string <i>Cmd</i> as a configuration file command. This option does not follow the same syntax as other options. It must start with character "+" and it cannot have a trailing "+" or "-". The string <i>Cmd</i> may not contain blanks or the whole option must be quoted.
Rc	Configuration report, default: enabled. Controls the generation of a configuration report in the LOC file.
Rr	Raw configuration report, default: disabled. If enabled, all configuration file lines read will be listed in the configuration report of the .LOC file, even if they are not executed due to an #ifdef 0.
Ri	When this option is enabled, all preprocessor symbols are replaced by their respective values in the configuration report.
Ry	When this option is disabled with -Ry-, the preprocessor symbol list at the end of the configuration report is suppressed.
Re	EXE file report, default: enabled. Controls the generation of an EXE file report in the LOC file.
Rf	Fixup table report, default: disabled. Controls the generation of a fixup table report in the LOC file.
Rd	Dynamic link report, default: disabled. Controls the generation of a link report for imported functions.
RL	Locate report, default: enabled. Controls the generation of a locate report in the LOC file.

R _o	Compression report, default: enabled. Controls the generation of a compression report in the LOC file.
R _p	Page table detailed report, default: disabled. Controls the generation of a detailed page table report in the LOC file.
R _s	Summary page table report, default: enabled. Controls the generation of a summary page table report in the LOC file.
R _b	Boot code configuration report, default: enabled. Controls the generation of a boot code configuration report in the LOC file. The option is ignored if the application does not contain boot code.
R _a	Application output report, default: enabled. Controls the generation of an application output report in the LOC file.
R	Reports: all reports. Enables or disables all reports in the LOC file.
?	Shows a help screen with an options summary.

Options Command

Options are usually supplied on the RTLoc command line. However, you can also place them in a configuration file using the Options command:

```
Options = Option [,Option...]
```

Up to 15 options can be specified on one line and several Options commands can appear in a configuration file. Please note, however, that options placed in a configuration file take effect only after they have been processed. For example, if you wish to disable the Configuration Report using the Options command in a configuration file, all lines read up to the respective Options command will be included in the report.

Example for the Options command:

```
Options = -o -Rp
```

Configuration Files

The main source of information for RTLoc are configuration files. These files are line-oriented. Each line starts with a keyword followed by one or more parameters separated by spaces, tabs, commas, or equal signs. If a parameter contains embedded spaces, tabs, commas, or equal signs, it must be enclosed in single or double quotes. Blank lines are ignored. A comment can be placed in the file by preceding it with a double slash (/), slash asterisk (/*) or a semicolon (;). The comment extends to the end of the line.

Parameters placed in square brackets [] are optional. For example, the command

```
COMPort = Port [,Baudrate [,IRQ [,IOBase]]]
```

can accept 1, 2, 3, or 4 parameters. However, it is not valid to specify only the third, but not the second parameter. For example, if you want to specify the IRQ of the port, you **must** also supply the desired baud rate.

Specifying Numeric Values

Numeric parameters are assumed to be decimal by default. The following prefixes to numeric parameters can change the base of a number:

- o octal
- 0x hexadecimal
- \$ hexadecimal

The following suffixes are supported:

- b binary
- h hexadecimal

- k kilo (multiplies the number by $2^{10} = 1024$)
- p page (multiplies the number by $2^{12} = 4096$)
- M Mega (multiplies the number by $2^{20} = 1048576$)
- G Giga (multiplies the number by $2^{30} = 1073741824$)

For example, the following numbers are all identical: 16M, 0x1000p, 040000k, 16777216.

Embedded underscore characters in numeric values are ignored. For example, value 3E98A4Ch is identical to 0011_1110_1001_1000_1010_0100_1100b.

Numeric parameters may contain simple arithmetic expressions using operators +, -, *, /, %, &, |, ^ . For example, the following directives are legal:

```
Region  BootROM  64M-64k  64k-16  ROM
Region  Reset    64M-16      16      ROM
Virtual VMem      1G+1M+4k
HexFile Hex       4G-64k      64k
```

The rules for numeric expressions are:

- Pre- and postfixes are processed first.
- Operators are processed left to right without precedence rules. Parenthesis are not supported.
- There must not be any white space characters within a single numeric expression. For example "1k + 16" would be interpreted as three parameters: number 4096, string "+", and number 16.

Region names with an attribute can be used in place of a numeric value. Attributes .Start (address of start of region), .Size (size of region), and .End (address after the region) are supported.

Example 1:

```
Region  EPROM      4G-64k      64k  ROM
HexFile Program  EPROM.Start  EPROM.Size
```

The second line would evaluate to:

```
HexFile Program  FFFF0000h    10000h
```

Example 2:

```
Region  NullPage    0          4k          RAM
Region  MoreLowMem  640k-128k    128k          RAM
Region  LowMem      NullPage.End  MoreLowMem.Start-NullPage.End RAM
Region  Reset       4G-16        16          ROM
Region  Boot        4G-64k      64k-Reset.Size  ROM
Region  EPROM       Boot.Start-64k  64k          ROM
...
HexFile Program  EPROM.Start    EPROM.Size+Boot.Size+Reset.Size
```

which evaluates to:

```
Region  NullPage    00000000h  00001000h  RAM
Region  MoreLowMem  00080000h  00020000h  RAM
Region  LowMem      00001000h  0007F000h  RAM
Region  Reset       FFFFFFFFh  00000010h  ROM
Region  Boot        FFFF0000h  0000FF0h   ROM
Region  EPROM       FFFE0000h  00010000h  ROM
...
HexFile Program  FFFE0000h  00020000h
```

Preprocessor Directives

RTLoc supports the following basic preprocessor commands with the same syntax as C/C++:

```
#include Filename

#define Symbol [Value]
#defineN Symbol Expression
#undef Symbol

#if Expression
#elif Expression

#else
#endif

#ifdef Symbol
#ifndef Symbol
#elifdef Symbol
#elifndef Symbol

#ifsection [ModuleName.]SectionName
#elifsection [ModuleName.]SectionName

#error String
```

Expressions for `#if` and `#elif` are numeric unsigned 32-bit integers. The supported operators are `+`, `-`, `|`, `&` (high precedence) and `==`, `>`, `>=`, `<`, `<=`, `||`, `&&` (low precedence). Parenthesis are not supported.

`#defineN` evaluates the specified expression and then assigns the result to *Symbol*. By contrast, `#define` merely associates one string with another without any interpretation or evaluation.

RTLoc will expand symbols defined with `#define`/`#defineN` or RTLoc's command line option `-D`. However, such expansions cannot change the number of tokens. A single token is always expanded to a new single token, whereby numeric expressions are considered a single token. The following symbols are predefined:

```
OUTNAME = <base output directory and file name without extension>
APPLICATION = <Application name specified on RTLoc command line>
RTT32_VER = <RTTarget-32 version multiplied by 100>
RTLOC = TRUE
TRUE = 1
FALSE = 0
```

`#ifsection` and `#elifsection` evaluate to true if section *SectionName* exists in the DLL specified by *ModuleName*. If no *ModuleName* is given, the main .EXE file is assumed.

Macros

Configuration file macros can be defined using the Macro and EndM keywords:

```
Macro Name [Parameter...]
...
EndM
```

Up to 8 parameters and 256 source lines per macro are supported. Macro declarations cannot be nested.

Macros are expanded by specifying the macro name followed by parameters. Numeric parameters are evaluated and then passed (they are not passed as strings). Not all parameters need to be specified. Inside the macro, `#ifdef` can be used to test if a parameter was specified. Macro expansions can be nested up to 15 levels deep. Please refer to configuration file `Bin\Bootdbg.cfg` for several examples of using macros.

Defining the Target Hardware

To locate a program, RTLoc must know the types of physical memory available at specific addresses. This is done using the *Region* command in a configuration file. In addition, RTLoc can remap some memory by creating virtual regions and appending unused RAM to enlarge a region.

Region Command

The Region command defines the properties of a region of consecutive physical address space. The syntax is

```
Region = RegionName, Address, Size, MemType [,Access]
```

Parameter *RegionName* is a string with a name you wish to assign to this region. The only restriction on name usage is that no two regions can have the same name.

Parameter *Address* is a numeric value specifying the start address of the region. *Size* is the number of bytes it occupies. Parameter *MemType* can have any of the following values:

RAM The region contains RAM. RTLoc assumes that this region can be used for any part of a program requiring read only or read/write access.

ROM The region contains ROM. RTLoc assumes that this region can only be used for read only data such as code.

Device The region contains a device. RTLoc makes no assumptions about such regions. Use this memory type for video RAM, memory-mapped devices, etc.

The optional parameter *Access* can be used to specify the type of access your program needs for the region. If not specified, RTLoc will assume *Assign* access:

NoAccess The region is inaccessible. RTLoc will issue a warning if you attempt to locate parts of your program in this region.

SysRead The region cannot be accessed by the application, but the RTTarget-32 Boot Code has read only access. This access is useful for protected data such as the boot code itself.

System The region cannot be accessed by the application, but the RTTarget-32 Boot Code has read and write access. This access should be used for protected data managed by the boot code (e.g., the boot data or the page table).

ReadOnly The application will have read only access to this region.

ReadWrite The application will have read/write access to this region.

Assign RTLoc will assign appropriate access rights to parts of the region as they are allocated to program entities. For example, if a code section is located into this region, that part will be assigned read only access. If a data section is also located here, **only** the data section will be assigned read/write access. Unused parts of the region get NoAccess and - if the region consists of RAM - can be remapped using the FillRAM command. It is recommended to use Assign for all RAM and ROM regions.

Access rights are checked statically by RTLoc and at run time by the CPU if paging is enabled (see section *PageTable* for details). RTLoc will make sure any access rights you have assigned are compatible with the respective memory type.

Examples for the Region command:

```
Region = RealModeVectors    0    1k RAM    NoAccess
Region = BIOSDataArea      1k    3k RAM    NoAccess
Region = LowMem             4k 636k RAM    Assign
Region = ColorGraphic A0000h 64k Device NoAccess
Region = MonoText          B0000h 32k Device NoAccess
Region = ColorText         B8000h 4k Device ReadWrite
Region = Ethernet          D8000h 16k Device ReadWrite
Region = BIOS              F0000h 64k ROM    NoAccess
Region = HighMem           1M    3M RAM    Assign
```

The above Region commands are typical for an AT class PC with 4 megabytes of memory. It assumes that no monochrome video hardware is installed (or shouldn't be used) and that the application will not use graphics mode.

Virtual Command

The Virtual command defines a region of memory which does not exist physically, but will be created by remapping physical memory:

```
Virtual = RegionName, Address
```

Parameter *RegionName* is a string with a name you wish to assign to the virtual region.

Parameter *Address* is a numeric value specifying the start address of the virtual region. It must be page (4096 byte) aligned. Since no two regions (either physical or virtual) may overlap, it is recommended to use an address higher than any physical memory on the target.

Virtual regions initially have a size of zero and no physical memory is associated with them. As program entities are located to the region, some physical memory is remapped from a physical region to successfully build the virtual region. Virtual regions can also acquire memory through the FillRAM command. The advantage of virtual regions is that they can consist of different physical memory portions (for example, a mixture of RAM and ROM). The address space created with a virtual region is independent of the underlying physical memory structure.

Paging must be enabled using the Locate PageTable command to be able to use virtual regions (see section *PageTable* in this chapter). All program entities to be allocated to a virtual region must be page-aligned. The only entities supported in virtual regions are Section, NTSection, Stack, Heap, File, and Nothing.

Example for the Virtual command:

```
Region  LowMem      4k 636k RAM    Assign
Region  MyEPROM F0000h 64k ROM    NoAccess
Region  HighMem     1M   3M RAM    Assign
Virtual VMem        4M
FillRAM VMem
Locate  NTSection CODE VMem->MyEPROM
Locate  NTSection DATA VMem->HighMem
Locate  Stack      Stack VMem->LowMem 16k
Locate  Heap       Heap  VMem
```

Three physical and one virtual regions are defined. The first three Locate commands place entities in the virtual region and also specify the physical region the memory should be taken from. The heap does not specify a physical region, but since region VMem receives all unused RAM through the FillRAM command, all available memory will be allocated to the heap.

FillRAM Command

The FillRAM command instructs RTLoc to remap unused RAM memory. The syntax is:

```
FillRAM = RegionName
```

Parameter *RegionName* is the name of a physical or virtual region to which all RAM pages remapped from other regions should be appended. You can use this command to create larger regions of consecutive RAM if your RAM address space is fragmented. For example, an AT class computer has RAM from 0 to 640k and some more starting at 1MB. If not all RAM in the region below 640k is used, the unused pages of memory can be appended to the extended memory region above 1MB.

Example:

```
Region = LowMem      4k 636k RAM
Region = HighMem     1M   3M RAM
FillRAM = HighMem
```

The following example creates a region consisting of remapped pages only:

```
Region = LowMem      4k 636k RAM
Region = HighMem     1M   3M RAM
Region = Remapped   16M   0   RAM
FillRAM = Remapped
```

or alternatively:

```
Virtual = Remapped 16M
FillRAM = Remapped
```

Please note, however, that remapped RAM can only be used for the program's stack and heap. FillRAM requires paging (see section *PageTable* in this chapter).

Defining Program Location

After the hardware memory layout has been defined, RTLoc needs to know whether the program requires additional DLLs and in what regions which components of your program are to be located. This is done using the DLL, Align, Reserve, and Locate commands described in this section.

DLL Command

Apart from the application's .EXE file, RTLoc can be instructed to add a DLL to the program's image:

```
DLL = DLLName
```

If DLLName has no file name extension, .DLL is assumed. If no path information is supplied, RTLoc will search for the DLL in the default directory and then RTLoc's load directory. Up to 31 DLLs can be used by an application. All sections of the DLL required by the program must be located using *Locate Section* or *Locate NTSection*, just like for the main EXE.

For additional information about DLLs, please refer to Chapter 9, *Using DLLs through RTLoc*. An alternate method of using DLLs is described in Chapter 9, *Loading DLLs through a File System*.

Align Command

Align specifies the default alignment for the mapping process:

```
Align = Value
```

The starting address of each entity to be mapped is rounded up to a multiple of Parameter *Value*, which must be a power of two. The smallest supported value is 4, the default is 4096 (one page). Note that you can override the alignment in each *Locate* command.

RTLoc uses one page as the default alignment to make sure that each page will never be used by more than one program entity. This allows optimum page protection. For example, assume you have a CODE and a DATA section located to the same region. Code usually has read only access while data needs read/write access. If the alignment is less than one page, the start of DATA could be located in the same page as the end of CODE, forcing the access for that particular page to have the higher value (read/write). Thus, parts of your code would not be protected from write accesses.

Of course, the disadvantage of using a large alignment value is wasted memory. For example, if the next available address of a region is just one byte following a page boundary, 4095 bytes are wasted to map a new entity.

Reserve Command

The Reserve command instructs RTLoc to reserve memory for another program:

```
Reserve = ApplicationName
```

Parameter *ApplicationName* specifies the name of the application whose address range must be protected. RTLoc will search for the file *ApplicationName*.RTB in the default directory and in RTLoc's load directory.

This command is typically used to reserve space for the RTTarget-32 Monitor when the program is to be debugged or run via download. When running under the debugger's control, two programs must coexist on the target: the Monitor and the application under test.

Example:

```
Reserve = Monitor
```

The reserved program must not have remapped pages (thus, FillRAM or virtual regions cannot be used by it; however, they **can** be used by the application) and its heap (if any) must have Read/Write access. It must **not** use RTTarget-32's uncommitted memory support. The reserved program and the application must both use paging or must both run without paging.

Locate Command

The Locate command maps a program entity to one of the memory regions of the target hardware. The general syntax is:

```
Locate Entity Name Region[->PRegion] [Size [Align [Access [Alloc]]]
```

Parameter *Entity* specifies what kind of entity is to be located. All possible values of *Entity* are discussed in the following sections. Parameter *Name* specifies the name of the entity. For some entities, the name can have a special meaning (see the sections on different entities below). Parameter *Region* specifies in which region the entity will be mapped. *Region* must have been defined in a previous Region or Virtual command. If *Region* references a virtual region, the name of the physical region for the entity must also be specified as parameter *->PRegion*. There must be no spaces between parameters *Region* and *->PRegion*. Parameter *Size* specifies how many bytes RTLoc should allocate to the entity. This parameter is optional, since RTLoc can automatically determine the size for most entities; specify it only to override RTLoc's defaults. If zero is given, RTLoc will try to determine the size automatically. Parameter *Align* can be used to override the default alignment for this particular entity. If zero is specified, the default alignment is used. Optional parameter *Access* specifies the type of access you want RTLoc to assign to this entity. Supported values are: *Assign*, *NoAccess*, *SysRead*, *System*, *ReadOnly*, *ReadWrite* (see section *Region Command* for details). The default is *Assign* which instructs RTLoc to select the appropriate access value automatically. Optional parameter *Alloc* supports values *AllocDefault*, *BottomUp*, and *TopDown*. If it is not specified, *AllocDefault* is assumed which in turn causes discardable entities in RAM to be allocated *TopDown* and all others *BottomUp*. *BottomUp* allocated entities will receive the lowest available address in the specified region which *TopDown* entities will receive the highest available address.

All supported values for parameter *Entity* are discussed in the following sections.

Section

Locate Section and Locate NTSection will map data from the application .EXE or a .DLL file. The data in PE files (Win32's portable executable file format) is divided into named sections. The *Name* parameter in a Locate Section command maps to a name in the PE file.

Example:

```
Locate Section CODE LowMem
```

This command will map section CODE from the EXE file to region LowMem. RTLoc will determine the type of access required and the size of section CODE.

What sections are required by a program depends on the compiler/linker used. RTLoc will write a complete list of all the PE file's sections to the LOC file in the EXE File Report. Appendix A contains an overview of the sections generated by each compiler/linker.

The complete syntax for the Name parameter in *Locate Section* and *Locate NTSection* commands is:

```
[ModuleName.]SectionName or  
[ModuleName.]#SectionNumber
```

The name of the section to be mapped may be prefixed with the name of the module containing the respective section. If omitted, the application's main .EXE file is assumed. Example:

```
Locate Section MyDLL.dll..idata LowMem
```

This command will map section .idata of module MyDLL.DLL. MyDLL.DLL must have been referenced in a previous DLL command.

Some linkers may produce several sections with the same name or the name may contain unprintable characters. In this case, the section's number can be specified instead of its name. Section numbering starts at 1 (see the EXE File Report in the .LOC file for a complete list of all sections). Examples:

```
Locate Section .text LowMem
Locate Section SomeDLL.dll.DATA HighMem
Locate Section MyLIB.DLL.#3 MyROM
```

The module's and the section's names are not case sensitive.

The *Size* parameter has a special meaning for Sections and NTSections: it is used as the segment index into the map file. If specified, RTLoc will try to determine the size of the section by interpreting the module's MAP file.

Example:

```
Locate Section CODE LowMem 1
```

The map file produced by the linker might contain the following information:

Start	Length	Name	Class
0001:00000000	000005969H	__TEXT	CODE
0002:00000000	000001300H	__DATA	DATA
0002:00001300	000000000H	__TLSCBA	TLSCBA
0002:00001300	000000024H	__INIT__	INITDATA
0002:00001324	000000000H	__INITEND__	INITDATA
0002:00001324	000000006H	__EXIT__	EXITDATA
0002:0000132A	000000000H	__EXITEND__	EXITDATA
0002:0000132C	000000000H	CONST	CONST
0002:0000132C	000000624H	__BSS	BSS
0002:00001950	000000000H	__BSEND	BSS

RTLoc will find that segment 1 has a size of 5969h and will use this value instead of that from the PE file. This has the advantage of reducing the program's memory requirements, since some Win32 linkers will round up a section's size to a multiple of 512.

NTSection

NTSections are similar to Sections. However, RTLoc will make sure the relative offsets of the sections are not changed.

Example:

```
Region MyRAM 1M 3M RAM Assign
Locate NTSection CODE MyRAM
Locate NTSection DATA MyRAM
```

If, for example, the PE file specifies section CODE to start at address 20000h and section DATA at 30000h, RTLoc will ensure that DATA is located exactly 10000h after CODE. This strategy of mapping is highly compatible with the method used by Win32. If you intend to use source-level debugging, *Locate NTSection* must be used instead of *Locate Section*.

The exact syntax for the *Name* parameter is identical to the *Locate Section* command (see previous section).

The disadvantage of NTSections is that all NTSections must be located in the same region and that more address space is used if the linker uses a generous alignment (e.g., Borland C++ 4.5 uses 64k alignment). However, wasted RAM can be recovered if FillRAM is used. Programs located using NTSections are ROMable only if used with the *Locate Copy* command (see section *Copy*) for all sections or if located to a virtual region.

Generally, command *Locate NTSection* is recommended over *Locate Section* (see also Chapter 9, *Choosing a Locate Method*).

Header

Every application needs a header record which is automatically generated by RTLoc. It contains information about how the program is mapped. This information is needed by the boot code to correctly initialize and invoke the program.

The *Locate Header* command tells RTLoc where to place the header. Parameter *Name* is ignored but must be present (you can use any arbitrary name). Parameter *Size* is also ignored; RTLoc determines the size automatically. The default *Access* is *ReadOnly*.

Example:

```
Locate Header "Test Program Header" LowMem
```

BootCode

If an application must be booted (e.g., it doesn't run under the Monitor which has booted the target hardware already), you must include boot code. The *Name* parameter is the filename of a DOS EXE file containing the boot code. Three standard boot codes, located in the BIN directory, come with RTTarget-32:

BOOT.EXE	This boot code is suitable for targets that boot in real mode from ROM (EPROM or flash). It contains no BIOS dependencies or BIOS support.
BIOSBOOT.EXE	Boots in real mode with BIOS support (e.g. booting from disk, DOS, or BIOS extension). This boot code includes support for RTLoc's GMode command and function RTGetGMode(), function RTGetExtMem(), function RTCMOSExtendHeap(), and A20 switching.
PMBOOT.EXE	Boots in protected mode. This boot code is used on NS486 systems or when the CPU has been put in 32-bit flat protected mode by some other means. This boot code expects to be invoked at CPL 0 with CS, DS, ES, and SS set to zero-based flat 32-bit segments.

The boot code must be paragraph (16-byte) aligned.

Depending on the desired boot method, the following boot code and other entities must be located:

Boot from disk	BIOSBOOT.EXE, BootData, and DiskBuffer
Boot from BIOS extension	BIOSBOOT.EXE, BootData, and BIOSVector
Boot From MS-DOS	BIOSBOOT.EXE, BootData, optionally a DiskBuffer
Boot from ROM	BOOT.EXE or PMBOOT.EXE, BootData, and BootVector

By default, all three standard boot codes will detect an installed FPU, initialize interrupt controllers at port addresses 20h and A0h, install dummy interrupt handlers for IRQ 0 and 1, and will set up the CPU to run applications at CPL 3. BIOSBOOT.EXE also enables A20, if it is locked. See section *BOOTFLAGS Command* later in this chapter on how to change this behaviour.

Apart from the interrupt controllers, none of the standard boot codes contain any chipset initialization. Demo programs ExLED, HelloSC400, HelloSC520, and NSHello show how to initialize a controller using configuration file commands.

BootData

If boot code was located, a boot data section is also required. It must also have at least 16-byte alignment and be located in address space addressable in real mode.

Parameters *Name* and *Size* are ignored.

BootVector

If the target must boot without BIOS support, a boot vector must be located using this command. It will usually be located at the end of the physical address space minus 16, the location all Intel 80x86 CPUs boot from.

Parameter *Name* is ignored. Parameters *Align* and *Size* must be at least 16 (*Size*'s default is 16). *Access* should be *SysRead*.

If RTLoc finds that the application uses real-mode boot code, it will generate a 16-bit near jump to the boot code's entrypoint. This requires that the boot code and the boot vector are located within the same 64k segment. If the protected mode boot code is used (e.g., for the NS486SXF CPU, which boots in protected mode), a 32-bit near jump is used.

BIOSVector

BIOSVector will generate a far jump to the boot code's entrypoint. However, unlike the *Locate Boot-Vector* command, the produced code has the format of a BIOS extension. It should be located at an address scanned by the BIOS for an extension. These are usually all 2k aligned addresses in the range C8000h - DF800h.

Parameter *Name* is ignored. Parameters *Align* and *Size* must be at least 16 (*Size*'s default is 16). *Access* should be SysRead.

The BIOS extension's size is automatically rounded up to a multiple of 512 bytes by RTLoc (this is required by the BIOS). Since this can be wasteful, RTLoc can automatically combine the BIOSVector with the boot code if these two entities are located exactly adjacent to one another in the same region.

Example:

```

Region NullPage      0      1p RAM
Region LowMem        4k    636k RAM    Assign
Region ColorGraphic A0000h   64k Device ReadWrite
Region MonoText     B0000h    4k Device ReadWrite
Region ColorText    B8000h    4k Device ReadWrite
Region BIOSExt      CA000h    8k ROM    Assign
Region SomeROM      D0000h   64k ROM    Assign
Region BIOS         F0000h   64k ROM    NoAccess
Region HighMem      1M      1M RAM    Assign

Locate BIOSVector BootVector  BIOSExt 0 16 Assign BottomUp
Locate BootCode   BIOSBOOT.EXE BIOSExt 0 16 Assign BottomUp
Locate BootData   BootData    LowMem
...

```

In this example, a BIOS extension of 8k is defined at address CA000h. Both BIOSVector and BootCode are located with 16-byte alignment in this region. RTLoc finds that they are adjacent and will combine the two.

Alternatively, BIOSVector and BootCode could have been placed in separate regions (e.g., BIOSExt and SomeROM, respectively).

DiskBuffer

The *Locate DiskBuffer* directive is required for all programs that boot from hard disk or diskette. The name of this entity is ignored by RTLoc, but must be present. The disk buffer is required by the disk loader to read the application from disk. It must reside in memory addressable in real mode, must have a size of at least 8k and needs a minimum alignment of one disk sector (512 bytes). Example:

```
Locate DiskBuffer "Disk IO Buffer" MoreLowMem
```

The disk buffer is a discardable entity with a default size of 64k. If booted from floppy disk, a disk buffer which does not span a 64k boundary may result in improved boot time.

Stack

Every program needs a stack. Parameter *Name* is ignored. *Access* must be ReadWrite if specified. If parameter *Size* is not specified or zero, RTLoc will assign all unused memory of the region to the stack after all other entities have been located. Please note that if the stack and heap are located in the same region, at least one of the two **must** specify a size.

If the stack is mapped to a virtual region and command FillRAM is specified for the same region, the specification of a physical region can be omitted. In this way, the stack can span several disjoint areas of physical memory.

Examples:

```

Region  LowMem  4k  636k RAM
Region  HighMem 1M    1M RAM

Virtual VMem      2M
FillRAM VMem

Locate Stack S VMem           // all unused RAM
Locate Stack S VMem->LowMem    // all unused LowMem
Locate Stack S LowMem         // ditto
Locate Stack S VMem 16k       // 16k of any RAM
Locate Stack S VMem->HighMem 16k // 16k of HighMem

```

Heap

Programs linked with a run-time system need a heap. Parameter *Name* is ignored. If parameter *Size* is not specified or 0, RTLoc will assign all unused memory of the region to the heap after all other entities have been located. If stack and heap are located in the same region, at least one of the two **must** specify a size.

The default value for parameter *Access* is NoAccess. The heap manager must explicitly change page attributes at run time to make the memory accessible. This prevents the application from accessing heap memory before it has been allocated using malloc, new, or a similar function. However, this requires the page table to be located in RAM. If you plan to locate the page table in ROM, ReadWrite access must be specified for the Heap. In addition, each program that will run under another program's control (i.e., the program is referenced with a *Reserve* command by another program), must also commit its heap, because it will run using a foreign page table.

If the heap is mapped to a virtual region and command FillRAM is specified for the same region, the specification of a physical region can be omitted. In this way, the heap can span several disjoint areas of physical memory.

The examples given in section *Stack* work identically for the *Locate Heap* command.

PageTable

If you intend to use paging, a Locate PageTable command must be present in the configuration file. RTLoc will create a page table that will subsequently be used by the boot code when the program is run. A page table is required for RAM remapping, virtual regions, and page-level protection.

RTLoc will determine the size of the page table automatically. However, it can arrive at an incorrect value if the FillRAM command has been used or the configuration uses virtual regions. FillRAM or virtual regions can actually extend the available linear address space, making a larger page table necessary. In this case, RTLoc issues an error message. The problem is fixed by explicitly specifying the size. The required size is calculated as follows:

$$\text{Page Table Size} = 4096 * (1 + \text{Number of 4M regions touched})$$

Consider the linear address space to be divided into divisions of 4 megabytes each. The first division covers addresses in the range 0 to 4M - 1, the second addresses 4M to 8M - 1, etc. The whole 32-bit address space has 1024 such divisions. Each division overlapped by any region (physical or virtual) defined in the configuration file contributes to the *number of 4M regions touched* in the formula given above.

Example: Suppose you have a PC with 4M of RAM installed in the address range 0 to 4M - 1. The formula above will give:

$$4096 * (1 + 1) = 8192$$

This is the size of the page table calculated by RTLoc. However, if only a single page is appended to the last RAM region by a FillRAM command, the second 4M division of the linear address space is actually used. Thus, the correct page table size would be:

$$4096 * (1 + 2) = 12k$$

which must be specified in the Locate command.

If you are not sure of the proper page table size for your system, RTLoc can calculate it for you. Do not specify any size in the `Locate PageTable` command and run RTLoc. If you get one or more *Page table too small* errors, just specify the required value given in the error message in the `Locate PageTable` command and run RTLoc again.

The alignment for the page table must be 4k or larger. This is required by the 386 CPU. Parameter *Name* is ignored.

If no page table is located, paging is disabled for the application. In this case, the following features of RTTarget-32 and the CPU are not available:

- No run-time checks for read/write accesses.
- System data structures such as IDT and GDT cannot be protected against corruption.
- FillRAM, virtual regions, and RTTarget-32's virtual memory manager cannot be used.

If the page table is placed in ROM, the following restrictions apply:

- RTTarget-32's virtual memory manager cannot be used.
- RTTarget-32's memory mapping functions (e.g., `RTMapMem`) do not work.

Copy

To create ROMable programs, some entities must be copied from ROM to RAM before a program can run. In particular, this is true for initialized data. The `Locate Copy` command instructs RTLoc to create a copy of an entity already given in the configuration file. Optionally, the copy can be compressed to save ROM or disk space. Compression is used if decompression code and data are also located and compression has not been disabled using option `-c`. Parameter *Name* specifies the entity to copy.

Example:

```
Locate Section DATA LowMem
Locate Copy    DATA MyEPROM
```

Here, section DATA is allocated in region LowMem. However, the data associated with section DATA is placed in Region MyEPROM and copied to LowMem before the application is started.

Parameter *Size* is ignored. The only *Access* value supported is `SysRead` (which is also the default).

For the program's data, the `Locate Copy` command must be used to make the program ROMable. However, `Locate Copy` can be used for any program entity that has data associated with it. For code, this can be advantageous if the installed RAM is faster than ROM. If the code is copied from ROM to RAM (and consequently executed from RAM), the program will run faster. During the development phase of a program, copied entities can significantly reduce download times.

DecompCode

To enable data compression for copied sections, decompression code and decompression data must be located on the target. Example:

```
Locate DecompCode Expand LowMem
Locate DecompData Buffer HighMem
```

Both entities are discardable and thus will not reduce the amount of memory available to the application.

Parameter *Name* is ignored. The *Size* parameter's default value supplied by RTLoc cannot be overridden.

The following table summarizes all available entity types and specifies whether they have an image, can be copied or compressed, and whether they are discardable:

Entity	Image	Discardable	Copiable	Compressible
Section	Yes	No	Yes	Yes
NTSection	Yes	No	Yes	Yes
PageTable	Yes	No	Yes	Yes
File	Yes	No	Yes	Yes
Header	Yes	No	Yes	No
BootCode	Yes	No	Yes	No
BootVector	Yes	Yes	No	-
DecompCode	Yes	Yes	No	-
Copy	Yes	Yes	No	-
DiskBuffer	No	Yes	-	-
DecompData	No	Yes	-	-
BootData	No	No	-	-
Stack	No	No	-	-
Heap	No	No	-	-
Nothing	No	No	-	-

Just like the page tables, compressed page tables need some special treatment. Again, if a page table is compressed, RTLoc must make an estimate on how large the compressed image will be. Since the size of the page table can change during the locate process and because the achievable compression ratio varies, this estimate may be inaccurate. In this case, RTLoc will issue an appropriate error message and the size of the copied page table must be specified in the configuration file explicitly.

RTLoc will estimate the compressed image size of a page table to be 3% of the page table's size rounded up to the copied section's alignment. If this size is too small, an error message is issued. If a smaller value would save memory at the current alignment value, an information message is issued.

Please note that RTLoc's compression algorithm for page tables is very efficient. Typical compression ratios are 1% - 2%. In addition, the decompression of a page table is frequently faster than copying it without decompression.

The achievable compression of other entities varies with the type of data. Program code will typically compress to 50% - 60% of its original size.

DecompData

To enable data compression for copied sections, decompression data must be located along with decompression code. Parameter *Name* is ignored and RTLoc's default for the *Size* parameter cannot be overridden.

For additional information, please see the previous section on command *Locate DecompCode*.

File

RTLoc can include a data file in the program image. Parameter *Name* in a *Locate File* command specifies the File to include. At run time, the program can use standard file I/O operations to read the file. If ReadWrite access is specified (ReadOnly is the default), write access to the file is also possible, but the file's size cannot be changed. The program should open such a file using the name given in the *Locate File* command. Any path information is ignored at run time (but not by RTLoc).

Nothing

RTLoc can reserve memory on the target without associating it with anything using the *Locate Nothing* command. Parameter *Name* can be an arbitrary string. You will have to supply the size parameter because there is no default value. The default Access value is NoAccess.

Nothing sections can be located at run time using the `RTLocateSection` function. Use `Locate Nothing` when you need to reserve memory in a particular region.

Defining Program Options

Some additional options for program location can be defined using the commands *Init* and *Link* defined in this section. Both commands deal with imported or exported functions of the program.

Set Command

RTTarget-32 supports a static program environment. A configuration file can contain any number of SET commands to define environment variables:

```
SET Variable Value
```

where *Variable* is the name of the environment variable to define and *Value* is its value. Please note that you must quote "Value" if it contains blanks, semicolons, slashes, or any other whitespace or comment character.

Example:

```
SET PATH="C:\;C:\BIN"  
SET COMSPEC myprog.exe  
SET OS=RTTarget-32
```

Environment variables defined in this manner can be queried at run-time using the Win32 API function `GetEnvironmentStrings` or `GetEnvironmentVariable`, or equivalent functions of the run-time system.

The program environment cannot be changed dynamically at run time.

Commandline Command

RTTarget-32 also supports a command line string with command:

```
Commandline string
```

where *string* is the command line passed to the program. Please note that you must quote "string" if the command line contains blanks, semicolons, slashes, or any other whitespace or comment character.

Example:

```
CommandLine "hello.exe -q SomeParm"
```

The first part of the command line should always be the program's name. If no `CommandLine` directive is specified, RTLoc uses the .EXE file's filename.

The command line is available at run time through Win32 API function `GetCommandLine`. Each portion of the command line is passed to function `main()` in arguments `argc` and `argv`.

Init Command

Using the `Init` command, a function can be defined that will be executed before control is passed to the program's entrypoint:

```
Init [ModuleName.]FunctionName
```

FunctionName must be the name of an exported function without any parameters. If *ModuleName* is not specified, the function is assumed to reside in the main program. Otherwise, *ModuleName* must specify the main program's .EXE file name or the name of a DLL specified in a previous DLL command. Function name matching is case sensitive; module names are not. Exporting a function is achieved using the `__export` keyword for Borland and Watcom C/C++, with `_declspec(dllexport)` for Microsoft Visual C/C++, or listing them under the `EXPORTS` keyword in Delphi. Alternatively, a .DEF file can also be used for C/C++ programs. Please consult your compiler's documentation or the RTTarget-32 examples for details.

Some important restrictions apply to init functions: since they execute prior to the run-time system's initialization, they cannot use run-time system functions that require the startup code to have been executed. For example, an init function cannot use heap allocation or file I/O functions. In addition, an init function cannot permanently change uninitialized data, since the startup code of the run-time system (which executes after the init function) will set all uninitialized data to zero. However, initialized global data can be changed and all such changes will be preserved.

Init functions are useful for initializations required for the run-time system. For example, the installation of a floating point emulator must be done in an init function because the run-time system performs floating point operations in its startup code. Another application could be extending the program's heap using `RTCMOSExtendHeap`.

Here is an example of an init function:

```
#ifdef _MSC_VER
_declspec(dllexport) void MyInitFunction(void)
#else
void __export __cdecl MyInitFunction(void)
#endif
{
    int Pages;

    RTSetFlag(RT_MM_VIRTUAL, 1);    // force virtual memory manager
    Pages = RTCMOSExtendHeap();    // extend the heap
    RTDisplayString("Heap extended by ");
    RTDisplayInt(Pages*4);
    RTDisplayString("k bytes.\r\n");

    RTEmuInit();                  // initialize emulator
    RTDisplayString("Emulator is up and running!\r\n");
}
```

For Microsoft Visual C/C++, the configuration file must contain the line:

```
Init MyInitFunction
```

For Borland and Watcom, use:

```
Init _MyInitFunction
```

Init functions are executed before the C/C++ startup code. The Watcom compiler generates calls to a stack check routine at the entry of each function by default. However, this stack check routine does not work correctly before the run-time system's initialization has run. Thus, all code which can be executed from an Init routine must be compiled with stack checking off (command line option -s).

Link Command

`RTLoc` will usually resolve DLL imports to DLL exports the same way as Win32 program loaders would. However, to achieve greater flexibility, the Link command may be used to change such linkage.

The syntax of the Link command is:

```
Link ModuleName.ImportedName [ModuleName.]ExportedName
```

or

```
Link ModuleName.Ordinal [ModuleName.]ExportedName
```

ModuleName.ImportedName or *ModuleName.Ordinal* specifies that a DLL function referenced by one or more modules of the application is to be replaced by function *[ModuleName.]ExportedName*. If *[ModuleName.]* is not specified, the main program is assumed. Function name matching is case sensitive; module names, however, are case-insensitive.

Parameters *ModuleName*, *ImportedName*, and *ExportedName* may be replaced by the wildcard `''`. In this case, `RTLoc` will match up imports against exports. If several Link commands are used, they are processed in the order given in the configuration file. If *ModuleName* is `''`, it matches any module. If *ImportedName* is `''`, it matches any DLL import. If both *ImportedName* and *ExportedName* are `''`, the two names must be identical for a match. *ExportedName* must not be `''` if *ImportedName* is not `''`.

Parameter *Ordinal* cannot contain wildcard characters; it must be a single numeric value. Please note that RTLoc does not support ordinal exports. If a module contains ordinal imports (which are discouraged by Win32), you must use the Link command to resolve them.

Examples:

```
Link SomeLIB.DLL.SomeFunc MyFunc
// all calls to SomeLIB.DLL.SomeFunc are rerouted to MyFunc

Link SomeLIB.DLL.* OtherDLL.DLL.MyFunc
// all calls to any function of module SomeLIB.DLL rerouted
// to MyFunc in OtherDLL.DLL

Link *.SomeFunc MyFunc
// calls to function SomeFunc of any module are rerouted
// to MyFunc of the main program

Link *.* MyFunc
// all calls to any imported functions are rerouted to MyFunc

Link SomeLIB.* *
// all calls to imported functions from module SomeLIB are
// rerouted to functions with the same name in the program

Link *.* *
// all calls to imported functions are rerouted to functions
// of the program with the same name
```

Assume you want to link functions A and B of DLL SomeDLL to your own functions MyA and MyB, function C of OtherDLL to MyOtherC, and all other functions to a handler for unsupported calls:

```
Link SomeDLL.DLL.A      MyA
Link SomeDLL.DLL.B      MyB
Link OtherDLL.DLL.C     MyOtherC
Link *.*                UnsupportedAbort
```

The Link command requires the import table of the program to be located with a Locate command. The import table is usually named .idata.

If you are unsure about the exact names of imported or exported functions, use RTLoc option -Rd and look them up in the .LOC file's .EXE file report. Naming conventions may vary depending on the compiler, compiler version, and calling conventions.

RTLoc attempts to resolve static DLL references with the following methods in the given order:

- All Link commands are processed as described above.
- Function and module names are matched up exactly as a Win32 program loader would.
- Any references to modules KERNEL32.DLL, USER32.DLL, ADVAPI32.DLL, OLEAUT32.DLL, and RTT32DLL.DLL are resolved against functions in the module containing RTTarget-32's Win32 emulation (library RTT32.LIB or RTT32DLL.DLL).

IgnoreMsg Command

The IgnoreMsg directive in configuration files can be used to suppress the generation of information or warning messages:

```
IgnoreMsg "string"
```

"string" should be replaced with the beginning of a message to be suppressed. All information or warning messages which begin with the given string (not case sensitive) will be ignored. They will not be displayed and are not listed in the .LOC file.

Example: The Debug Monitor shipped with RTTarget-32 does not use a heap or a C/C++ run-time system which might need a heap internally. Thus, RTLoc does not need to allocate any heap space for the Monitor. However, this causes warning message "No heap region specified" to be issued. Since this warning can safely be ignored for the Monitor (but not for programs using a run-time system), the directive:

```
IgnoreMsg "No heap"    // ignore all messages which start with "No heap"
```

can be placed in the Monitor's configuration file. This way, the Monitor can be built successfully without generating any warnings.

This directive must be used with care. Use it only if you are 100% sure that an information or warning message can be ignored. If possible, the cause of the message should be corrected. For example, the warning message

Section *Something* has no size and is not mapped

is usually due to superfluous "Locate NTSection *Something* ..." commands which should be removed.

Defining Boot Code Options

If boot code is included in an application, a number of options for it can be set in the configuration file. Each of these options has a default; therefore, you need include only the options whose defaults are not appropriate for a target.

BOOTFLAGS Command

The BootFlags command specifies boot code options:

BOOTFLAGS = Value

Value can be an "ored" combination of the following values:

BF_CPL_0	Current privilege level 0. The application will run in ring 0 instead of 3. At CPL 0, the program can execute all privileged instructions (such as HLT, LGDTR, etc.). However, memory access to read only and system pages is not protected at run time. Please note that the Halt instruction can only be executed at CPL 0 if the target hardware actually supports Halt. For example, the Halt instruction is not supported by some 386EX boards, because these boards do not detect the Halt bus cycle and do not generate the required Ready signal to acknowledge the cycle. Please refer to functions RTHalt and RTHaltCPL3 for further information.
BF_NO_FPU	No 387 compatible floating point unit. The boot code should not attempt to detect an FPU. This is required on many 386EX and AMD Élan targets because any attempt to communicate with a non-existent FPU can hang the CPU. This flag can also be used to test the 387 FPU emulator on a target which has an FPU.
BF_NO_A20	No A20 enabling. The boot code BIOSBOOT.EXE should not attempt to enable A20. Use this flag if the boot code's algorithms to enable A20 are incompatible with the target. If this flag is specified and A20 is disabled, and access to memory above 1M is required at run-time, use OUT and InitCode commands to enable A20 instead. By default, BIOSBOOT.EXE tries BIOS function int15h/ax=2401h, the PS/2 method via port 92h, and the keyboard controller command D1h to enable A20. Boot codes BOOT.EXE and PMBOOT.EXE never attempt to enable A20 and ignore this flag.
BF_NO_KEYBRD	No PC compatible keyboard. When this flag is specified, the boot code will not install and enable a handler for IRQ 1. BIOSBOOT.EXE will not attempt to use the keyboard controller to enable A20. In addition, this flag is also checked by the keyboard driver of RTTarget-32.
BF_NO_PCTIMER	No PC compatible 8253 timer chip. When this flag is specified, the boot code and RTTarget-32's Win32 function GetTickCount will not install and enable a handler for IRQ 0. Programs not using RTKernel-32 must override function GetTickCount of library RTT32.LIB to use any time functions. RTKernel-32 programs need custom clock and high resolution timer drivers.

BF_NO_SLAVE_PIC	No PC compatible slave 8259 programmable interrupt controller at port address A0h. When this flag is specified, the boot code will not initialize the slave PIC found on PC compatible systems and it will not enable IRQ 2 on the master PIC.
BF_NO_MASTER_PIC	No PC compatible master 8259 programmable interrupt controller at port address 20h. When this flag is specified, the boot code will not initialize the master PIC found on PC compatible systems. Note that functions RTEnableIRQ, RTDisableIRQ, and RTIRQEnd must be replaced in library RTT32.LIB and the Debug Monitor must be recompiled if no 8259a master PIC is available. RTKernel-32 programs must be linked with a suitable replacement interrupt driver.
BF_NO_VESA_LFB	This flag can be used to allow the BIOS boot code to accept VESA graphics modes without a linear frame buffer. The RTPEG-32 and MetaWINDOW drivers can support such modes if the required video RAM address space is smaller than or equal to 64k. The boot code applies this flag automatically to VESA mode 102h (800x600x16 = 60000 byte addresses in 4 bit planes), but not to others.

Example:

```
BOOTFLAGS = BF_NO_FPU|BF_NO_KEYBRD    // no blanks around "|" operator
```

VideoRAM Command

The boot code supports API calls to display characters and strings. Application programs can inquire the location of the video RAM from the boot code for their own screen I/O. For example, function printf or object cout will use these functions. To accomplish this, RTTarget-32's boot code must know where the video RAM (if any) is located. The syntax for the VideoRAM command is:

```
VideoRAM = RegionName | None
```

where parameter *RegionName* must have been defined in a previous Region command. The video RAM is assumed to be located in the given region. If *None* is specified, the boot code will assume that no display adapter is installed and program output will be sent to the serial port specified with the *COMPort* command.

If an application is booted with boot code BIOSBOOT.EXE and the configuration file does not contain a VideoRAM = None command, the boot code will enquire the location of the video RAM from the BIOS at startup, overriding whatever value was specified in the configuration file. This allows the same program configuration to run on targets with color or monochrome displays. However, this feature only works if **both** video RAM areas at B0000h and B8000h have ReadWrite access permission.

The default value for the VideoRAM is *None*.

GMode Command

If the BIOS boot code BIOSBOOT.EXE is used, the boot code can place the video hardware into a different graphics mode during the boot process using BIOS interrupt 10h. Calling int 10h at run-time is not possible, because the BIOS can only operate in real mode or virtual 8086 mode, both of which are not supported by RTTarget-32.

The GMode directive has the following syntax:

```
GMode Mode [Mode...]
```

Up to 15 BIOS video modes may be specified. At boot time, the boot code will attempt to set each graphics mode specified until one is found which is accepted by the video hardware. At run time, the graphics mode actually set can be enquired with function RTGMode().

The mode parameters must be in the range 0..7Fh for standard BIOS modes or in the range 100h..17Fh for VESA BIOS modes. If the boot code successfully sets a VESA mode, it will query the VESA BIOS for the VbeInfoBlock and the ModeInfoBlock and store them at physical address 00000C00h and

00000E00h, respectively. Thus, it is important not to use the first page of physical memory for other purposes. RTLoc will issue a warning if a GMode parameter is 100h or larger (a VESA mode) and address range 3k - 4k has been allocated to something other than a Nothing section.

The hardware configuration file Graphpc.cfg (used by all RTPEG-32 demos) defines most commonly supported VESA modes. Graphpc.cfg attempts to set graphics modes in the order 105h 103h 101h 102h 12h (256 color modes with resolutions 1024x768, 800x600, 640x480, and then 16 color modes with resolutions 800x600 and 640x480).

Only VESA modes with a linear frame buffer are supported, which requires a VESA BIOS version 2.0 or higher to be present on the graphics card (see boot flag BF_NO_VESA_LFB in Chapter 3, *BOOTFLAGS Command* to change this behavior). Program VESATEST.COM can be used to check which modes are supported. VESATEST can be executed under MS-DOS or a DOS Box of Windows 3.1/95/98/Millennium to display all available VESA graphics modes and a list of MetaWINDOW and RTPEG-32 drivers which will support the respective modes. Use this program to determine which parameters the RTLoc directive GMode will support on a specific target.

If a non-text mode is set, directive *VideoRAM None* should be included in the same configuration file, because RTTarget-32's char out handler cannot display characters in graphics mode. Instead, all text output is then sent to the Host.

COMPort Command

The COMPort command can be used to supply configuration information for a serial port to be used for cross debugging or as an output device to display fatal errors if no display is available:

```
COMPort = Port [,Baudrate [,IRQ [,IOBase]]]
```

or

```
COMPort = None // No serial port is available
```

Parameter *Port* can be any of the strings COM1, COM2, COM3, or COM4. It is used to initialize the default values for parameters *IRQ* and *IOBase*. Optional parameter *Baudrate* specifies the speed of host <-> target communication (default: 115200). Parameters *IRQ* and *IOBase* specify the Interrupt Request and port I/O address used by the UART. These parameters need only be specified if non-standard values are used.

Parameter *Port* is ignored if all optional parameters are specified. The default is:

```
COMPort = COM1
```

RTTarget-32 assumes that the UART uses a clock input of 1.8432MHz (the value normally used on PCs). If a different input clock frequency is used by the target hardware, the baud rate specified in the COMPort command must be adjusted according to the following formula:

$$B = (1,843,200 / \text{Clock Frequency}) * \text{Baudrate}$$

where *B* is the value to be supplied in the COMPort command and *Baudrate* is the effective baud rate to be used.

Example: The target computer has a 386EX CPU running at 25Mhz, and you want to use COM2 (the second internal serial port of the 386EX) at 57600 baud. The input frequency for the serial ports is CLK2 (50Mhz in this example) divided by 4, giving 12.5 MHz. The value required in the COMPort command is:

$$(1.8432 / 12.5) * 57600 = 8493$$

Creating Output Files

RTLoc will usually create an RTTarget-32 binary file with extension .RTB (RTTarget-32 Binary). These files are used by program BootDisk, the RTTarget-32 debugger, and the download utility RTRun. Alternatively, RTLoc can also produce HEX files with its *HexFile* command or Binary Images using the BINFile command. Such files can be processed by most EPROM programming devices or EPROM emulators.

Output Command

RTLoc's default output file format is .RTB. The Output command can specify a path and location for the .RTB and .LOC file:

```
Output Name
```

Parameter *Name* may optionally contain a path, but it should not specify a file name extension. If this command is not used, the name used is the Application command line parameter of RTLoc.

HexFile Command

The HexFile command instructs RTLoc to produce one or more Intel HEX files containing the program image:

```
HexFile Name Address Size [,Split [,RecLen]]
```

Parameter *Name* is the base name of the hex file to be produced. *Address* specifies the location in the target's address space where the HEX file should start. This will usually be the start address of an EPROM to be programmed with the HEX file. Parameter *Size* specifies the number of bytes to be covered in the target's address space. Optional parameter *Split* may be set to 1, 2, or 4 and defines how many EPROMs are used in parallel. For example, if you use two 8-bit EPROMs to implement a 16-bit memory system, *Split* must be set to 2. For 32-bit systems built with 8-bit EPROMs, 4 must be specified. The default value is 1 (no splitting). Parameter *RecLen* can be used to change RTLoc's default HEX file record length of 16. Any value which is a power of two in the range 4 to 64 may be used.

Any number of HexFile commands can be used in a single configuration file. Each command should describe a portion of the address space implemented by EPROMs. The number of files actually produced per HexFile command depends on the *Split* value. If *Split* is equal to 1, a single file named *Name*.HEX is produced. For other *Split* values, one file is produced per EPROM. They will have the names *Name*.HE0, *Name*.HE1, etc. The last digit in the file name specifies the byte offset of the data contained in each file.

RTLoc does not fill unused portions of an EPROM with a specific pattern. Unused parts of the EPROM's address space are not written to the HEX file(s).

Example 1:

The target system uses a single 64k EPROM at address F0000h to implement 8-bit wide memory:

```
HexFile MyEPROM F0000h 64k
```

RTLoc will produce a single HEX file named MyEPROM.HEX. It will contain any data mapped into the given address range.

Example 2:

The target system uses 16-bit memory implemented with 8-bit EPROMs. One EPROM pair is located at address F0000h to cover 64k (using 32k EPROMs), and a second pair is located at 100000h and implements 512k with 256k EPROMs:

```
HexFile EPROM1 F0000h 64k 2
HexFile EPROM2 100000h 512k 2
```

RTLoc will produce four HEX files named EPROM1.HE0, EPROM1.HE1, EPROM2.HE0, and EPROM2.HE1. EPROM1.HE0 will contain the byte data at address F0000h, F0002h, F0004h, etc. EPROM1.HE1 will contain the byte data at F0001h, F0003h, F0005h, etc.

BinFile Command

The BinFile command instructs RTLoc to produce one or more binary images of the target's address space:

```
BinFile Name Address Size [,Split [,FillByte]]
```

This command is very similar to command HexFile. Please refer to the previous section for a detailed parameter description.

Parameter *FillByte* specifies the value to use for target memory without any data. It defaults to CCh (x86 opcode to trigger a breakpoint software interrupt 3).

Depending on the Split value, the output file name is *Name.BIN* or *Name.BI0*, *Name.BI1*, or *Name.BI0*, *Name.BI1*, *Name.BI2*, and *Name.BI3*.

RTLoc option -t can be used to truncate bin files to the actual required size.

Initializing Target Hardware

OUT and InitCode commands described below can be placed in an RTLoc configuration file to be executed by the standard boot codes at power-up or reset. OUT and InitCode commands can be used to initialize the target's chipset, such as RAM/ROM and all required peripherals.

Most target initializations should only need to use OUTB, OUTW and Delay. Please refer to files Ex386ini.cfg, Sc400ini.cfg, Sc520ini.cfg, and Ns486ini.cfg of demos ExLED, HelloSc400, HelloSc520, and NSHello for complete examples using these commands to initialize a controller. Configuration file Bin\Bootdbg.cfg contains a few macros for debugging a chipset initialization.

OUT Commands

Out commands can be used to send 8, 16, or 32 bit values to I/O ports at boot time:

```
OUTB Port Value
OUTW Port Value
OUTD Port Value
```

OUTB sends the byte *Value* to port address *Port*. OUTW sends the word *Value* to port address *Port*. OUTD sends the dword *Value* to port address *Port*.

Delay Command

The delay command pauses further processing of OUT or InitCode commands by the boot code:

```
Delay Value
```

Parameter *Value* specifies the value loaded into register CX before the assembler code sequence

```
L1: LOOP L1
```

is executed. *Value* must be in the range 0..FFFFh. On an Intel 386EX at 25MHz, each loop iteration requires 11 clock cycles, which is approximately 0.44 microseconds. On a Pentium 133MHz, we have measured one iteration to take about 0.038 microseconds.

The use of the Delay command may be required for chipset initializations to become effective (e.g., RAM refresh, etc.).

InitCode Commands

For advanced initializations, the following commands can be used to have the boot code execute additional code on the target:

```
InitCode
InitCode  Filename
InitCodeB Value [,Value...]
InitCodeW Value [,Value...]
InitCodeD Value [,Value...]
InitCodes "String" [,"String"...]
```

Each of these commands emit code to be executed by the boot code. Code sequences are grouped in *InitCode sections*. The order of OUT... and InitCode sections is observed. All adjacent InitCode commands contribute to the same InitCode section. If the command InitCode (without parameters) or another command except InitCode is specified, the current InitCode section is closed. The *InitCode Filename* command always creates a unique InitCode section which is not concatenated with other sections. When a section is closed, RTLoc will append a `jmp [e]bx` instruction to the section, since register [e]bx contains the return address.

InitCode Filename appends the content of the MS-DOS .EXE file *Filename* to the *InitCode* data. The program must not contain fixups and its entry point must be the start of the program. *InitCodeB/W/D* emit bytes, word, or dwords, respectively. *InitCodeS* writes strings to the *InitCode* data and interprets embedded escape sequences following the C/C++ rules for string constants (e.g. `\n`, `\r`, `\x123`, etc). No zero byte is appended to such strings.

The offset with the *InitCode* data to receive the next emitted code can be changed with command:

```
InitCodeOrg Value
```

Pseudo symbol `$` can be used to determine the current offset. Example:

```
Macro LONG_DELAY COUNT
    InitCodeB 66h 8Bh C1h      ; mov eax, ecx      ; preserve ecx
    InitCodeB 66h B9h          ; mov ecx, <dword>
    InitCodeD COUNT
    #defineN LDL $
    InitCodeB 67h E2h LDL-$-2   ; loop $          ; use ecx
    InitCodeB 66h 8Bh C8h       ; mov ecx, eax     ; restore ecx
    #undef LDL
EndM
```

BOOT.EXE and BIOSBOOT.EXE invoke *InitCode* sections in 16-bit real mode with the following register context. PMBOOT.EXE invokes *InitCode* sections in 32-bit protected mode:

Register	Contains	Preserve
ss:[e]sp	boot code stack	must be preserved
ds	ss	must be preserved
ebp	reserved	must be preserved
cs:[e]si	address of current <i>InitCode</i> section	must be preserved
cs:[e]di	start of <i>InitCode</i> table	may be clobbered
[e]bx	near return address	may be clobbered
eax	undefined	may be clobbered
edx	undefined	may be clobbered
ecx	preserved	may be clobbered
es	preserved	may be clobbered

Registers *es* and *ecx* are not modified by the boot code while it processes *InitCode* commands. They can be used to pass information from one *InitCode* section to another.

Note that RAM might not be usable as long as the chipset initialization has not been completed. In this case, not even the stack can be used. The boot code itself will not use the stack or any other RAM until all *OUT...* and *InitCode* commands have been processed.

The LOC File

Apart from the .RTB and/or .HEX and .BIN files, RTLoc generates a LOC file with extension .LOC. The LOC file has a purpose similar to that of a MAP file produced by a linker. It contains detailed information about how the program is mapped to the target hardware.

The LOC file is divided into several reports. Each report can be enabled or disabled with the -Rx options (see section *RTLoc Options* in this chapter).

The *Configuration Report* contains a copy of all configuration files lines processed with blank, comment, and inactive lines removed.

The *EXE File Report* contains information about the application's PE file(s). In particular, parts of the PE file header and a list of all sections found is included. If the Dynamic Link Report is enabled, this report also lists all exported and imported functions. RTLoc produces a separate EXE File Report for the main program and each DLL of the application.

The *Fixup Table Report* lists all fixups processed by RTLoc. For each fixup, its location in the PE file's address space and in the target image's address space, the original value, and the new value are given. This report is disabled by default, because it can become rather long. It is usually required for trouble-shooting only.

The *Dynamic Link Report* shows how RTLoc has matched DLL imports to DLL exports. For each DLL import, the importing module, the import's module and function name, the matched exporting module and function name, and the address of the exported function are given. In addition, this report lists all exported functions which are never referenced. However, these unreferenced functions could still be required at run-time if they are called locally or are accessed through functions LoadLibrary/GetProcAddress. This report is disabled by default. However, whenever any Link commands are changed or added, it is strongly recommended to enable the Dynamic Link Report at least temporarily to verify that the linkage is correct.

The *Compression Report* lists all program entities which have been compressed. This includes the entities' names, full size, compressed size, ratio of compressed to full size in percent, and the time required to compress and decompress the data. The times are given in milliseconds. The times for decompressing the data are measured on the host when RTLoc verifies that the compressed data indeed decompresses to its original. The time required on the target will depend on the processing speed of the target. The numbers in the Compression Report are supplied to allow estimates of how long the target will need to initialize.

The *Relocation Report* is probably the most interesting report. It contains a list of all regions and program entities with their addresses, sizes, images' sizes, and effective access privilege levels. In this report, you can see how your program has been mapped onto the target. For regions, the *Image* column contains the size of the region which is actually being used by non-discardable entities. Discardable entities are marked with an asterisk (*) character in front of their respective names.

The *Page Table Detailed Report* lists all pages of memory with their respective linear and physical addresses and access rights. Since this report can be rather long, it is disabled by default.

The *Page Table Summary Report* shows the number of pages with identical properties (e.g., pages with the same access rights, etc.).

If boot code has been included, the *Boot Code Configuration Report* lists all configuration options and parameters available for the boot code.

The *Application Image File Report* includes information about the binary file produced by RTLoc. The application header is shown followed by a list of program entities and modules with their respective attributes.

If RTLoc produces any error, warning, or information messages, they are listed in order of their occurrence in the *Message Report*. Please be sure to read this section carefully and understand all warnings (if any). A list of all possible messages is given in Appendix C. If any errors or fatal errors are encountered, no .RTB or .HEX file is created.

The Locate Process in Detail

This section contains some information about how RTLoc relocates the application. This information helps to better understand the sequence used to map program items and why the sequence of Locate commands can be important.

RTLoc processes all command line parameters from left to right. Each configuration file is processed before the rest of the command line. If no configuration file is specified, the default configuration file (AppName.CFG) is processed after the command line.

Next, RTLoc will attempt to process map files for all modules with segment numbers given in their *Locate Section* or *Locate NTSection* commands. The respective sizes of such sections are calculated in this step.

The .EXE file of the main program and all DLLs are read. The sizes of all sections that don't have a size yet are calculated now. The image of each section is also saved.

If *Locate PageTable* was specified, the page table is created and - if no size was given - its size is determined from the *Region* commands. Since RAM remapping and the creation of virtual regions take place later, the size found may be too small.

If boot code is included, the boot code .EXE file is read and the size of the boot code and data is determined.

If a *Reserve* command was given, the reserved application image file is read and its application header retrieved. RTLoc will go through all program entities of the reserved application and look for overlaps between it and regions defined for the current locate process. If overlaps are found, the first available address of the region is set to the first byte following the overlap. This algorithm is used because RTLoc does not support fragmented regions. Each region is filled from low to high addresses. RTLoc maintains a high water mark for each region to which the next entity can be mapped. Thus, if the reserved application and the current application have been built using different *Region* commands, memory can be wasted.

In addition to processing the program entities of the reserved application, RTLoc will also analyze the imported page table (if any) and merge it with the current page table.

The application header is initialized and its size is calculated.

If any entities are mapped to virtual regions, the sizes of the pseudo entities to hold the physical data of these entities are determined.

The next step involves mapping all entities except copies and entities without a fixed size (e.g., stack and heap are not located yet if no size was specified for them). All entities are allocated to their respective regions in the sequence the entities were created. Discardable entities in RAM are allocated top-down (as opposed to bottom-up for all other entities).

All program entities that can contain fixups are now mapped and fixups are applied. The method depends on whether *Locate Section* or *Locate NTSection* is used. With *Locate NTSection*, all fixups are simply incremented by the difference between the image base address of the PE file and the image base assigned by RTLoc. For *Locate Section*, RTLoc determines the location and target of each fixup location and adds the difference of the PE-file's target location and RTTarget-32's target location to the fixup. All DLL references are also resolved in this step.

If any *Locate Copy* commands are present, the copies are created (with compression, if appropriate) and mapped now. Again, their sequence is observed in the mapping process.

If stack and heap have not been mapped and do not map to the FillRAM region, they are mapped now.

If a *FillRAM* command was specified, RTLoc will now go through all regions and look for pages of RAM which are not used and have *Assign* access. Such pages are remapped and appended to the region given in the *FillRAM* command. Please note that this process may require the Page Table to grow, which can lead to an error at this stage.

If the stack and heap are still unmapped, they are processed now. Since this is performed after RAM remapping, they can benefit from the enlarged region that has received all unused pages.

Chapter 4

Running a Program on the Target

Chapter 3 described how a program is prepared to run on the target. This chapter describes how it actually gets there.

Before the program can be invoked, the computer's CPU and vital hardware components must be initialized. Then, the program data must be initialized (e.g., copied entities must be copied or decompressed, uninitialized data must be set to 0, etc.). These steps are performed by the RTTarget-32 boot code. Four forms of booting are supported: booting from disk, from a BIOS extension, from MS-DOS, and booting through the CPU's reset vector. In addition, RTTarget-32's Remote Debug Monitor can be used to download programs via a serial link.

Booting from Disk

The BIOS boot code BIOSBOOT.EXE supports booting from hard disk, floppy disk, or any other kind of disk device supported by the target computer's BIOS. To configure a program to boot from disk, simply include a *Locate BootCode BIOSBOOT.EXE*, a *Locate BootData*, and a *Locate DiskBuffer* in the application's configuration file. All program entities should be located in RAM regions.

Booting from disk is the simplest boot method. It does not require any ROM or EPROM and applications do not have to be ROMable. Updating and changing applications is easily accomplished by simply overwriting the diskette.

Program BootDisk

Command line utility BootDisk must be used to transfer an application to a bootable floppy or hard disk. The target disk must have been formatted as a standard DOS disk (FAT-12, FAT-16, or FAT-32). The application to be placed on the disk must contain the BIOS boot code disk and a sufficiently large disk buffer. To invoke BootDisk, use the command line:

```
BootDisk Application Drive[:] [BIOS_ID [Loader]]
```

or

```
BootDisk /remove Drive[:]
```

Parameter *Application* is the name of the program to write to the target drive. If no path information is given, the .RTB file is searched in the default directory and then in the directory BOOTDISK resides in.

Parameter *Drive* is the logical (not physical!) drive to receive the boot image. BOOTDISK does not check whether the target drive is bootable. For example, you cannot boot from a logical drive in an extended partition; it must be a primary partition.

Optional parameter *BIOS_ID* specifies the BIOS ID byte the bootstrap loader should use during booting to load itself. If not specified, BOOTDISK will assume value 0 (for diskette drive A:) for parameter *Drive* 'A' and 'B'. For all other drive letters, *BIOS_ID* 128 (80h) for hard disk drive C is assumed. If your BIOS boots from a drive other than 'A' or 'C', you must specify this parameter.

Optional parameter *Loader* is the file name of an MS-DOS .EXE file which will load the application from disk at boot time. When this parameter is not specified, the disk loader built into BootDisk is used. The source code of the default disk loader is contained in file BootDiskload.asm.

If parameter */remove* is specified instead of an application's name, BOOTDISK will delete all .RTA files from the target drive and restore the original boot sector which was saved by BootDisk in file BOOT-SECT.RTT the first time it wrote an RTTarget-32 application to the disk.

Program BootDisk will write the boot sector, a disk loader program, and the application to a file named *Application.RTA* (RTTarget-32 Application) in the target drive's root directory. In addition, BootDisk will check that the file written is located in a single chain of clusters. This is required because the disk loader code is unable to load fragmented files. If the check fails, delete all files on the target drive and try again (sometimes, simply trying again without deleting any files will also work).

Disks prepared by BootDisk are standard DOS disks and you can store other files on them. However, you should **never** delete an .RTA file and you should **never** use any method other than BootDisk to write an .RTA file. If you wish to use the diskette for other purposes, restore the original boot sector with

```
BootDisk /remove Drive[:]
```

or reformat it using FORMAT, or run SYS on it to write a new MS-DOS boot sector to the disk.

RTTarget-32 in conjunction with RTFiles-32 also supports writing boot images to a disk under program control using function RTMakeBootDisk described in Chapter 7.

Booting from a BIOS Extension

If the target has a BIOS but no mass storage device such as a diskette, hard disk, or EPROM disk, RTTarget-32 can be booted from a BIOS extension. After the BIOS has completed its initialization, it scans the address range 0C8000h to 0DFFFFh in 2k increments for an expansion ROM signature. Such a signature is produced by the *Locate BIOSVector* command in the configuration file. If the BIOS finds an expansion ROM, it transfers control to it.

A BIOS extension signature consists of the following byte sequence: 55h, AAh, and the number of 512 byte blocks in the BIOS extension. At offset 3, the actual BIOS extension code starts. The BIOS reads the signature and then calculates the checksum over the complete BIOS extension. Control is passed to the entrypoint only if the checksum is zero. The *Locate BIOSVector* ensures that the BIOS extension has the required format and provides the correct checksum.

To boot from a BIOS extension, BIOSVector, BootCode, and BootData must be specified in Locate commands. The BIOSVector must be located at an address searched by the BIOS. All program entities having an image must be located in ROM.

The boot code BIOSBOOT.EXE delivered with RTTarget-32 is suitable for booting by this method for targets with PC compatible hardware.

Booting from the CPU Reset Vector

This process is similar to booting from a BIOS extension. However, no BIOS is required. RTTarget-32 controls the complete boot process which starts at 16 bytes before the end of the physical address space after power-on or reset.

To boot using the reset vector, ResetVector, BootCode (either BOOT.EXE or PMBOOT.EXE), and BootData must be specified in Locate commands. The ResetVector must be located at the end of physical address space minus 16. All program entities having an image must be located in ROM. The boot code must take care of all system initializations, since no BIOS code is executed. In particular, the motherboard chip set and - if dynamic RAMs are used - the RAM refresh must be initialized before any RAM can be accessed. This is achieved using *OUT..* and *InitCode* commands.

Downloading

To download a program from the host to the target, the program Monitor included with RTTarget-32 must be installed on the target computer. The monitor can be generated from file Bin\Monitor.exe supplied with RTTarget-32 and suitable configuration files (sample configuration files for the Monitor are supplied in the Demo.. directories). Subsequently, the Monitor can be booted using one of the methods described above (e.g., booted from diskette).

An application to be downloaded or debugged using the cross debugger or the download utility should not contain any boot code (the Monitor has already booted the target) and it **must** contain the command

```
Reserve Monitor
```

in its configuration file.

Program RTRun

Once the Monitor is installed on the target, programs can be downloaded using program RTRun. RTRun is invoked using the following command line:

```
RTRun [-d-] [-q+] [Application]
```

Parameter *Application* should have no file name extension. RTRun will look for the file *Application.RTB* in the default directory and send it to the target.

Option *-d* (detach) controls whether RTRun should wait for any program output sent from the target to the host. The default *-d+* causes RTRun to detach immediately and terminate after the program has been downloaded. With *-d-*, RTRun will wait and any program output from the target is displayed on the host and recorded in file *RTTARGET.LOG*. In this case, RTRun will terminate either when the target program terminates or the user presses Ctrl-C.

Please note that the RTTarget-32 run-time system sends program output to the host only if the boot code option *VideoRAM None* has been specified.

Option *-q* (quiet) can be used to suppress the display of download progress information.

Communication parameters to use by the host (e.g., serial I/O port, baud rate) are retrieved from file *RTTARGET.INI*. See Chapter 5, section *File RTTARGET.INI* for details.

RTRun can also be started without any command line parameters. In this case, it will merely wait for and display/log any raw data coming from the target. You can use RTRun in this way to check whether a displayless target is booting properly. For example, if the target has no display or runs in graphics mode and is expected to boot the Debug Monitor, RTRun previously started without command line parameters would display the following output:

```
RTTarget-32 3.0 32-Bit Boot Code (c) 1996,2000 On Time Informatik GmbH
RTTarget-32 Debug Monitor 3.0 (c) 1996,2000 On Time Informatik GmbH
Monitor Header at: 00046A54, Current CPL: 3
Port: IOBase: 03F8, IRQ: 4, Baudrate: 115200
```

To terminate RTRun, press Cntrl-C. If RTRun is unable to communicate with the Debug Monitor, you must press Cntrl-C three times to terminate. If the Monitor was booted successfully, a program can then be downloaded using RTRun, RTD32, or Visual Studio 6.0.

Booting from MS-DOS

Program RTTBOOT allows booting an RTTarget-32 program from MS-DOS. RTTBOOT.COM is a small MS-DOS loader program which loads an RTB file given on its command line and transfers control to it. The file is read via DOS and all program entities are copied to their target addresses. Subsequently, a far jump to the application's boot code is performed.

RTTBOOT cannot load and boot every RTTarget-32 program. In particular, the following restrictions apply:

- The computer must be running in real mode (i.e., not in virtual 8086 mode).
- No program entity with an associated image can overlap any address range currently occupied by DOS or RTTBOOT.COM itself.

To satisfy the first requirement, no memory manager such as EMM386, QEMM, etc. may be loaded (EMM memory managers run the computer in virtual 8086 mode).

The second requirement will usually also not allow using an XMS memory manager such as HIMEM.SYS, since HIMEM will allocate all available extended memory. The only exception are RTTarget-32 programs which do not load any images into extended memory.

Apart from a suitably configured DOS system, an application to be loaded via RTTBOOT must also be configured correctly:

- BIOSBOOT.EXE must be used as boot code.
- BIOSBOOT.EXE and the boot data must be allocated within the first MB of address space.
- Memory allocated by DOS and RTTBOOT must not be used for program images.

Since RTTBOOT transfers control to the application, a boot vector is not required. If one is used, RTLoc will issue a warning that the vector is not located at a suitable address. If no boot vector is used, RTLoc will warn about a missing boot vector. Both warnings can safely be ignored. Locating a DiskBuffer can suppress the warning (and keep the program compatible with booting from disk).

DOS will typically occupy the first 64k of real-mode address space. RTTBOOT is a COM file and thus will occupy exactly another 64k. Therefore, on a minimal DOS system, it is sufficient to reserve the first 128k for DOS (note: these 128k can still be used for RAM remapping, as in example HELLODOS, or for entities not requiring an image). However, to be on the safe side, it is recommended to reserve at least 256k for DOS. Example:

```
Region  DOSMem      0  256k RAM
Region  LowMem      256k 256k RAM
Region  MoreLowMem  512k 128k RAM
Region  Mono        B0000h 4k Device ReadWrite
Region  Color       B8000h 4k Device ReadWrite
Region  HighMem     1M   1M RAM

FillRAM  HighMem

Locate   BootCode   BIOSBOOT.EXE LowMem
Locate   BootData   BootData      LowMem

Locate   PageTable  PageTable     HighMem
Locate   Header     Header        HighMem
Locate   NTSection  CODE          HighMem
Locate   NTSection  DATA         HighMem
Locate   Stack      Stack          HighMem 16k
Locate   Heap       Heap          HighMem

VideoRAM Color
```

This example uses the scarce conventional memory only for entities which must reside in real mode address space: boot code and boot data.

RTTarget-32 takes complete control of the hardware and thus cannot return to DOS. When the RTTarget-32 application terminates, the PC is rebooted.

Demo program HELLODOS shows how to use RTTBOOT.

Program RTTBOOT

RTTBOOT loads and starts an RTTarget-32 program from DOS:

```
RTTBOOT [Options] RTBFileName
```

The following options are available:

- A- Disable A20 control. With this option, RTTBOOT will not attempt to enable A20, and it will not check whether A20 is enabled. Use this option only on systems which do not have any extended memory or which always have A20 enabled. A20 control requires BIOS int 15h support as well as a keyboard controller. If you need to use this option, RTLoc command *BOOTFLAGS=BF_NO_A20* will also be required.
- F- Disable floppy motor control. This option instructs RTTBOOT not to attempt to turn off the diskette motor(s). Floppy motor control requires a BIOS with floppy disk support.
- X- Disable XMS memory overlap check. This option prevents RTTBOOT from checking whether any extended memory area needed by the loaded application is currently in use. The XMS memory overlap check requires BIOS int 15h support.
- M- Disable DOS memory overlap check. This option prevents RTTBOOT from checking whether any conventional memory area needed by the loaded application is currently in use. The DOS memory overlap check requires BIOS int 12h support.
- R- Disable real-mode execution check. This option will prevent RTTBOOT from executing instruction SMSW to detect the current CPU mode.

Parameter *RTBFileName* must be the file name - with file name extension - of the .RTB file to load.

Example:

```
RTTBOOT hello.rtb
```


Chapter 5

Cross Debugger RTD32

RTTarget-32 supports source-level cross debugging with its debugger RTD32 and with Microsoft Visual Studio. Cross debugging means that the debugger runs on the host and controls the program running on the target. This chapter describes configuring the host - target communication, the Debug Monitor, and the cross debugger RTD32. Information on using Visual Studio for cross-debugging is available in Chapter 6.

File RTTARGET.INI

The cross debugger and download utility RTRun must know how to communicate with the target computer. This information is read from file RTTARGET.INI located in the default directory or RTTarget-32's BIN directory. In addition, options for the debugger can be specified in this configuration file.

RTTARGET.INI is an ASCII text file divided into sections. Each section starts with a section name in square brackets, followed by keywords and their respective values.

Section [COM] supports the following keywords:

Keyword	Values	Description
Port	COM1 : COMx	Serial port used by the host to communicate with the target. The default value is COM1.
Baudrate	any numeric value	Speed of host <-> target communication. The value specified must match the value given in Monitor's configuration file. If both host and target have buffered UARTs, it should not be necessary to use a baud rate lower than 115200. If the host <-> target communication is unreliable, try using a lower value. The default value is 115200.
Start-Timeout	1000 ..	Timeout value to apply for program initialization in milliseconds. After downloading, RTRun or RTD32 have to wait until the program has initialized. Especially with compressed entities, the required time may vary significantly depending on program size, target processing speed, etc. If you get a timeout error after program download, increase this value. The default is 10000 (10 seconds).
OverlappedIO	0 or 1	Specifies whether the host <-> target communication link can be used for simultaneous send and receive. If this option is set to 0, targets with low processing power may be able to run at higher baud rates. Download performance is reduced slightly. This option defaults to 1 (enabled).
DBGBaud	any numeric value	Alternate host baud rate to use after initialization of the debugger. With this value, the communication speed can be changed without rebuilding the Monitor. The default value is <i>Baudrate</i> .
MonBaseBaud	any numeric value	Initial baud rate used by the Monitor. Only required if <i>DBGBaud</i> is different from <i>Baudrate</i> and the target uses a non-standard UART frequency. The default value is <i>Baudrate</i> .
MonBaud	any numeric value	Alternate baud rate to be used by the Monitor after initialization. Only required if <i>DBGBaud</i> is different from <i>Baudrate</i> and the target uses a non-standard UART frequency. The default value is <i>DBGBaud</i> .

Section [Options] supports the following keywords:

Keyword	Values	Description
SwapVectors	0 or 1	Controls whether the Debug Monitor should always install its own interrupt vectors when it gains control and restore the application's vectors when it resumes. The default value 1 (enable) ensures that no part (not even hardware interrupts) of the target program runs when the debugger has suspended it. Setting this value to 0 will allow application hardware interrupts to continue even when the program has encountered a breakpoint.
RTKPreemptOff	0 or 1	This option is only relevant for RTKernel-32 programs and defaults to 0. If set to 1, the Debug Monitor will execute RTKPreemptionsOFF() each time a breakpoint is encountered and restore preemptions when it resumes. This option is not required when SwapVectors is set to 1, because RTKernel-32 will not get any interrupts for preemptive task switches. However, if SwapVectors is set to 0, this option may be required to prevent other tasks from running while the debugger has suspended the program.
SendBreak	0 or 1	If set to 0 (default is 1), the debugger will never send a BREAK signal to the target. This may be required on some target computers with UARTs not 100% 16450 compatible. By default, the debugger sends a BREAK before each program download and when the running target needs to be interrupted.
TargetLog	Filename	If the target computer has no screen (or <i>VideoRAM = None</i> has been specified for some other reason), all program output is sent to the host and is displayed on the host's screen. In addition, it is written to the file given in this option. If the name is blank, no target log is maintained. The default value is RTTarget.log.
KillVirtual-Console	0 or 1	If set to 0 (default is 1), the debugger will close the Target Virtual Screen on the host immediately when the target program terminates. By default, the Target Virtual Screen remains on the desktop until a new debug session is started or the debugger is closed.
Rows	0..127	Defines the number of screen window rows to be used by RTD32. The default value is 0, which instructs RTD32 to select a suitable default.
Columns	0..127	Defines the number of screen window columns to be used by RTD32. The default value is 0, which instructs RTD32 to select a suitable default.
RestoreScreen	-1, 0, 1, 2	Defines how RTD32 should restore the window when it terminates. Value 0 causes the screen window not to be restored. 1 will restore the window, and value 2 instructs RTD32 to run in a separate console to be closed when the program terminates. -1 (the default) instructs RTD32 to pick a suitable default depending on the host operating system (1 for NT/2000, 2 for Win9x).

Example RTTARGET.INI file:

```
[COM]
Port=COM3
Baudrate=57600
OverlappedIO=1
StartTimeout=5000

[Options]
SwapVectors=0
RTKPreemptOff=1
```

Prerequisites for Cross Debugging

To use the debugger, the following conditions must be met:

- The RTTarget-32 Debug Monitor must be installed on the target.
- The program to be tested must be compiled with full debug information. Please consult your compiler documentation for details. It is recommended to disable compiler optimization for debugging. Otherwise, the compiler's elimination of common subexpressions, multiple register allocation to variables, etc., can make debugging difficult.
- The program must be located with the same target hardware definition (Region commands) as the Monitor. Usually, this is ensured by using the same configuration file containing all Region commands for the Monitor and the test program.
- The configuration file of the test program must contain the command:

```
Reserve Monitor
```

This prevents RTLoc from allocating the same memory to both the Monitor and your test program.

- The program must be located using command *Locate NTSection* rather than *Locate Section*. This restriction only applies to EXEs and DLLs for which source level debugging is required. CPU level debugging also works with *Locate Section*.
- The configuration file of the test program must not include boot code. The Monitor already has boot code.
- Both the .EXE/.DLL file(s) and the .RTB file of the application to be tested must reside in the default directory. DLL files may also optionally be on the current path.
- The debugger's configuration file RTTARGET.INI must contain the host's COM port configuration.
- The application to be tested must not use the same serial port as the Monitor.

The Debug Monitor

The Monitor will initialize the COM port on the target according to the parameters specified in the *COMPort...* command in its configuration file. In addition, it understands the following case sensitive command line options:

Halt	While the Monitor is idle (e.g., waiting for commands to be sent by the host debugger), CPU instruction HLT shall be executed. Executing Hlt can reduce power consumption and heat generation on the target. If the Monitor is running on a virtual target of VMware, it will require significantly less CPU time of the host OS.
NoFIFO	Instructs the Monitor not to use the FIFO of 16550 compatible UARTs. Use this option if the target's UART contains too many bugs for the Monitor to handle.

Instructions on how the Monitor is best located are given in the Monitor.cfg file included with each On Time RTOS-32 demo.

Differences from Borland's TD32

The RTTarget-32 debugger RTD32 behaves just like Borland's TD32 for Win32 would for local debugging, with the following exceptions:

- Pressing Ctrl-C while the program under test is running will suspend execution immediately. This can be useful to get out of endless loops. Ctrl-C only works if interrupts are not disabled on the target. If the target no longer responds, pressing Ctrl-C more than three times will abort the debug session.
- A *non maskable interrupt* (NMI) is treated like a breakpoint, interrupting program execution. This feature can be useful to regain control when the program has entered a dead loop with interrupts disabled. Of course, it must be possible to trigger an NMI manually on the target to use this feature.
- Resetting the debugger will restore all interrupt vectors to their original values.

- Program output of displayless targets is displayed in a separate window on the host and written to file RTTARGET.LOG. RTTARGET.LOG is overwritten each time RTD32 is started. The log file name can be specified in RTTARGET.INI.
- Menu options *View | Global Descriptors* and *View | Interrupt Descriptors* have been added.
- In the CPU window, local menu option *I/O* for reading from or writing to I/O ports has been added.

A Quick Example

To get a general idea of using the debugger, please try to reproduce the following little debug session using demo program SerInt.

First, the program must be compiled. Change into one of the DEMO... directories and type:

```
MAKE SERINT
```

or whatever the make utility of your compiler is called. Then, the Debug Monitor must be installed on a boot diskette for the target PC:

```
BOOTDISK MONITOR A:
```

With this diskette, the target PC can be rebooted. Make sure the file RTTARGET.INI contains correct port parameters for your host <-> target communication. Then you can start the debugger:

```
RTD32 SERINT
```

The debugger will download the program and execute it until it has reached function main(). A module source window is opened automatically and the cursor is positioned at the current execution position.

Use the mouse or *Alt-W, S* to make the source window a bit smaller. Then, open the Interrupt Descriptor View with *Alt-V, I* and position it below the source window. In the Interrupt Descriptor View, press *Alt-F10* or the right mouse button to see the local menu for this window. Select *Goto...* and enter number 67 (or 0x43) to see the descriptor of vector 43h, which is the vector which will be modified by the program. Switch back to the source window by clicking on it or by pressing *Alt-1*. Move the cursor to source line SetupSerial(...) and hit *F4* (run to cursor). The debugger will execute the program until the requested source line has been reached. You will notice that the interrupt 67 vector in the IDT display has changed, it now displays *_ASMHandler* next to it. *Alt-F10, V* in the IDT windows will position the cursor in the source window to the source code of that interrupt handler (*_ASMHandler* in *SERISR.ASM* in this case). Press *Ctrl-O* (or *Alt-F10, O*) to get back to the current execution point. Press *F7* to step into the function to be called (SetupSerial). Pressing *F8* (step over) twice will take you to the first *RTOut()* statement. Position the cursor on variable *Divider* (click the left mouse button anywhere on the identifier or use the cursor keys) and press *Ctrl-I* to open a data inspector for that variable. Alternatively, use *Ctrl-F4* which would also allow modifying the value.

To continue the program, press *F9*. Because no breakpoints are set, the debugger will not regain control automatically, but you can interrupt it by pressing *Ctrl-C*. Since it is unlikely that the *Ctrl-C* interrupt has hit the program exactly on a C or Pascal source line, the debugger will open a CPU window and position the cursor on the interrupted instruction. Go back to the source window and position the cursor on the first statement of the program loop and press *F4* to run to that statement.

Suppose we want to restart the debug session: Just press *Ctrl-F2* or select *Run | Program reset* from the menu.

To exit the debugger, press *Alt-X* or select menu option *File | Quit*.

Debugger Reference

This section describes how to use the RTTarget-32 debugger RTD32.

RTD32 Command Line

The debugger is started with command line:

```
RTD32 [options] [Program]
```

Parameter *Program* is the name of the program to be debugged. It should be located in the current directory. If no file name extension is specified, .EXE is assumed. If no program is given on the command line, you can load one using the *File | Open* command.

The following options are available:

- cfile Use configuration file *file*.
- h or -? Display short help screen.
- l Assembler startup. For C/C++ program, this option causes the debugger not to run to function main(). Instead, the debugger stops the program at the run-time system entry-point.

Please note that it is not easily possible to debug an RTTarget-32 Init routine or DllEntry-Point functions of statically referenced DLLs since they are executed even before the run-time system startup code. To debug such code, you must explicitly call Win32 function DebugBreak in such a function.
- sddir Search for source files in *dir*.
- tdir Use *dir* as the initial default directory (as opposed to the directory the program is located in).

Navigating in RTD32

RTD32 can be controlled with a mouse or the keyboard. The menu options in the menu bar at the top of the screen contain pull-down menus, all of which can be selected by left mouse clicks or the highlight letter with the Alt-key pressed.

The status line at the bottom of the screen lists frequently used function keys, which are shortcuts for menu commands. Holding down the Alt or Ctrl key will display what Alt- or Ctrl-keys are available, depending on the current context. It is recommended to frequently press Alt or Ctrl to see the available shortcuts.

Data is presented by RTD32 in windows described later in this chapter. Some of the windows are further divided into panes. Each window or pane has a local menu which contains commands specific to this pane. The local menu is activated with the right mouse button or *Alt-F10*. You should have a look at the local menu in each pane to get familiar with all of the debugger's available features.

RTD32 has a context sensitive, hyperlink help system. You can always press *F1* to get help about the window, menu option or whatever has the focus. Menu option *Help* can be used to access a help index.

Expressions

Frequently, the debugger will prompt you for a value. This can be a source line, an address, or any other kind of value.

The syntax with which values are supplied can depend on the programming language used. RTD32 defaults to the language of the current source file, but you can also force a particular language under *Options | Language*.

Under some circumstances, you must override the scope of a variable. Consider debugging inside a function which contains a variable *X*, but there is also a global variable *X*, which you want to inspect. Just typing '*X*' into the *Data | Evaluate* dialog box will access the value of the local *X*. However, with a suitable scope override, the global *X* can also be accessed. Example (for C/C++):

```
#MyModule#X
```

Or for Pascal:

```
MyModule.X
```

The full syntax for scope overrides is:

```
[#module[#filename]]#linenumber[#variablename]
```

or

```
[#module[#filename]][#functionname]#variablename
```

The optional components *module*, *filename*, *linenumber* and *functionname* specify the lexical scope for interpreting the expression. The default for each value is the program's current execution location.

Numeric values follow the language conventions of the currently selected language. The default is always decimal. Hex can be specified with a *0x* prefix for C/C++, a *\$* prefix for Pascal, and an *h* postfix for assembler.

In some cases, address ranges are required. They can be specified either as

`address, size`

or

`address address`

The second case specifies the range's start and end addresses. Of course, each address can be a numerical value or a symbol of the program.

More information about expressions is available in RTD32's online help.

Menu Commands

This section briefly introduces the main menu commands available in RTD32. To get a feeling for them, it is recommended to try each one in a real debug session.

Menu commands can be activated with the left mouse button or the keyboard (using either *F10* or *Alt* with the highlighted menu option key pressed).

File

The *Open* option allows loading a program (either because no program was specified on the command line or because you want to debug a different program). Option *Quit* exits the debugger.

Edit

The Edit menu contains commands to copy and paste highlighted items to the RTD32 clipboard or the log.

View

The View menu is used to open different kinds of windows such as source modules, CPU, watches, etc. All available windows are described later in this chapter.

Please note that most windows can have only one instance open at any given time. To open another instance of a Module, Dump, or File window, please use the *View / Another* option.

Run

The Run menu allows executing the program in a variety of ways as described below:

Command	Shortcut	Description
Run	F9	Run until breakpoint, program termination, or Ctrl-C.
Go to cursor	F4	Run to cursor location in the current source or CPU window.
Trace into	F7	Execute one line, follow function calls.
Step over	F8	Execute one line, skip function calls.
Execute to	Alt-F9	Prompt for an address to run to (e.g., a function name).
Until return	Alt-F8	Execute the program until the current function or procedure returns to its caller.
Animate		Auto-repeating <i>Trace Into</i> command. Instructions or source lines are executed continuously until a key is pressed. RTD32's display changes to reflect the current program state between each trace, which allows you to watch the flow of control in your program. You are prompted for the rate at which to step.
Back trace	Alt-F4	Undo the last instruction or source statement. The processor and memory state are restored to the way they were before the instruction or source line was traced over. The history pane of the Execution history window contains a list of saved steps and allows you to control when stepping and tracing information is saved. The <i>Full history</i> option in the Execution history windows must be selected for this command to work.
Instruction Trace	Alt-F7	Executes a single machine instruction. After using this command, you are usually taken to the CPU window.
Program reset	Ctrl-F2	Reload (if required) and reinitialize the program.

Breakpoint

The Breakpoint menu can be used to set, delete, or change various breakpoint types. Please note that you can also set breakpoints using *F2* in the source module or CPU window.

Command	Shortcut	Description
Toggle	F2	Sets or clears a breakpoint at the current cursor position in the source module or CPU window.
At	Alt-F2	Sets a breakpoint at a specific location. A dialog box is displayed which allows you to specify the location, group ID, and other options of the breakpoint. The group ID allows grouping several breakpoints together. In the local menu of the Breakpoint window, you can change attributes of complete breakpoint groups.
Change memory global		Stops the program when a memory location changes value. This command sets a hardware breakpoint for write accesses. The program is stopped when the value changes. Only memory ranges of one, two, or four bytes are supported.
Expression true global		Sets a breakpoint that occurs when the value of an expression becomes true. You are prompted for the expression that must become true for the program to stop. Program execution is significantly slowed down by such breakpoints.
Hardware breakpoint		Sets a breakpoint using the 386 debug registers. The great advantage of such breakpoints is that they do not incur any run-time overhead. The program can run at full speed and stops when the breakpoint condition is met. Up to four such breakpoints can be set on memory Write, Access, or Fetch instruction accesses. Memory location sizes of one, two, or four byte sizes are supported.
Delete all		Removes all currently defined breakpoints.

Data

The Data menu allows inspecting and changing the program's variables.

Command	Shortcut	Description
Inspect	Ctrl-I	The Inspect command opens an Inspector to show the value of a variable or a memory-referencing expression. You are prompted for the variable or memory-referencing expression to inspect. If the cursor was in a source module window when you issued this command, the prompt is initialized to contain the variable under the cursor, if any. If you have marked an expression using the <i>Ins</i> key, the prompt is initialized to the marked expression.
Evaluate/ modify	Ctrl-F4	The Evaluate/Modify command evaluates an arbitrary expression. Enter an expression followed by an optional format control string that is separated from the expression with a comma (.). The value appears in the middle pane. You can move to this pane to scroll long value displays or error messages. If the result is changeable, you can change the expression's value by moving to the bottom pane using the <i>Tab</i> key, typing a new value, and pressing Enter.
Add watch	Ctrl-W	The Data Add Watch command places an expression or variable on the watch list displayed by the Watches window. If the cursor was in a source module window when you issued this command, the prompt is initialized to contain the variable under the cursor, if any. If you have marked an expression using the <i>Ins</i> key, the prompt is initialized to the marked expression.
Function return		This command lets you examine the value that is about to be returned by the current function. You can only issue this command just before the function returns.
Heap tracking		The Data Heap tracking command lets you collect information about the memory that your program allocates and frees on the heap. This is useful for finding locations in your program where you allocate memory but fail to free it at a later time. The online help contains more information about this feature. Heap tracking is only supported for C/C++ programs.

Options

This menu allows customizing RTD32 to your needs.

Command	Shortcut	Description
Language		Defines the programming language to use for the syntax of expressions.
Macro		Lets you set and clear keystroke macros assigned to different keys.
Display Options		Lets you customize the Debugger's display screen.
Path for source		This option lets you set where the Debugger looks for the source files that make up your program. Source files are searched for first where the compiler found them, then in each directory specified by this command or the -sd command line option, then in the current directory, and finally in the directory that contains the program you are debugging.
Set restart options		The Set restart options command lets you control how the Debugger will use the restart information saved from the last time you were debugging a program. A separate restart file is maintained for each program you have been debugging.
Save options		Saves the current options to a configuration file.
Restore options		Restores options from a configuration file.

Window

The Window menu allows managing the desktop.

Command	Shortcut	Description
Zoom	F5	Enlarges the current window to its maximum size, or, if it is already maximized, restores its default size.
Next	F6	Moves the focus to the next window. Each window has a number displayed in the upper right-hand corner. Pressing <i>Alt-Number</i> or clicking on the window moves the focus to the respective window directly.
Next pane	Tab Shift-Tab	If the current window contains several panes, the focus moves to the next pane. Shift-Tab moves one pane backwards.
Size/move	Ctrl-F5	Change the size and/or position of the current window. Use the cursor keys to move the window and Shift plus the cursor keys to resize it. Dragging the lower right-hand corner of a window with the mouse also resizes it. Dragging the title bar moves the window.
Iconize/restore		Makes the current window as small as possible and parks it at the bottom of the screen. This command acts as a toggle, so that if a window is already iconized, this command restores it to its previous size and position on the screen. This command is useful when you want to free up screen space and memory, but don't want to lose the context of an open window.
Close	Alt-F3	Closes the current window.
Undo close	Alt-F6	Restores the last window closed. Only the last closed window can be restored.
User screen	Alt-F5	Display program output. This command only works under DOS/Windows 3.1 and is thus obsolete.

Help

Use the help menu to access the online help. You can also press F1 at any time to get context sensitive help.

Debugger Windows

Most of the debugger's windows are opened using the View menu. The following sections briefly describe the information available in each window, along with information about local menu commands.

Source Module

The *View / Module* command allows selecting a module from all currently loaded EXEs and DLLs. If available, the source code of the module is displayed. Source lines with associated code (which is required to set breakpoints) are marked with a bullet in the window's left-most column. The local menu (accessed with *Alt-F10* or the right mouse button) allows maneuvering in the module. The online help explains all local menu options in detail.

Inspect

The Inspect window is opened through *Data / Inspect*, or by pressing *Ctrl-I* in the source module window with the cursor positioned on the variable to inspect. The inspect window displays the variables content and its type. For structured types such as arrays, structures, or classes, you can select a member with the highlighted bar followed by a carriage return and inspect the member. With this technique, complex data structures can be walked through.

Watch

The watch window can contain a list of variables to be displayed continuously. Press *Ctrl-W* in the source module window to add the variable under the cursor to the watch window. Local menu options allow editing, deleting, and inspecting watched variables.

Breakpoints

The breakpoint window lists all currently defined breakpoints in the left pane with their respective properties in the right pane. Local menu options allow adding, deleting, and changing breakpoints.

Stack

The stack windows display all currently active stack frames which have been built up by function calls. The stack display only works for functions which set up a standard EBP-based stack frame. Most compilers can be instructed to produce such stack frames with compiler options. In many cases, disabling all optimization will also produce standard stack frames.

Selecting a particular stack will display the corresponding source code (or CPU instructions if no source code is available).

Local menu options allow inspecting the local variables of active functions.

Log

The log window displays a log of actions the debugger has taken. You can also write information to the log using menu command *Edit* or the log window's local options. A local menu command allows saving the log to a disk file for later inspection. In addition, logging can be enabled or disabled.

Variables

The variables window shows a list of symbols and their values for the current module and function, as well as a list of your program's global symbols and their values. The pane in the lower part of the window contains all local variables of the current function (if any). Use the *TAB* key to change between the two panes. Local menu options allow inspecting, watching, etc. these symbols.

File

Use the file window to inspect the contents of an arbitrary file. The data can be displayed in ASCII or hex format. The local menu allows selecting the format, searching, and positioning.

CPU

The CPU window has five panes displaying data on the CPU (as opposed to source) level.

Code Pane

This pane displays assembler instructions interleaved with source lines (if available). This pane's local menu allows setting display options, positioning the cursor (current EIP value, caller, previous position, searching, etc.), setting a new EIP value, assembling (modifying) instructions, and inspecting/modifying the CPU's I/O ports.

Register Pane

All user mode accessible CPU registers are displayed here and can be modified using local option commands. Highlighted registers have changed during the last program execution.

Flags Pane

This pane displays the state of user mode accessible CPU flags. Highlighted flags have changed during the last program execution. Local options allow changing flag values.

Data Pane

The data pane displays an arbitrary memory region on the target. Question marks (????) indicate that the respective linear address is not accessible. Local menu commands allow setting display options (such as BYTE, WORD, floating point, etc.), searching for data, editing data, and setting a block of data to a constant or file supplied values.

Stack Pane

The stack pane contains the topmost double words on the stack. The current ESP value is indicated with a small triangular mark in the pane's left-hand column (press *Ctrl-O* to position the cursor on that line). Local menu commands allow positioning and changing values.

Register

The register window is identical to the register and flag pane of the CPU window.

Numeric Processor

View | Numeric Processor displays the state of the coprocessor or emulator (if installed). The three panes contain the register stack, control, and status flag. Each pane has a local menu allowing manipulation of all values.

Dump

The data dump window is identical to the data pane of the CPU window (see above).

Execution History

The execution history window shows the last CPU instructions that were executed. Previously executed instructions are only recorded when you use the Run/Trace command (F7). If you are tracing through source code in a source module window, you must make sure that you have Full history set to Yes in the local menu of this window. If the executed instruction corresponds to a source line, you will see the source code, otherwise the disassembled machine code is shown. Local menu commands allow viewing the associated code, running the code backwards and enabling/disabling execution history recording.

Please note that execution history recording can slow down program execution dramatically, because the debugger must execute the program one CPU instruction at a time and save the program state after each step. Enable this option only when needed.

Class Hierarchy

The hierarchy window shows an alphabetical list of all the object types in the program. The local menu allows inspecting and viewing a class hierarchy tree view for each type.

The right-hand pane of this window shows all the object types in the program as a tree, indicating ancestor/descendant relationships between object types.

Global Descriptor Table

This window displays the contents of the global descriptor table (GDT). GDT values cannot be modified except by the user program. Undefined Descriptors are displayed as blank lines. The local menu allows positioning the cursor on a particular selector value.

Interrupt Descriptor Table

This window displays the contents of the interrupt descriptor table (IDT). IDT values cannot be modified except by the user program. Undefined Descriptors are displayed as blank lines. The local menu allows positioning the cursor on a particular selector value or positioning the source module or CPU window on a particular handler.

Clipboard

This window shows all the items that have been put on the clipboard with the *Edit | Copy* command. The local menu has commands to inspect, remove, delete, or freeze values in the clipboard.

Keyboard Shortcuts

The following table lists the most frequently used (but by no means all) keyboard shortcuts with their respective actions for the source module, CPU, and dump windows. *Alt-Letter* will always select the main menu with the respective highlighted letter.

Shortcut keys are always displayed in the status line at the bottom of the debugger's window. Ctrl- and Alt-keys are displayed when the respective key is pressed.

Key	Action	Key	Action
F1	Help	Alt-F1	Last help topic
F2	Toggle breakpoint	Alt-F2	Breakpoint at
F3	Open source module	Alt-F3	Close window
F4	Run to cursor	Alt-F4	Back trace
F5	Zoom window	Alt-F5	User screen
F6	Next window	Alt-F6	Undo close
F7	Step into	Alt-F7	Instruction
F8	Step over	Alt-F8	Run to return
F9	Run	Alt-F9	Run to
F10	Menu	Alt-F10	Local menu
Ctrl-B	Block operations	Ctrl-M	Module select
Ctrl-C	Caller	Ctrl-N	Next
Ctrl-D	Display options	Ctrl-O	Origin
Ctrl-F	Follow	Ctrl-P	Previous
Ctrl-G	Goto	Ctrl-S	Search
Ctrl-I	Inspect	Ctrl-V	View source
Ctrl-L	Line number select	Ctrl-W	Watch
Ins	Mark text		

Chapter 6

Using Microsoft Visual Studio

Microsoft Visual Studio 6.0 or higher can be used to develop On Time RTOS-32 programs. All steps required during the development phase can be performed within the IDE. This chapter describes how to set up Visual Studio projects and how to use the Visual Studio integrated debugger to debug On Time RTOS-32 programs running on the target.

Program DBGShell

Program DBGShell intercepts Visual Studio's debugger to use RTTarget-32's host <-> target communication module for debugging. You must use program DBGShell to start Visual Studio to be able to perform cross debugging. The command line syntax of DBGShell is:

```
DBGShell [IDE] [ProjectFile]
```

Parameter IDE is the full path name of Visual Studio (the path to Msdev.exe). Parameter ProjectFile should be a workspace file to load. Both parameters are optional, but at least one of them must be specified. If the IDE parameter is missing, DBGShell will attempt to locate Visual Studio by looking for an "open" or "open with Msdev" association for the given project file. If only the IDE parameter is specified, but no project or workspace file, the IDE is started without loading a project. Examples:

```
dbgshell "C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin\msdev.exe"  
Hello.dsw
```

or

```
dbgshell hello.dsw
```

Ideally, On Time RTOS-32 projects should be launched through a Start Menu or Desktop shortcut which invokes DBGShell. The On Time RTOS-32 installation has created such shortcuts for the demo projects.

Another advantage of DBGShell is that it defines environment variable RTTARGET to point to the On Time RTOS-32 installation directory and that it places the On Time RTOS-32 Bin directory on the PATH before Visual Studio is started.

Setting up a Project

On Time RTOS-32 programs are essentially Win32 console mode programs. Thus, setting up an On Time RTOS-32 project in Visual Studio is best performed by creating a new Win32 console mode project and then modifying it to fit On Time RTOS-32's needs. This section describes step-by-step how to set up such a project. Additional in-depth information about the Microsoft compiler and linker and general compile/link guidelines are available in Appendix A.

An On Time RTOS-32 project must have the following properties:

- The project type is Win32 Console Application.
- The On Time RTOS-32 include path must be known to the compiler.
- On Time RTOS-32 libraries must be linked.
- The linker must be instructed to produce a relocation table (option */fixed:no*).
- After an .EXE file has been created, RTTarget-32's locator must be run. We will define a second project called "Target" for this purpose.

In addition, it would be desirable to write boot diskettes directly from the IDE. We will define a custom build steps for RTB files to accomplish this.

The sample project will have two configurations: Win32 Debug and Win32 Release. The Debug configuration assumes that it will run under the debugger's control with the RTTarget-32 Debug Monitor installed on the target. The Release configuration will produce a self-booting version of the application. The project will also build a suitable Debug Monitor.

The project setup described below assumes that environment variable RTTARGET is defined to contain On Time RTOS-32's installation directory. If Visual Studio is started using DBGShell, DBGShell will define RTTARGET correctly.

Of course, the easiest way to set up such a project is to copy one of On Time RTOS-32's examples. Nevertheless, here are step-by-step instructions to create the Hello project shipped with RTTarget-32.

- Create a new Workspace (named Hello in this example).
- Add a new, empty Win32 Console Application project to the Workspace. Place it into the same directory as the Workspace. We will also call this project "Hello".
- Add a new Utility project to the Workspace. Again, place it into the same directory as the Workspace. We will call this project "Target".
- For projects Hello and Target, verify that the Intermediate and Output files for the Win32 Debug and Win32 Release Configuration are "Release" and "Debug", respectively.
- Make Project Target depend on Project Hello. This will ensure that the application's .EXE file is built before the locator is run.
- Add all required source and header files to the "Source File" and "Header Files" folders of project Hello.
- Create a new file folder called "Config Files" in project Target and add all RTTarget-32 configuration files to this folder. In this example, we need the files Hello.cfg, Demopc.cfg, and Monitor.cfg.
- In the project settings for project Hello, All Configuration, tab C/C++, Category Preprocessor, Additional include directories, enter \$(RTTarget)\Include.
- In the project settings for project Hello, All Configuration, tab Link, Category Input:
 - Object/library modules: delete all libraries and enter rttheap.lib¹² rtt32.lib¹³
 - Ignore Libraries: enter kernel32.lib
 - Force symbol references: enter _malloc,_EnterCriticalSection@4,_RTFileSystemList¹⁴
 - Additional library path: enter \$(RTTarget)\Libmsvc
- If the project is a multithreaded RTKernel-32 application, select the Multithreaded run-time library on tab C/C++, Category Code Generation, for both configurations. Do not use the DLL version of the run-time system.
- In project settings for project Hello, both Win32 Release and Win32 Debug Configurations, tab Link, Project Options: manually add option /fixed:no.
- In project Target, settings for file Monitor.cfg, Win32 Debug configuration, tab Custom Build, enter:
 - Description: Locate Monitor
 - Commands: "\$(RTTarget)\Bin\RTLloc" -DBOOT \$(OutDir)\Monitor Demopc.cfg Monitor.cfg
 - Outputs: \$(OutDir)\Monitor.rtb
 - Dependencies: Demopc.cfg
- In settings for file Hello.cfg Win32 Debug configuration, Custom Build, enter:
 - Description: Locate Hello
 - Commands: "\$(RTTarget)\Bin\RTLloc" -g- \$(OutDir)\Hello Demopc.cfg Hello.cfg

¹² Linking RTTHeap is optional but recommended. More information about RTTarget-32's alternate heap manager RTTHeap is available in Chapter 7, section *RTTarget-32's Memory Managers*.

¹³ If you need more On Time RTOS-32 libraries, insert them here. Please observe the rules for the order of On Time RTOS-32 libraries explained in Appendix A.

¹⁴ These settings are required to work around a Microsoft linker bug. More information is available in Appendix A, section *Microsoft Visual C++*. Do not force a reference to these symbols if they reside in a different .DLL or .EXE file.

Dependencies: \$(OutDir)\Hello.exe, Demopc.cfg

- In settings for file Hello.cfg, Win32 Release configuration, Custom Build, enter:

Description: Locate Hello

Commands:

"\$(RTTarget)\Bin\RTLoc" -g- -DBOOT \$(OutDir)\Hello Demopc.cfg Hello.cfg

Dependencies: \$(OutDir)\Hello.exe, Demopc.cfg

- Create folder "Boot Diskettes" in project Target and Add files Debug\Monitor.rtb and Release\Hello.rtb. Ignore any warnings such as "the file does not exist" and choose to add the files to the project anyway.

Settings for files Debug\Monitor.rtb and Release\Hello.rtb, All Configurations:

Description: Creating Boot Diskette for \$(InputPath)

Commands:

"\$(RTTarget)\Bin\BootDisk" \$(InputPath) A:
if not errorlevel 1 echo New boot disk > \$(OutDir)\BootDisk.txt

Outputs: \$(OutDir)\BootDisk.txt

The active configuration should be set to "Target - Win32 Debug" or "Target - Win32 Release". To produce a new boot diskette either for the Monitor or the application itself, simply right-click the corresponding .RTB file in project Boot Diskette and click Compile.

Here are a few recommendations to streamline the project:

- Remove all subdirectories under the workspace directory other than Debug and Release. They have been created by Visual Studio before the correct paths had been defined.
- Delete the "Resource Files" folder of the Hello project.
- In the Hello project, Linker, General, disable incremental linking. This will reduce program size by up to 64k and thus reduce download times. However, this will also require to change Debug Info in the Win32 Debug Configuration, C/C++, General to "Program Database".
- In the Hello project, Win32 Debug Configuration, C/C++, Code Generation, Use Run-time library, select a non-Debug Library. This will avoid linking the Microsoft debug libraries, which are very large. However, they can be useful if you encounter bugs related to run-time system functions.
- If you plan to use the Microsoft debug libraries, remove RTTHEAP.LIB from the Win32 Debug Configuration.
- Instruct the linker to generate MAP files and add the MAP files to the project. A MAP file is frequently useful to analyse whether the correct libraries have been linked. A MAP file can be enabled in Link category General.
- Add separate folders for the Debug and Release configurations to contain all .MAP and .LOC files. These files should be inspected frequently to verify that linking and locating is performed as intended.

Cross Debugging

When Visual Studio has been started using DBGShell (see the corresponding section earlier in this chapter), its integrated debugger is automatically configured to act as a cross debugger for On Time RTOS-32. Chapter 5 has described the prerequisites that must be met for cross debugging. This section only addresses issues specific to Visual Studio.

To start debugging, make sure the active project configuration is Win32 Debug and that the project is up-to-date. Menu command Build, Start Debugger, can then be used to start a debug session. When Visual Studio asks for the executable file to debug, enter Debug\<project name>.exe. The program will

be downloaded to the target and all of the Visual Studio Debugger's features are available. Please note that the project should **not** be configured for remote debugging within Visual Studio. Redirecting the debugger is performed by DBGShell and not by Visual Studio.

The Visual Studio Debugger used with On Time RTOS-32 is not thread-aware. When the target is suspended at a breakpoint, the debugger's context is set to the current thread. If you need to inspect local variables of another thread, set a breakpoint in that thread and run to that breakpoint. Global data is always accessible in the debugger, regardless of the current thread.

Chapter 7

RTTarget-32 Library

RTTarget-32 applications must contain library RTT32.LIB. It contains functions to communicate with RTTarget-32's boot code, emulation of about 200 Win32 API functions, and functions for interrupt-driven serial communication. RTT32.LIB may be linked directly into the main program's .EXE file or as a DLL. The DLL RTT32DLL.DLL supplied in RTTarget-32's BIN directory is such a DLL; it contains RTT32.LIB completely. For more information about multi-module applications, please refer to Chapter 9, *Using DLLs through RTLoc*.

RTTarget-32 Flags

RTTarget-32 maintains a global 32-bit flags variable initialized to zero by default. The application can simply declare its own instance of RTTarget32Flags to override RTTarget-32's default.

Example:

```
DWORD RTTarget32Flags = RT_MM_VIRTUAL | RT_CLOSE_FIND_HANDLES;
```

A module containing such a declaration can be linked into the EXE or DLL containing the RTTarget-32 library RTT32.LIB.

The following flags are defined:

RT_KEY_BY_INTERRUPT	Force RTTarget-32's keyboard driver to translate scan codes to virtual key codes within the keyboard interrupt handler.
RT_MOUSE_BY_INTERRUPT	Force RTTarget-32's text mode mouse driver to translate raw mouse data to Win32 mouse events and to update the mouse cursor on the screen within the mouse interrupt handler.
RT_MM_FIXED	Force use of RTTarget-32's fixed memory manager.
RT_MM_VIRTUAL	Force use of RTTarget-32's virtual memory manager with uncommitted memory support.
RT_CRT_NO_ACCESS	Instructs RTTarget-32 not to attempt accessing the CRT controller to position the cursor.
RT_CLOSE_FIND_HANDLES	Win32 API function FindNextFile should automatically close find handles when no more files are found. This feature can be useful to avoid running out of handles in programs that do not close handles created and returned by FindFirstFile. For example, programs using Borland's run-time system functions findfirst/findnext are affected by this bug.
RT_DBG_OUT_TO_HOST	This flag controls how strings passed to the Win32 API function OutputDebugString should be displayed. If set, all strings are sent to the host if the program is running under the Debug Monitor and ignored otherwise. By default, such strings are displayed locally if the Debug Monitor is not present.
RT_DBG_OUT_NONE	With this flag set, all calls to OutputDebugString are ignored.
RT_KEYS_US	Use US keyboard layout (default).
RT_KEYS_GERMAN	Use German keyboard layout.
RT_KEYS_FRENCH	Use French keyboard layout.
RT_NO_LANG_HOTKEYS	Do not intercept Left Ctrl-Alt-F1 and Left Ctrl-Alt-F2 to switch keyboard language.
RT_NUMLOCK_OFF	Set the initial NUMLOCK state to off. Other keyboard options can be set with function RTSetKeyboard.

RT_NO_KEYB_LEDS Instructs the keyboard driver never to send data to the keyboard controller to control keyboard LEDs or to set the keyboard repeat rate.

RT_HEAP_MIN_BLOCK_SIZE_32 Sets the minimum block alignment of memory allocated through
RT_HEAP_MIN_BLOCK_SIZE_64 Win32 Heap functions. The default value is 16 bytes. Larger
RT_HEAP_MIN_BLOCK_SIZE_128 values can improve speed at the expense of memory efficiency.

Flags **RT_KEY_BY_INTERRUPT** and **RT_MOUSE_BY_INTERRUPT** improve response times for keyboard and mouse events at the cost of higher interrupt latencies. More information is available in section *Console Input Event Management* of this chapter. Additional information about RTTarget-32's memory managers is available in section *RTTarget-32's Memory Managers* later in this chapter.

RTTarget-32's Native API

The functions described in this section can be used by applications to display data on the screen, install interrupt handlers, perform port I/O, reboot the computer, etc. Prototypes for all of these functions are given in include files **RTTARGET.H** and **RTTARGET.PAS** which must be included in every source file using any of these functions.

Function RTSetFlags

This function can be used to change RTTarget-32's run-time options:

```
DWORD RTSetFlags(DWORD Flags, int SetReset)
```

Parameter **Flags** must be a combination of the values given in the previous section. Parameter **SetReset** specifies the value for the the respective flag(s) (1 or 0).

RTSetFlags must be used to change any of the RTTarget-32 flags from an .EXE or .DLL which has not linked **RTT32.LIB**.

The return value contains all flags set after the call. To merely inquire the current flags, Call **RTSetFlags(0, 0)**.

By default, RTTarget-32 will choose the memory manager to be used when the first memory allocation function is called. If the application uses paging and the application consists of more than one module (.EXE/.DLL), the virtual memory manager is used. Once the memory manager has been chosen, it cannot be changed. Since the run-time systems frequently call memory allocation functions before the main program is called, **RTSetFlags** must be called in an *Init* routine to be effective, or you should define your own instance of global variable **RTTarget32Flags**. *Init* routines are described in Chapter 3, *Init Command*.

Function RTSetDisplayHandler

RTSetDisplayHandler installs one of the predefined display handlers to handle characters to be displayed by TTY type functions (e.g., **WriteFile** for **stdout/stderr**, **printf**, **writeln**, **RTDisplayString**, etc.). Console I/O functions such as **WriteConsoleOutput** are not affected by the display char handler.

```
RTDisplayType RTSetDisplayHandler(RTDisplayType Type);
```

Parameter **Type** can have one of the following values:

RT_DISP_DETECT If a video RAM is present, the **RT_DISP_SCREEN** handler is installed. If command *VideoRAM None* was specified and the program is running under the Debug Monitor's control, the **RT_DISP_HOST** handler is installed. If neither a video RAM nor the Debug Monitor are present, the **RT_DISP_NONE** handler is used.

RT_DISP_NONE Any data to be displayed is discarded.

RT_DISP_SCREEN Characters are displayed by writing them to the video RAM. The output position for the next character is advanced automatically. Characters **CR**, **LF**, **TAB**, and **BS** only move the output position and do not display any text. Character **FF** clears the screen and returns the output position to the upper left-hand corner. The physical cursor is not used.

RT_DISP_HOST The data is passed on to the Debug Monitor which will in turn send it to the host. If the debugger or RTRun is running in non-detached mode, the data is displayed on the host.

The function will not install the requested display handler if the resources required by that handler are not available. Instead, RT_DISP_DETECT is assumed (for example, RT_DISP_HOST is requested and the program does not run under the Debug Monitor, or RT_DISP_SCREEN is used, but VideoRAM = None was specified in the configuration file).

When this function is never called by a program, RT_DISP_DETECT is assumed.

The return value of the function corresponds to the actual handler used. It will never be RT_DISP_DETECT.

If none of the predefined char out handlers are suitable for your program, you can install a custom handler in global variable:

```
extern void (RTTAPI * RTCharOutHandler)(char c);
```

For example, such a handler could be used to redirect all screen output to a serial port.

Function RTDisplayChar

This function calls RTTarget-32's display char handler whenever a single ASCII character is to be displayed:

```
void RTDisplayChar(char c);
```

Parameter c is the character to be displayed. RTDisplayChar calls (*RTCharOutHandler)(c) internally. Please see the previous section on function RTSetDisplayHandler on details about RTTarget-32's available display char handlers.

Function RTDisplayString

This function displays a zero-terminated ASCII character string on the target's screen:

```
void RTDisplayString(const char * s);
```

Parameter s is a pointer to the string to be displayed. Function RTDisplayString calls RTDisplayChar internally.

Function RTDisplayInt

This function displays a decimal integer on the target's screen:

```
void RTDisplayInt(int i);
```

Parameter i is a 32-bit integer to be displayed.

Function RTDisplayHex

This function displays an unsigned integer in hex format on the target's screen:

```
void RTDisplayHex(DWORD h);
```

Parameter h is a 32-bit unsigned integer to be displayed.

Function RTDisplayHexW

This function displays an unsigned short integer in hex format on the target's screen:

```
void RTDisplayHexW(WORD h);
```

Parameter h is a 16-bit unsigned integer to be displayed.

Function RTSaveVector

This function saves an entry of the Interrupt Descriptor Table:

```
void RTSaveVector(BYTE Vector, RTInterruptGate * Gate);
```

Parameter Vector specifies which entry to save. Gate is a pointer to a buffer to which the descriptor is copied. Use this function to save/restore vectors which are modified by the program.

Function **RTRestoreVector**

Function **RTRestoreVector** restores an Interrupt Descriptor previously saved by **RTSaveVector**:

```
void RTRestoreVector(BYTE Vector, const RTInterruptGate * Gate);
```

Function **RTSetIntVector**

RTSetIntVector installs an application interrupt handler in the Interrupt Descriptor Table:

```
typedef void (RTTAPI * RTIntHandler)(void);  
void RTSetIntVector(BYTE Vector, RTIntHandler Handler);
```

Parameter **Vector** specifies the interrupt to be processed by **Handler**. **Handler** is a pointer to a routine that will be invoked when the specified interrupt occurs. Please note that hardware interrupts are mapped to interrupt vectors 40h to 4Fh under **RTTarget-32**.

The address of the specified handler is directly placed in the IDT. The handler must be terminated with an **IRETD** instruction and must save/restore all registers it uses. See Chapter 2, *Descriptors and Descriptor Tables* for an overview of all interrupt vectors. Demo program **SERINT.C** shows how to use **RTSetIntVector** to install a hardware interrupt handler.

Function **RTSetTrapVector**

This function is similar to **RTSetIntVector**:

```
void RTSetTrapVector(BYTE Vector, RTIntHandler Handler);
```

However, the handler is installed as a trap gate and not as an interrupt gate. The handler will be entered without changing the interrupt flag in the flags register.

Function **RTInstallISR**

This function installs a handler for a hardware interrupt:

```
void RTInstallISR(int IRQ,  
                 RTIntHandler HighLevelHandler,  
                 RTIntHandler LowLevelHandler);
```

Parameter **IRQ** should be in the range 0 .. 15 and specifies the Interrupt Request on which the handler should be installed. **HighLevelHandler** should be a C function handler without interrupt stack frame. **LowLevelHandler** should contain a complete interrupt stack frame (save/restore all registers, return with **IRETD**).

RTInstallISR behaves differently depending on the presence of **RTKernel-32**. Without **RTKernel-32**, **RTInstallISR** does:

```
RTSetTrapVector(RTIRQ0Vector+IRQ, LowLevelHandler);
```

With **RTKernel-32**, this function does:

```
RTKSetIRQHandler(IRQ, HighLevelHandler);
```

Function **RTEnableIRQ**

RTEnableIRQ instructs the interrupt controller to pass interrupts for a specific **IRQ** to the CPU:

```
void RTEnableIRQ(int IRQ);
```

Parameter **IRQ** must be in the range 0 .. 15. **RTTarget-32** disables all **IRQs** except 0, 1, and 2 at boot time. Please note that there is no **IRQ 2** on PC compatible systems. Hardware that claims to generate **IRQ 2** actually uses **IRQ 9**.

RTEnableIRQ assumes the target hardware to have IBM-PC compatible interrupt controllers at port addresses 20h and A0h.

Function RTDisableIRQ

RTDisableIRQ instructs the interrupt controller not to pass interrupts for a specific IRQ to the CPU:

```
void RTDisableIRQ(int IRQ);
```

Parameter IRQ must be in the range 0 .. 15. RTTarget-32 disables all IRQs except 0, 1, and 2 at boot time. Please note that there is no IRQ 2. Hardware that claims to generate IRQ 2 actually uses IRQ 9.

RTDisableIRQ assumes the target hardware to have IBM-PC compatible interrupt controllers at port addresses 20h and A0h.

Function RTIRQEnd

RTIRQEnd disables interrupts and notifies the master and slave interrupt controllers that processing of an interrupt has been completed:

```
void RTIRQEnd(int IRQ);
```

Each hardware interrupt handler must perform this call. If it is omitted, no additional interrupts of the same or lower interrupt priorities can be accepted.

RTIRQEnd assumes the target hardware to have IBM-PC compatible interrupt controllers at port addresses 20h and A0h.

Function RTDisableInterrupts

This function executes CPU instruction CLI to disable interrupts.

```
void RTDisableInterrupts(void);
```

Function RTEnableInterrupts

This function executes CPU instruction STI to enable interrupts.

```
void RTEnableInterrupts(void);
```

Function RTSaveAndDisableInterrupts

This function disables interrupts and returns the previous interrupt state. This value can later be passed to function RTRestoreInterrupts to restore the interrupt state (interrupts enabled or disabled).

```
DWORD RTSaveAndDisableInterrupts(void);
```

Function RTRestoreInterrupts

This function restores the interrupt state previously read with RTSaveAndDisableInterrupts.

```
void RTRestoreInterrupts(DWORD IntState);
```

Functions RTIn, RTInW, RTInD

These functions read a byte, word, or dword value from an I/O port:

```
BYTE RTIn (unsigned int addr);  
WORD RTInW(unsigned int addr);  
DWORD RTInD(unsigned int addr);
```

Parameter addr specifies the I/O port to read from.

Functions RTOut, RTOutW, RTOutD

These functions write a byte, word, or dword value to an I/O port:

```
void RTOut (unsigned int addr, BYTE val);  
void RTOutW(unsigned int addr, WORD val);  
void RTOutD(unsigned int addr, DWORD val);
```

Parameter val is written to port addr.

Function RTReboot

RTReboot restarts the target computer:

```
void RTReboot(void);
```

RTReboot merely puts the CPU in shutdown mode through a triple fault. This mode is signalled by a special bus cycle. Most motherboards detect this cycle and generate a hardware reset. Thus, the specific reaction to shutdown mode can depend on your motherboard's design.

Function RTHalt

RTHalt waits for a hardware interrupt:

```
void RTHalt(void);
```

If the program runs at privilege level 3, RTHalt returns immediately and does nothing. However, if running at CPL 0, instruction HLT is executed. HLT stops the CPU until a hardware interrupt or a hardware reset occurs. During the wait, the CPU is in *Halt Mode*. The bus is free to be accessed by other hardware such as DMA. Also, only a fraction of the CPU's normal power consumption is used, significantly reducing heat generation.

All loops waiting for an external event (e.g., keyboard input, a specific time, data to be received, etc.) can call RTHalt to reduce power consumption and heat emission. Alternatively, you can assign function RTHalt to RTTarget-32's idle handler (see section *Function RTWait*).

Function RTHaltCPL3

RTHaltCPL3 also waits for a hardware interrupt. However, unlike function RTHalt(), it can switch to CPL 0 temporarily to execute the CPU HLT instruction:

```
void RTHaltCPL3(void);
```

If the program runs at privilege level 0, HLT is executed. Otherwise, RTHaltCPL3 sets up a custom IDT, switches to ring 0 and executes HLT. When a hardware interrupt occurs, the function switches back to ring 3 and re-raises the interrupt to allow the application's handler to process the interrupt.

RTHaltCPL3 makes the advantages of the HLT instruction available without sacrificing protection. However, RTHaltCPL3 requires a privilege level transition before an interrupt is actually serviced. For example, on a Pentium 100MHz, using function RTHaltCPL3 would increase the interrupt latency by about 1 - 2 microsecond.

RTHaltCPL3 can also be used as an RTTarget-32 idle handler (see the following section).

Function RTWait

RTWait is called by various modules contained in RTT32.LIB whenever the software must wait for some external event to occur. It is intended to allow the installation of idle processing.

```
void RTWait(void);
```

RTWait() calls the current *IdleHandler*, or, if that is NULL, returns immediately. The IdleHandler is declared as follows:

```
extern void (RTTAPI * RTIdleHandler)(void);
```

Its default value is NULL. You may assign your own idle handler or function RTHalt to RTIdleHandler to install your own idle handler.

All loops that wait for an external event can call RTWait to signal that they are currently idle.

Function RTLocateSection

RTLocateSection can search for a program section created by RTLoc:

```
RTAppSection * RTLocateSection(int Index, int Type, const char * Name);
```

Parameter Index specifies the section to select if several sections match the following search criteria. Index is 1-based. Type must be one of the following constants:

RT_ST_ANYSECTION Matches any section type

RT_ST_BOOTCODE	section containing boot code
RT_ST_BOOTDATA	section containing boot data
RT_ST_APPCODE	section containing application code
RT_ST_APPDATA	section containing application data
RT_ST_APPHEAP	section containing the application heap
RT_ST_APPSTACK	section containing the application stack
RT_ST_NOTHING	section created by <i>Locate Nothing</i>
RT_ST_COPY	section containing a copy of another section
RT_ST_APPHEADER	section containing the application header
RT_ST_FILEDATA	section containing file data
RT_ST_PAGETABLE	section containing the page table
RT_ST_BOOTVECTOR	section containing the boot vector
RT_ST_PHYSICAL	shadow section in the physical address space for a section in a virtual region
RT_ST_DECOMPCODE	section containing decompression code
RT_ST_DECOMPDATA	section containing decompression data buffer
RT_ST_COPYCOMPRESSED	section containing a compressed section
RT_ST_DISKBUFFER	section containing a disk buffer
RT_ST_REGION	pseudo section describing a region
RT_ST_RESERVED	pseudo section describing memory used by a reserved application

All sections available at run-time are listed in the Relocation Report and Application Image Report of the LOC file created by RTLoc.

Parameter Name may either be NULL to match any section, or it must point to the name of the desired section as given in the program configuration file. Name matching is not case-sensitive.

The return value is a pointer to a structure defined in RTTARGET.H with the following layout:

```
typedef struct {
    DWORD   Flags;
    DWORD   SectionImageLen;
    DWORD   SectionAllocLen;
    DWORD   SectionAddr;
} RTAppSection;
```

SectionAddr is the location of the section in the virtual address space. SectionAllocLen is its size. If initialized data is associated with the section, its size is given in SectionImageLen. For regions, SectionImageLen is the **physical** size of the region. Thus, this value is always zero for virtual regions and smaller than SectionAllocLen for regions which have been extended with command FillRAM.

If the desired section is not found, RTLocateSection returns NULL.

Example: The following function retrieves a pointer to the page table:

```
RTPageTableEntry * PageTablePtr(void)
{
    RTAppSection * Section;
    Section = RTLocateSection(1, RT_ST_PAGETABLE, NULL);
    if (Section == NULL) // no paging
        return NULL;
    else
        return (RTPageTableEntry*) Section->SectionAddr;
}
```

The following code lists all available RAM files on the screen:


```
for (i=1;;i++)
{
    RTAppSection * Section = RTLocateSection(i, RT_ST_FILEDATA, NULL);
    if (Section == NULL)
        break;
    printf("File name: %s\n", RTSectionName(Section));
}
```

Function RTSectionName

Function RTSectionName can determine the name of a section located using RTLocateSection:

```
char * RTSectionName(const RTAppSection * Section);
```

If the section exists, the function returns a pointer to its name; otherwise, NULL is returned.

Function RTGetExtMem

This function can retrieve the amount of extended memory reported by the BIOS at boot time:

```
DWORD RTGetExtMem(void);
```

The return value is the amount of installed RAM above address 1M or zero if the BIOS does not support BIOS function int 15h, ah=E820h or ah=E801h.

Extending the heap at run-time is best done with function RTCMOSExtendHeap described later in this chapter. Function RTCMOSExtendHeap calls RTGetExtMem internally.

Function RTGetGMode

This function can retrieve the current BIOS graphics mode:

```
int RTGetGMode(void);
```

If the GMode directive was used, this function returns the graphics mode set during the boot process. Please note that the returned mode may actually be a text mode (e.g., value 3 or 7), if none of the graphics modes were accepted by the BIOS.

On targets which do not have a BIOS, value 0 or -1 is returned.

For more information about the GMode command, see Chapter 3, *GMode Command*.

Function RTGetVideoRAMAddr

Function RTGetVideoRAMAddr returns the video RAM address specified in the VideoRAM command:

```
void * RTGetVideoRAMAddr(void);
```

If no video RAM is present or used, NULL is returned.

Function RTLoadRTBFile

Function RTLoadRTBFile reads an RTB file produced by RTLoc and copies all entities it contains to their respective target addresses:

```
typedef void * (RTTAPI * RTDataCallback)(void * dest,
                                         const void * src,
                                         unsigned int n);
RTAppHeader * RTLoadRTBFile(const char * Name,
                           RTDataCallback DataCallback);
```

Parameter Name is the file name of the RTB file to process (note that RTFiles-32 is required to read disk files instead of RAM/ROM files). Optional parameter DataCallback can point to a function which RTLoadRTBFile() will call to transfer data to the target. Supply a function if special processing is required (e.g., programming the data into flash memory, etc.). If not specified, RTLoadRTBFile will simply memcpy() the data directly to the addresses given in the RTB file. The return value is a pointer to the application's header or NULL if the function fails.

Note that this function is not guaranteed to correctly read RTB files produced by different versions of RTTarget-32. If cross version compatibility is required, use BIN files instead of RTB files to transfer On Time RTOS-32 program images.

Demo program Loader demonstrates the use of this function.

Function RTRunProgram

RTRunProgram can execute a child application contained in a separate program image produced by RTLoc:

```
int RTRunProgram(const RTAppHeader * Header);
```

Parameter Header must point to the header of the child application. The return value is the exit code of the child.

The child application referenced by parameter Header must have been completely loaded into the physical address space. This can be done with function RTLoadRTBFile as in demo program Loader or by loading a BIN file as demonstrated by program BootProg.

RTRunProgram will save and restore the following interrupt vectors before/after the child runs: 7h, 61h, and 40h - 4Fh.

A program which intends to execute function RTRunProgram (called the loader or parent) must adhere to the following restrictions:

- It must reside completely in the physical address space. This means that it cannot use virtual regions or the FillRAM command in its configuration file.
- It must not use RTTarget-32's virtual memory manager with uncommitted memory support. Preferably, it should not use the run-time system and a heap at all to preserve memory for the program to be loaded. Please refer to *Function RTSetFlags* earlier in this chapter on how to force use of the fixed memory manager.
- While the child program is running, the parent's heap is not accessible unless it has been assigned ReadWrite access in the configuration file.
- The child program must not overlap any memory used by the parent. This is best achieved by using the *Reserve* command in the child's configuration file.
- RTRunProgram is not reentrant and cannot be called simultaneously by several tasks in a multi-tasking environment. However, it may be used again by a child program.
- For applications running at CPL 3, RTRunProgram may disable interrupts for several seconds to expand compressed entities.

Example program Loader demonstrates how to use RTRunProgram.

Function RTBootRM and RTBootPM

These functions allow one program to boot into another previously loaded program. They are similar to RTRunProgram, but the newly loaded program overlays the loading program and it cannot return to its loader:

```
void RTBootRM(WORD BootVectorSegment, WORD BootVectorOffset);  
void RTBootPM(void * BootVectorAddress);
```

RTBootRM reboots the target into a new program and switches to real mode before transferring control to the 16:16 real mode address *BootVectorSegment:BootVectorOffset*. RTBootPM reboots the target into a new program in 32-bit protected mode. The function's parameters are the physical addresses of the boot vectors of the child application. Both functions completely reset the target and begin execution at the given addresses just like after power-on or reset.

RTBootRM expects a real mode address as its parameter. For children built with RTTarget-32, parameter BootVectorSegment must be bits 19..16 of the physical address of the boot vector shifted right by 4 bits. Parameter BootVectorOffset must be the low 16 bits of the physical address of the boot vector. Example (assuming the boot vector has been located to the very start of section ChildImage):

```
DWORD BootVectorAddress = LoadBinFile("child.bin", "ChildImage");  
RTBootRM((BootVectorAddress & 0x0F0000) >> 4,  
         BootVectorAddress & 0x0FFFF);
```

All of the child's data must have been loaded at the correct addresses on the target before these functions can be called. These functions never return. The child is booted and will overwrite the parent in memory.

The child must include a boot vector, boot code, and boot data just like a self-booting application. A disk buffer is not required but does no harm. RTBootRM supports boot codes BOOT.EXE and BIOS-BOOT.EXE, while RTBootPM supports PMBOOT.EXE. RTBootRM requires the boot vector and boot code to reside below 1M.

To use RTBootRM or RTBootPM:

- The parent must be located such that it reserves some target address space to hold the child's image.
- If the child includes BIOSBOOT.EXE as its boot code, the parent must not clobber the BIOS data area in the first 4k of address space. This is best achieved by allocating a Nothing section in region NullPage with NoAccess access rights. Some BIOS extensions may require additional memory at the end of the DOS memory area below address A0000h. Thus, it is recommended to also allocate the last 64k or 128k of DOS memory using a Nothing section.
- The child should be located such that all of its images will reside in the address space reserved for its image.
- The parent must load the child's image before calling RTBootRM/RTBootPM. It can use function RTLoadRTBFile to do this or simply load a BIN file to the correct address.
- Call RTBootRM for children containing real mode boot code (BOOT.EXE or BIOSBOOT.EXE) or RTBootPM for children containing protected mode boot code (PMBOOT.EXE).

It is not possible to debug a child started with RTBootRM/RTBootPM using the cross debugger.

Demo program BootProg demonstrates function RTBootRM.

Function RTDLLThreadEvent

Function RTDLLThreadEvent calls a specific or all entrypoints (function DllEntryPoint) of DLLs currently in use:

```
void RTDLLThreadEvent(HMODULE Handle, DWORD Win32Event);
```

Parameter Handle is the handle of the DLL's entrypoint to be called. Handle may be NULL. In this case, the entrypoints of all currently loaded DLLs (DLLs with a current reference count greater 0) are called.

Parameter Win32Event is passed on to the DllEntryPoint functions as parameter fdwReason. It may have one of the values DLL_THREAD_ATTACH, DLL_THREAD_DETACH, or DLL_PROCESS_DETACH.

The reference count of the DLL(s) is not modified by this function. Thus, passing DLL_PROCESS_DETACH is not recommended (use FreeLibrary instead), except at program termination.

RTDLLThreadEvent is typically used in multithread applications to inform DLLs of thread creation/termination.

More information about using DLLs is available in Chapter 9, *Using DLLs through RTLoc* and Chapter 7, *Win32 DLLs*.

Function RTLockHeap

RTLockHeap can be used to protect RTTarget-32 memory management functions from being reentered in multi-thread applications:

```
void RTLockHeap(void);
```

All of RTTarget-32's Win32 memory management functions (VirtualAlloc, VirtualFree, HeapAlloc, HeapFree, HeapRealloc, LocalAlloc, LocalFree, etc.) use an internal critical section to protect RTTarget-32's internal memory management state. All of the run-time system's memory management functions map onto these functions. However, RTTarget-32's memory management and mapping functions (RTFindPhysMem, RTReserveVirtualAddress, RTReleaseVirtualAddress, RTMapMem, RTExtendHeap,

and RTCMOSExtendHeap) do not use this critical section automatically. If any of these functions are called while any other memory management function could be called by another thread, the RTTarget-32 heap must be locked explicitly with this call and unlocked with function RTUnlockHeap.

Use of this function is only required in multi-thread applications. In single-thread programs, it does nothing.

Function RTUnlockHeap

RTUnlockHeap unlocks RTTarget-32's memory management functions previously locked with RTLockHeap:

```
void RTUnlockHeap(void);
```

Please refer to the previous section for information about locking RTTarget-32's memory management.

Function RTCallRing0

This function allows executing a function at CPL 0 within a program running at CPL 3 or 0:

```
typedef DWORD (__fastcall * RTRing0Function)(void * P);  
DWORD RTCallRing0(RTRing0Function Ring0Func, void * P);
```

The function to be called has a single 32-bit parameter, a 32-bit function result, and must use register calling conventions. RTCallRing0 will pass its parameter P to Ring0Func in EAX and will return the value returned by Ring0Func in EAX.

If the calling program runs at CPL 0, RTCallRing0 will simply call Ring0Func. If the program executes at CPL 3, RTCallRing0 will use a call gate to transfer control to Ring0Func. In addition, interrupts are disabled and the interrupt state is restored when the function returns.

A few things should be noted if this function is used in programs running at CPL 3:

- Ring0Func may call other functions. Such called functions will also execute at CPL 0.
- Interrupts are disabled and must not be reenabled. Thus, the function should be short.
- You cannot set debugger breakpoints within the Ring0Func or any function it calls.
- The function will execute on the boot code's ring 0 stack, which has a size of only 256 bytes. Please be sure not to use more stack space.
- All privileged instructions except HLT can be executed. Write protection for read-only and system pages is not enforced.
- Code executing at CPL 0 may never raise an exception, be it a Win32 structured or C++ exception. Exceptions cannot be propagated across privilege levels.
- You cannot execute floating point instructions if the emulator is being used.
- Selector 3Bh (index 7) is used for the call gate and must not be used by the application.

Function RTFindPhysMem

Function RTFindPhysMem scans the page table for a physical address range:

```
void * RTFindPhysMem(void * Physical, unsigned Bytes, int MinAccess);
```

Parameter Physical is the address of the physical memory to find. Parameter Bytes defines its size. MinAccess specifies the minimum access level required. Values RT_PG_NOACCESS, RT_PG_SYSREAD, RT_PG_SYSREADWRITE, RT_PG_USERREAD, and RT_PG_USERREADWRITE are supported.

The return value is a pointer of the virtual address space to the requested physical address. If the function fails, NULL is returned.

Demo program MapDemo demonstrates how to use RTFindPhysMem.

Function RTReserveVirtualAddress

Function RTReserveVirtualAddress allocates a virtual address range:

```
int RTReserveVirtualAddress(void ** Virtual, unsigned Bytes, unsigned Flags);
```

Parameter Virtual is a pointer to a pointer containing the address of the virtual address space. Parameter Bytes specifies the size of the required address range. Parameter Flags specifies options. The only flag currently defined is RT_MAP_NO_RELOCATE. It prevents ReserveVirtualAddress from allocating a different address to *Virtual.

*Virtual should be initialized by the application before ReserveVirtualAddress is called to supply a preferred virtual address. If this address is available, it will be used. Otherwise, if RT_MAP_NO_RELOCATE is not set, ReserveVirtualAddress scans the virtual address space starting at zero for a free area large enough to satisfy the request. If the original value of *Virtual is not page-aligned, the page offset of *Virtual is always preserved.

When this function succeeds, *Virtual contains the address of the reserved address range.

The function return value can be one of the following:

RT_MAP_SUCCESS	The function succeeded.
RT_MAP_NO_PAGING	Paging is not enabled. This function requires paging.
RT_MAP_VADDR_NOT_AVAIL	The requested virtual address is not free and RT_MAP_NO_RELOCATE was specified, preventing RTReserveVirtualAddress from searching for a different address.
RT_MAP_OUT_OF_MEM	The function was unable to allocate memory for a larger page table.

Demo program MapDemo demonstrates how to use RTReserveVirtualAddress.

Function RTReleaseVirtualAddress

Function RTReleaseVirtualAddress marks an address range to be available for future reservation:

```
int RTReleaseVirtualAddress(void * Virtual, unsigned Bytes);
```

Parameters Virtual and Bytes define location and size of the address range to release. The address range is unmapped from memory, causing any subsequent accesses to trigger page faults.

The function return values can be RT_MAP_SUCCESS or RT_MAP_OUT_OF_MEM (see section *Function RTReserveVirtualAddress*).

Function RTMapMem

Function RTMapMem maps physical to virtual addresses:

```
int RTMapMem(void * Physical, void * Virtual, unsigned Bytes, int Access);
```

Parameters Physical and Virtual define the addresses in the respective address spaces. If parameters Physical and Virtual are not page-aligned, their respective offsets from a page must be identical. Parameter Bytes specifies the size of the address range. Parameter Access specifies the desired access rights to the virtual address range.

If paging is disabled, the function succeeds only if Physical == Virtual addresses. Otherwise, the page table is updated to map the given physical address to the requested virtual address.

This function performs no additional parameter validation. If the given virtual address is used by some other program entity or the physical address does not exist, the results will be unpredictable.

In addition to the return codes documented for function RTReserveVirtualAddress, this function can return the following value:

RT_MAP_PARAM_ERROR The page offset of parameters Physical and Virtual are not identical.

Demo program MapDemo demonstrates how to use RTMapMem.

Function RTextendHeap

Function RTextendHeap can add memory to the heap at run-time. RTextendHeap allows a program built with the same hardware configuration to use more heap memory if it finds more installed:

```
int RTextendHeap(void * Physical, unsigned Bytes);
```

Parameters Physical and Bytes define location and size of the physical memory to be added to the heap. If paging is not enabled, parameter Physical must match the end of the heap section. In this case, the heap is simply extended by Bytes bytes. If paging is used, the given physical memory is mapped to the virtual address immediately following the heap. The function checks that no memory after the current heap is overlapped. Thus, less memory than requested could be appended (possibly zero if some other section is right after the heap).

Please note that this function cannot perform any checks on the passed parameters. If the specified memory does not exist or it has already been used by RTLoc, memory will be corrupted.

If the function succeeds, the number of pages (not bytes) is returned. For error return codes, see the RT_MAP... function codes given for function RTReserveVirtualAddress. In addition, the following return codes are possible:

RT_MAP_HEAP_TOP_USED The application has allocated memory from the end of the heap. The heap cannot be grown in this case.

RT_MAP_HEAP_UNALIGNED The end of the heap is not page-aligned. Therefore, it is impossible to add pages to the heap without leaving a gap.

In general, RTextendHeap can only be called within an Init function before any heap allocation functions have been called to avoid problem RT_MAP_HEAP_TOP_USED.

Please note that RTTarget-32 will never set up virtual memory at addresses which have physical memory. Thus, if you intend to use RTextendHeap or RTCMOSExtendHeap, the address range to add to the heap should not be used by RTLoc. For example, if your configuration file specifies extended memory to end at 2M, and you want to add additional extended memory to the heap at run time, no program entities should be mapped in the address range 2M - 64M through virtual regions.

For systems equipped with a BIOS and CMOS RAM, function RTCMOSExtendHeap is recommended instead.

Demo program MapDemo demonstrates how to use RTextendHeap.

Function RTRaiseCPUException

This function can install exception handlers which will raise a Win32 structured exception when the CPU signals an exception:

```
void RTRaiseCPUException(BYTE Vector);
```

Parameter Vector can be any value in the range 0 to 16. Win32 Function RaiseException will be called when the specified exception occurs. RTRaiseException may be called for any number of vectors in the range 0 .. 16.

CPU exceptions handled in this manner will not be seen by the debugger RTD32, which will usually trap them and halt the program. To conditionally install exception handlers only when the program is not running under the control of the debug monitor, use the following code:

```
if (RTCallDebugger(RT_DBG_MONITOR, 0, 0) != -1)
{
    // the debug monitor is not running; install handlers
    RTRaiseCPUException(0); // divide error
    RTRaiseCPUException(4); // INTO Overflow
    RTRaiseCPUException(5); // range check
    RTRaiseCPUException(6); // invalid opcode
    RTRaiseCPUException(13); // GPF
    RTRaiseCPUException(14); // page fault
    RTRaiseCPUException(16); // floating point error
}
```

Unit RTTarget used in Pascal programs automatically executes the above code at program startup. The Pascal run-time system will map all structured exceptions generated through this mechanism to run-time error exceptions which can be handled by Pascal's exception handling.

Please note that RTRaiseCPUException will happily install handlers for any exception in the range 0..16, but normal applications cannot successfully handle all of them. For example, a stack fault (exception 12) can only be handled if the handler is executed on a different stack, which in turn requires handling it at a higher privilege level. However, the handlers of the application must run at the same privilege level as the application. The same is true for a doubles fault, which could, for example, be triggered by a page fault caused by the stack pointer. Again, the error can only be handled at a higher privilege level.

For such exceptions, it is recommended to leave the default handlers of RTTarget-32's boot code in place. These handlers run at CPL 0 and will display a register dump on the screen, allowing offline analysis.

For additional information about handling structured exceptions, please refer to your compiler's or the Win32 API documentation.

Function RTCMOSRead

This function can read one byte of battery-backed CMOS RAM:

```
BYTE RTCMOSRead(BYTE Addr);
```

Parameter Addr is the byte offset of the value to retrieve. Example:

```
printf("The equipment byte of this PC is: %u\n", RTCMOSRead(0x14));
```

RTCMOSRead only works on systems equipped with an MC146818A Real-Time Clock or compatible device. The contents of the CMOS RAM area are valid only if it has been initialized by a BIOS or by the program itself.

Function RTCMOSWrite

This function can write one byte of battery-backed CMOS RAM:

```
void RTCMOSWrite(BYTE Addr, BYTE Value);
```

Parameter Addr is the byte offset of the value to write.

RTCMOSWrite only works on systems equipped with an MC146818A Real-Time Clock or compatible device. The contents of the CMOS RAM area are valid only if it has been initialized by a BIOS or by the program itself.

Function RTCMOSReadTime

RTCMOSReadTime reads the current date and time of day from the Real-Time Clock:

```
void RTCMOSReadTime(SYSTEMTIME * T);
```

Parameter T points to the SYSTEMTIME structure defined in window.h and rttarget.h (respective windows.pas for Delphi).

This function requires an MC146818A Real-Time Clock on the target.

Function RTCMOSWriteTime

RTCMOSWriteTime writes the given date and time of day to the Real-Time Clock:

```
void RTCMOSWriteTime(const SYSTEMTIME * T);
```

Parameter t points to the SYSTEMTIME structure defined in windows.h and rttarget.h (respective windows.pas for Delphi).

This function requires an MC146818A Real-Time Clock on the target.

Function RTCMOSSetSystemTime

RTCMOSSetSystemTime reads the current date and time of day from the Real-Time Clock and calls the Win32 function SetSystemTime:

```
void RTCMOSSetSystemTime(void);
```

Function SetSystemTime must be called at least once at program initialization if the application intends to use time-of-day functions of the run-time system or Win32 emulation. RTCMOSSetSystemTime performs this function.

This function requires an MC146818A Real-Time Clock on the target.

Function RTCMOSExtendHeap

RTCMOSExtendHeap uses information from the BIOS and CMOS RAM to determine the amount of physical memory installed. If more memory than currently used is found, the extra RAM is added to the heap:

```
int RTCMOSExtendHeap(void);
```

Internally, this function calls RTGetExtMem, RTCMOSRead, and RTEExtendHeap. For possible return codes and additional information, please refer to *Function RTEExtendHeap* earlier in this chapter. RTCMOSExtendHeap should only be called from Init functions.

This function requires a PC compatible BIOS with support for int 15h function E820h or E801h, or an MC146818A Real-Time Clock on the target.

Function RTSetKeyboard

This function can set a few keyboard options:

```
void RTSetKeyboard(DWORD LockState, int RepeatDelay, int RepeatRate);
```

Parameter LockState can be any combination of NUMLOCK_ON, SCROLLLOCK_ON, and CAPS-LOCK_ON defined in windows.h.

Parameter RepeatDelay must be in the range 0 .. 3. It specifies the delay after which keyboard input is automatically repeated when a key is held down. At zero, the delay is set to 0.25 seconds; at 3, the delay is 1 second.

Parameter RepeatRate must be in the range 0 .. 31. It specifies the rate at which keys are generated when a key is held down. At zero, the rate is 30 characters per second; at 31, the rate is 2 characters per second.

When this function is never called, RTSetKeyboard(NUMLOCK_ON, 0, 0) is assumed. The NUMLOCK state can also be changed with the RTTarget-32 flags (see section *RTTarget-32 Flags* earlier in this chapter).

Function RTSetKeyboardTables

RTTarget-32's keyboard driver is preconfigured to support US, German, and French keyboard layouts using the OEM (437) code page. Function RTSetKeyboardTables is used to install and activate custom translation tables for the keyboard driver:

```
typedef const struct {           // generic translator, first entry: {0, #Entries}
    WORD Key;                    // must be sorted on Key
    WORD Value;
} RTKeyTable;

typedef const struct {           // lookup tables for keyboard driver
    int BasedOn;                 // based on language index BasedOn
    DWORD Flags;
    RTKeyTable * SCToVK;         // scan code -> virtual key code
    RTKeyTable * VKToLower;      // virtual key code -> Unicode character
    RTKeyTable * VKToUpper;      // virtual key code -> character (upper)
    RTKeyTable * AltCar;         // virtual key code -> character (AltCar)
} RTKeyLanguage;

RTKeyLanguage * RTSetKeyboardTables(int Index, RTKeyLanguage * Table);
```


Variable size arrays of type `RTKeyTable` are used as generic translation tables. The first entry's `Value` member must have the number of following value pairs. The table must be sorted on member `Key`.

Type `RTKeyLanguage` contains all lookup tables the keyboard driver needs to map keyboard scan codes supplied by the keyboard controller to character values. Member `BasedOn` specifies which other keyboard table should be used if no translation for a particular value is found. `BasedOn` implements a linked list of `RTKeyLanguage` structures of up to 16 entries. Value `-1` specifies that no other tables should be scanned. Member `Flags` currently supports 2 bits: when bit `0x00000001` is set, the `CapsLock` key behaves like a shift lock and applies not only to letters. If bit `0x00000002` is set, pressing a shift key unlocks the `CapsLock` key. Table `ScToVK` is used to map device scan codes to Win32 virtual key codes. Standard virtual key codes are declared in `winuser.h`. If you need to define custom virtual key codes, use values in the range `0x0100..0xFFFFE`. Table `VKToLower` maps virtual key codes to Unicode character values. Table `VKToUpper` maps virtual key codes to Unicode character values when shift is pressed. Table `AltCar` is used to translate virtual key codes when the right Alt key is being held down. To explicitly disable a key in a particular table, translate it to value `0xFFFF`.

Function `RTSetKeyboardTables` is used to install and activate the translation tables for the keyboard driver. Parameter `Index` may be in the range `0..15`. Indices `0..7` are reserved for US, German, and French, respectively, and for future extensions. They should not be used (but may be used to remove and override a predefined language). It is recommended to install custom keyboard mappings on higher indices such as `15` or `11`. After installation, different languages can be activated by calling `RTSetFlags(Index << 8, 1)`; or by pressing `Ctrl-Alt-Fx`, where the `x` is `Index+1`. The function's return value is a pointer to the previously installed language structure on the specified index.

Function `RTSetCodepageTranslation`

Function `RTSetCodepageTranslation` installs a translation table to map Unicode character values to 8-bit ASCII values:

```
void RTSetCodepageTranslation(RTKeyTable * Table);
```

Unicode values not found in the table will merely be truncated to 8 bits. By default, functions `ReadConsoleInputA` and `PeekConsoleInputA` use code page 437 (OEM code page) mapping, just like Windows NT/2000. However, RTPEG-32 will change the input code page to 1252 (ANSI), which is largely identical to Unicode.

Function `RTInitMouse`

Function `RTInitMouse` installs and initializes RTTarget-32's mouse driver. This driver supports Microsoft compatible serial and PS/2 mice. It does not display a mouse cursor on the screen, but merely generates mouse events in the program's console event queue:

```
void RTInitMouse(int PortIOBase, int PortIRQ,
                 int DoubleClickSpeed,
                 int ScaleX, int ScaleY);
```

Parameters:

<code>PortIOBase</code>	The port I/O address of the serial port used by the mouse. For standard ports, you should use <code>3F8h</code> (COM1), <code>2F8h</code> (COM2), <code>3E8h</code> (COM3), or <code>2E8h</code> (COM4). For a PS/2 mouse, this parameter must be <code>-1</code> .
<code>PortIRQ</code>	The interrupt request line of the serial port. Typical values are <code>4</code> , <code>3</code> , <code>4</code> , and <code>3</code> for the ports COM 1, 2, 3, and 4, respectively. PS/2 mice always use IRQ 12; thus, <code>RTInitTextMouse</code> ignores this value for PS/2 mice.
<code>DoubleClickSpeed</code>	Maximum number of milliseconds between left button clicks to form a double click. This value may be zero. In this case, a default value of <code>300</code> is used.
<code>ScaleX</code>	The factor by which raw horizontal mouse motion is multiplied. Large values cause fast mouse motion.
<code>ScaleY</code>	The factor by which raw vertical mouse motion is multiplied. Large values cause fast mouse motion.

Please see section *Console Input Event Management* later in this chapter for information on how mouse and keyboard events are processed by RTTarget-32.

All RTPEG-32 and MetaWINDOW demo programs use the mouse driver.

Function RTInitTextMouse

Function RTInitTextMouse installs and initializes RTTarget-32's mouse driver. Unlike RTInitMouse, this driver also displays a text mode mouse cursor if the program uses console I/O functions to write to the screen:

```
void RTInitTextMouse(int PortIOBase, int PortIRQ,
                    int DoubleClickSpeed,
                    int ScaleX, int ScaleY,
                    char PointerChar,
                    unsigned char PointerColor);
```

Parameters:

PortIOBase	The port I/O address of the serial port used by the mouse. For standard ports, you should use 3F8h (COM1), 2F8h (COM2), 3E8h (COM3), or 2E8h (COM4). For a PS/2 mouse, this parameter must be -1.
PortIRQ	The interrupt request line of the serial port. Typical values are 4, 3, 4, and 3 for the ports COM 1, 2, 3, and 4, respectively. PS/2 mice always use IRQ 12; thus, RTInitTextMouse ignores this value for PS/2 mice.
DoubleClickSpeed	Maximum number of milliseconds between left button clicks to form a double click. This value may be zero. In this case, a default value of 300 is used.
ScaleX	The factor by which raw horizontal mouse motion is divided. Large values cause slow mouse motion. If set to zero, the default of 3 is used.
ScaleY	The factor by which raw vertical mouse motion is divided. Large values cause slow mouse motion. If set to zero, the default of 8 is used.
PointerChar	The character value to represent the mouse pointer. Zero will produce a blank pointer.
PointerColor	The screen attribute of the mouse pointer. Zero will produce a black mouse pointer.

Please see section *Console Input Event Management* later in this chapter for information on how mouse and keyboard events are processed by RTTarget-32.

The Turbo Vision example included with RTTarget-32 demonstrates how to use this text mode mouse driver.

Function RTSetMousePos

RTSetMousePos informs the mouse driver of a new pointer position:

```
void RTSetMousePos(int X, int Y);
```

The mouse driver will prevent the cursor from having negative coordinates, but if RTInitMouse was used to initialize the mouse, the driver does not know the screen resolution and thus cannot prevent the mouse cursor from leaving the screen. Use RTSetMousePos() to bring the mouse cursor back onto the screen. The application will usually not need this function. For MetaWINDOW, RTGetMetaWEvents() calls RTSetMousePos() whenever required. For RTPEG-32, the internal event manager also uses it.

Function RTMouseDone

Once the mouse driver initialized with RTInitMouse is no longer needed, RTMouseDone must be called to disable interrupts on the serial port or PS/2 port used and to restore the original interrupt vector:

```
void RTMouseDone(void);
```

Function RTTextMouseDone

This function is identical to RTMouseDone().

Function RTMakeBootDisk

RTMakeBootDisk can create a bootable diskette or hard disk:

```
int RTMakeBootDisk(char      LogicalDrive,
                   int       BIOSDevice,
                   const char * RTBFileName,
                   char *     Buffer,
                   int        BufferSize,
                   DWORD      Flags);
```

RTMakeBootDisk requires On Time's embedded file system RTFiles-32 and is not included in RTTarget-32's preconfigured system DLL RTT32DLL.DLL.

Parameter LogicalDrive specifies the disk drive from which to boot. The new boot image file (.RTA file) will be created in the root directory of that drive.

Parameter BIOSDevice specifies the physical device identification used by the BIOS for the device at boot time. If this parameter is set to -1, RTMakeBootDisk will choose an appropriate default. If LogicalDrive is 'A' or 'B', BIOSDevice is set to 0 (the first diskette drive). For all other drives, BIOSDevice is set to 80h (the first hard disk).

Parameter RTBFileName points to the file name of the RTB file to install on the bootable drive.

Parameter Buffer must point to a temporary buffer to be used as a disk buffer by RTMakeBootDisk. It should have a size of at least 8k (larger sizes may improve performance for large .RTB files).

Parameter BufferSize specifies the size in bytes of the buffer referenced by parameter Buffers.

Parameter Flags can be zero or specifies options for the function. Currently, only the following option is defined:

RT_BDISK_DEL_RTA RTMakeBootDisk should delete all .RTA files it finds on the target drive before the new .RTA file is created. If not specified, existing RTA files are erased at the end of the operation.

The function return value can be one of the following:

RT_BDISK_SUCCESS	The new boot image was installed successfully.
RT_BDISK_OUT_OF_MEM	The supplied buffer is too small.
RT_BDISK_INVALID_RTB	The RTB file is invalid or was produced with an older version of RTTarget-32.
RT_BDISK_ERROR_OPEN_DEVICE	RTFiles-32 reported an error on the attempt to open a drive file for the target drive.
RT_BDISK_NO_BOOT_CODE	The RTB file or the target drive contain no valid boot code.
RT_BDISK_ERROR_CREATING_RTA	RTMakeBootDisk was unable to create the boot image file on the target drive. This can happen when there is insufficient disk space on the target drive. Flag RT_BDISK_DEL_RTA may be able to fix this problem.
RT_BDISK_NOT_CONTIGUOUS	The boot image file is not contiguous. This error should never occur.
RT_BDISK_INVALID_SECTOR_SIZE	The target drive has a sector size other than 512.
RT_BDISK_ERROR_WRITE_BOOT_SECTOR	RTMakeBootDisk was unable to write the new boot sector to disk.
RT_BDISK_OLD_RTB	The RTB file is invalid or was produced with an older version of RTTarget-32.
RT_BDISK_INVALID_BOOT_CODE	The boot code found in the .RTB file is corrupted.
RT_BDISK_RTB_NOT_FOUND	RTMakeBootDisk was unable to open the given .RTB file.

RT_BDISK_UNSUPPORTED

RTMakeBootDisk is not supported because it was built without RTFiles-32.

When flag RT_BDISK_DEL_RTA is not specified, RTMakeBootDisk guarantees that the new program is properly installed or, in case of an error, any previously existing boot image is still bootable. With flag RT_BDISK_DEL_RTA set, the disk may be left in an unbootable state when the function fails. In particular, the following steps are performed:

- If no backup boot sector is present and the currently installed boot sector is not an RTTarget-32 boot sector, the boot sector is saved as file BOOTSECT.RTT on the target drive.
- If flag RT_BDISK_DEL_RTA is set, the backup copy of the boot sector is installed (if present) and all .RTA files in the target drive's root directory are deleted. If the boot sector backup is present, the disk would now boot its original operating system, or if the backup boot sector is not present, any attempt to boot from the disk would fail.
- A temporary file to receive the boot image is created in the root directory of the target drive and is written.
- If not done already, all .RTA files in the target drive's root directory are deleted.
- The temporary file with the boot image is renamed to the RTB file's name with extension .RTA instead of .RTB.
- If the boot image was written successfully, the new boot sector is written to the disk.

Function RTRestoreBootSector

Function RTRestoreBootSector can restore a previously saved boot sector on an RTTarget-32 boot disk:

```
int RTRestoreBootSector(char LogicalDrive);
```

RTRestoreBootSector requires On Time's embedded file system RTFiles-32.

Parameter LogicalDrive specifies the drive letter of the drive containing the RTTarget-32 boot disk.

This function will only succeed if file BOOTSECT.RTT is present on the target drive. This file is created by function RTMakeBootDisk or RTTarget-32 command line utility BOOTDISK.EXE if a non-RTTarget-32 boot sector is found.

Function RTRestoreBootSector can return any of the following error codes as described in the previous section:

```
RT_BDISK_SUCCESS  
RT_BDISK_NO_BOOT_CODE  
RT_BDISK_ERROR_OPEN_DEVICE  
RT_BDISK_ERROR_WRITE_BOOT_SECTOR
```

Function RTPrinterSetIOBase

Function RTPrinterSetIOBase defines the I/O port address of a parallel (Centronics) port:

```
BYTE RTPrinterSetIOBase(int Port, WORD IOBase);
```

Parameter Port must be one of the values RT_LPT1, RT_LPT2, RT_LPT3, or RT_LPT4. When this function is never called for a parallel port, RTTarget-32 defaults to I/O port address 378h, 278h, and 3BCh, respectively. There is no default address for RT_LPT4.

The function return value is the printer's status. See function RTPrinterStatus for details.

Function RTPrinterInit

RTPrinterInit resets and initializes a printer:

```
BYTE RTPrinterInit(int Port);
```

Parameter Port must be one of the values RT_LPT1, RT_LPT2, RT_LPT3, or RT_LPT4. The function return value is the printer's status. See function RTPrinterStatus for details.

Function RTPrinterStatus

Function RTPrinterStatus returns the current status of a printer:

```
BYTE RTPrinterStatus(int Port);
```

Parameter Port must be one of the values RT_LPT1, RT_LPT2, RT_LPT3, or RT_LPT4.

The return value is a standard BIOS parallel port status byte. The following bits can be set:

RT_LPT_READY	The printer is ready.
RT_LPT_ACK	The printer has acknowledged data.
RT_LPT_OUT_OF_PAPER	The printer is out of paper.
RT_LPT_SELECTED	The printer is selected.
RT_LPT_IO_ERROR	An error occurred sending to the printer. This error is also returned if the parallel port is not installed.
RT_LPT_TIMEOUT	A timeout error occurred while sending data to the printer.

Function RTPrintByte

Function RTPrintByte sends a byte of data to a parallel (Centronics) port:

```
BYTE RTPrintByte(int Port, BYTE Data, DWORD Timeout);
```

Parameter Port must be one of the values RT_LPT1, RT_LPT2, RT_LPT3, or RT_LPT4, which RTTarget-32 maps to I/O port address 378h, 278h, and 3BCh, respectively. There is no default address for RT_LPT4. If you need a fourth parallel port, you can use RT_LPT4 only after having called RTPrinterSetIOBase(RT_LPT4, ...);

Parameter Data is the byte to output to the printer. Parameter Timeout specifies in milliseconds how long the function should wait for the printer not to signal busy before returning a timeout error.

The function return value is the printer's status, see function RTPrinterStatus for details.

Serial I/O Functions

Since many embedded systems applications require serial I/O, RTTarget-32 supplies functions for this purpose. Module RTTCOM has the following features:

- Asynchronous, interrupt-driven send and receive.
- Arbitrary send and receive buffer sizes.
- Up to four ports can be handled simultaneously.
- Supports XOn/XOff, RTS/CTS, and DTR/DSR handshake.
- Automatically detects 16550 UARTs and supports their internal FIFO buffer.
- Full control over all modem control lines.

RTTCOM can work with 8250, 16450, 16550 and compatible UART chips. It expects an 8259A compatible interrupt controller at port address 20h. If IRQs greater 7 are used, a slave interrupt controller is assumed at port address A0h.

RTTCOM's functions are described in the following sections. Prototypes for all of them are given in header file RTTCOM.H.

If you use RTKernel-32 with RTTarget-32, RTTarget-32's RTTCOM should **not** be used. RTKernel-32's module RTCOM is optimized for serial communication in a real-time multitasking environment.

Demo program SERDEMO.C demonstrates how to use RTTCOM.

Function RTInitCOMPort

RTInitCOMPort initializes a port and prepares it for communication:

```
void RTInitCOMPort(int Port,
                  int IOBase,
                  int IRQ,
                  int Baudrate,
                  int Parity,
                  int StopBits,
                  int WordLength,
                  int ReceiveBufferSize,
                  int SendBufferSize,
                  int Protocol);
```

Parameters:

Port	RT_COM1 .. RT_COM4 (0 .. 3).
IOBase	Any 16-bit integer. The array RTDefaultCOMIOBase can be used for 'standard' ports.
IRQ	0 .. 15. The array RTDefaultCOMIRQ can be used for 'standard' ports. Interrupt sharing is only supported for port pairs COM1/COM3 and COM2/COM4. For best performance, individual IRQs are recommended.
Baudrate	50 .. 115200. A baud rate input clock frequency of 1.8432 Mhz (the PCs default) is assumed. If a different value is used by the target, you must scale the baud rate accordingly. See Chapter 3, section <i>COMPort Command</i> for details.
Parity	Values PARITY_NONE, PARITY_ODD, PARITY_EVEN, PARITY_MARK, and PARITY_SPACE (all defined in Rttcom.h and windows.h) are supported.
StopBits	Values 1 and 2 are supported.
WordLength	Values 5, 6, 7 and 8 are supported.
ReceiveBufferSize	Any size between 1 and 2G is supported. The buffer is allocated from the default Win32 process heap.
SendBufferSize	Any size between 1 and 2G is supported. The buffer is allocated from the default Win32 process heap.
Protocol	RT_NO_PROT, RT_XON_XOFF, RT_RTS_CTS, or RT_DTR_DSR.

Parameters IOBase, IRQ, ReceiveBufferSize, and SendBufferSize are evaluated only the first time InitCOMPort is called for a particular port. Subsequent calls can only be used to change the UART initialization. If InitCOMPort detects a 16550 UART, its FIFO is automatically enabled at trigger level 8.

Function RTCloseCOMPort

Once a COM port is no longer used, RTCloseCOMPort should be called to disable interrupts for the port and to restore the interrupt vector in the interrupt descriptor table.

```
void RTCloseCOMPort(int Port);
```

Parameter Port must have value RT_COM1 .. RT_COM4. The port must have been initialized previously.

If several COM ports sharing the IRQ have been initialized, it is important that RTCloseCOMPort is called in the reverse order as RTInitCOMPort. Otherwise, interrupt vectors are not restored correctly.

Example:

```
RTInitCOMPort(RT_COM1, ...);
RTInitCOMPort(RT_COM3, ...);
...
RTCloseCOMPort(RT_COM3);
RTCloseCOMPort(RT_COM1);
```

Function RTSendChar

RTSendChar places a character in the send buffer and returns immediately:

```
void RTSendChar(int Port, Byte Data);
```

The function does not wait for the data to be sent. Rather, the data is sent from the send buffer by a send interrupt as soon as the transmit register of the UART becomes empty. However, the function does wait until space becomes available if the send buffer is full.

Function RTSendCharTimed

RTSendCharTimed is similar to RTSendChar, but a timeout for the operation can be specified:

```
BOOL SendCharTimed(int Port, Byte Data, int Timeout);
```

The function returns FALSE if the data could not be placed in the send buffer within the time given in parameter Timeout in milliseconds. Please note that the data is merely placed in the send buffer; it is not guaranteed to have been sent even when this function returns TRUE.

Function RTSendBlock

RTSendBlock transfers a block of data to the send buffer:

```
void RTSendBlock(int Port, void * Data, int Length);
```

Parameter Data points to the data to be transmitted. Length specifies the size of the block in bytes. This is equivalent to

```
for(i=0; i<Length; i++)
    RTSendChar(P, Data[i]);
```

but is faster.

Function RTSendBlockTimed

RTSendBlockTimed allows specifying a timeout for a block of data to be sent:

```
int RTSendBlockTimed(int Port, void * Data, int Length, int Timeout);
```

This function returns the number of bytes placed in the send buffer without getting a timeout. If all data was transferred to the send buffer, Length is returned. The timeout in milliseconds applies to the complete data block.

Function RTSendBufferCount

The status of the send buffer can be enquired with this function:

```
int RTSendBufferCount(int Port);
```

RTSendBufferCount returns the number of bytes currently in the send buffer for the corresponding port. Please note that the last byte may still be in the send shift register even if this function returns 0. To be sure all data has been transmitted, check bit RT_TX_SHIFT_EMPTY in the Line Status register. Example:

```
while (RTSendBufferCount(P) > 0)
{
    printf("waiting for the send buffer... ");
    RTWait();
}

while (!(RTLineStatus(P) & RT_TX_SHIFT_EMPTY))
    printf("waiting for the UART... ");
```

Function RTReceiveBufferCount

The status of the receive buffer can be enquired with this function:

```
int RTReceiveBufferCount(int Port);
```

RTReceiveBufferCount returns the number of bytes currently in the receive buffer for the corresponding port.

Function RTReceiveChar

RTReceiveChar retrieves a byte from the receive buffer:

```
RTCOMData ReceiveChar(int Port);
```

If the receive buffer is empty, this function waits until data comes in. The return value contains the actual received data in the low byte and any error information in the high byte (see RTTCOM.H for all possible values). If the high byte is 0, no errors have occurred. For complete error checking, use code such as this to receive:

```
C = RTReceiveChar(PORT);
if ((C & 0xFF00) == 0)          // No errors?
    printf("received: %c\n", C); // display received char
else                            // handle receive errors
    printf(RTCOMError(C));       // display receive error message
```

Function RTReceiveCharTimed

RTReceiveCharTimed receives data with a timeout:

```
RTCOMData RTReceiveCharTimed(int Port, int Timeout);
```

If no data is received within the given timeout, the function returns with the RT_TIMEOUT bit set in the high byte of the return value.

Function RTCOMError

RTCOMError can be used to display error information for received data:

```
char * RTCOMError(RTCOMData Data);
```

The return value is a pointer to a string corresponding to the most severe error set in Data.

Function RTLineStatus

Function RTLineStatus queries the Line Status Register of a port:

```
BYTE RTLineStatus(int Port);
```

See the status mask constants in RTTCOM.H for all possible return values. Several bits may be set simultaneously.

Function RTModemStatus

Function RTModemStatus queries the Modem Status Register of a port:

```
BYTE RTModemStatus(int Port);
```

See the status mask constants in RTTCOM.H for all possible return values. Several bits may be set simultaneously.

Function RTModemControl

Function RTModemControl can change control lines of the UART:

```
void RTModemControl(int Port, int SetToOneZero, int NewValue);
```

See the status mask constants in RTTCOM.H for all possible values. If Parameter SetToOneZero == 1, the value is "ored" with the register; otherwise, it is "not-anded".

PCI BIOS Functions

PCI I/O cards may require configuration at run-time, or software may need to query configuration information about such cards. Protected mode PCI BIOS version 2.1 services can be used to access the PCI configuration address space on a PCI bus.

All PCI BIOS functions made available by RTTarget-32 are declared in header file RTTBIOS.H. For Pascal, the driver's API is defined in unit RTTBIOS.PAS.

The PCI BIOS returns standard status codes. The following codes can be returned:

RTT_BIOS_SUCCESSFUL	The function succeeded.
RTT_BIOS_FNC_NOT_SUPPORTED	The BIOS does not support the requested function.
RTT_BIOS_BAD_VENDOR_ID	An invalid vendor ID was specified.
RTT_BIOS_DEVICE_NOT_FOUND	The requested device was not found.
RTT_BIOS_BAD_REG_NUM	The specified register does not exist.
RTT_BIOS_SET_FAILED	The BIOS was unable to write a configuration register value.
RTT_BIOS_BUFFER_TOO_SMALL	The supplied return buffer space was too small.

Additional information about the PCI BIOS is available in the PCI BIOS Specification. It can be ordered from the PCI Special Interest Group (see <http://www.pcisig.com>).

Demo Program BIOSDemo demonstrates how to use PCI BIOS services.

Function RTT_BIOS_Installed

This function checks whether a PCI BIOS is installed:

```
DWORD RTT_BIOS_Installed(void);
```

If a PCI BIOS is present, a non-zero value is returned.

Function RTT_BIOS_FindDevice

This function returns the location of PCI devices that have a specific device ID and vendor ID. Given a vendor ID, device ID and an index (N), the function returns the bus number, device number and function number of the Nth device function whose vendor ID and device ID match the input parameters.

```
int RTT_BIOS_FindDevice(WORD Vendor,
                       WORD DeviceID,
                       int Index,
                       BYTE * Bus,
                       BYTE * DeviceFunc);
```

All devices having the same vendor ID and device ID can be found by making successive calls to this function starting with Index set to zero and incrementing it until the return code is RTT_BIOS_DEVICE_NOT_FOUND. If the function succeeds, the function returns the device's bus in *Bus, its device number in bits 7 .. 3 of *DeviceFunc, and its function number in bits 2 .. 0 of *DeviceFunc.

Function RTT_BIOS_FindClassCode

This function returns the location of PCI devices that have a specific class code. Given a class code and an Index (N), the function returns the bus number, device number, and function number of the Nth device/function whose class code matches the input parameters.

```
int RTT_BIOS_FindClassCode(WORD ClassCode,
                          int Index,
                          BYTE * Bus,
                          BYTE * DeviceFunc);
```

All devices having the same class code can be found by making successive calls to this function starting with Index set to zero and incrementing it until the return code is DEVICE_NOT_FOUND. For return values, see function RTT_BIOS_FindDevice.

Function RTT_BIOS_GetInterruptRouting

This function returns the PCI interrupt routing options available on the system board and information about interrupts exclusively assigned to PCI devices:

```
int RTT_BIOS_GetInterruptRouting(RTT_BIOS_IRQ_ROUTING * RoutingInfo,
                                WORD * Entries);
```

Parameter `RoutingInfo` must point to an array of `RTT_BIOS_IRQ_ROUTING` structures to receive the returned information. Parameter `Entries` must be initialized with the number of elements in this array. When the function returns successfully, `*Entries` is changed to the actual number of entries returned. If the function fails with `RTT_BIOS_BUFFER_TOO_SMALL`, `*Entries` is set to the number of entries required to satisfy the call.

More information about this function is available in the PCI BIOS Specification.

Function `RTT_BIOS_SetPCIInt`

This function causes the specified hardware interrupt IRQ to be connected to the specified interrupt pin of a PCI device:

```
int RTT_BIOS_SetPCIInt(BYTE PCIInt, BYTE IRQ, BYTE Bus, BYTE DeviceFunc);
```

Parameter `PCIInt` must be in the range 0 .. 3 for `INTA#` .. `INTD#`. Parameter `IRQ` is an IRQ number in the range 0 .. 15.

More information about this function is available in the PCI BIOS Specification.

Function `RTT_BIOS_GenSpecialCycle`

This function will broadcast a PCI special cycle to all devices on the specified PCI bus:

```
int RTT_BIOS_GenSpecialCycle(BYTE Bus, DWORD Data);
```

More information about this function is available in the PCI BIOS Specification.

Function `RTT_BIOS_ReadConfigData`

This function allows reading individual bytes, words, or dwords from the configuration space of a specific device:

```
int RTT_BIOS_ReadConfigData(BYTE Bus,
                             BYTE DeviceFunc,
                             int Register,
                             int Width,
                             void * Value);
```

Parameter `Width` may be 1, 2, or 4 for byte, word or dword access. Parameter `Register` must be divisible by `Width` and cannot be larger than 255. If the function succeeds, the read data is returned at `*Value`.

Function `RTT_BIOS_WriteConfigData`

This function allows writing individual bytes, words, or dwords into the configuration space of a specific device:

```
int RTT_BIOS_WriteConfigData(BYTE Bus,
                              BYTE DeviceFunc,
                              int Register,
                              int Width,
                              DWORD Value);
```

Parameter `Width` may be 1, 2, or 4 for byte, word or dword access. Parameter `Register` must be divisible by `Width` and cannot be larger than 255.

Plug-and-Play BIOS Functions

Plug-and-play I/O cards may require configuration at run-time, or software may need to query configuration information about such cards. 16-bit protected mode PnP BIOS services can be used for this purpose.

All PnP BIOS functions made available by RTTarget-32 are declared in header file `RTTPNP.H`. For Pascal, the driver's API is defined in unit `RTTPNP.PAS`.

Important (multithreaded programs only): if an RTKernel-32 program uses any PnP BIOS functions, the CPU driver `CPU386` must be used instead of the default driver `CPU386F`. This is required because the PnP BIOS reloads segment registers with selectors referring to 16-bit segments. This is not supported by the default `CPU386F` driver.

Details about the Plug-and-Play BIOS are available in the Plug-and-Play BIOS Specification freely available from Compaq Computer Corporation, Phoenix Technologies Ltd., or Intel Corporation, as well as Microsoft's MSDN Library.

Function RTT_PNP_Installed

This function checks whether a PnP BIOS is installed:

```
int RTT_PNP_Installed(void);
```

This function will return a non-zero value if the BIOS has protected mode PnP BIOS support, and zero otherwise.

Function RTT_PNP_CallPnPBIOs

This function performs a PnP BIOS call:

```
int RTT_PNP_CallPnPBIOs(void * Data, WORD DataSize, WORD Parameters[]);
```

Parameter Data can point to a data buffer through which the BIOS and the application can exchange information. All data to which pointers are passed to the BIOS must be located in this data area. Parameter DataSize specifies the size of this buffer. RTTarget-32 will create an appropriate 16-bit segment spanning the data to be passed to the 16-bit BIOS.

The array Parameters must contain the WORD parameters to be passed to the BIOS in the order they appear in the BIOS function declaration. 16-bit FAR pointers must be expanded to offset:segment values according to the following rules: The segment to parameters in *Data is RTT_PNP_STUB16DS. Their offset is the offset relative to *Data. Parameter BiosSelector is RTT_PNP_BIOS16DS.

An example to call PnP BIOS function 0 (GetSystemNodes) follows. The PnP BIOS Specification declares function 0 as:

```
int FAR (*entryPoint)(Function, NumNodes, NodeSize, BiosSelector);

int Function;                // PnP BIOS Function 0
unsigned char FAR *NumNodes;  // Number of nodes the BIOS will return
unsigned int FAR *NodeSize;   // Size of the largest device node
unsigned int BiosSelector;    // PnP BIOS readable/writable selector
```

Thus, a total of 6 WORD parameters are required (each 16-bit far pointer needs two 16-bit values). The parameters need to be filled as follows:

```
P[0] = 0;                      // function number is zero
P[1] = RTT_STRUCT_OFS(WorkSpace, Nodes); // offset Nodes
P[2] = RTT_PNP_STUB16DS;       // seg Nodes
P[3] = RTT_STRUCT_OFS(WorkSpace, Size);  // offset Size
P[4] = RTT_PNP_STUB16DS;       // seg Size
P[5] = RTT_PNP_BIOS16DS;       // BIOS data selector
```

where WorkSpace is defined as:

```
struct {
    WORD Nodes;
    WORD Size;
} WorkSpace;
```

The BIOS can then be called with:

```
Result = RTT_PNP_CallPnPBIOs(&WorkSpace, sizeof(WorkSpace), P)
```

Demo program BIOSDemo demonstrates how to call the PnP BIOS and contains additional examples of using function RTT_PNP_CallPnPBIOs.

PC Cards (PCMCIA)

RTTarget-32 supports one or two Intel 82365SL compatible PC card controllers with one to four PCMCIA card slots. RTTarget-32 has functions to detect card status changes (e.g., card removal or insertion), identify a card type, and to map its resources into the computer's address space. Currently, only 16-bit PC Cards are supported (no 32-bit Cardbus Cards).

Nevertheless, RTTarget-32 is not a plug-and-play operating system. RTTarget-32 has no built-in support for resource allocation and management. Applications must ensure that PC Cards are mapped in such a way that no resource conflicts occur.

Applications to support PCMCIA PC Cards must perform the following actions:

- React to card insertion/removal events passed to the application through a callback of the PCMCIA driver.
- Initialize and power-up a card.
- Read information from the card's configuration space memory to identify the card type.
- Allocate I/O port addresses or memory addresses as well as an IRQ to the card.
- Configure the device driver to handle the card.

RTTarget-32 demo programs PCCard and PCCardMT (for multithreaded applications) show how to do this. In multithreaded programs, it is strongly recommended to set up a separate thread to perform these steps. Single threaded programs must poll for and react to insertion/removal events periodically.

The PCMCIA driver is **not** reentrant. Thus, the steps outlined above may be performed only by a single thread and they must not be performed by interrupt handlers. In particular, the PCMCIA driver's callback to signal insertion/removal events is called by an interrupt handler and thus cannot perform card configuration. Rather, the callback handler must record the event and inform a separate thread or the main program's event loop to process the event at a later time.

The PCMCIA driver needs some address space to map a card's Configuration Information Space (CIS) into the computer's address space. If paging is enabled, such a region with a size of 4k can be allocated automatically by the driver. However, it is frequently desirable to assign a fixed address. This can be achieved by allocating a suitable region named PCMCIA in the RTLoc configuration file. Example:

```
Region PCMCIA C8000h 4k Device ReadWrite
```

Such a PCMCIA region must be located below 16M and must have at least 4k size. You can choose any address not used for any other purpose (in particular, it should be an address with no associated RAM, ROM, or other device).

If the PCMCIA controller is a PCI cardbus controller, the RTTarget-32 PCMCIA driver also requires memory to map the controller's registers into the computer's address space. If paging is used, such a region with 4k size per PCMCIA slot can be allocated automatically. However, it is frequently desirable to assign a fixed address. This can be achieved by allocating a suitable region named CARDBUS in the RTLoc configuration file. Example:

```
Region CARDBUS 3G 8k Device ReadWrite
```

Such a CARDBUS region must have a size of at least 4k (8k for supporting two sockets). You can choose any address not used for any other purpose (in particular, it should be an address with no associated RAM, ROM, or other device).

All data structures, constants, and functions made available by the PCMCIA driver are declared in header file RTPCMCIA.H. For Pascal, the driver's API is defined in unit RTPCMCIA.PAS.

More information about PC Cards and PCMCIA are available in the PC Card Standard available from the *Personal Computer Memory Card International Association*, 2635 North First Street, Suite 209, San Jose, CA 95134, USA, <http://www.pc-card.com>.

Function RTPCInit

Function RTPCInit initializes the PCMCIA driver and determines the type of controller installed. It must be called exactly once before the driver can be used:

```
int RTPCInit(int IRQ, int IOBase, int Sockets, RTPCCardEventHandler Handler);
```

Parameter IRQ specifies the IRQ the driver should use to signal insertion/removal events. Any value except 0 can be specified. A value typically used on many PCs is 5.

Parameter IOBase defines the first of two I/O port addresses used by the PCMCIA controller. If 0 is specified, the default value 3E0h is assumed.

Parameter `Sockets` specifies the number of sockets supported by the controller. This parameter may be 1, 2, or 4.

Parameter `Handler` is the address of a callback handler function which the PCMCIA driver will call whenever a card status change event is detected. The handler must be defined as:

```
void RTTAPI RTPCCardEventHandler(int Socket, BYTE Event);
```

When an event is detected, the handler will be called with parameter `Socket` containing the zero-based number of the socket that has caused the event. Parameter `Event` can have any of the following bits set, possibly several simultaneously:

`RTPC_BATTERY_DEAD` The battery dead signal of the card has changed state.

`RTPC_BATTERY_WARN` The battery low warning state of the card has changed.

`RTPC_READY` The card's ready signal has changed state.

`RTPC_CARD_DETECT` A card has been inserted or removed.

The handler is called from an interrupt handler with interrupts enabled. The handler may not call any PCMCIA driver functions. Instead, it should record the event or signal a thread to enable the event to be processed later.

Applications that do not need to process events may set the `Handler` parameter to `NULL`.

If this function succeeds, a value identifying the PCMCIA controller found is returned. If the return value is 0, no controller was detected.

Function RTPCShutdown

This function may be used to unmap any currently active cards and to uninstall the PCMCIA driver's interrupt handler:

```
void RTPCShutdown(void);
```

Function RTPCCardPresent

`RTPCCardPresent` may be called to check if a card is fully inserted in a card slot:

```
int RTPCCardPresent(int Socket);
```

The function returns a non-zero value if the specified socket (zero-based) holds a card.

Function RTPCPowerUp

Function `RTPCPowerUp` resets a card and applies power:

```
int RTPCPowerUp(int Socket);
```

If the card has already been powered up, this function returns `TRUE` immediately without accessing the card. Otherwise, power is applied and the function waits until the card signals that it is ready. The complete power-up sequence can take up to about one second, depending on the type of card.

If the card signals ready within one second, the return value is `TRUE`; otherwise, it is `FALSE`.

Function RTPCGetFunctionID

Function `RTPCGetFunctionID` reads the function identifier from the card's CIS:

```
int RTPCGetFunctionID(int Socket);
```

The card must have been powered up successfully before this function can be called. The following function IDs can be returned:

`RTPC_FUNCID_MEMORY` The card is a memory card.

`RTPC_FUNCID_SERPORT` The card is a serial port or modem.

`RTPC_FUNCID_PARPORT` The card is a parallel port.

`RTPC_FUNCID_FDISK` The card is a hard disk (but not necessarily ATA).

`RTPC_FUNCID_VIDEO` The card is a video card.

RTPC_FUNCID_NETWORK The card is a network adapter.

In case of an error, the following values can be returned:

RTPC_BAD_SOCKET The given socket number is invalid. Parameter **Socket** is a zero-based index of the socket to be addressed.

RTPC_NO_CARD The card has been removed.

RTPC_NO_MORE_ITEMS The card's information space does not contain a function ID tuple. The card is probably defect.

Function RTPCGetFirstTuple

RTPCGetFirstTuple searches a data tuple in a card's Configuration Information Space (CIS):

```
int RTPCGetFirstTuple(int          Socket,
                     BYTE          DesiredTuple,
                     RTPCTupleInfo * Handle,
                     BYTE          * Tuple);
```

Parameter **DesiredTuple** is the tuple identifier to look for. If this parameter is set to **RTPC_CISTPL_ANYTUPLE** (FFh), the first tuple of the CIS will be returned.

Parameter **Handle** must point to a variable of type **RTPCTupleInfo**. The function stores housekeeping information here. Do not access any data stored here.

If the function succeeds, the tuple found is stored at ***Tuple** and the function return value is **RTPC_SUCCESS**. If it fails, the return value is one of the error codes **RTPC_BAD_SOCKET**, **RTPC_NO_CARD**, **RTPC_NO_MORE_ITEMS** (see previous section for details).

Function RTPCGetNextTuple

RTPCGetNextTuple searches subsequent data tuples in a card's CIS. You must call **RTPCGetFirstTuple** before this function can be used:

```
int RTPCGetNextTuple(int          Socket,
                     BYTE          DesiredTuple,
                     RTPCTupleInfo * Handle,
                     BYTE          * Tuple);
```

Parameter **DesiredTuple** is the tuple identifier to look for. If this parameter is set to **RTPC_CISTPL_ANYTUPLE** (0xFF), the next tuple in the CIS will be returned.

Parameter **Handle** must point to a variable or type **RTPCTupleInfo** which must have been initialized by a previous call to **RTPCGetFirstTuple** or **RTPCGetNextTuple**.

If the function succeeds, the tuple found is stored at ***Tuple** and the function return value is **RTPC_SUCCESS**. If it fails, the return value is one of the error codes **RTPC_BAD_SOCKET**, **RTPC_NO_CARD**, **RTPC_NO_MORE_ITEMS** (see previous sections for details).

Function RTPCGetTupleData

RTPCGetTupleData retrieves the data of a tuple previously located with **RTPCGetFirstTuple** or **RTPCGetNextTuple**:

```
int RTPCGetTupleData (int          Socket,
                     RTPCTupleInfo * Handle,
                     void          * Data,
                     int           MaxDataLen,
                     int           * Len);
```

Parameter **Handle** must point to an **RTPCTupleInfo** structure previously initialized by a successful call to **RTPCGetFirstTuple** or **RTPCGetNextTuple**. Parameter **Data** points to a data buffer to receive the tuple. **MaxDataLen** specifies the length of the supplied buffer. If the function succeeds, the length of the tuple is returned in ***Len** and the function return value is **RTPC_SUCCESS**. If it fails, the return value is one of the error codes **RTPC_BAD_SOCKET**, **RTPC_NO_CARD**, **RTPC_NO_MORE_ITEMS** (see previous sections for details).

More information about the structure of CIS tuples is available in the PCMCIA Standard.

Function RTPCSetConfigRegister

Function RTPCSetConfigRegister sets a register in a card's CIS to a specific value:

```
int RTPCSetConfigRegister(int Socket, int Register, BYTE Value);
```

At least the Option Config Register (register 0) must be set before a card can be used. Standard CIS configuration registers (defined in RTPCMCIA.H) are:

```
#define RTPC_CFGREG_OPTION      0
#define RTPC_CFGREG_STATUS     1
#define RTPC_CFGREG_PIN_REPLACE 2
#define RTPC_CFGREG_SOCKET_COPY 3
```

Complex cards may define additional registers.

This function must parse the CIS to find the register. If parsing the CIS fails, the function returns one of the error codes RTPC_BAD_SOCKET, RTPC_NO_CARD, RTPC_NO_MORE_ITEMS. If the function succeeds, RTPC_SUCCESS is returned.

Function RTPCUnmapCIS

When an application has finished analyzing the CIS, the CIS memory window should be unmapped:

```
void RTPCUnmapCIS(int Socket);
```

Functions RTPCGetFirstTuple, RTPCGetNextTuple, RTPCGetTupleData, and RTPCSetConfigRegister will map the CIS into the computer's address space as needed, so there is no call to explicitly map the CIS.

When the CIS of the specified socket is not currently mapped, this function has no effect.

Function RTPCMapMemoryWindow

Function RTPCMapMemoryWindow maps a memory region of the PC card into the computer's address space:

```
void * RTPCMapMemoryWindow(int Socket,
                           int Window,
                           void * HostAddress,
                           DWORD CardAddress,
                           DWORD Size);
```

Parameter Window must be in the range 0 to 3. Although PCMCIA controllers support five memory windows, the last window is reserved by the driver to map the card's CIS. Parameter HostAddress points to the address of the window in the computer's address space. The application must initialize this value to a suitable location (i.e., an address not currently in use by any installed memory or device). If this value is initialized to NULL and paging is enabled, function RTPCMapMemoryWindow will allocate a suitable region of uncommitted memory. Parameter CardAddress is the address of the memory to map in the PC card's address space. Parameter Size specifies the window's size in bytes. HostAddress, CardAddress, and Size must be multiples of 4096. HostAddress + Size and CardAddress + Size must be below 64M as the PCMCIA controller only supports 26-bit addresses.

The function returns the address of the mapped window, or NULL if the function has failed.

Function RTPCMapIOWindow

Function RTPCMapIOWindow maps I/O address space of the PC card into the computer's I/O address space:

```
void RTPCMapIOWindow(int Socket,
                    int Window,
                    WORD HostAddress,
                    WORD CardAddress,
                    WORD Size,
                    BYTE Flags);
```

Parameter Window must be 0 or 1. Parameter HostAddress is the address of the I/O window in the computer's I/O address space. Parameter CardAddress specifies the address in the PC Card's I/O address space. Parameter Size specifies the window's size. Parameter Flags specifies additional options for the I/O window and can be a combination of:

RTPC_IO_WINDOW_8BIT 8-bit port accesses are performed in this I/O range (default).

RTPC_IO_WINDOW_16BIT 16-bit port accesses are performed in this I/O range.

RTPC_IO_WINDOW_AUTO Automatically detect 8- or 16-bit I/O access.

RTPC_IO_WINDOW_TIMER0 Use timing register 0 (default).

RTPC_IO_WINDOW_TIMER1 Use timing register 1.

Flags value 0 is equivalent to (RTPC_IO_WINDOW_8BIT | RTPC_IO_WINDOW_TIMER0).

Function RTPCEnableIRQ

Function RTPCEnableIRQ programs the PCMCIA controller to route interrupts of an inserted card to a specific ISA interrupt:

```
void RTPCEnableIRQ(int Socket, int IRQ);
```

Parameter IRQ specifies the interrupt to use. Only the following values are supported: 3, 4, 5, 7, 9, 10, 11, 12, 14, and 15.

Function RTPCUnmapSocket

Function RTPCUnmapSocket unmaps any memory windows, I/O windows, the CIS, and interrupts assigned to a socket and disables the card's power supply:

```
void RTPCUnmapSocket(int Socket);
```

This function should be called when the card is removed from a socket or when an application decides not to support an inserted card.

Function RTPCIsATA

This function analyzes the CIS of an inserted card and returns 1 if it is an ATA or CompactFlash disk card:

```
int RTPCIsATA(int Socket);
```

If this function returns 1, RTPCMapATA can be called to initialize the card and RTFiles-32's IDE device driver can work with the card.

If the card is not an ATA or CompactFlash disk, this function returns 0.

Function RTPCIsUART

This function analyzes the CIS of an inserted card and returns 1 if it is a serial port or modem:

```
int RTPCIsUART(int Socket);
```

If this function returns 1, RTPCMapUART can be called to initialize the card.

If the card is not a serial port or modem, this function returns 0.

Function RTPCMapUART

This function maps and enables all required resources to use a serial port PCMCIA card:

```
void RTPCMapUART(int Socket, WORD IOBase, int IRQ);
```

This function should be called after an application has verified that a serial port card type has been inserted.

Parameter IOBase specifies the I/O address at which the serial port should appear. Parameter IRQ is the interrupt to be used by the port to signal interrupts. For example, if you want to use the port as COM2, use:

```
RTPCMapUART(0x2F8, 3);
```


Function RTPCMapUART's source code is:

```
void RTPCMapUART(int Socket, WORD IOBase, int IRQ)
{
    RTPCSetConfigRegister(Socket, RTPC_CFGREG_OPTION, 0x41);
    RTPCUnmapCIS(Socket);
    RTPCMapIOWindow(Socket, 0, IOBase, 0x3F8, 8, RTPC_IO_WINDOW_8BIT |
                                                    RTPC_IO_WINDOW_TIMER1);
    RTPCEnableIRQ(Socket, IRQ);
}
```

Function RTPCMapATA

This function maps and enables all required resources to use an ATA disk card:

```
void RTPCMapATA(int Socket, int DriveNumber, int IRQ);
```

This function should be called after an application has verified that an ATA disk card has been inserted.

Parameter DriveNumber specifies the RTFiles-32 drive number for the drive. Values 0..7 are supported. (DriveNumber / 2) specifies the IDE channel/controller and (DriveNumber % 2) specifies whether the drive should be a master (0) or slave (1). For example, DriveNumber 3 would configure an ATA card to be a slave drive of the second IDE channel.

The application must ensure that the requested resources are free. For example, a PCMCIA disk can only be configured to be on the primary IDE channel if no IDE host adapter is in the system (e.g., on the motherboard). In addition, it must be observed that not all PCMCIA ATA disks support being configured as a slave drive. Thus, it is recommended to use DriveNumbers 0 and 2 for embedded targets without IDE controller or DriveNumbers 2 and 4 for targets with a single IDE controller.

Parameter IRQ is the interrupt on which the disk should signal events. The default values are 14, 15, 11, and 10 for up to four IDE controllers. If the default IRQ is not available (either in use by some other peripheral or not supported by the PCMCIA controller), a custom value must be selected.

Important: If a custom IRQ value is used, you must also inform RTFiles-32's IDE driver of the IRQ used!

The following example shows how to configure a PCMCIA ATA disk as the master on the secondary IDE channel with a custom IRQ value. The PCMCIA ATA disk has drive number 2 and can coexist with up to two IDE disks on the primary IDE channel installed on the target's motherboard.

```
#include <rttarget.h>
#include <rtpcmcia.h>
#include <rtfiles.h>

#define PCMCIA_ATA_IDX 2 // entry #2 in device list
#define PCMCIA_ATA_IRQ 10

static RTFDrvIDEData IDEDrive0Data = {0}; // master disk on moboard
static RTFDrvIDEData IDEDrive1Data = {0}; // slave disk on moboard
static RTFDrvIDEData IDEDrive2Data = {0, 0, 0, 0, 0, 0, 0, PCMCIA_ATA_IRQ};

RTFDevice RTFDeviceList[] = {
    { RTF_DEVICE_FDISK, 0, 0, &RTFDrvIDE, &IDEDrive0Data },
    { RTF_DEVICE_FDISK, 1, 0, &RTFDrvIDE, &IDEDrive1Data },
    { RTF_DEVICE_FDISK, 2, RTF_DEVICE_REMOVABLE |
                          RTF_DEVICE_NO_MEDIA |
                          RTF_DEVICE_NEW_LOCK, &RTFDrvIDE, &IDEDrive2Data },
    { 0 }
};

void ConfigureDisk(int Socket)
{
    if (RTPCIsATA(Socket))
    {
        RTPCMapATA(Socket,
                    RTFDeviceList[PCMCIA_ATA_IDX].DeviceNumber,
                    PCMCIA_ATA_IRQ);
    }
}
```

```
        // tell RTFiles-32 that this disk is now available
        RTFRawSetMedia(PCMCIA_ATA_IDX, 1);
    }
}
```

Function RTPCMapATA's source code is:

```
void RTPCMapATA(int Socket, int DriveNumber, int IRQ)
{
    static WORD DefaultPortBases[] = { 0x1F0, 0x170, 0x0F0, 0x070 };
    static BYTE DefaultIRQs[]      = { 14, 15, 11, 10 };

    int Controller = DriveNumber / 2;
    int MasterSlave = DriveNumber % 2;

    // Set Socket_Copy Register (3)
    // Set to Socket_Copy register to Copy << 4 | Socket
    RTPCSetConfigRegister(Socket,
                          RTPC_CFGREG_SOCKET_COPY,
                          (MasterSlave << 4) | Socket);

    // Set Option Config Register (0)
    RTPCSetConfigRegister(Socket, RTPC_CFGREG_OPTION, 0x40 | 2); // config 2
    RTPCUnmapCIS(Socket);
    RTPCMapIOWindow(Socket, 0,
                    DefaultPortBases[Controller],
                    DefaultPortBases[0],
                    8, RTPC_IO_WINDOW_AUTO);
    RTPCMapIOWindow(Socket, 1,
                    DefaultPortBases[Controller] + 0x206,
                    DefaultPortBases[0] + 0x206,
                    2, RTPC_IO_WINDOW_8BIT);

    if ((IRQ == -1) || (IRQ == 0))
        RTPCEnableIRQ(Socket, DefaultIRQs[Controller]);
    else
        RTPCEnableIRQ(Socket, IRQ);
}
```

DOS Emulation

RTTarget-32 emulates only two DOS functions: Interrupt 21h, functions 25h and 35h (Set and Get Interrupt Vector). They are provided to be able to support third-party libraries that may install interrupt handlers using these functions.

Hardware interrupt vectors are remapped to reflect the mapping of IRQ <=> vector under RTTarget-32, which is different than under DOS. If the application attempts to install or query a handler on vectors 08h .. 0Fh, vectors 40h .. 47h are used. Likewise, vectors 70h .. 77h are mapped to 48h .. 4Fh.

DPMI Emulation

RTTarget-32 emulates six DPMI functions: Interrupt 31h, functions 0202h, 0203h, 0204h, 0205h, 0210h, and 0212h. They are provided in order to support third-party libraries that may install interrupt or exception handlers using these functions. Vector remapping is done for hardware interrupt handlers just as for DOS emulation (see above).

Win32 Emulation

To allow the compiler supplied run-time systems to be used under RTTarget-32, all Win32 functions referenced must be supplied by RTTarget-32. Usually, these functions reside in KERNEL32.DLL, USER32.DLL, GDI32.DLL, and some other system DLLs. RTTarget-32 supplies these functions in its library RTT32.LIB or in the RTTarget-32 System DLL RTT32DLL.DLL.

For all Win32 functions expecting a character string, only the ASCII versions are supplied. RTTarget-32 does not support Win32 emulation for Unicode programs. Any security attribute parameters are ignored by RTTarget-32. Application programs may specify NULL (this is also supported by Win32).

The different areas of the Win32 API covered by RTTarget-32 are described in the following sections, followed by a list of all supported Win32 functions.

Win32 Handles

RTTarget-32's Win32 emulation includes functions to manage Win32 handles. Various Win32 functions can allocate handles (e.g., CreateFile or CreateHeap). CloseHandle is normally used to close handles and objects associated with a handle.

Handles are indirect references to an object. Several different handles can refer to a single object (for example, when a handle was duplicated with function DuplicateHandle). RTTarget-32's handle manager maintains tables to keep track of the different handles in use. However, these tables have a fixed size and thus limit the number of handles that can be open at any one time. You can adjust the number of available handles by including the following lines in the module which links library RTT32.LIB:

```
#include <rttarget.h>

#define MAXHANDLES 64
#define MAXOBJECTS 64
#define MAXTYPES 32

RTW32Handle RTHandleTable[MAXHANDLES] = {{0}};
int RTHandleCount = MAXHANDLES;

RTW32Object RTOBJECTTable[MAXOBJECTS] = {{0}};
int RTOBJECTCount = MAXOBJECTS;

RTW32Types RTTypeTable[MAXTYPES] = {{0}};
int RTTypeCount = MAXTYPES;
```

Constants MAXHANDLES, MAXOBJECTS, and MAXTYPES define the sizes of RTTarget-32's handle manager. The default values are given above. MAXHANDLE defines the maximum number of handles that can be open at any one time. Please consider that several predefined handles are opened automatically (e.g., for stdin and stdout). MAXOBJECTS defines how many objects can be referenced by handles. MAXTYPES specifies how many different object types (e.g., console file, RAM files, threads, events, etc.) will be supported.

Function RTHandleInfo

Function RTHandleInfo can calculate the degree to which the handle tables are being used:

```
void RTHandleInfo(int * FreeHandles, int * FreeObjects, int * FreeTypes);
```

All three parameters must point to integers which will receive the number of available handles, objects, and types, respectively.

Use this function to analyze the handle requirements of an application at run time and to adjust the tables' sizes appropriately.

Win32 Memory Management

RTTarget-32 defines functions for Win32's virtual memory (VirtualAlloc, VirtualFree), Win32 heaps (HeapCreate, HeapAlloc, etc.), and the local heap (LocalAlloc, etc.).

The Win32 memory management is implemented using one of two available RTTarget-32 memory managers: the fixed memory manager and the virtual memory manager with uncommitted memory support (both are described in section *RTTarget-32's Memory Managers* later in this chapter). Win32 memory management normally requires uncommitted memory support, and many compilers' run-time systems rely on it. Thus, if the fixed memory manager is used, problems can occur if:

- The application uses several different Win32 allocation function groups (for example, Heaps **and** VirtualAlloc).
- Several run-time systems are used (for example, because the main .EXE **and** additional DLLs with run-time systems have been linked).

In such situations, memory may be exhausted quickly with the fixed memory manager due to excessive allocation of uncommitted memory. RTTarget-32's virtual memory should be used in this case.

Win32 File I/O

RTTarget-32 supports RAM files (see Chapter 3, section *File*), LPT files, screen output via stdout and stderr, and keyboard input via stdin through functions such as CreateFile, ReadFile, WriteFile, etc. Screen output can be redirected to the host during software development for targets without a display (see Chapter 7, *Function RTDisplayChar*).

The file searching functions FindFirstFile and FindNextFile are also available and support any RAM file. These functions will also find any DLLs which are included in the application to support programs which first search for DLLs before they load them. However, a DLL which is reported to exist cannot be opened with CreateFile; it can only be loaded with LoadLibrary.

Win32 Console I/O

RTTarget-32 installs its own keyboard interrupt handler on IRQ 1 when any file I/O or console I/O function is first called (usually through the run-time system's startup code). Reading keyboard input will only work if the target hardware has an IBM-PC compatible keyboard controller at the default I/O port addresses. RTTarget-32's keyboard interrupt handler is optimized for low interrupt latency. US, German, and French keyboard layouts are supported. Switching keyboard layout is done using function SetThreadLocale. The default keyboard layout is US. Example:

```
SetThreadLocale(MAKELCID(
    MAKELANGID(LANG_GERMAN, SUBLANG_GERMAN),
    SORT_DEFAULT));
SetThreadLocale(MAKELCID(
    MAKELANGID(LANG_FRENCH, SUBLANG_FRENCH),
    SORT_DEFAULT));
```

Alternatively, the keyboard layout can be defined with the RTTarget-32 system flags (see Chapter 7, section *RTTarget-32 Flags*). The keyboard driver also supports switching languages using hotkeys left Ctrl-Alt-F1 (US), left Ctrl-Alt-F2 (German), or left Ctrl-Alt-F3 (French). Installing alternate keyboard translation tables is supported through function RTSetKeyboardTables (see Chapter 7, *Function RTSetKeyboardTables*).

The default input code page used is 437 (OEM). Function RTSetCodepageTranslation (see Chapter 7, *Function RTSetCodepageTranslation*) can be used to change the code page.

RTTarget-32's keyboard driver is linked to the application by default, but it does occupy quite a lot of memory. For targets which do not need a keyboard, it can be eliminated by including the following function into the EXE/DLL which links RTT32.LIB:

```
void RTTAPI RTGetKeyEvents(void) {}
```

Mouse events are only available if the program has explicitly called function RTInitTextMouse() (see section *Function RTInitTextMouse*).

When the program is waiting for keyboard input, RTTarget-32 will display a cursor. However, this feature only works with a standard CRT monochrome or color display adapter. By default, this feature is automatically enabled if a video RAM at address B0000h or B80000h is used. If a video controller incompatible with the IBM-PC is used, but you still want to use a video RAM at those addresses, RTTarget-32 flag RT_CRT_NO_ACCESS must be set either through RTSetFlags or through a global instance of RTTarget32Flags.

Console Input Event Management

RTTarget-32 contains a flexible user event (keyboard and mouse) management. Both the keyboard and mouse drivers are interrupt driven. By default, the interrupt handlers will only retrieve the raw event data from the hardware and place it in a buffer. Each time the application program checks for user input, these buffers are processed and interpreted to build Win32 event structures (structure INPUT_RECORD). Drivers under DOS or Windows work differently: they will perform this processing inside the interrupt handler, which is one of the reasons for their poor interrupt latency.

The advantage of RTTarget-32's delayed event processing is a very low interrupt latency; however, events are not always processed immediately. For example, if the application does not check for user events for a long time, mouse movements on the screen will be erratic. If you prefer smooth mouse movements and can tolerate higher interrupt latencies, you can instruct the drivers to process events within the interrupt handlers (that is, immediately). This is achieved by setting the RTTarget-32 system flags `RT_KEY_BY_INTERRUPT` and/or `RT_MOUSE_BY_INTERRUPT` using function `RTSetFlags`.

The Win32 event reading functions such as `ReadConsoleInput` will call the following RTTarget-32 function to update the event queue:

```
void RTProcessEvents(void);
```

`RTProcessEvents` is actually not required if both drivers run with their respective `RT_KEY_BY_INTERRUPT` and `RT_MOUSE_BY_INTERRUPT` flags set. However, if they are not set, the application can also call `RTProcessEvents` to update the event queue and process all pending keyboard and mouse events.

RTTarget-32 defines a few event hooks which are called on various events:

```
extern void (RTTAPI * RTGetMouseEvents)(void);
extern void (RTTAPI * RTSignalEvent)(void);
extern void (RTTAPI * RTWaitEvent)(void);
extern void (RTTAPI * RTNewEvents)(void);
```

*`RTGetMouseEvents` is called by `RTProcessEvents` to get mouse events. *`RTSignalEvent` is called by interrupt handlers of the drivers to signal that `RTProcessEvent` should be called because new unprocessed event are now available. *`RTWaitEvent` is called by `ReadConsoleInput` when no events are available. *`RTNewEvents` is called by `WriteConsoleInput` when new `INPUT_RECORD` events are placed in the event queue.

All hooks given above are initialized to point to dummy (do nothing) routines. The mouse driver will install a routine on `RTGetMouseEvents` which will process any pending mouse events and write them to the user event queue using `WriteConsoleInput`.

Win32 Time Management

Function `GetTickCount` is fully supported and returns the number of milliseconds elapsed since `GetTickCount` was first called. RTTarget-32 expects an interrupt to occur at 18.2 Hertz on IRQ 0 for this feature to work. The RTTarget-32 boot code usually initializes the timer hardware to generate such a timer interrupt. The interrupt handler is installed by function `GetTickCount` the first time it is called.

If a timer frequency other than 18.2 Hz is used, the calibration constant `RTTickFactor` must be assigned the actual number of milliseconds per timer interrupt multiplied by 65536 before `GetTickCount` is called the first time. Example (for a 10ms timer interrupt):

```
RTTickFactor = 10 * 65536;
```

RTTarget-32 also supports the date and time functions such as `Get/SetSystemTime`. `Get/SetLocalTime` are identical to `Get/SetSystemTime`. However, after program reset, the Win32 system time must be initialized at least once. This is achieved with Win32 function `SetSystemTime`. For targets equipped with an MC146818A Real-Time Clock (which is the case for most PC-compatible systems), RTTarget-32's function `RTCMOSSetSystemTime()` can be used for this purpose.

Please note that `RTKernel-32` overrides RTTarget-32's function `GetTickCount`.

Win32 DLLs

RTTarget-32 supports `LoadLibrary`, `FreeLibrary`, `GetModuleHandle`, `GetModuleFileName`, and `GetProcAddress` for any modules supplied in the locate process through the `RTLLOC` DLL directive or for DLLs loaded through a file system (see Chapter 9 for details).

`GetModuleHandle` will return a handle for the module containing `RTT32.LIB` for module names `KERNEL32.DLL` and `USER32.DLL` (except if there are modules with these names present on the target).

`GetProcAddress` only works if the `.edata` section of the module searched for a function has been mapped.

The entrypoints of all DLLs which are statically referenced are called before the entrypoint of the main program is called. The entrypoints of other DLLs are called by the first call to LoadLibrary and again when a matching FreeLibrary call occurs.

Care must be taken with repeated calls to LoadLibrary/FreeLibrary for DLLs linked into the program image. Many run-time systems of DLLs will not release their resources (for example, allocated memory) when FreeLibrary unloads the DLL. Repeated loading/unloading of such DLLs can quickly exhaust memory. Some other DLLs cannot be initialized twice, because RTTarget-32 cannot bring their initialized data into the state it had at boot time. Thus, it is recommended to load all required DLLs only once without unloading them (or unload them only once at program termination).

More information about using DLLs is available in Chapter 9, *Using DLLs through RTLoc and Loading DLLs through a File System*.

Win32 Exception Handling

Win32's structured exception handling (and C++'s or Object Pascal's exception handling which relies on the operating system's exception handling) are fully supported. However, by default, CPU exceptions do not generate structured exceptions. If this is required, call RTRaiseCPUException.

Win32 Thread Local Storage (TLS)

RTTarget-32 will correctly set up any TLS data found in the main application .EXE file (not, however, in any DLLs). In addition, Win32 functions which operate on TLS data are supported (e.g., TlsAlloc, TlsSetValue, Get/SetLastError, etc.).

Please note that RTTarget-32 does not contain a scheduler for multithreading. However, RTTarget-32's TLS management is fully compatible with RTKernel-32, which can be used to implement separate TLS data and `__thread` variables for each thread.

When using DLLs in a multithread environment, it must be considered that the DLL_THREAD_ATTACH and DLL_THREAD_DETACH events are **not** passed to the DllEntryPoint functions. However, you can explicitly perform these calls with function RTDLLThreadEvent.

Win32 API Function Cross Reference

The following table summarizes the functions available and the level of functionality they offer. The letters preceding each function name have the following meaning:

- F Fully implemented. The function behaves the same as under Win32.
- P Partial. A subset of the functionality is available.
- D Dummy. The function does nothing and returns immediately.
- A Abort. Calling the function aborts the program.

Please note that RTTarget-32's Win32 emulation covers most file I/O functions. However, the actual degree of support will depend on the file systems drivers installed. The list below will indicate the level of support for RTTarget-32's default file system configuration (console files, RAM files, and LPT files). If additional file systems are available (e.g., RTFiles-32), most "dummy" file I/O functions will actually be fully supported.

F	CharNextA	D ⁶	CreateMutexA
F	CharUpperBuffA	D	CreateProcessA
F	CharToOemA	F ⁶	CreateThread
F	CharUpperA	F	DebugBreak
F	CloseHandle	D ⁶	DeleteCriticalSection
P	CompareStringA	D ¹	DeleteFileA
D	CompareStringW	D	DestroyWindow
D ¹	CreateDirectoryA	D ¹	DeviceIOControl
D ⁶	CreateEventA	D	DisableThreadLibraryCalls
P ¹	CreateFileA	F	DosDateTimeToFileTime
D	CreateFileW	F	DuplicateHandle

D ⁶	EnterCriticalSection	F	GetLargestConsoleWindowSize
D	EnumCalendarInfoA	F	GetLastError
D	EnumSystemLocales	D	GetLocaleInfoA
D	EnumThreadWindows	D	GetLocaleInfoW
F	ExitProcess	F	GetLocalTime
A ⁶	ExitThread	F	GetLogicalDrives
F	FatalAppExit	F	GetModuleFileNameA
F	FileTimeToDosDateTime	D	GetModuleFileNameW
F	FileTimeToLocalFileTime	F	GetModuleHandleA
F	FileTimeToSystemTime	F	GetNumberOfConsoleInputEvents
F	FillConsoleOutputAttribute	D	GetNumberOfConsoleMouseButtons
F	FillConsoleOutputCharacterA	D	GetOEMCP
F ¹	FindClose	F ¹²	GetProcAddress
F ¹	FindFirstFileA	F	GetProcessHeap
F ¹	FindNextFileA	F	GetStartupInfoA
F ¹⁵	FindResourceA	F	GetStdHandle
F ¹⁵	FindResourceExA	D	GetStringTypeA
F	FlushConsoleInputBuffer	D	GetStringTypeW
D ¹	FlushFileBuffers	D	GetStringTypeExA
D	FormatMessageA	D	GetSystemDefaultLangID
D	FreeEnvironmentStringsA	D	GetSystemDefaultLCID
D	FreeEnvironmentStringsW	D	GetSystemInfo
F	FreeLibrary	D	GetSystemMetrics
D	FreeResource	F	GetSystemTime
D	GetACP	D ¹	GetTempFileNameA
D	GetActiveWindow	D	GetThreadContext
F	GetCommandLineA	D	GetThreadLocale
D	GetCommandLineW	F ⁶	GetTickCount
F	GetConsoleCursorInfo	P ¹³	GetTimeFormatA
F	GetConsoleMode	D	GetTimeZoneInformation
F	GetConsoleScreenBufferInfo	F	GetUserDefaultLCID
D	GetCPInfo	F	GetVersion
D ¹	GetCurrentDirectoryA	F ²	GetVersionExA
D	GetCurrentProcess	D ¹	GetVolumeInformationA
D	GetCurrentProcessId	F	GlobalAlloc
D ⁶	GetCurrentThread	F	GlobalFree
D ⁶	GetCurrentThreadId	F	GlobalHandle
P ¹³	GetDateFormatA	F	GlobalLock
D ¹	GetDiskFreeSpaceA	F	GlobalMemoryStatus
F	GetDriveTypeA	F	GlobalReAlloc
F	GetEnvironmentStrings	F	GlobalUnlock
D	GetEnvironmentStringsW	F	HeapAlloc
F	GetEnvironmentVariableA	F	HeapCompact
D	GetExitCodeProcess	P ¹⁰	HeapCreate
F ¹	GetFileAttributesA	F ¹⁰	HeapDestroy
D	GetFileAttributesW	F ¹⁰	HeapFree
D	GetShortPathNameA	F ¹⁰	HeapReAlloc
F ¹	GetFileSize	F ¹⁰	HeapSize
F ¹	GetFileTime	F ¹⁰	HeapValidate
D	GetFileTitleA	D ⁶	InitializeCriticalSection
F	GetFileType	F	InterlockedDecrement
F ¹	GetFullPathNameA	F	InterlockedExchange
D	GetKeyboardType	F	InterlockedIncrement

F	IsBadCodePtr	D ¹	SetCurrentDirectoryA
F	IsBadReadPtr	D ¹	SetEndOfFile
F	IsBadWritePtr	D	SetEnvironmentVariableA
D	IsValidCodePage	D	SetEnvironmentVariableW
D	IsValidLocale	D ⁶	SetEvent
D	LCMapStringA	F ¹	SetFileAttributesA
D	LCMapStringW	F ¹	SetFilePointer
D ⁶	LeaveCriticalSection	D ¹	SetFileTime
F	LoadLibraryA	D ³	SetHandleCount
F	LoadLibraryExA	F	SetLastError
F ¹⁵	LoadResource	F	SetLocalTime
F ¹⁵	LoadStringA	F	SetStdHandle
F	LocalAlloc	F	SetSystemTime
F	LocalFileTimeToFileTime	P ¹¹	SetThreadLocale
F	LocalFree	F	SetUnhandledExceptionFilter
F	LocalReAlloc	D ¹	SetVolumeLabelA
D	LockFile	D	SHGetFileInfoA
F ¹⁵	LockResource	F ¹⁵	SizeofResource
F	IstrcmpA	F ⁶	Sleep
F	IstrcmpiA	D	SysAllocStringLen
F	IstrcpyA	D	SysFreeString
F	IstrcpynA	D	SysStringLen
F	IstrlenA	D	SysReAllocStringLen
P ⁵	MessageBoxA	F	SystemTimeToFileTime
D ¹	MoveFileA	F	TerminateProcess
D	MultiByteToWideChar	F ⁷	TlsAlloc
F	OemToCharA	F	TlsFree
F ¹⁴	OutputDebugStringA	F	TlsGetValue
P ⁴	PeekConsoleInputA	F	TlsSetValue
F	RaiseException	D	UnhandledExceptionFilter
F ⁴	ReadConsoleInputA	D	UnhookWindowsHookEx
F ⁴	ReadConsoleInputW	D	UnlockFile
P ¹	ReadFile	D	VariantClear
F	ReadProcessMemory	D	VariantCopy
D	RegCloseKey	D	VariantCopyInd
D	RegOpenKeyA	F	VirtualAlloc
D	RegOpenKeyExA	F ⁸	VirtualFree
D	RegQueryValueExA	F	VirtualProtect
D	ReleaseMutex	P ⁸	VirtualQuery
D ¹	RemoveDirectoryA	D	WaitForSingleObject
D ⁶	ResumeThread	D	WideCharToMultiByte
F	RtlUnwind	F	WriteConsoleA
D	SetConsoleCtrlHandler	F	WriteConsoleW
F	SetConsoleCursorInfo	F	WriteConsoleInputA
F	SetConsoleCursorPosition	F	WriteConsoleOutputA
D	SetConsoleMode	P ¹	WriteFile
D	SetConsoleScreenBufferSize	P ⁹	wsprintfA
D	SetConsoleWindowInfo		

- 1 The level of support for file I/O functions depends on the file system drivers installed. By default, file I/O is supported for the console (CONIN\$ and CONOUT\$), RAM/ROM files, and LPT files. If RTFiles-32 is used, the RAM file system is replaced by a real file system. RTFiles-32 fully implements all Win32 file I/O function, even those marked *D* or *P* above.
- 2 GetVersionExA returns RTTarget-32's version * 100 in dwBuildVersion. dwPlatformId is 0 and szCSDVersion is set to "RTTarget-32". You can use GetVersionExA to check whether the program is running under RTTarget-32 or Win32.
- 3 SetHandleCount is dummy and does not change the number of available handles. To change the number of available Win32 handles, see section *Win32 Handles* in this chapter.
- 4 Only keyboard events are supported by default. Mouse events are supported only if RTTarget-32's mouse driver has been initialized.
- 5 The message is displayed on the screen. The program does not wait for any user intervention but returns constant IDOK immediately.
- 6 These thread API functions are replaced and fully implemented by RTKernel-32.
- 7 Only 16 TLS slots are available.
- 8 VirtualFree is dummy for RTTarget-32's fixed memory manager. VirtualQuery only supports the stack region.
- 9 The format string is copied to the buffer without any formatting.
- 10 If multiple heaps are allocated, they are maintained in the same address space. The maxsize parameter is ignored. HeapDestroy deallocates all of a heap's allocated blocks, but it does not decommit the memory. Such blocks can only be reused by other Win32 Heap functions.
- 11 SetThreadLocale does not affect threads, but the whole program. It can be used to change the keyboard mapping between US (default) and German.
- 12 The target module's .edata section must be present on the target.
- 13 Only the default (English/American) locale is supported. The dwFlags parameter is ignored. Format picture parameter *gg* is not supported.
- 14 The exact action of OutputDebugStringA depends on RTTarget-32 flags RT_DBG_OUT_TO_HOST and RT_DBG_OUT_NONE. See section *Function RTSetFlags* for details.
- 15 The .rsrc section must be located on the target to be able to use resources.

Adding other Win32 Functions

If a program requires a Win32 function (a function defined in one of Windows' system DLLs), and the function is not provided by RTT32.LIB or RTT32DLL.DLL, the function must be supplied by your program.

The following steps are required:

- Create a source file for the new function.
- Declare the function **exactly** as it is defined in the Windows header file.
- Implement the function.
- Implement a call stub (not required for Borland C/C++).

Example:

Suppose you need the function CharUpperA, which is not included in RTTarget-32's library. The following source code could be used to implement it:

```
#define _USER32_           // required by Microsoft C
#define _KERNEL32_        // to prevent importing this function
#include <windows.h>
```

```
WINUSERAPI LPSTR WINAPI CharUpperA(LPSTR lpsz)
{
    LPSTR p = lpsz;
    while (*p)
    {
        if ((*p >= 'a') && (*p <= 'z'))
            *p -= 'a' - 'A';
        p++;
    }
    return lpsz;
}
```

This file should be compiled and linked with your program.

For Watcom and Microsoft C, an assembler call stub is also required:

```
.386
EXTRN _CharUpperA@4:near
PUBLIC __imp__CharUpperA@4
PUBLIC _CharUpperA
_TEXT SEGMENT DWORD USE32 PUBLIC 'CODE'
ASSUME CS:_TEXT, DS:_TEXT
__imp__CharUpperA@4 DD offset _CharUpperA@4
_CharUpperA: JMP _CharUpperA@4
_TEXT ENDS
END
```

The numeric value following the at sign (@) represents the number of bytes the function expects as parameters. The value is decimal and is usually four times the number of parameters. Function CharUpperA requires just one parameter, so 4 bytes are pushed onto the stack.

The call stub must be assembled and also linked to the program.

As an alternative to a call stub, it is also possible to use RTLoc's Link command. In this case, the import table's call stub is used.

RTTarget-32's Memory Managers

RTTarget-32 provides two different memory managers with different properties. Usually, RTTarget-32 will automatically select the memory manager to be used. Alternatively, you can use function RTSetFlags to force the use of a specific memory manager. By default, the Virtual Memory Manager is used only if the application runs with paging enabled and uses DLLs.

The reason for two different memory managers is Win32's requirement for uncommitted memory support, which cannot be implemented without paging. Uncommitted memory is allocated address space with no associated physical memory. A typical C/C++ or Pascal run-time system for Win32 implements its heap by allocating large regions of uncommitted memory which is then committed in smaller chunks as memory is allocated by the application. The allocation of uncommitted memory is typically performed even when no or only very little heap space is required by the application.

The following sections describe the differences between the two memory management strategies.

Fixed Memory Manager

The fixed memory manager uses exactly the virtual memory assigned to the heap by the locator. The .LOC file can be inspected to see in which single consecutive address range the heap is located. The fixed memory manager will not attempt to remap pages of memory. Thus, it is able to run without paging.

Reserving uncommitted address space starts at the low end of the heap area. Reserving will succeed even if the requested address range exceeds the available physical heap area. Committing memory will, of course, fail if the requested address exceeds the available heap area.

Example 1:

RTLloc has allocated a heap area of 2M size starting at address 1M. The program's run-time system initially reserves 4M of address space. Although 4M are not available, RTTarget-32 will allow this and return address 1M (start of the heap area) to the run-time system. During the course of program execution, the run-time system commits memory starting at address 1M. This will succeed until the available heap area of 2M is exhausted.

Example 2:

The application uses a DLL which contains its own copy of the run-time system. The DLL is initialized first and reserves 4M for application heap allocations. Then the .EXE's startup code also reserves 4M for the same purpose. Both calls will succeed (addresses 1M for the DLL and 5M for the .EXE), although only 2M of physical heap space are available but 8M have now been reserved. Committing memory for the DLL will succeed until all available physical memory is exhausted, but any attempt of the .EXE to commit will fail, because no physical memory is present at addresses at or above 5M.

The advantages of the fixed memory manager are:

- Paging is not required.
- It is faster than the virtual memory managers, because the page table need not be scanned and no pages are remapped.
- No memory is consumed by extending the page table due to large reserved memory areas.

Thus, the fixed memory manager is well suited for most applications not using DLLs or for programs using only DLLs that do not contain their own run-time system (such as RTT32DLL.DLL, for example).

Virtual or Uncommitted Memory Manager

The virtual memory manager with uncommitted memory support is not limited to the address space allocated for the heap by RTLloc. Rather, at initialization, it will completely decommit all memory in the heap area. This is done by mapping all heap memory pages to their physical location and marking them as inaccessible in the page table with appropriate page table attributes.

When the application's run-time system reserves uncommitted memory, the page table is scanned for an appropriate unused address range large enough to fulfil the request. If one is found, the address range is then marked as reserved in the page table. This process may require the page table to grow. Thus, only reserving address space can fail due to lack of memory.

Committing memory is implemented by mapping unused physical pages of RAM to address ranges reserved in the previous step. Committing will fail as soon as no free physical pages of memory are found.

The advantages of the virtual memory manager are:

- High degree of compatibility with Win32's memory management.
- No physical memory is wasted in large uncommitted areas.
- Many large uncommitted areas of memory can be supported with only little physical memory usage (reserving 4M of address space requires only 4k of physical memory).

The virtual memory manager is well suited for applications using one or more DLLs.

Alternate Heap Manager RTTHeap

The C/C++ heap managers included in the run-time libraries can have problems with very small heaps, especially when RTTarget-32's fixed memory manager is used. The run-time system will usually attempt to preallocate a fair amount of heap space, even if the program does not make use of it. If insufficient heap space is available, the run-time system may refuse to start up even when the available heap is sufficiently large. Even with larger heaps, the wasted memory in reserved - but never committed - memory blocks remains a problem.

Since many embedded systems cannot tolerate such a waste of memory, RTTarget-32 is shipped with an alternate heap manager in library file RTTHEAP.LIB. Its use is recommended for heap sizes less than 64k. RTTHEAP is usually not required for programs using RTTarget-32's virtual memory manager with uncommitted memory support.

To use RTTHEAP, simply link RTTHEAP.LIB before the compiler's run-time system libraries. For Pascal programs, "use" unit RTTHeap as the first unit. There are no source code changes required.

RTTHeap is currently supported only for Borland and Microsoft compilers.

Chapter 8

Demo Programs

Numerous example programs are shipped with RTTarget-32. Each of them resides in a separate directory under directories Demobc, Demomsvc, Demomsdev, Demowat, and Demodel for the respective compilers Borland C/C++, Microsoft Visual C/C++ (command line tools), Microsoft Visual Studio 6.0, Watcom C/C++, and Borland Delphi.

In addition to the source code of each example program, there are also a number of configuration files included. These configuration files are preconfigured for five different typical target computer configurations: a PC-compatible target computer with at least 2M of RAM, an i386EX based evaluation board with 128k RAM and ROM, the AMD Élan Sc400 Evaluation Board, the AMD Élan Sc520 Evaluation Board, and the National Semiconductor Evaluation Board for the NS486SXF microcontroller. Sections below describe how to prepare each target to run the demos.

The demo configuration files are set up such that defining preprocessor symbol BOOT on RTLoc's command line will include boot code in the program build. If BOOT is not specified, the Debug Monitor is "reserved" with command *Reserve Monitor* to enable the program to be run with RTRun or under the Debug Monitor.

Running Demos with Command Line Tools

The On Time RTOS-32 installation has created shortcuts in the Start Menu to launch suitably configured command prompts for all demos. The shortcuts will define all required environment variables (PATH and RTTARGET) and set the current directory to the respective demo's directory. To build the demo, follow these steps:

- Connect the host and target with an RS232 NULL modem cable. If you do not want to use COM1 on the host, change the line "Port=COM1" in the RTTarget.ini file (click on "Edit Settings" in the On Time RTOS-32 Start Menu folder).
- Invoke the MAKE utility that came with your compiler (MAKE for Borland/Inprise, NMAKE for Microsoft, and WMAKE for Watcom). The makefile for each demo contains all commands to compile, link, and relocate each demo and a Debug Monitor.
- Install the Debug Monitor on the target (see below, depends on the target type).
- Reboot the target.
- Invoke RTRun to download and execute the demo or call RTD32 to debug the program.

All demos are built to run under the Debug Monitor. To build a demo to be self-booting, relocate it using RTLoc's command line option -DBOOT. Example:

```
RTLoc -DBOOT Hello Demopc.cfg Hello.cfg
BootDisk Hello A:
```

All bootable demos intended to run on a standard PC can also be started from DOS with:

```
RTTBOOT Hello.rtb
```

Running Demos in Visual Studio 6.0

The On Time RTOS-32 installation has created shortcuts in the Start Menu to launch Microsoft Visual Studio through RTTarget-32's DBGShell program. The shortcuts load the required workspace and project files for each demo. Please note that you must start MsDev.exe through DBGShell to be able to use the integrated debugger. To build a demo, follow these steps:

- Connect the host and target with an RS232 NULL modem cable. If you do not want to use COM1 on the host, change the line "Port=COM1" in the RTTarget.ini file (click on "Edit Settings" in the On Time RTOS-32 Start Menu folder).
- Set the active configuration to **Target - Win32 Debug** or **Target - Win32 Release**.

- If the demo is intended to run on a standard PC, place an empty formatted diskette in drive A:.
- Build the project with the *Build* command.
- Install the Debug Monitor on the target (see below, depends on target type).
- Reboot the target.
- If you have selected the Debug configuration, you can use any debugging command to start a remote debug session (e.g., *Debug | Step Into*). The program will be downloaded and executed. If you are prompted to enter the program's name, enter `Debug\<project>.exe`.
- If you have selected a Release configuration, the boot diskette contains the demo program as a bootable image for any demos intended to run on a standard PC. Demos intended to boot from ROM will generate a boot image as an Intel Hex file in the Release directory.

Preparing a Standard PC to Act as a Target

Most demos will run on a standard PC target. To install the Debug Monitor on a second PC, follow these steps:

- Build the demo program Hello.
- For any of the command line demos, run command:

`Bootdisk Monitor A:`

with an empty formatted diskette in drive A:. Visual Studio projects perform this step automatically as part of the build process.

- Reboot the target with this diskette.
- Use COM1 of the target to connect to the host.

Preparing a Standard PC to Act as a Target for GUI Demos

All RTPEG-32 demos require the target to boot into graphics mode. To install the Debug Monitor on a second PC in graphics mode, follow these steps:

- Build the demo program PegDemo.
- For any of the command line demos, run command:

`Bootdisk GraphMon A:`

with an empty formatted diskette in drive A:. Visual Studio projects perform this step automatically as part of the build process.

- Reboot the target with this diskette.
- Use COM1 of the target to connect to the host.

Preparing the AMD Élan SC400 Evaluation Board

If you intend to use the Élan SC400 Evaluation Board, you must first install the Debug Monitor on the board using the following steps:

- Build the demo program HelloSc400.
- Program MonSc400.hex into a 128k EPROM and place it in the DIP socket.
- Configure the board to connect ROMCS0 to the DIP socket.
- Use COM1 of the Eval Board to connect to the host.

Preparing the AMD Élan SC520 Evaluation Board

If you intend to use the Élan SC520 Evaluation Board, you must first install the Debug Monitor on the board using the following steps:

- Build the demo program HelloSc520.

- Program MonSc520.hex into a 256k EPROM and place it in the BIOS DIP socket.
- Configure the board to connect ROMCS0 to the BIOS DIP socket (this is the default).
- Use **COM2** of the Evaluation Board to connect to the host. COM1 is configured to be RS422 by default and is therefore not suitable for communication with a RS232 port of the host.

Preparing the NS486 Evaluation Board

If you intend to use the NS486SXF Evaluation Board, you must first install the Debug Monitor on the board using the following steps:

- Build the demo program NSHello.
- Install jumper W2 on the Evaluation Board. The monitor needs interrupt-driven serial I/O on the DEBUG COM port.
- Use the DEBUG COM port to connect to the host.
- Run the Flashldr program supplied with the board and execute the following commands:

```
program nsmon.hex
vector=0FFFF0000p
setboot nsmon
boot nsmon
exit
```

Program Hello

Hello is the simplest of all demo programs. It contains a single printf statement to display the string "Hello, RTTarget-32!" on the screen. You can use it to verify that RTTarget-32 is correctly installed and configured on your system.

Program Hello2

Hello2 performs the same task as Hello. However, the program does not rely on the run-time system. Instead, low-level RTTarget-32 functions are used to display a string on the screen. Since Hello2 uses no run-time system functions (such as screen or file I/O, the heap, etc.), Hello2 is linked without the run-time system and RTTarget-32's simplified startup code C0RTT.OBJ is used. The make file contains the complete command lines to compile and link Hello2. Please note that although Hello and Hello2 do the same thing, Hello2 requires much less memory.

The Delphi version of Hello2 does not actually eliminate the run-time system. Rather, the program shows how to access RTTarget-32's native API from a Pascal program.

Program SerInt

Program SerInt demonstrates how to install a hardware interrupt handler with RTTarget-32. As an example, this program installs a very simple interrupt handler to receive data on serial port COM2.

Since most Win32 compilers lack support for interrupt handling (because interrupts are not supported for application programs by Win32), the low-level interrupt thunk is written in assembler and included in SERISR.ASM or provided as inline assembler code for Delphi and Microsoft C. It contains a little function named _ASMHandler which saves all registers, loads ES and DS, and then calls the high-level handler written in C (respectively Pascal). The high-level handler will read the byte received from the UART chip and place it in a buffer.

The main program continually checks whether the user has requested program termination by pressing Escape on the keyboard, or whether data has been received on COM2.

Programs using RTKernel-32 should not handle interrupts by the method described in this demo. Rather, they should use RTKernel-32's interrupt API demonstrated in program RTKInt.

The program is kept very simple and lacks many features of high-performance serial I/O applications. However, it is a good example of interrupt processing under RTTarget-32. Please note that RTKernel-32 applications should use the RTKernel-32 API to install hardware interrupt handlers.

Program SerDemo

SerDemo shows how to use RTTarget-32's serial I/O support. It initializes the target's COM2 for 9600 baud, 8 data bits, 1 stop bit, no parity, and RTS/CTS hardware handshake. All data received is displayed on the screen and echoed to the same port. If transmission errors are detected, corresponding error messages are displayed on the screen. Each time a carriage return character is received, a complete line of text is echoed to the sender.

The program can be terminated by entering any character on the target's keyboard.

Please refer to Chapter 7, section *Serial I/O Functions* for more information about RTTarget-32's serial I/O support. RTKernel-32 should not use module RTTCom for serial I/O; instead, use module RTCom as demonstrated in demo program COMDemo.

Program MAPDemo

MAPDemo shows how RTTarget-32's memory mapping functions work. The program uses an Init function to extend the heap with function RTextendHeap. In addition, it maps different parts of the display memory into its virtual address space.

Please note that MAPDemo assumes the program is built using the preconfigured configuration files. If DEMOPC.CFG is modified, modifications to MAPDemo's source code may also become necessary. Please refer to MAPDEMO.C or MAPDEMO.PAS for details.

Program EmuDemo

This program shows how to use the 387 FPU Emulator RTEmu.

Program DLLDemo

DLLDemo shows how an application can consist of several DLLs. The main program links LIB1.DLL statically and LIB2.DLL dynamically with LoadLibrary. For the RTTarget-32 Library, RTT32DLL.DLL is used.

Apart from an Init function (which is not executed under Win32), DLLDemo does not call any RTTarget-32 native API functions. Thus, DLLDemo can actually be executed under Win32.

Program DLLDemo2

DLLDemo2 is similar to DLLDemo, but RTT32DLL.DLL is not used. Instead, the RTTarget-32 system library RTT32.LIB is linked into the main program. All functions of RTT32.LIB needed by LIB1.DLL and LIB2.DLL are exported from the main .EXE.

DLLDemo2 also demonstrates how utility program MakeDef is used to generate an RTTarget-32 system module which exports only the API functions actually required by the main program and the two DLLs.

Program DLLDemo3

DLLDemo3 is also similar to DLLDemo, but instead of RTT32DLL.DLL, it uses a custom RTTarget-32 system DLL. All functions of RTT32.LIB needed by LIB1.DLL and LIB2.DLL are exported from that DLL. DLLDemo3 also demos the use of DLMs. Unlike DLLDemo and DLLDemo2, LIB2.DLL is loaded as DLM file through RTTarget-32's RAM file system.

Program SysDemo

This demo is only available for Microsoft Visual C++ and Borland C++. SysDemo shows how a system DLL can be constructed to minimize download times, in particular for programs using several On Time RTOS-32 components. The demo builds a system DLL only with those Win32 and native API functions actually called by the application. The main program then uses this DLL instead of linking the On Time RTOS-32 libraries. Since the system DLL will rarely change, it will usually not be re-downloaded, even if the much smaller program .EXE file has been modified. The Borland version of this demo even places the C/C++ run-time system into the system DLL, resulting in a very small .EXE file size. However, file RTL.TXT must be maintained manually with a list of all run-time system functions available for the .EXE.

This demo uses four On Time RTOS-32 components (RTTarget-32, RTKernel-32, RTFiles-32, and RTPEG-32), but removing unneeded components is easily accomplished by removing the respective libraries and API files from the make file or project file.

Program Loader

Program Loader shows how to use function `RTRunProgram`. The loader program includes a binary image file of a child program to start at run-time. The loader copies the files to the address it was located to by `RTLLoc` and then starts it.

The build process for the two programs (Loader and Child) is a bit unusual since they are mutually dependent. The Child program must be built with a *Reserve Loader* command; this requires the loader to be built first. On the other hand, the loader contains a *Locate File Child.rtb* command, which requires the Child to be built first. To get a valid configuration, both programs must be built several times. Real-world applications which download the loaded child program will probably not have this problem, since only the loaded program will depend on the loader but not vice-versa.

The method of starting one program from another with function `RTRunProgram` has the advantage that the Child can return to the Loader, but this does require quite a bit of memory. A more efficient method is used by function `RTBootRM` and `RTBootPM` introduced by demo program `BootProg`.

Program BootProg

Demo program `Bootprog` demonstrates using function `RTBootRM()`. The locator produces a BIN file of program Child which is then loaded and executed by program `Bootprog`.

This demo can be built in three different configurations:

- **Debug Bootprog**

The target is booted with the Debug Monitor and program `Bootprog` can be executed by downloading. When program `Bootprog` calls `RTBootRM()`, `Bootprog` and the Monitor are overwritten by the child. By default, this configuration is built by the command line versions of this demo with commands:

```
RTLLoc -DBOOT Monitor Demopc.cfg Monitor.cfg
RTLLoc Child Demopc.cfg Child.cfg
RTLLoc BootProg Demopc.cfg BootProg.cfg
```

- **Debug Child**

The target is booted with the Debug Monitor and program Child can be executed by downloading. This configuration is built by the "Win32 Debug" configuration of the Visual Studio version of this demo with commands:

```
RTLLoc -DBOOT Monitor Demopc.cfg Monitor.cfg
RTLLoc -DDEBUG Child Demopc.cfg Child.cfg
```

- **Release**

The target is booted with program `Bootprog` which will in turn boot to program Child. This configuration is built by the "Win32 Release" configuration of the Visual Studio version of this demo with commands:

```
RTLLoc Child Demopc.cfg Child.cfg
RTLLoc -DBOOT Bootprog Demopc.cfg Bootprog.cfg
```

Program BIOSDemo

This program shows how to use the PCI BIOS and PnP BIOS functions of `RTTarget-32`.

Program PCCard

This program shows how to use `RTTarget-32`'s PCMCIA driver. By default, this program requires `RTFiles-32` to support ATA-Flash disks, but it can easily be reconfigured not to use `RTFiles-32` (see the source code and makefile for details). The demo also supports PC cards which implement a UART (e.g., serial port cards, modems).

Please note that this demo shows how PCMCIA events can be handled in a single-threaded environment. For multi-threaded programs, a separate thread should handle card insertion/removal events as shown by the PCCardMT demo program.

Program PCCardMT

This program shows how to use RTTarget-32's PCMCIA driver in a multithreaded program. It is similar to program PCCard, but it uses a dedicated thread instead of a polling loop to handle PCMCIA events. RTKernel-32 is required to run this demo.

Program EXLED

Program EXLED is intended for the Incosys EMU386EX Evaluation Board. Without this board, you cannot run this demo program. However, you can study the configuration files DEMOEX.CFG and EXLED.CFG as examples for programs which will boot the target directly from EPROM, without any BIOS support. EXLED.CFG also demonstrates the use of virtual regions to run code in ROM and still locate with the *Locate NTSection* command.

The configuration file DEMOEX.CFG contains the region definitions for the Incosys board. It has 128k of static RAM and a 128k EPROM (the board's flash memory is not used by EXLED). The 128k EPROM is always written to a hex file with the same base name as the program being built and extension .HEX. Thus, when the Monitor ExMon is built, Intel hex file ExMon.hex is generated. When ExLED is built as a self-booting program, file ExLED.HEX is produced. This file has to be programmed into the last 128k of physical address space on the target to run it.

The configuration of the EX386 board and the ExLED demo program are examples only. Since this particular board has only 128k of RAM, the demo may fail to build due to an overflow of the SRAM region. The program is only intended as an example and a starting point for real projects. Please do not attempt to run this example unmodified if you do not have the Incosys EMU386EX Evaluation Board.

Program HelloSc400

This demo program can run on the AMD Élan SC400/SC410 Evaluation Board without a BIOS. The included configuration file Sc400ini.cfg contains an example chipset initialization for the AMD Élan SC400/410 CPU family.

Program HelloSc520

This demo program can run on the AMD Élan SC520 Evaluation Board without a BIOS. The included configuration file Sc520ini.cfg contains an example chipset initialization for the AMD Élan SC520 CPU.

Program NSHello

This program is configured to run on the NS486SXF evaluation board. To run it, you must first install the Debug Monitor for the NS486SXF (NSMON) on the evaluation board (see description at the beginning of this chapter).

NSHello displays some information about the target hardware configuration and checks that the on-chip real-time clock is working properly and has been set.

Program TVDemo

TVDemo shows how to use Borland's Turbo Vision class library for text mode user interfaces. TVDemo can only be used with Borland C/C++ and was originally supplied by Borland with Turbo Vision.

The Turbo Vision library itself is not included with RTTarget-32. It can be downloaded from Borland's Web site (<http://www.borland.com>) or CompuServe forum (GO BCPPLIB, library 19, file TV.ZIP or the patched version TV2BUG.ZIP).

TVDemo can be compiled, linked, and located using the supplied makefile. Two versions of TVDEMO can be built: one with and another without RTKernel-32. To build without RTKernel-32, just type:

```
make tvdemo
```

To build the program with RTKernel-32, use command line:

```
make -DRTK32 tvdemo
```

Program ClassDemo

This demo is only available for Microsoft Visual C++ 6.0 and Borland Delphi. The Visual C++ version demonstrates the use of some non-GUI MFC classes and templates. Supported classes and templates include CString, CTime, CTimeSpan, CFile, CArchive, CArray, CCriticalSection, CList, CMap, and CMapStringtoOb, as well as most classes derived from these. To use non-GUI MFC classes, a program must adhere to a few restrictions:

- All Win32 API libraries such as kernel32.lib, user32.lib, shell32.lib, and advapi32.lib, must be excluded using linker option /nodefaultlib.
- The Microsoft debug run-time system libraries cannot be used.
- RTTHeap cannot be used.
- The application must define replacements for MFC functions AfxMessageBox and AfxGetFileName.

Please refer to demo program ClassDemo.cpp for further details.

The Delphi version of this program demonstrates how some non-GUI classes of VCL unit Classes can be used in On Time RTOS-32 programs. The use of classes TList, TThreadList, TBits, THandleStream, TFileStream, TMemoryStream, TResourceStream, TStrings, and TStringList is demonstrated.

Program MetWorld

The MetWorld program demonstrates how to use Metagraphics' graphics library MetaWINDOW with RTTarget-32. The graphics demos only work if you have purchased and installed the MetaWINDOW library from MetaGraphics.

The graphics example programs expect the MetaWINDOW library to reside in directory \METAWIN. If it is not found here, define environment variable METAW to be the top level MetaWINDOW directory.

The graphics examples require a differently configured Debug Monitor. Unlike the Monitor for all other examples, it configures its boot code to select video graphics modes 105h 103h 101h 102h 12h (256 color modes with resolutions 1024x768, 800x600, 640x480, and then 16 color modes with resolutions 800x600 and 640x480). The makefile accompanying the demos builds the required monitor.

This demo can only be compiled with Microsoft Visual C++ if the COFF version of MetaWINDOW is installed. Registered MetaWINDOWs users can download the COFF library MET_MR3L.LIB from Metagraphics' Web site at <http://www.metagraphics.com>.

Program HelloGUI

The HelloGUI demo program was originally supplied by Metagraphics and has been slightly modified to make it work under RTTarget-32. HelloGUI demonstrates how to implement a simple, mouse and keyboard driven Graphical User Interface (GUI) with the MetaWindow library.

Please note that both graphics demo programs are supplied by MetaWINDOW. If you have any questions about these programs or the MetaWINDOW library, please contact MetaGraphics.

Chapter 9

Advanced Topics

This chapter discusses some of the services provided by RTTarget-32 and some techniques for creating reliable embedded systems applications.

Choosing a Locate Method

RTLoc offers a rich set of options to map your application to the target hardware. While these options provide a high degree of flexibility, you must decide how to map your program. Frequently, several different approaches are possible with varying advantages and disadvantages.

The following sections discuss some of the required design decisions.

Locate Section or NTSection

The only major disadvantage of *Locate NTSection* is that all sections must be placed in the same region. Thus, for ROMable applications, all sections having an image must be copied to RAM, which duplicates at least the code section, and possibly other read-only sections. Another problem could be that there is no region large enough to contain the whole program. However, both problems can be solved by using a virtual region. An advantage of this approach is the Windows NT compatible fixup method. *Locate NTSection* is required if you want to debug your program.

The disadvantage of *Locate Section* is the different mapping algorithm required for fixing-up. Since this fixup method can fail (though only in very rare circumstances), some programs cannot be mapped in this way. The advantage is its high degree of flexibility. *Locate Section* sections can be located in different regions, making ROMable applications possible without copying the code or other read-only sections. Paging is not required to fully exploit the mapping flexibility of *Locate Section*.

NTSection should be used whenever possible. Not all programs can run with *Locate Section*, since the compiler or run-time system may rely on the way Windows NT loads programs. If you want to use *Locate Section* anyway, make sure the program runs with *Locate NTSection* first. If it works with *Locate NTSection* but not with *Locate Section*, then you will probably not be able to use *Locate Section*.

Physical or Virtual Regions

The disadvantages of virtual regions are the requirement to use paging, a minimum alignment of 4096 bytes, and the possible need for the page table to be enlarged to accommodate a potentially larger linear address space.

Virtual regions have no advantages if the program is mapped with *Locate Section*. However, if *Locate NTSection* is used, there are two circumstances that could make use of a virtual region necessary: if the program is to run completely in RAM and no physical RAM region is large enough to contain the program, or if parts of the program should reside in ROM.

Running with or without Paging

RTTarget-32 supports paging through the *Locate PageTable* command. However, this feature is optional. Some advantages of paging are:

- Memory protection. Each 4k page of memory has its own access rights. In this way, memory which should not be accessed or modified at run time can be protected. Since every invalid memory access causes an exception, this feature is very useful for debugging a program with stray pointers or similar bugs.
- Virtual Regions. Some programs can be mapped more efficiently in a virtual region.
- RAM remapping. Unused pages of memory can be appended to another region. This has two advantages: regions larger than physically available can be created for program entities that must be located in consecutive address space (such as stack or heap). In addition, memory that would be wasted due to alignment restrictions without paging can be used.

- Uncommitted memory support. RTTarget-32's virtual memory manager can be used only if paging is enabled.

Disadvantages of paging are:

- Memory overhead. The page table itself requires memory on the target system. For example, the page table for a computer with 4M of memory needs 8k. If RAM remapping is used, the page table grows to 12k.
- Run-time overhead. Paging can lead to slightly slower program execution. The amount of overhead incurred depends on the program's use of memory and the CPU used. However, the overhead is hardly measurable and can be ignored on most systems.

Generally, the use of paging is recommended. The extra level of protection usually outweighs the small overhead incurred. If very low resource requirements are a major design goal, paging could be used during software development only and disabled for the production release. For applications using DLLs, paging is recommended to be able to use the virtual memory manager.

Running at CPL 0 or 3

Similar to paging, selecting the program's privilege level is a trade-off between maximum protection and performance.

Advantages of privilege level 3 are:

- Maximum page level protection. At CPL 3, the CPU distinguishes four different access rights for memory pages. System data structures can be completely protected. At CPL 0, the CPU allows read and write access to all pages actually mapped to memory. For example, there is no protection against corruption of the system tables or the code segment.

Advantages of privilege level 0 are:

- The program can execute privileged instructions (CLTS, HLT, LGDT, LIDT, LLDT, LMSW, LTR, MOV to/from CR0/DRn/TRn). While most privileged instructions have little value for application programs, HLT can be useful. Multitasking systems such as RTKernel-32 can execute HLT in their idle task. Any program that waits for an interrupt can call function RTHalt, which executes HLT. While in the HLT state, the CPU consumes only a small fraction of the power it would need otherwise. Note that function RTHaltCPL3 can be employed to execute Halt while running at CPL 3.
- Lower interrupt latency. Some functions of RTTarget-32's native API are handled by the boot code and accessed through a software interrupt. To guarantee proper interrupt processing, the boot code executes with interrupts disabled if called from CPL 3, but with interrupts enabled if called from CPL 0. The effect on the interrupt latency depends on which functions of RTTarget-32's boot API are actually used. There is no penalty for applications that do not use it.

CPL 3 is recommended at least during the development phase of a program. If low power consumption or a very low interrupt latency are of great importance, CPL 0 may be considered for the production release.

Installing Hardware Interrupt Handlers

The RTTarget-32 boot code installs interrupt handlers for all 16 IRQs and disables all IRQs except 0, 1, and 2. The *BOOTFLAGS* command can be used to modify this behavior. On IRQ 0 (the timer interrupt), the master interrupt controller is reset. On IRQ 1 (the keyboard interrupt), the keyboard controller is read and the master interrupt controller is also reset. If the scan code pressed is the DEL key on the numeric keyboard, function RTReboot() is called. All other scan codes are ignored. The interrupt handlers on all other IRQs reset the interrupt controller(s) and display a warning message.

RTTarget-32's Win32 emulation library can install interrupt handlers on IRQ 0 and 1, overriding the boot code's handlers. The first call to GetTickCount() will install a handler on IRQ 0. The handler simply increments an integer which is evaluated by function GetTickCount. The first call to any function that

might read keyboard input (for example, any file I/O function, `kbhit()`, `getch()`, or any of the console I/O functions) will install a handler on IRQ 1 to read and interpret keyboard scan codes. This handler does **not** reboot the target on any scan code such as DEL or Ctrl-Alt-DEL.

It is important to be aware of the installation sequence of these handlers if you plan to install your own handlers. For example, if you wish to install your own timer interrupt handler and you also want to use function `GetTickCount()`, you **must** call `GetTickCount` once **before** you install your own handler and your handler should chain to the previously installed handler (otherwise, `GetTickCount` will no longer work). You should also consider that you can only chain to interrupt handlers running at the same privilege level as your program (except if you reinstall them on a different vector and chain with the INT instruction). Thus, you should not chain to a handler installed by the boot code (which runs at CPL 0).

Catching NULL Pointer Assignments

The nastiest type of bug results from the use of uninitialized or corrupted pointers. The most frequent incorrect value of a pointer is NULL (0). The simplest method to catch these errors is to make the first page of memory inaccessible. Of course, paging must be used to achieve this.

The simplest method is to define a region of 4k size at address 0 and give it *NoAccess* access rights. Example:

```
Region  FirstPage 0      4k RAM NoAccess
Region  LowMem     4k 636k RAM Assign
...
```

This will guarantee that all references to address 0 will trigger an exception at run-time.

If you want to avoid wasting a whole page of memory, you could just remap it:

```
Region  FirstPage 0      4k RAM Assign
Region  LowMem     4k 636k RAM Assign
Region  HighMem    1M    3M RAM Assign
FillRAM HighMem
...
```

If you don't allocate anything to region `FirstPage`, `RTLoc` will append the physical memory page at address 0 to the end of `HighMem`, making it available for the heap and stack.

However, you should consider that this approach will destroy the real mode interrupt vector table and the real mode BIOS data area. This is usually no problem, but if your program needs to look up some information in this area at run time, you can't use this approach. Instead, region `FirstPage` would need *ReadOnly* or *ReadWrite* access.

Catching Stack Overflows

Protecting from stack overflows works in a similar way. The stack grows from high address to low address. To detect a stack overflow, the memory immediately preceding the stack should have less than read/write access. For example, the stack could be placed following the code section or in a region separated from other regions by at least one page. Example:

```
Region  FirstPage 0      4k RAM Assign
Region  LowMem     4k 636k RAM Assign
Region  HighMem    1M    3M RAM Assign

Virtual ProgMem    4M                      // for code and data
Virtual StackMem   5M                      // for the stack
Virtual HeapMem    6M                      // for the heap which gets
FillRAM HeapMem                                // all remaining memory
...
Locate Stack S StackMem->LowMem 16k
Locate Heap  H HeapMem
```

Both the stack and the heap are placed in separate virtual regions, completely isolating them from other program entities. Any stack overflow will cause the stack pointer to leave region `StackMem` and will trigger a page fault.

Running with or without Run-Time System

RTTarget-32 fully supports the run-time systems of the supported compilers. This simplifies porting programs that use the run-time system extensively. However, there is some overhead involved. Depending on the application's use of the run-time system, up to 100k of memory can be required to support it.

Simple applications that don't need the heap can be significantly reduced in size if the run-time system is eliminated. However, all calls to run-time system functions (such as malloc, free, printf, etc.) must be removed.

Advantages of the run-time system are:

- Easy porting. Existing software containing calls to the run-time system can run under RTTarget-32, often with no or only minor modifications.
- Ease of programming. The functionality of the run-time system can be used for the benefit of the application.
- Availability of the heap. Dynamic memory management using malloc, free, realloc, etc., is fully supported.
- RAM files. Programs requiring read access to files can run even though RTTarget-32 does not include a file system.
- C++ features can be used. The run-time system takes care of automatic constructor/destructor calls, exception handling, etc.
- Run-time checks of the run-time system can be used. For example, most compilers can be instructed to generate calls to run-time system routines for stack overflow or other checks.

Benefits of running without the run-time system are:

- Lower resource requirements. Programs without the run-time system are much smaller.
- No Heap. Most Win32 programs expect a heap of at least 64k to be available, even if the application requires much less heap space. Without the run-time system, there is no need to allocate a single byte of heap memory.

RTTarget-32 does not support eliminating the Delphi run-time system for Pascal programs. However, due to Delphi's smart linker, the Pascal run-time system's size can be next to negligible. If unit SysUtils is not used, the Delphi run-time system does not allocate any heap space for its internal housekeeping.

Generally, the use of the run-time system is recommended. It should be eliminated only if low resource requirements must be met.

Avoid Repeated Downloads

If a program is frequently tested using the cross debugger, repeated downloads can become quite time-consuming, even at 115200 baud with data compression. However, the debugger will always check whether downloading data repeatedly is actually required by performing a CRC check on the different parts of the program on the target. If all data resides in read-only memory, the program needs to be downloaded only once and never again.

Some program entities never change anyway (for example, the code). For other entities, the *Locate Copy* command can be used to keep protected and compressed copies on the target. Example (for Borland C++):

```
#include "demopc.cfg"
FillRAM HighMem
Reserve Monitor
```

```
Locate PageTable Pages LowMem
Locate Header Header HighMem
Locate NTSection CODE HighMem
Locate NTSection DATA HighMem
Locate Stack Stack HighMem 16k
Locate Heap Heap HighMem

Locate DecompCode Expand LowMem
Locate DecompData ExBuf LowMem
Locate Copy Pages LowMem
Locate Copy DATA LowMem
Locate Copy CODE LowMem
```

The only two entities that can change at run-time are the page table and the DATA section. However, since copies of these are available on the target, repeated downloads after a program reset are not required. Copying the CODE section compresses it, reducing download times.

Switching between Configurations with and without Debug Monitor

Typically, the RTTarget-32 Debug Monitor will be resident on the target computer during the development phase of a project to allow downloading and source level debugging. On the other hand, the final configuration will probably require including boot code instead.

RTLc's preprocessor can be used to easily switch between these configurations without the need to edit any configuration files.

Suppose we have the following configuration files to build the monitor and program TestProg to run on a PC, booted from disk:

HARDWARE.CFG:

```
Region NullPage 0 4k RAM
Region LowMem 4k 636k RAM
Region VideoM B0000h 4k Device ReadWrite
Region VideoC B8000h 4k Device ReadWrite
Region HighMem 1M 3M RAM

#ifdef BOOT
    Locate BootCode BIOSBOOT.EXE LowMem
    Locate BootData SystemData LowMem
    Locate DiskBuffer Buffer LowMem
#else
    Reserve Monitor
#endif

BOOTFLAGS = BF_NO_FPU
VideoRAM VideoC
COMPort COM1 115200
```

MONITOR.CFG:

```
Locate Section CODE LowMem 1
Locate Header Monitor LowMem 0 4
Locate Section DATA LowMem 2
Locate Stack Stack LowMem 1k 4

Locate PageTable PageTable LowMem
Locate DecompCode Expand LowMem
Locate DecompData ExBuffer LowMem

Locate Copy CODE LowMem
Locate Copy DATA LowMem
Locate Copy Pages LowMem
```


TESTPROG.CFG:

```

Virtual ProgMem 4M
Virtual StackMem 5M
Virtual HeapMem 6M
FillRAM HeapMem

Locate Header      TestProg LowMem
Locate PageTable   Paging    LowMem 16k

Locate NTSection   CODE      ProgMem->HighMem
Locate NTSection   DATA     ProgMem->LowMem
Locate Stack       Stack     StackMem->HighMem 16k
Locate Heap        Heap      HeapMem

Locate DecompCode   Expand    LowMem
Locate DecompData   ExBuffer  LowMem

Locate Copy         CODE      LowMem
Locate Copy         DATA     LowMem
Locate Copy         Paging    LowMem

```

Now you can build the Monitor, TestProg for debugging, and TestProg's release version with the following commands:

```

RTLoc -DBOOT Monitor Hardware.cfg Monitor.cfg
RTLoc          TestProg Hardware.cfg TestProg.cfg
RTLoc -DBOOT TestProg Hardware.cfg TestProg.cfg

```

Using Data Compression

Except for headers, the boot code, and the decompression code, any program entity containing an image can be compressed by RTLoc. The following prerequisites must be satisfied for data compression:

- *Locate DecompCode* and *Locate DecompData* commands must be present.
- Entities to be compressed must be copied using the *Locate Copy* command. The copied entity will be compressed. At boot time, the copied entity is expanded to the original entity.

The extra entities required for compression (DecompCode, DecompData, and Copies) are all discardable. Thus, if they reside in RAM, their memory can be reused by the application's stack and heap without causing any memory overhead.

Apart from reducing program image sizes, data compression has an impact on application initialization times. Decompressing program code and data may require several seconds (the .LOC file contains the times needed for decompression, as measured on the host). The only exception is the page table, which will typically decompress faster than it can be copied. Of course, compression also requires time on the host, slowing down RTLoc. However, with modern desktop systems equipped with a Pentium or higher CPU, this can be neglected for most systems. Again, the Compression Report in the .LOC file shows these times.

Boot time may be an issue for systems which must be able to start up very quickly. The following sections discuss how boot times and overall memory requirements vary depending on how the application is loaded.

Downloading and Cross Debugging

Compressed applications can typically be downloaded twice as fast. Since all discardable entities are located to RAM, there is no memory lost on the target. Thus, compression is highly recommended during the development phase.

Applications Booted from Disk

The disk space requirement is typically 50% of an uncompressed application, and there is no extra memory required, since all extra program entities required for compression are discardable and reside in RAM.

Even program load time will be faster with compression for most systems, because less data has to be read from the boot device and decompressing is typically faster than reading data from disk. However, the net gain or loss in load time depends on the speed of the boot device and the speed of the CPU. If booting from floppy disk, compression will certainly improve startup time, even on slow CPUs. The same is true for most hard disk based systems with 486 or higher CPUs. Silicon disks, however, may be faster than RTTarget-32's decompression algorithm (with the exception of page tables).

Applications copied from ROM to RAM

Applications which are booted from ROM, but then completely copied to RAM will benefit from significantly reduced ROM space requirements with no extra RAM needs. Thus, the total memory requirement is actually reduced. Boot time, however, will be higher with compression.

Applications Running in ROM

If all read-only entities remain in ROM, data compression can only be used for initialized data and the page table. Since the decompression code itself requires about 2k of memory (and cannot be reused because it resides in ROM), the total ROM space requirement may actually increase if no paging is used and the initialized data size is not reduced by more than 2k.

Memory savings and boot time differences may be small for such applications and depend heavily on initialized data size.

Using DLLs through RTLoc

The RTTarget-32 locator RTLoc supports the *DLL* command which adds DLLs to an application image. Both static references to DLL exported functions and dynamic linking using LoadLibrary/GetProcAddress are supported. Loading DLLs as files on the target is supported if a file system is used. For details about loading DLLs through a file system, see section *Loading DLLs through a File System* later in this chapter.

To use DLLs successfully, a thorough understanding of Win32's DLL mechanism is mandatory. If you are not familiar with terms such as import library, DLL exports, DLL imports, static DLL references, LoadLibrary, GetProcAddress, or DllEntryPoint, please consult your compiler's documentation before trying to use RTTarget-32's DLL support.

To successfully use DLLs located into a program's image, the following prerequisites must be satisfied:

- For each DLL to be used, a separate *DLL* command must appear in the configuration file.
- All code and data sections of all modules must be mapped using separate *Locate Section* or *Locate NTSection* commands.
- For each module (EXE or DLL) which statically imports DLL entrypoints, the *.idata* section must be mapped with *Locate Section* or *Locate NTSection* commands. RTLoc will issue warning messages if this is not the case.
- For each module whose exported entrypoints must be available to GetProcAddress at run-time, the *.edata* section must be mapped with *Locate Section* or *Locate NTSection* commands. The *.edata* sections are **not** required for static fixups which are processed by RTLoc.
- The RTTarget-32 library RTT32.LIB must be linked into exactly one module (either the main program or one of the DLLs). The preconfigured DLL RTT32DLL.DLL may be used for this purpose. All RTTarget-32 native API and Win32 emulation functions which are referenced by other modules must be exported using a module definition file (DEF file).
- Any module which needs to call an RTTarget-32 native API function must define symbol RTT32DLL before including header file RTTARGET.H and must link import library RTT32DLL.LIB instead of RTT32.LIB. For Delphi programs, access to the RTTarget-32 API is available through unit RTTarget.

In addition, it is recommended to link only one run-time system, preferably into the main EXE. Multiple run-time systems are supported, but may lead to very inefficient heap memory usage, especially when no paging is used. DLLs without a run-time system can be produced with RTTarget-32's special startup code C0RTTD.OBJ. For optimal heap management in multi-module applications, it is recommended to use RTTarget-32's alternate heap manager RTTHeap in all DLLs and the main EXE. RTTHeap will insure that only one memory manager is used for all modules.

Here are some advantages of using DLLs:

- Programs and DLLs which do not need access to RTTarget-32's native API can be binary compatible with Win32. For example, the DLLDemo example program shipped with RTTarget-32 consists of four modules: DLLDEMO.EXE, LIB1.DLL, LIB2.DLL, and RTT32DLL.DLL. The application can be executed under Win32 without relinking.
- Projects can be composed of modules developed with different compilers and even programming languages. For example, RTTarget-32's Delphi support relies on Pascal programs using RTTarget-32's run-time support implemented with a DLL written in C and assembler.
- DLLs which were not developed for RTTarget-32 can be used as long as these DLLs do not call Win32 API functions not supported by RTTarget-32.
- Reduced download times. Although applications with DLLs are typically larger, repeated downloads during software development can be accelerated if code modifications do not affect all modules. In this case, unmodified modules are not downloaded again.

Disadvantages of DLLs are:

- The complexity of an application can increase significantly. Most applications will need custom linker definition files.
- Memory requirements increase because more memory is lost in section alignment.
- Application image size and memory requirements increase due to redundant code in several modules.
- If several run-time systems are used, heap memory is potentially used less efficiently.

The following sections describe several different configuration options available.

Using RTT32DLL.DLL

RTT32DLL.DLL is a preconfigured module in RTTarget-32's BIN directory. It exports RTTarget-32's native API as well as RTTarget-32's Win32 API emulation. To use RTT32DLL.DLL, the corresponding import library RTT32DLL.LIB can be linked. Please note that import library RTT32DLL.LIB does not contain import records for the Win32 emulation. The linker will resolve such calls against the Win32 import library supplied with the compiler (e.g., IMPORT32.LIB, KERNEL32.LIB, or unit Windows for Delphi). If RTT32DLL.DLL is used, `#define RTT32DLL` must be included in the source files immediately before `#include <rttarget.h>`.

RTT32DLL.DLL contains all functions provided by RTTarget-32, even those which might never be used by the application. Thus, linking RTT32.LIB instead of using RTT32DLL.DLL may result in smaller applications.

Example program DLLDemo shows how to use RTT32DLL.DLL.

Linking RTT32.LIB into the EXE

When RTT32.LIB is linked into the main program, other DLLs can access RTTarget-32's functions in the EXE if they are exported. This generally means that a module definition file has to be used when linking the EXE. The DEF file must list all functions (RTTarget native and Win32 emulation) needed by other modules with EXPORT directives. The advantage of this method is that only those parts of RTT32.LIB actually used by the application are linked. The disadvantage is that maintaining DEF files can be cumbersome. However, utility MakeDef.exe can be used to automate DEF file maintenance (see section *Utility MakeDef* later in this chapter).

RTLoc's Dynamic Link Report (which must be explicitly enabled with option -Rd+) in the LOC file can be used to analyze exactly which functions must be exported. All missing functions will be reported in warning messages. If you have exported functions not required by the application, they will be listed under *unreferenced entrypoints* at the end of the Dynamic Link Report.

Example program DLLDemo2 shows how to link RTT32.LIB and export RTTarget-32 functions to other DLLs.

Using a Custom RTTarget-32 System DLL

A mixture of the two strategies described above is to create a custom DLL containing RTTarget-32's API. You can create an application specific DLL which contains a subset of RTT32DLL.DLL. This method avoids linking code which is never used. On the other hand, it also requires maintaining a custom DEF file for those functions which are required, or using MakeDef.

Example program DLLDemo3 shows how to create and use an application specific RTTarget-32 system DLL.

Utility MakeDef

Commandline utility MakeDef can significantly ease the maintenance of .DEF files. MakeDef reads function names from one or more function list files and converts them into a DEF file suitable for exporting or importing these function for various compilers. Regardless of the used compiler, MakeDef always uses the same naming conventions as Win32: no function names pre- or postfixes.

MakeDef's command line:

```
Makdef [Options] DLLName DefFileName F.Lists...
```

Available options are:

- M Generate a .DEF file for Microsoft Visual C++ (imports and exports use the same format).
- Bi Generate an import .DEF file for Borland C/C++.
- Be Generate an export .DEF file for Borland C/C++.
- Wi Generate an import .LBC file for Watcom C/C++.
- We Generate an export .LBC file for Watcom C/C++.
- EExeName Use ExeName's import table as a function filter. Only the functions required by the given .EXE file(s) will be included in the resulting .DEF file. ExeName can be the name of any PE file. Thus, it also supports DLLs. Several -E options can be specified.
- IPath Search for Filelist files in directory Path.

Parameter *DLLName* is the name (with file name extension) of the DLL for which an import or export library is to be generated.

Parameter *DefFileName* is the name of the output file.

Parameter *F.Lists* is one or more function name list file. Function name list files are line oriented. Each line must start with a function name without leading underscore. If the function uses the stdcall calling convention, the number of bytes pushed onto the stack as parameters must follow. Blank lines or lines starting with a colon ":" are ignored. Example:

```
; sample function list file for MakeDef
CreateFile 28 ; Win32 API functions use stdcall
malloc      ; this is a cdecl function
```

On Time RTOS-32 is shipped with function list files for all APIs it makes available. The following files are available in directory Source:

- Rtt32api.txt RTTarget-32's native API.
- Win32api.txt RTTarget-32's Win32 emulation API.
- Rtk32api.txt RTKernel-32's native API.

W32apimt.txt RTKernel-32's Win32 emulation API (only functions not covered by Win32api.txt).

Rtf32api.txt RTFiles-32 native API.

Demo programs DLLDemo2 and DLLDemo3 show how MakdeDef can be used.

Differences from Win32

The following properties of RTTarget-32's DLL support differ from Win32's implementation:

- LoadLibrary always supports DLLs included in the locate process. It can load DLLs as files only if they are converted to DLMs and if the file system RTFiles-32 is used.
- GetProcAddress finds exported entrypoints only in modules which have their .edata section mapped on the target.
- GetModuleHandle will return the handle of the module containing RTT32.LIB for module names KERNEL32.DLL, USER32.DLL, ADVAPI32.DLL, OLEAUT32.DLL, and RTT32DLL.DLL.
- At program termination, the DllEntryPoint functions are not called for event DLL_PROCESS_DETACH. If this is required by an application, it should call FreeLibrary or RTDLLThreadEvent(NULL, DLL_PROCESS_DETACH) instead.
- In multithreaded applications, the DLL_THREAD_ATTACH and DLL_THREAD_DETACH events are not passed automatically to the DllEntryPoint functions. However, the application can explicitly call these with function RTDLLThreadEvent.
- The data segment of a DLL is not reloaded when a DLL is initialized more than once.

Loading DLLs through a File System

RTTarget-32 can load DLLs as files in addition to loading them from the program image. This feature can be useful for applications using RTFiles-32, which need to dynamically load only a subset of available DLLs.

The actual DLL file format used by RTTarget-32 is not identical (but similar) to the PE format used by Win32. Rather, DLM (Dynamically Loadable Modules) are used. Command line utility MakeDLM can convert a standard Win32 DLL to a DLM. It's command line syntax is:

```
MakeDLM [Options] DLLName
```

The following options are available:

- c[+|-] Compression, default is on. This option controls whether the DLM should be compressed. Typically, compression reduces the file size by a factor of 2.
- q[+|-] Quiet, default is off. Controls whether compression statistics should be displayed.
- g[+|-] Debug symbol conversion, default is on. Controls whether MakeDLM should prepare debug symbol tables for RTD32. Symbol table conversion is only required for Microsoft and Watcom compilers. Disabling this option can speed up MakeDLM if you do not need to debug DLMs.

Parameter DLLName must be the name of the DLL to process. If no filename extension is given, .DLL is assumed. The file is first searched in the default directory, then in the directory MakeDLM is loaded from. The resulting DLM file will reside in the same directory as the original DLL with filename extension .DLM.

Please note that program MakeDLM is **not** redistributable. It may only be used by RTTarget-32 license owners. For information on distributing MakeDLM with your applications to your customers, please contact On Time.

The use of compression for DLMs entails a tradeoff between disk space efficiency, load time, and memory requirement at run-time. Obviously, compressed DLMs will always need much less disk space. Compressed DLMs must be decompressed at run time. For fast disks and/or slow CPUs, compression will slow down DLM load time. However, for slow disks (e.g., diskettes) and/or fast CPUs, compressed

DLMs will load faster. Another issue is that LoadLibrary will need some temporary storage to expand the DLM. Depending on the installed memory management options, this temporary storage may not be available for subsequent allocations of the program. Please refer to the following table.

	Fixed Memory Manager	Virtual Memory Manager
Run-Time System Heap Manager	The temporary storage is returned to the Win32 default heap and is not available to allocations through malloc or new. It is available for allocations through HeapAlloc only.	The temporary storage is decommitted and deallocated and can be reused by any allocation method.
RTTHEAP	The temporary storage is returned to the Win32 default Heap and is available to allocations through malloc, new, and HeapAlloc.	The temporary storage is decommitted and deallocated and can be reused by any allocation method.

The rows and columns specify the memory manager and heap manager used by your program at run time.

Win32 API function LoadLibrary should always be passed the original DLL's file name; it will automatically convert extension DLL to DLM if the DLL is to be loaded as a file. LoadLibrary first checks whether the DLL is loaded already; if so, its reference count is incremented. If it is not found, LoadLibrary attempts to open the corresponding .DLM file. The file is first searched in the default directory, then in all directories given in the PATH environment variable (see RTLoc command SET on how to specify a path on the target). If the file is found, it is loaded. If no file is found, LoadLibrary attempts to locate the DLL in the program's image. Thus, a program image can contain a DLL which can be replaced with a DLM file without rebuilding the program image.

DLLs loaded as DLM files cannot be statically referenced by the program EXE or any DLL the main EXE statically references, either directly or indirectly. DLMs can **only** be loaded through LoadLibrary. However, a DLM can reference other DLMs and DLLs of the program image. A single call to LoadLibrary can load any number of DLLs through the file system or from the program image.

LoadLibrary does not support imports or exports by ordinal. The only exception are ordinal imports of DLLs linked into the program image that have been resolved through RTLoc's LINK command.

When LoadLibrary loads a DLM, it will require approximately 2k of stack space. The application should ensure that enough stack space is available.

The maximum number of DLLs which can be loaded simultaneously is 31. This limit applies to all DLLs, regardless of how they were loaded.

DLLs loaded as DLM files can be debugged with RTD32 just like DLLs loaded from the program image. However, the debugger will need access to the original DLL and to the associated debug symbols on the host. For Borland compilers, the symbol tables can reside within the DLL or in a separate TDS file. To reduce the memory requirements of a DLM at run time, it is recommended to move the debug symbol tables to a TDS file using Borland's command line utility TDSTRP32 prior to running MakeDLM. For Microsoft and Watcom compilers, the TDS file is by default generated by MakeDLM.

Demo program DLLDEMO3 shows how DLMs can be used. DLLDEMO3 does not actually need RTFiles-32. Instead, it loads a DLM file through RTTarget-32's RAM file file system driver.

RTTarget-32 offers alternate methods to use DLLs: either linking DLLs into the program image described in the previous section using the DLL command in the RTLoc configuration file, or by loading them as DLM files. Both methods have advantages and disadvantages:

Advantages of DLMs

- DLLs which may not always be required do not use up any memory on the target.
- DLMs can be replaced through the file system without the need to rebuild the application image.

Disadvantages of DLMs

- A DLL loaded as a DLM will usually need more memory on the target because all of its sections are always loaded. With RTLoc, only those sections actually required at run time will be included using the Locate NTSection or Locate Section commands. Examples of unneeded information are relocation tables, debug symbol tables, etc.
- No checks for static DLL references. RTLoc will verify that all static imports/exports can be matched up. If exports are missing, an error will be issued. In addition, the Link command can be used to change the standard method of matching imports against exports. However, DLMs loaded by LoadLibrary cannot do this; when exports are missing, LoadLibrary will simply fail.
- The application has no control over how DLMs are mapped in memory. LoadLibrary will allocate the required address space through VirtualAlloc. For example, program code of a DLM cannot be placed in ROM. When DLMs are frequently loaded and unloaded, heap fragmentation can become a problem. However, heap fragmentation can be minimized by using RTTarget-32's virtual memory manager by calling:

```
RTSetFlags(RT_MM_VIRTUAL, 1);
```

from an Init routine. For even better memory management, RTTarget-32's alternate heap manager RTTHEAP and the use of paging is strongly recommended.

Installable File System

This section is only relevant for applications which need to interface with a third-party file system or if you need to change the default file system configuration of RTTarget-32 (or RTFiles-32, if applicable).

RTTarget-32 can make its Win32 API emulation for file I/O functions available even for files it cannot handle. This is achieved through installable file I/O drivers. RTTarget-32 is shipped with drivers for console files, LPT files, and RAM files. For example, On Time's product RTFiles-32 adds a file I/O driver for files on devices with a FAT file system.

A file system driver consists of structure RTFileSystemHandlers defined in RTTARGET.H. Its first member must contain the size of the structure. Members FileHandleType and FindHandleType must be -1. The rest of the structure consists of function pointers which will handle the various file I/O operations. Functions which are not required or not applicable to a particular file system can be set to NULL. In this case, RTTarget-32 will simply return an appropriate error when such a function is called.

Each file I/O function has a file name or a file handle as a parameter. RTTarget-32 uses this parameter to determine which file system (if several are installed) should handle the request. For file handles, it will simply pass the request to the file system which created the handle. For file names, two different criteria are used. First, the type of file is determined (e.g., disk file, console, communication device, etc.). For disk files, the desired drive letter is also determined.

For each installed driver, RTTarget-32 knows which file types and which logical drives it can handle and will dispatch the call to the appropriate driver accordingly.

The file system drivers do not deal with Win32 file handles. RTTarget-32 requires that functions CreateFile and FindFirstFile return unique 32-bit values which will be passed to subsequent handle-based functions. RTTarget-32 will automatically allocate true Win32 handles for such values, to be passed back to the application. All functions of the driver must otherwise behave just like the corresponding Win32 functions. This includes setting an appropriate error value using SetLastError if the function fails.

The CloseFile and FindClose functions of the driver do not use the Win32 __stdcall calling conventions, but RTTAPI (__cdecl) instead.

The list of available file I/O drivers is retrieved from global variable RTFileSystemList:

```
typedef struct {
...    // see RTTARGET.H
} RTFileSystemHandler;
```

```
typedef struct {
    DWORD Flags;
    DWORD Drives;
    DWORD PhysicalDisks;
    RTFileSystemHandlers * Handlers;
} RTFileSystem;

// file systems shipped with RTTarget-32. All are installed by default
extern RTFileSystemHandlers RTConsoleFileSystem;
extern RTFileSystemHandlers RTRAMFileSystem;
extern RTFileSystemHandlers RTLPTFileSystem;
```

The default file system installation is:

```
static RTFileSystem Console =
{ RT_FS_CONSOLE, 0, 0, &RTConsoleFileSystem };

static RTFileSystem LPTFiles =
{ RT_FS_LPT_DEVICE, 0, 0, &RTLPTFileSystem };

static RTFileSystem RAMFiles =
{ RT_FS_FILE | RT_FS_IS_DEFAULT, 0x00000004, 0, &RTRAMFileSystem };

RTFileSystem * RTFileSystemList[] =
{
    &Console,
    &LPTFiles,
    &RAMFiles,
    NULL
};
```

This configuration assigns drive letter 'C' to the RAM file system, which holds the initial default directory. The console and printer file systems cannot handle logical drives.

The Flags field for each RTFileSystem can have a combination of the values given below. It indicates which file types are supported:

RT_FS_FILE	Standard data files.
RT_FS_CONSOLE	Console files CONIN\$ and CONOUT\$.
RT_FS_DISK_DEVICE	Device files such as \\.\A: or \\.\PHYSICALDRIVE0.
RT_FS_PIPE	Pipes such as \\.\pipe\name.
RT_FS_MAILSLLOT	Mailslots such as \\.\mailslot\name.
RT_FS_NET_FILE	Remote network files such as \\servername\name.
RT_FS_COM_DEVICE	COM devices such as COM1 or \\.\COM1.
RT_FS_LPT_DEVICE	LPT devices such as LPT1 or \\.\LPT1.
RT_FS_IS_DEFAULT	This flag indicates that the initial default directory is held by this driver.

The Drives field is used for file systems which support data files. Each bit set indicates a logical drive supported by this driver. Bit 0 corresponds to drive 'A', bit 1 corresponds to drive 'B', etc. RTTarget-32's file I/O emulation supports up to 32 logical drives.

The PhysicalDisks field is used for file systems which support disk device files. Each bit set indicates a physical disk supported by this driver. Bit 0 corresponds to drive 'A', bit 1 corresponds to drive 'B', etc. RTTarget-32's file I/O emulation supports up to 32 physical disks.

Multithread Applications

RTTarget-32 does not contain a scheduler, but it does support RTKernel-32, On Time's real-time multitasking kernel. In multitasking applications, the following issues must be considered:

- RTTarget-32 implements a number of Win32 thread API functions as dummy functions. These must be replaced by the multitasking system. This is achieved by linking the multitasking library **before** the RTTarget-32 library.

- If DLLs are used, both the RTTarget-32 and the multitasking library must reside in the same EXE or DLL. Thus, RTTarget-32's predefined system DLL RTT32DLL.DLL cannot be used. You must create your own custom system DLL as in example programs DLLDemo2 and DLLDemo3.
- If DLLs are used, the DLL_THREAD_ATTACH and DLL_THREAD_DETACH events are not passed to the DllEntryPoint functions automatically. If you are using DLLs which require these calls, each thread using such DLLs must explicitly call function RTDLLThreadEvent.
- RTTarget-32's Win32 memory management functions are thread safe (they are protected with a critical section). However, RTTarget-32's memory mapping functions (RTFindPhysMem, RTReserveVirtualAddress, RTReleaseVirtualAddress, RTMapMem, RTExtendHeap, and RTCMOSEExtendHeap) are not. If any of these function can execute simultaneously with other memory management or memory mapping functions, they must be protected with function RTLockHeap and RTUnlockHeap. However, in most applications, they can be executed before any threads are created to avoid reentrance problems.

Please consider that the run-time systems (in particular the run-time system heaps) might not be thread safe. Please refer to your compiler's documentation and the multitasking system's documentation for information about how to solve reentrance problems in this area.

Using the MetaWINDOW Graphics Library

With a few run-time restrictions, the graphics library MetaWINDOW by MetaGraphics - which was originally designed for DOS and DOS extenders - can be used with RTTarget-32. However, to use MetaWINDOW, it is important to understand how the support for it has been implemented. For professional GUIs, On Time RTOS-32 component RTPEG-32 is recommended instead of MetaWINDOW.

MetaWINDOW performs BIOS calls to communicate with the graphics adapter, in particular to change the graphics mode. However, RTTarget-32 is a pure protected mode environment which does not support such BIOS calls (the PC BIOS consists of 8086 real mode code which cannot be executed in protected mode).

To overcome this problem, RTTarget-32 can set a desired graphics mode during the boot process before switching to protected mode. This is achieved with the GMode directive (see *Chapter 3, GMode Command*). At run-time, RTTarget-32's function RTGetGMode() can be used to enquire which mode has been set.

This mechanism is used to place the display hardware in graphics mode. To prevent the MetaWINDOW library from performing int 10h calls, a dummy int 10h handler is installed which will trap all such calls. The handler simulates success, but actually does nothing except to pass some diagnostics information to the host debugger (if desired).

Prerequisites

To successfully run graphics programs under RTTarget-32, the following conditions must be satisfied:

- The target computer must have a BIOS and a graphics display adapter.
- The first 4k of physical address space must not be used, because MetaWINDOW needs to access the BIOS data area located here. Reserving this memory area is best achieved with the following directive:

```
Region BIOSMem 0 4k RAM NoAccess
```

- Any graphics video memory required for the graphics modes to be supported must have read/write access. Example:

```
Region ColorGraphic A0000h 64k Device ReadWrite
Region MonoText     B0000h 32k Device ReadWrite
Region ColorText     B8000h 32k Device ReadWrite
```

- The RTTarget-32 configuration file used to build the program containing the boot code (either the Debug Monitor or the graphics application itself) must contain directive

```
VideoRAM = None
```

to suppress text mode style screen I/O. All text mode screen output will be sent to the host debugger (if present). (Note: you can re-enable text mode I/O at run-time by supplying your own OutCharHandler, see Chapter 7, *Function RTDisplayChar*).

- The same configuration file must contain a GMode directive with at least one BIOS graphics mode to be set. Example for VGA 640x480, 16 colors:

```
GMode 12h
```

- All required MetaWINDOW driver DLLs must be included in the program image with the *DLL* command. Example:

```
DLL \MetaWindow\metwnd05.dll
```

The driver DLLs can also be loaded as DLMs as done by the example programs.

- All required font files must be mapped with Locate File commands. Example:

```
Locate File \MetaWindow\fonts\system00.fnt HighMem
```

- The program's source code should #include header file RTMETAW.H supplied with RTTarget-32 and function RTMetaWInit() must be called before any MetaWINDOW function is called (see below).
- The program must be linked with the RTTarget-32 library RTT32.LIB, RTMETAW.LIB and a suitable MetaWINDOW library (e.g., one for DPML-32).
- Selector 7Bh (or 78h at CPL 0) is reserved for MetaWINDOW and must not be used by the application.
- If the graphics program also uses RTKernel-32, you **must** use the CPU386 driver and **not** CPU386F which is the default. This is very important! MetaWINDOW frequently changes segment registers. If CPU386F is used, random and evasive to debug program crashes will occur.

Initialization

RTMetaWInit() must be called to initialize MetaWINDOWS. Example:

```
int main(void)
{
    int    i;
    printf("Current BIOS video mode: %04X\n", RTGetGMode());
    i = RTMetaWInit(RT_METAW_INIT_GRAPHICS);
    RTIdleHandler = RTGetMetaWEvents; // update event queue
    if (i != 0)
    {
        printf("MetaWINDOW InitGraphics error - %d\n", i);
        exit(1);
    }
    SetDisplay( GrafPg0 );      /* switch display to graphics mode */
    ...
}
```

Limitations

Several features of MetaWINDOW are not available under RTTarget-32, mainly because no BIOS calls are possible at run-time. In particular, these are the limitations:

- Mode switches are not possible at run time. The hardware is placed to a particular mode at boot time which cannot be changed later.
- Only display page 0 (GrafPg0) can be used and switching pages is not supported.
- Functions QueryGraphics() and FindBestGraphics() are not supported.
- Functions QueryMouse() and FindBestMouse() are not supported.
- The mouse drivers MsDriver and Joystick are not supported. However, MsCOM1, MsCOM2, MoCOM1, and MoCOM2 are supported.

Function RTMetaWInit

RTMetaWInit must be called once before the MetaWindow library can be called. The function's prototype is defined in Rtmataw.H as follows:

```
int RTMetaWInit(unsigned long Flags);
```

The Flags parameter may have any combination of the following values:

- | | |
|------------------------|---|
| RT_METAW_KEYSTOMETAW | Keys-To-MetaWINDOW. It instructs RTTarget-32 to send all key events to MetaWINDOW's event queue. This is done by the keyboard interrupt handler and is therefore not recommended. The additional processing load for the keyboard interrupt handler is severe and can degrade the interrupt latency of the system significantly. A better approach is to call RTGetMetaWEvents in a loop whenever the program expects user input. |
| RT_METAW_SHOW_INT10 | This flag is supplied for debugging purposes. If set, all int 10h calls performed by MetaWINDOW are displayed if the program runs under the Debug Monitor. Use this flag if you encounter problems selecting a particular graphics mode or if you suspect that MetaWINDOW uses the BIOS for other graphics operations. |
| RT_METAW_INIT_GRAPHICS | This flag will map a linear frame buffer, call InitGraphics(), and configure MetaWINDOW to use the linear frame buffer. When this flag is used (which is always recommended), RTMetaWInit() will return the return code of MetaWINDOW's InitGraphics() function. |

Function RTGetMetaWEvents

This function will transfer all currently pending keyboard events from RTTarget-32's Win32 compatible event queue to MetaWINDOW's event queue.

```
void RTGetMetaWEvents(void);
```

It is not required to call this function if flag RT_METAW_KEYSTOMETA was specified in the call to RTMetaWInit. Typically, you should call this function when the program is waiting for user input.

Using the 387 Emulator

RTTarget-32 contains the 387 Floating Point Emulator RTEmu. Although RTEmu is bundled with RTTarget-32, it is a separate product with different licensing terms.

This emulator has been derived from the DJGPP emulator EMU387 version 1.12, Copyright (c) DJ Delorie, 24 Kirsten Avenue, Rochester, NH 03867-2954, USA, email dj@ctrn.com. DJGPP is released under the GNU General Public Licensing terms, which also apply to this emulator. Please read the complete licensing terms given later in this section before considering the use of this emulator.

For Borland C++, Microsoft Visual C++, and Delphi, two versions of the emulator are supplied: a single-thread version and a multi-thread version. The single-thread version can be used in a multitasking system only if the emulator's context is swapped with FNSAVE/FRSTOR in every task switch. The multi-thread version is completely reentrant on the task level and requires no swapping. However, it uses TLS data, which, unfortunately, is not supported by Watcom. The reentrant emulator is compatible with RTKernel-32's floating point driver FLTEMUMT.LIB.

Both emulator versions are non-reentrant for interrupt handlers. If you intend to do floating point calculations in interrupt handlers, FNSAVE/FRSTOR are required to preserve the foreground thread's floating point context.

The emulator never disables interrupts. Thus, it is well-suited for real-time systems requiring a low interrupt latency.

In addition to the emulator, source file `FPEXH.C` contains a sample exception handler for 387 exceptions. It can be used for the emulator as well as for actual floating point hardware. It traps int 16 and displays some information about the exception on the screen. Then it continues, allowing the FPU or emulator to take its default action. `FPEXH.C` is recommended only for debugging. Real applications should mask all floating point exceptions.

Linking the Emulator in C/C++ Programs

Since the run-time systems contain floating point instructions in their startup code, the emulator should be installed before the startup code executes. This can be accomplished using RTTarget-32's `INIT` directive in the application's RTTarget-32 configuration file. Example:

```
INIT _RTEmuInit
```

or, for Microsoft C:

```
INIT RTEmuInit
```

Function `RTEmuInit` is exported by module `RTEMU.OBJ`, which must be linked to the program. It installs the emulator's exception 7 handler as a trap gate. If you must call several functions at init time, you must write your own exported `Init` function, which in turn calls all `Init` functions required by your program.

Applications which can guarantee that no floating point operations are required by application code before `main()` is called (e.g., in constructors of global objects, DLL initialization code, etc.), can alternatively call `RTEmuInit()` explicitly in function `main()`, followed by `freset()`.

Apart from module `RTEMU.OBJ`, you must also link `RTEMU.LIB` (single-thread) or `RTEMUMT.LIB` (multithread). Please note that the multithread library requires TLS data. If you use `RTKernel-32`, you must select an appropriate floating point driver. For `RTEMUMT.LIB`, use driver `FLTEMUMT`; for `RTEMU.LIB`, use driver `FLT387`.

There are no source code modifications required to use the emulator. You only need to link it (`RTEMU.OBJ` + `RTEMU.LIB` or `RTEMUMT.LIB`) and make sure it is initialized through the RTTarget-32 `INIT` directive. If you use the multithread version, be sure to include all sections generated for TLS by the compiler and linker.

If you want to use the floating point exception handler included with the emulator, install it using the following call:

```
#include <rttarget.h>

int main(void)
{
    RTFPInstallExHandler();
    ...
}
```

To make sure you actually see all exceptions, unmask all FPU exceptions in the development phase of your project with:

```
_control87(0, 0xFFFFFFFF);
```

Demo program `EmuDemo` shows how to use the emulator.

Linking the Emulator in Delphi Programs

Using the emulator in Pascal programs is much simpler. Just add unit `RTEmu` (or `RTEmuMT` for multithread support) to the main program's `USES` clause. The emulator will be installed automatically by the unit's initialization code.

Only if floating point instructions are executed at an earlier stage (e.g., in the initialization code of a statically referenced DLL), you must call `RTEmuInit` as an `Init` function. To do this, add

```
Init RTEmuInit
```

to your program configuration file.

Demo program `EmuDemo` shows how to use the emulator.

Emulator Licensing Terms

Although you have obtained this floating point emulator bundled with RTTarget-32, it is a separate product with different licensing terms.

On Time Informatik GmbH and the original author of the emulator, DJ Delorie, supply this software without any warranty.

The emulator is made available to all RTTarget-32 users free of charge. It may be further distributed under the terms of the GNU General Public License, with the following exceptions:

- You may distribute this emulator programming library only if you also make available its source code under the GNU General Public Licensing terms.
- Any existing copyright or authorship information in any given source file must remain intact. If you modify a source file, a notice to that effect must be added to the authorship information in the source file.
- You can distribute executable applications linked with the emulator library without restrictions. In particular, there is no need to supply any object, library, or source files or references to the emulator's authors or the GNU licensing terms with such executables.

Please refer to file Source\Emu\Copyping for the GNU General Public License.

Appendix A

Compiling and Linking with On Time RTOS-32

This appendix describes how to compile and link embedded systems programs. Whenever applicable, special considerations for a particular compiler are detailed. All compiler options apply identically to the command line tools and respective IDEs.

Complete examples are given in directories Demobc, Demomsvc, Demowat, and Demodel. Examples for compiling/linking programs with Microsoft's Visual Studio and Visual C++ 6.0 or higher are included under directory Demomsdev.

General Rules

This section describes some general considerations to keep in mind for compiling and linking On Time RTOS-32 applications.

- On Time RTOS-32 .EXE and .DLL files are Win32 Console Mode executables and are compiled and linked as such.
- The startup code and run-time libraries for Win32 Console applications supplied with the supported compilers are used in unmodified form.
- If you do **not** intend to use the compiler supplied run-time system (for example, to save memory), RTTarget-32's custom startup code C0RTT.OBJ (for EXE files) or C0RTTD.OBJ (for DLLs) is used as a replacement for the compiler-supplied startup code. This option is not available for Delphi programs. RTTarget-32's custom startup code is not compatible with the run-time systems and cannot be used if you need the run-time system.
- An application consists of a single .EXE. In addition, it can use up to 31 DLLs.
- The On Time RTOS-32 libraries must be linked to one and only one EXE or DLL of the application. All required On Time RTOS-32 libraries must reside in the same EXE or DLL.
- EXEs or DLLs which **do not** contain On Time RTOS-32 libraries and which **do not** need access to the On Time RTOS-32 native APIs are compiled and linked **exactly** as standard Win32 Console Mode modules. They can execute unmodified both under Win32 and On Time RTOS-32. No On Time RTOS-32 specific libraries need to be linked.
- EXEs or DLLs which **do not** contain On Time RTOS-32 libraries, but **do need** to access the On Time RTOS-32 native API can call such functions if
 - the EXE or DLL containing On Time RTOS-32 exports the required API functions;
 - they #define RTT32DLL before including RTTARGET.H or other On Time RTOS-32 header files.
 - they link an import library of the EXE or DLL containing the On Time RTOS-32 libraries.

Delphi programs always use import units to access the On Time RTOS-32 native APIs contained in RTT32DLL.DLL or a custom built system DLL.

Order of Libraries

The order of libraries can be very important. For a program or DLL to link directly against the On Time RTOS-32 libraries, the recommended order of libraries is as follows:

Object files of the application	Typically, the startup code and code of the application.
387 Emulator init object file	RTEMU.OBJ (only if the 387 emulator is needed).

Libraries of the application	If you have any application libraries, link them here.
RTPEG-32 library	PEG.LIB only if a GUI is required.
RTFiles-32 library	RTFILES.LIB only if a file system is required.
RTFiles-32 driver	Needed only if RTFILES.LIB is linked. RTFSRTT.LIB for single-threaded applications or RTFSK32.LIB for RTKernel-32 applications. DRVDOC.LIB must be linked if M-Systems DiskOnChip support is required.
RTKernel-32 library	Only required for multithreaded applications. RTK32.LIB for the RTKernel-32 Debug Version and RTK32S.LIB for the Standard Version.
RTKernel-32 drivers	Only required for multithreaded applications. DRVRT32.LIB or any set of alternate drivers followed by DRVRT32.LIB.
RTT32.LIB	Always required. RTTarget-32's native API and Win32 emulation.
RTTHEAP.LIB	Always optional. This library replaces the run-time system's heap manager with RTTarget-32's heap manager.
387 Emulator library	Only required if the application needs floating point support on CPUs without FPU. RTEMU.LIB or RTEMUMT.LIB.
MetaWINDOWS support library	Only required if MetaWINDOWS is also linked. Library RTMETAW.LIB.
Compiler-supplied run-time system	Whatever your compiler documentation recommends for console mode applications. For RTKernel-32 programs, use the multithread run-time system or the singlethread run-time system in conjunction with Automatic Library Protection.

EXEs or DLLs which do not link the On Time RTOS-32 libraries but must still have access to the On Time RTOS-32 API should link an import library. See examples DLLDemo2 and DLLDemo3 for details.

The order of object files is not important. The linker will unconditionally link all object files regardless of their relative order, but object modules contained in libraries are only linked if they are referenced. Thus, if two libraries define the same referenced symbol, the first symbol encountered in the list of libraries is linked.

The following symbols are redefined by various On Time RTOS-32 libraries:

Symbols	Defined in	Comment
Win32 API	run-time system libraries	All Win32 API functions are defined in import libraries supplied with the compiler (e.g. KERNEL32.LIB, IMPORT32.LIB, etc.).
	RTT32.LIB	RTT32.LIB contains replacement implementations of these functions.
Win32 thread API	RTT32.LIB	Implemented as dummies (do nothing or program abort).
	RTK32[S].LIB	Real implementations.
GetTickCount	RTT32.LIB	Simple implementation using an interrupt handler.
	RTK32[S].LIB	Implementations using RTKernel-32's clock driver.
RTFileSystemList	RTT32.LIB	Includes Console, LPT, and RAM file systems.
	RTFILES.LIB	Includes Console, LPT, and disk (FAT) file systems.
malloc, free, realloc	run-time system	C/C++ heap management routines.
	RTTHEAP.LIB	RTTarget-32 alternate heap manager.

The following examples list the On Time RTOS-32 libraries in the recommended link order.

An application using only RTTarget-32, no floating point:

```
RTT32.LIB
```

An application using RTTarget-32 and RTKernel-32's Debug Version with RTTarget-32's alternate heap manager:

```
RTK32.LIB DRVRT32.LIB RTT32.LIB RTTHEAP.LIB
```

An application using RTTarget-32, RTPEG-32, RTKernel-32's Standard Version, Pentium High-Resolution timer driver, need for FPU emulation, DiskOnChip support, and RTTarget-32's alternate heap manager:

```
RTEMU.OBJ PEG.LIB RTFILES.LIB RTFKS32.LIB DRVDOC.LIB RTK32S.LIB HRTPEM.LIB  
CLKPC.LIB FLTEMUMT.LIB DRVRT32.LIB RTT32.LIB RTTHEAP.LIB RTEMUMT.LIB
```

Borland C++

On Time RTOS-32's Libbc and Include directories must be added to the compiler library and include paths.

Important (multithreaded programs only): If you plan to use RTKernel-32's Automatic Library Protection instead of the multithreaded run-time library, you must install Automatic Library Protection first and make sure the linker searches On Time RTOS-32's Libbc directory before the compiler's Lib directory. In addition, you must specify compiler command line option `-WM-`. If you plan to use the multithread run-time libraries, use command line option `-WM+`.

Assuming On Time RTOS-32 is installed in directory C:\ONTIME, the example program HELLO.C can be compiled and linked using:

```
bcc32 -v -IC:\ONTIME\Include -LC:\ONTIME\Libbc hello.c rtt32.lib
```

If you prefer to call the linker explicitly, HELLO.C can be compiled and linked in two separate steps:

```
bcc32 -c -v -IC:\ONTIME\Include hello.c  
tlink32 -v -c c0x32.obj hello.obj, hello.exe, hello.map,  
C:\ONTIME\Libbc\rtt32.lib cw32.lib
```

ILINK32 can be used instead of TLINK32.

Demo program HELLO2.C demonstrates how to eliminate the run-time system using RTTarget-32's startup code C0RTT.OBJ. It can be compiled using:

```
bcc32 -c -v -IC:\ONTIME\Include hello2.c
```

and linked (enter on one line):

```
tlink32 -c -v C:\ONTIME\libbc\c0rtt.obj hello2.obj,  
hello2.exe, C:\ONTIME\Libbc\rtt32.lib
```

If RTT32DLL.DLL is used instead of RTT32.LIB, any EXE or DLL which needs to call RTTarget-32 native API functions must link import library RTT32DLL.LIB. RTT32DLL.LIB is not required to call Win32 emulation functions.

PE files produced by Borland C++ always contain two sections which must be located: section CODE and DATA. If Borland's incremental linker ILINK32 is used, these two sections are named `.text` and `.data`, respectively. Section CODE/.text contains the executable part of the program, requires only read-only access, and corresponds to segment number 1 in the map file. Section DATA/.data needs read/write access, contains both initialized and uninitialized global data, and corresponds to segment number 2 in the map file. If you want to debug the program, *Locate NTSection* commands must be used; otherwise, most programs should also work with command *Locate Section*. If the program uses statically referenced DLLs, section `.idata` must also be included. If exported functions must be available at run time using `GetProcAddress`, the `.edata` section is also required. If the run-time system is used, a heap with at least 64k is required. Otherwise, no heap is needed. A stack of at least 16k is recommended.

If the program contains TLS (thread) variables, sections `.tls` and `.rdata` must also be included in the program image.

If the program needs to access resources of the PE file at run-time, section `.rsrc` is also required.

A typical configuration file for a program suited for debugging is:

```
#ifsection .text          // redefine some section names for ILINK32
#define CODE .text
#define DATA .data
#endif

FillRAM HighMem

#ifndef BOOT
    Reserve Monitor
#endif

Locate  Header      Header      HighMem
Locate  PageTable   PageTable   HighMem

Locate  NTSection   CODE        HighMem    // program code
Locate  NTSection   DATA       HighMem    // program data
Locate  NTSection   .tls        HighMem    // needed for TLS data
Locate  NTSection   .rdata      HighMem    // needed for TLS data

Locate  Stack       Stack       HighMem 16k
Locate  Heap        Heap        HighMem
```

For a program which does not use the run-time system, contains no TLS data, and does not require debugging, you could use:

```
Locate  PageTable   PageTable   LowMem
Locate  Section     CODE        HighMem
Locate  Header      Header      HighMem 0 4
Locate  Section     DATA       HighMem
Locate  Stack       Stack       HighMem 16k 4
```

Unfortunately, the 32-bit linker integrated in the Borland C 4.5 IDE has some severe bugs. Therefore, it is strongly recommended to use the command line compiler or the command line linker for linking. However, if you prefer to use the integrated linker, always enable debug information. There is no run-time overhead if debug information is included in a PE file. If desired, the debug information can be removed after linking using command line utility TDSTRP32.EXE.

To enable source level debugging with RTD32, option -v should be added to BCC32/TLINK32/ILINK32 command lines.

Please refer to the sample make files in all directories under Demobc for numerous examples of building programs for On Time RTOS-32.

Microsoft Visual C++

On Time RTOS-32's Libmsvc and Include directories must be added to the compiler's library and include file search paths.

Microsoft's 32-bit linker has one serious bug: it does not strictly follow the order of libraries specified on the linker command line. Usually, C/C++ linkers will search for a symbol required by a program in all libraries in the order the libraries are supplied. With Microsoft's linker, the first occurrence of the symbol **after its reference** is used. Since RTKernel-32 and RTTarget-32 replace symbols usually present in run-time or Win32 system libraries, this can lead to the wrong modules being linked if that symbol is not referenced by any of the application's object file.

For example, library LIBC.LIB of Visual C++ contains the function malloc and also references to malloc. However, in On Time RTOS-32 programs, this function will usually be supplied by RTTHEAP.LIB, which is linked before LIBC.LIB. If the application itself does not reference malloc, the wrong version of malloc will be linked.

Consider the following libraries linked in the given order:

```
RTK32.LIB
DRVRT32.LIB
RTT32.LIB
RTTHEAP.LIB
LIBC.LIB
```

None of the On Time RTOS-32 libraries contain references to `malloc`, so LINK will not look for it until `LIBC.LIB` is parsed. `malloc` is located in `RTTHEAP.LIB`, but LINK missed that definition. Instead, it will link the version of `malloc` defined in `LIBC.LIB`.

To solve this problem, it is recommended to force the inclusion of symbols redefined by `RTFiles-32`, `RTKernel-32`, or `RTTarget-32` on LINK's command line using the `/include` option. It is sufficient to include `_RTFileSystemList`, `EnterCriticalSection@4`, and `_malloc`. The only reason not to reference these symbols would be programs known not to require them (e.g., programs not using a standard run-time system).

Library `KERNEL32.LIB` should be excluded with option `/nodefaultlib:kernel32.lib` to ensure that all required Win32 functions will be emulated by `RTKernel-32` or `RTTarget-32`.

Assuming On Time RTOS-32 is installed in directory `C:\ONTIME`, the example program `HELLO.C` can be compiled and linked using the following command line (enter on one line):

```
cl /IC:\ONTIME\Include /LC:\ONTIME\Libmsvc /Fm /Zi hello.c rtt32.lib
/link /nodefaultlib:kernel32.lib /fixed:no
```

If you prefer to call the linker explicitly, you can compile and link the program in two separate steps:

```
cl /c /IC:\ONTIME\Include hello.c
link /fixed:no /nodefaultlib:kernel32.lib hello.obj C:\ONTIME\Libmsvc\rtt32.lib
```

Demo program `HELLO2.C` shows how to eliminate the run-time system with `RTTarget-32`'s startup code `C0RTT.OBJ`. It can be compiled and linked using the following command line (enter on one line):

```
cl -I..\include hello2.c /link /entry:Start
..\libmsvc\c0rtt.obj ..\libmsvc\rtt32.lib
/fixed:no /map /nodefaultlib:kernel32.lib
```

If `RTT32DLL.DLL` is used instead of `RTT32.LIB`, any EXE or DLL which needs to call `RTTarget-32` native API functions must link import library `RTT32DLL.LIB`. `RTT32DLL.LIB` is not required to call Win32 emulation functions.

Starting with Visual C++ 5.0, the Microsoft linker will not write a fixup table to .EXE files by default. Because such programs cannot be processed by `RTLloc`, linker option `/fixed:no` is required starting with Visual C++ 5.0.

PE files generated by Microsoft Visual C++ contain four sections that must be located: sections `.text`, `.bss`, `.rdata`, and `.data`. Section `.text` contains the executable part of the program and only requires read-only access. Section `.bss` is only created by Visual C++ versions up to 2.2, needs read/write access, and contains uninitialized global data. Section `.rdata` contains read-only data. Section `.data` contains initialized data and needs read/write access. Visual C++ up to version 4.2 places the export table in section `.edata`; version 5.0 in `.rdata`.

If the program contains TLS (thread) variables, section `.tls` must also be included in the program image.

If the program needs to access resources of the PE file at run-time, section `.rsrc` is also required.

The Microsoft linker writes very precise information into PE files. Therefore, it is not necessary to use `.MAP` files to truncate unused information, and there is usually no need to specify any size information in the `Locate NTSection` or `Locate Section` commands. The default alignment is 4k, which is also `RTLloc`'s default. Therefore, using command `Locate NTSection` is recommended instead of `Locate Section`. The latter should only be used if several sections must be located into different memory regions and you do not want to use virtual regions.

If the run-time system is used, a heap with at least 64k is required by Visual C++ 2.0/2.2; Visual C++ 4.0/4.1 may run with less than 32k. If no run-time system is used, no heap is needed. A stack with at least 16k is recommended.

A typical configuration file for a Visual C++ program is:

```
FillRAM HighMem
#ifdef BOOT
  Reserve Monitor
#endif

Locate PageTable PageT LowMem
Locate Header Header LowMem

Locate NTSection .text HighMem // code section
Locate NTSection .rdata HighMem // read only data
Locate NTSection .data HighMem // read/write initialized data
#ifdef section .bss // not generated by all linker versions
  Locate NTSection .bss HighMem // uninitialized data
#endif
#ifdef ifsection .tls // only in multithreaded apps
  Locate NTSection .tls HighMem
#endif

Locate Stack Stack HighMem 16k
Locate Heap Heap HighMem
```

For a program that doesn't use the run-time system, the *Locate Heap* command can be omitted.

To enable source level debugging with RTD32, compiler command line switch */Zi* is required. RTLoc will by default convert debug symbol tables to the format required by RTD32. You can disable this behavior with RTLoc option *-g-* (recommended if you plan to use the Visual Studio 6.0 debugger). If you plan to use RTTarget-32's debugger RTD32 with programs compiled within the Visual Studio IDE, linker debug option "Separated Types" and compiler option Debug Info "Program Database for Edit and Continue" must both be disabled.

If RTLoc option *-g-* is not specified, RTLoc requires a DLL of your Visual C++ installation to read the debug symbol tables. The name of this DLL is Visual C++ version dependent:

VC++ 2.x	DBI.DLL
VC++ 4.0	MSPDB40.DLL
VC++ 4.1/4.2	MSPDB41.DLL
VC++ 5.0	MSPDB50.DLL
VC++ 6.0	MSPDB60.DLL

It will usually reside in the same directory as your Visual C++ compiler or IDE. Please make sure that the required DLL is on your path or in RTTarget-32's BIN directory.

Please refer to the examples under directory Demomsvc for numerous examples of building On Time RTOS-32 programs using command line tools. Examples for the Visual Studio 6.0 IDE are given in directory Demomsdev.

Watcom C/C++

On Time RTOS-32's Libwat and Include directories must be added to the compiler's library and include file search paths. It is recommended to add them to the environment variables INCLUDE and LIB. Example:

```
SET INCLUDE=C:\ONTIME\INCLUDE;%INCLUDE%
SET LIB=C:\ONTIME\LIBWAT;%LIB%
```

The example program HELLO.C can then be compiled and linked with:

```
wcl386 -s -d2 -hc hello.c rtt32.lib
```

If you prefer to call the linker explicitly, you can compile and link the program in two separate steps:

```
wcc386 -s -d2 -hc hello.c
```

```
wlink system nt Debug CodeView option cvpack file hello.obj library rtt32.lib
```

Demo program HELLO2.C demonstrates how to eliminate the run-time system with RTTarget-32's startup code C0RTT.OBJ. It can be compiled and linked with:

```
wcl386 -s hello2.c C:\ONTIME\Libwat\c0rtt.obj rtt32.lib
```

Please note that compiler option `-s` is required to prevent the compiler from generating calls to the stack checking routines of the run-time system (which is not available).

If `RTT32DLL.DLL` is used instead of `RTT32.LIB`, any EXE or DLL which needs to call `RTTarget-32` native API functions must link import library `RTT32DLL.LIB`. `RTT32DLL.LIB` is not required to call Win32 emulation functions. Due to Watcom's lack of support for module definition files (DEF files), the import library in the `Libbbc` directory must be used.

Init functions are executed before the C/C++ startup code. The Watcom compiler by default generates calls to a stack check routine at the entry of each function. However, this stack check routine does not work correctly before the run-time system's initialization has run. Thus, all code which can be executed from an Init function must be compiled with stack checking off (command line option `-s`).

PE files produced by Watcom C++ always contain three sections that must be located: section `BEGTEXT` (respective `AUTO` for Watcom C++ starting with version 11.0), `DGROUP`, and `.bss`. `BEGTEXT/AUTO` contains the executable part of the program and requires read-only access. Section `DGROUP` needs read/write access and contains initialized global data. Section `.bss` contains uninitialized data. Both *Locate NTSection* and *Locate Section* commands work fine. If the run-time system is used, a heap of at least 64k is usually required. Otherwise, no heap is needed. A stack of at least 16k is recommended.

The Watcom C++ linker generates separate sections for initialized and uninitialized data, but both sections are reported to reside in one segment in the map file. Therefore, `RTLloc` cannot determine the section sizes from the map file. You should not specify the size parameter in a *Locate Section* or *Locate NTSection* command.

A typical configuration file for a Watcom C/C++ is:

```
FillRAM HighMem

#ifdef BOOT
    Reserve Monitor
#endif

#ifsection AUTO                // for Watcom C 11.0
    #define BEGTEXT AUTO        // this section is renamed
#endif

Locate PageTable PageT LowMem 12k
Locate Header Header LowMem

Locate NTSection BEGTEXT HighMem
Locate NTSection DGROUP HighMem
Locate NTSection .bss HighMem

Locate Stack Stack HighMem 16k
Locate Heap Heap HighMem
```

For a program that doesn't use the run-time system, the command *Locate Heap* can be omitted.

To enable source level debugging for Watcom C programs, `wcl386` command line options `-d2` (full debug info) and `-hc` (CodeView style debug info) must be used. For linker `wlink`, use command "Debug CodeView". Unfortunately, the debug symbol table generator of Watcom contains a few bugs which may cause `wcl386` or `wlink` to fail with a `CVPACK` error. In this case, disabling debug info for some modules frequently helps.

`RTLloc` will by default convert debug symbol tables to the format required by `RTD32` (you can disable this behavior with `RTLloc` option `-g-`). The Watcom compiler does not generate debug symbols for code source lines located in `#include` files and for register variables. Turning off optimization can work around the latter problem.

Please refer to the numerous examples under directory `Demowat` on how to compile and link On Time RTOS-32 programs.

Borland Delphi

On Time RTOS-32 applications are compiled and linked as Win32 Console mode applications. Since the Delphi compiler by default generates GUI programs, command line option `-CC` must be specified.

Programs using any of On Time RTOS-32's import units located in directory `Libdel` must be compiled with command line option `-U` to enable the compiler to find these units. Assuming On Time RTOS-32 has been installed in directory `C:\ONTIME`, the example program `HELLO.PAS` can be compiled and linked using:

```
DCC32 -v -CC -UC:\ONTIME\Libdel hello.pas
```

Delphi programs cannot directly link any LIB files shipped with On Time RTOS-32. Instead, a system DLL containing these libraries must be used. For applications not using `RTKernel-32` or `RTFiles-32`, a preconfigured system DLL named `RTT32DLL.DLL` is included in the `Bin` directory. It contains `RTTarget-32`'s complete API except function `RTMakeBootDisk` (which requires `RTFiles-32`). If you need additional functions from `RTKernel-32` or `RTFiles-32` or if you want to link a smaller subset version of `RTT32DLL.DLL`, you must create a custom system DLL using a linker which can process `.LIB` files (e.g., `TLINK32` or `ILINK32` shipped with Borland C++ or Borland C++ Builder). Please refer to Chapter 9, section *Using a Custom RTTarget-32 System DLL* for details on how custom system DLLs are linked.

PE files produced by Delphi always contain six sections which must be located: section `CODE`, `DATA`, `BSS`, `.tls`, `.rdata`, and `.idata`. If exported functions are to be available at run time using `GetProcAddress`, the `.edata` section is also required. *Locate NTSection* commands must be used to build Delphi programs. A heap with at least 32k is required. A stack of at least 16k is recommended. The linker of Borland C++ Builder names section `CODE` and `DATA` `.text` and `.data`, respectively.

A typical configuration file for Delphi programs is:

```
FillRAM HighMem          // remap unused RAM

#ifdef BOOT
  Reserve Monitor        // leave room for the debug monitor
#endif

DLL      RTT32DLL.DLL    // include the RTTarget-32 system DLL

#ifdef section .text      // redefine some section names for BCB
  #define CODE .text
  #define DATA .data
#endif

#ifdef section rtt32dll.dll..text // redefine some section names for BCB
  #define rtt32dll.dll.CODE rtt32dll.dll..text
  #define rtt32dll.dll.DATA rtt32dll.dll..data
#endif

Locate PageTable PageT      LowMem
Locate Header Header       LowMem

Locate Stack Stack         HighMem 16k
Locate Heap Heap           HighMem

Locate NTSection CODE      HighMem
Locate NTSection DATA     HighMem
Locate NTSection BSS       HighMem
Locate NTSection .tls      HighMem
Locate NTSection .rdata    HighMem
Locate NTSection .idata    HighMem

Locate NTSection rtt32dll.dll.CODE HighMem
Locate NTSection rtt32dll.dll.DATA HighMem
```

Please refer to the demos under directory `Demodel` for numerous examples of building programs with Delphi.

Appendix B

Redistributable Components of RTTarget-32

The following files shipped with RTTarget-32 may be redistributed by RTTarget-32 license owners with their applications:

- The Debug Monitor as RTB, RTA, or Hex file produced by RTLoc, or an absolute image of the Monitor, also produced by RTLoc, burned into a ROM, EPROM, Flash, or similar non-volatile memory.
- BOOTDISK.EXE and BOOTD16.EXE.
- RTRUN.EXE, RTTDBG32.DLL, RTCOMNT.DLL, RTTARGET.INI.
- RTTBOOT.COM.

The above files may only be distributed under the licensing terms given in section *Licensing Terms and Liability*. In particular, they may be distributed exclusively for the purpose of loading RTTarget-32 binary files (RTB files) you have created using a full RTTarget-32 license.

Program MAKEDLM.EXE is **not** redistributable. If you want to distribute MAKEDLM.EXE, please contact On Time for additional information.

If you want to distribute BOOTDISK.EXE, RTRUN.EXE, or RTTBOOT.COM to allow your customers to start your RTTarget-32 program, please use the following table to determine which files are required. The columns specify the host operating system the customer will be using.

Program	MS-DOS	Windows 3.x	Windows 95/98/ME	Windows NT/2000
BOOTDISK	BOOTDISK.EXE	BOOTDISK.EXE	BOOTDISK.EXE BOOTD16.EXE	BOOTDISK.EXE
RTRUN	-	-	RTRUN.EXE RTTDBG32.DLL RTCOMNT.DLL RTTARGET.INI	RTRUN.EXE RTTDBG32.DLL RTCOMNT.DLL RTTARGET.INI
RTTBOOT	RTTBOOT.COM	-	-	-

RTRUN is not supported under MS-DOS or Windows 3.1. RTRun requires the RTTARGET.INI file to find the correct port parameters to be used for downloads. If the file is missing, COM1 at 115200 baud is assumed.

Appendix C

RTLoc Error, Warning, and Information Messages

RTLoc can issue information, warning, error, and fatal error messages. Information messages usually don't indicate a serious problem. Warnings can (and frequently do) indicate a problem. The cause of the message should be investigated to determine whether the message can safely be ignored. Error and fatal error messages indicate that the application cannot be located. RTLoc will not produce an .RTB, .HEX, or .BIN file if any errors are encountered. Fatal errors will immediately abort RTLoc.

All messages are written to stdout and to the .LOC file. This behavior can be modified by command line options (see Chapter 3, *RTLoc Options* for details).

The following sections contain all messages in alphabetical order. Message portions in *italic* typeface are replaced by more specific information. For example, in the message

Region *RegionName* not found

RTLoc will replace *RegionName* with the actual name of the region that could not be found.

#include files and/or macros nested too deeply

RTLoc's preprocessor has run out of resources. Possibly, macros and/or include files invoke each other recursively.

All NT sections must reside in same region, section: *SectionName*

RTLoc has encountered *Locate NTSection* commands specifying different regions. All NTSections of a specific EXE or DLL must be located in a single region. If you must locate sections in different regions, use the *Locate Section* command instead. If you prefer to use *Locate NTSection*, but the application must be ROMable, use the *Locate Copy* command to move the sections to ROM or use a virtual region.

All parameters after region name ignored

A *Locate NTSection* * command was used with optional parameters following the region name. RTLoc only supports the default values in this case.

Application header should have ReadOnly access

The application header has a higher access value than ReadOnly. Therefore, the application could accidentally overwrite it causing unpredictable results.

Application uses DLLs, but RTTarget32SystemInit is not exported

The application consists of several modules, but RTLoc did not find the function responsible for initializing those DLLs. Function RTTarget32SystemInit is contained in RTT32.LIB and declared as "export", but some linkers ignore this attribute for functions linked into .EXE files. In this case, some other means of exporting has to be used. For example, with Microsoft Visual C++, the function can be exported with linker option /export:RTTarget32SystemInit.

Boot code and boot code copy must be paragraph (16-byte) aligned

The boot code's alignment is set to a value below 16, which is not supported.

Boot code and boot vector must reside within the same 64k segment

Control is transferred from the boot vector to the boot code using a near jump, which limits the range to 64k.

Boot code contains fixups, file name: *FileName*

The .EXE file containing the boot code requires fixups; this is not supported by RTLoc. Change the source of the boot code to eliminate all memory references requiring fixups.

Boot code located twice

RTLoc has encountered two *Locate BootCode* commands. Only one may be specified.

Boot code must be paragraph (16-byte) aligned

The boot code alignment is set to a value below 16, which is not supported.

Boot code must be placed immediately after boot vector

RTLoc has found that both the BIOS boot vector and the boot code are located in the same region, but are not adjacent to the vector located first. Place the vector and the boot code in separate regions or rearrange them to make them adjacent.

Boot code must have at least SysRead access

The boot code has access value NoAccess, making its execution impossible.

Boot code should have SysRead access

The boot code has a higher access value than SysRead. Consequently, the boot code is not protected and could be corrupted at run time.

Boot data located twice

RTLoc has encountered two *Locate BootData* commands. Exactly one must be specified if a *Locate BootCode* command is used.

Boot data must be located in first megabyte

The boot data was mapped to memory inaccessible in real mode. Locate it to a region below 1M.

Boot data must be paragraph (16-byte) aligned

The boot data alignment is set to a value below 16, which is not supported.

Boot data must have at least System access

The boot data has access value NoAccess or SysRead, making it inaccessible for the boot code.

Boot data should have System access

The boot data has a higher access value than System. Consequently, the boot data is not protected and could be corrupted at run time.

Boot data too small, needs at least Size bytes

The boot data size specified in a *Locate BootData* command is less than the size required by the boot code. Do not specify a size and let RTLoc determine the correct value.

Boot vector not located at CPU boot location

A *Locate BootVector* command was used and the boot vector is not located at address FFF0h or FFFFFFFF0h. However, all Intel 80x86 compatible CPUs boot from this location. The boot process will fail if the vector is not located at the correct address. Ignore this warning only if your hardware maps the CPU's address to the actual physical address.

Boot vector section should have SysRead access

A *Locate BootVector* or *Locate BIOSVector* command has specified a different access value for the boot vector. The application can boot, but the boot vector is not protected against corruption.

Boot Vector too small, size specification ignored

A *Locate BootVector* or *Locate BIOSVector* command has specified an invalid size. The size should be at least 16 bytes.

Cannot copy SectionName

A *Locate Copy* command was applied to an entity which cannot be copied. *Locate Copy* supports only entities Section, NTSection, File, PageTable, Header, and BootCode.

Cannot link ModuleName.FunctionName in ModuleName: no import table

RTLoc was unable to resolve a static DLL reference because the .idata section was not mapped. Add command *Locate [NT]Section .idata*.

Cannot mix 'Locate Section' with 'Locate NTSection'

Both *Locate Section* and *Locate NTSection* commands were encountered for the same EXE or DLL. You must decide which of the two locate methods you wish to use.

Compress buffer overflow; cannot compress SectionName

The compression of data has failed, causing random memory locations to be corrupted. The given section cannot be compressed. Use option -c- to suppress compression for this particular section.

Compressed section *SectionName* precedes decompression code

The application header contains a list of program entities. This list is processed by the boot code to initialize the application. During this process, all copied sections are copied and/or decompressed. Before the decompression code is used for decompression, it must be at its final address. Thus, it must precede any compressed entries in the list. To fix this problem, move the *Locate DecompCode* and its *Locate Copy* to a location preceding any other *Locate Copy* commands in the configuration file.

Configuration file not *FileName* not found: *ErrorMessage*

A configuration file references another configuration file which could not be opened. RTLoc will search for configuration files in the default directory first and subsequently in the directory RTLoc was loaded from.

Copied sections should have NoAccess access

A *Locate Copy* command has specified a different access value for the copied section. The application can run, but the section is not protected optimally.

Copy of section *SectionName* larger than required, reduce from *Size* to *Size* to save memory

RTLoc has found that the size of a copied section could be reduced. To save memory, specify the recommended size for the given *Locate Copy* command.

Data in reserved application corrupted

RTLoc's verification of the data contained in a program image file specified in a *Reserve* command has failed. The file is corrupted and must be recreated using RTLoc.

Decompress code/data located twice

RTLoc has encountered more than one *Locate DecompCode* or *Locate DecompData* or *Locate DiskBuffer* command.

Decompress code/data should have NoAccess access

A *Locate DecompCode* or *Locate DecompData* command has specified a different access value for the section. The application can run, but the section is not protected optimally.

Decompression code/data located, but not required

Memory has been allocated for data decompression, but no sections are compressed. Either make copies of sections which can be compressed or remove the decompression code and/or data.

Disk buffer located twice

The disk boot code requires only one disk buffer.

Disk buffer must be sector (512 byte) aligned

This requirement is imposed by the disk loader and the BIOS.

Disk buffer outside real mode address space

The disk buffer must be accessible to the BIOS and must therefore be located below 1M.

Disk buffer should have NoAccess access

A *Locate DiskBuffer* command has specified a different access value for the section. The application can run, but the section is not protected optimally.

DLL dependency in *ProgramModule*: *ModuleName.FunctionName*

RTLoc encountered a DLL import, but was unable to resolve the reference against a DLL export. You must supply the missing function in your program (see Chapter 7, *Adding other Win32 Functions* for details). Alternatively, you can use RTLoc's LINK command to reroute the call to a different, existing function to be used as a replacement.

ENDM without matching MACRO

RTLoc encountered an ENDM directive but no matching MACRO directive.

Entities located in virtual regions must be page aligned, entity: *Name*

The given entity is located to a virtual region with an alignment value of less than 4096, which is not supported.

Error directive: *Message*

RTLoc has processed an #error command in a configuration file.

Error opening file *FileName*, errno: *ErrorMessage*

RTLoc was unable to open the specified file for the given reason.

Expression syntax error

The expression in an *#if* expression could not be parsed.

Failed to decompress section *SectionName*

RTLoc's verification of compressed data has failed: the decompressed data is not identical with the original data. Option *-c-* must be used to suppress compression for the given section.

File data should have at least *ReadOnly* access

The access value specified in a *Locate File* command is too low to allow the application to read the data.

File data size for file *FileName* too small; file truncated

The size given in a *Locate File* command is less than the actual file size. RTLoc disregards the rest of the file. If this is not desired, remove the size specification and let RTLoc assign the actual file size automatically.

FillRAM region must end on page boundary

The region specified in the *FillRAM* command does not end on a page (4k) boundary. Therefore, no pages can be added to the region without leaving a gap. Change the region specification accordingly.

FillRAM region not found, region: *RegionName*

The region specified in a *FillRAM* command was not found. Define the Region before the *FillRAM* command.

FillRAM specified twice

RTLoc encountered two *FillRAM* commands. Only one such command is supported.

Fixup target beyond section, target address: *Address*

A fixup referenced an address immediately following a section. Such a fixup is only valid if it was generated by a reference to a symbol which does not point to a storage location, but rather the first available address after a segment. In all other cases, use *Locate NTSection* instead of *Locate Section* to locate the program.

Function *_RTTarget32SystemInit* not found, must be exported

Function *_RTTarget32SystemInit* is required by all applications using DLLs to ensure proper DLL initialization. The module containing RTT32.LIB must export this function.

Header located twice

RTLoc encountered two *Locate Header* commands. Only one such command is supported.

Heap located twice

RTLoc encountered two *Locate Heap* commands. Only one such command is supported.

Heap should have at least 64k, size: *Size*

The application's heap is rather small. Many Win32 programs that use the run-time system will require at least 64k of heap space. However, if you know that your program can run with a smaller heap, this warning can be ignored.

hex/bin file *FileName* overlaps hex/bin file *FileName*

Two *HexFile* or *BinFile* commands specify overlapping address ranges. Please check the parameters of all *HexFile* and *BinFile* commands.

Import by ordinal *ModuleName.Ordinal*, ignored

RTLoc has encountered an import by ordinal, which is not supported. Use Import by name instead. The warning can be ignored if the imported function is never called.

Init function *FunctionName* not found, must be exported

The name given in an *Init* command does not reference a function exported by the .EXE or a .DLL file. Please check the spelling of the function name. Use RTLoc's option *-Rd* to get a detailed list of all exported and imported function names. Name comparison is case sensitive.

InitCode exe file *FileName* entrypoint must be 0:0, but is *seg:Ofs*

Exe files used with *InitCode* must have an entry point at the start of the file.

InitCode exe file contains fixups, file name: *FileName*

Exe files used with *InitCode* must use the tiny memory model and may not contain fixups.

InitCodeDump: no data to dump

An *InitCodeDump* directive was encountered, but no *InitCode* or *OUT...* commands generating *InitCode* data precede it.

InitCodeOrg not allowed outside code section; call InitCode first

The *InitCodeOrg* statement makes no sense in the context it was encountered.

InitTable overflow

The data generated by *InitCode* and *OUT...* commands is limited to about 60k bytes.

Internal: *Message*

RTLoc encountered an internal error. Please contact On Time's technical support for assistance.

Invalid boot code exe file, file name: *FileName*

The file name given in a *Locate BootCode* command does not reference a valid MS-DOS .EXE file or the file is corrupted.

Invalid boot code signature in file *FileName*

RTLoc did not find the expected boot code signature in the boot code file. Please make sure the file contains boot code with a valid header.

Invalid InitCode exe file, file name: *FileName*

The file name given in an *InitCode* command does not reference a valid MS-DOS .EXE file or the file is corrupted.

Invalid PE file: *FileName*

RTLoc did not find the expected signature in the application PE file (.EXE or .DLL). Either the file was not linked as a Win32 console mode program or the file is corrupted.

Invalid video mode: *Mode*

A parameter specified in a *GMode* command was not in the range 0..7Fh or 100h..17Fh.

Locate NTSection * not supported for virtual regions

The use of a wildcard name (*) in a *Locate NTSection* command is only supported for physical regions.

Location of fixup not found at: *Address*

RTLoc was unable to locate a fixup in the program image and ignores the fixup. The cause is probably that the fixup is located in a section not included in the program image. For example, if the program contains a DLL dependency, it will also contain an import table which is possibly not mapped (the import section is usually named .idata).

Location of *Item* not found, address: *address*

RTLoc attempted to apply a fixup to *Item*, but *Item* was not found to be part of the image. Verify that all sections required by the program are actually included with *Locate Section* or *Locate NTSection* commands. *Address* is the address in the PE file's virtual address space.

Macro cannot have keyword name: *Keyword*

You are trying to declare a macro with the name of a configuration file reserved word, which is not supported.

Memory access for section *SectionName* into region *RegionName* not possible.

It was attempted to locate the given section into a region that doesn't support the required access. For example, locating a data section requiring read/write access into a ROM Region is not possible.

Misplaced #else

The preprocessor was unable to resynchronize.

Missing ENDM for macro *MacroName*

Macro and *Endm* directives do not match.

Missing or misplaced #endif

#if and #endif directives do not match.

Missing parameter

The command encountered in a configuration file does not specify all required parameters.

ModuleName.FunctionName expected in Link command

The first parameter of a *Link* command did not have the required syntax.

More than one instance of RTT32.LIB linked

RTLoc has found functions of library RTT32.LIB in more than one EXE/DLL. Only one module may contain RTT32.LIB.

No application header located

The configuration file(s) did not contain a *Locate Header* command, which is required.

No boot code, application cannot be booted

RTLoc did not find any boot code which could start the application. Add a *Locate BootCode* command or a *Reserve* command for an application that contains boot code.

No boot code, application cannot be started

RTLoc has detected that you have not included any boot code and no *Reserve* command for an application with boot code. This warning message can be ignored only if some other means is employed to invoke the application.

No boot code, boot code configuration command(s) ignored

A *BOOTFLAGS*, *CodeSeg*, *DataSeg*, *CPL*, *NoFPU*, *COMPort*, *VideoRAM*, or *GMode* command was encountered in the configuration file, but there was no *Locate BootCode* command. Boot code configuration commands only have an effect in applications that include boot code. If boot code is indirectly imported through a *Reserve* command, this warning is only issued if the requested parameter differs from the value found in the imported boot code.

No boot data section for boot code

A *Locate BootCode* command was encountered, but no *Locate BootData*. Add a *Locate BootData* command.

No boot vector or disk buffer specified, application cannot boot

RTLoc has not found any means of transferring control to the boot code. Add the required *Locate BootVector*, *Locate BIOSVector*, or *Locate DiskBuffer* command. This warning message can be ignored only if some other means is employed to invoke the boot code (e.g. RTTBOOT).

No Decompression code/data located

Data compression has been requested, but no decompression code or data has been mapped to enable RTTarget-32 to decompress the data on the target. Add appropriate *Locate DecompCode* and/or *Locate DecompData* commands.

No fixup table found for ModuleName

RTLoc did not find a fixup table in the PE file of the application or a DLL. This is most likely due to a missing */fixed:no* option for the Microsoft Visual C++ linker.

If the linker did not include a fixup table because the program must be located at a specific address, RTLoc is unable to relocate the program. The only way to run such programs under RTTarget-32 is to locate them at the exact base address set by the linker.

No heap region specified

RTLoc did not find a *Locate Heap* command. This warning indicates an error if you intend to use the C/C++ run-time system. If no run-time system is used, the warning can be ignored.

No image base found for ModuleName

RTLoc was unable to find the code base in a PE file. A possible cause is that the section containing the program code is not included. The only other cause can be an error in the PE file.

No image base to allocate SectionName

RTLoc was unable to find the code base in a PE file. A possible cause is that the section containing the program code is not included or is not located before *SectionName*.

No MAP file found, file name: *MapFileName*

One or more *Locate Section* commands contain segment numbers, but RTLoc was unable to find a map file to calculate the actual segment sizes. RTLoc will use the sizes given in the PE file instead. The warning can be ignored.

No module found for init function *FunctionName*

RTLoc was unable to determine the module in which the Init function resides. Please verify the spelling of the module name in the *Init* command. The module name **must** be specified for Init functions in DLLs.

No module found for section *SectionName*

RTLoc was unable to determine the module in which a section referenced in a *Locate Section* or *Locate NTSection* command resides. Please verify the spelling of the module name in the *Locate* command. The module name must be specified for sections in DLLs.

No page table; FillRAM ignored

A *FillRAM* command was encountered, but the application does not use paging. The application is built without RAM remapping. Either remove the *FillRAM* command or add a *Locate PageTable* command to eliminate this warning.

No size information found in map file for section *SectionName*

A *Locate Section* command has specified a segment number that was not found in the map file. RTLoc will use the size given in the PE file instead.

No size reduction in compressing *SectionName*, storing instead

RTLoc has attempted to compress the given section, but was unable to reduce the image size. Very small sections or data which is already compressed may show this property. Use option *-c-* for the given section to suppress compression.

No stack region specified

RTLoc did not find a *Locate Stack* command required by every program.

No VideoRAM command, assuming 'VideoRAM = None'

No VideoRAM command was encountered. RTLoc assumes that the target has no display.

Numeric parameter expected but found *Parameter*

The given parameter of the given command line in a configuration file must be numeric.

Numeric parameter too large: *Parameter*

The given parameter exceeds the upper limit of the valid range.

Numeric parameter too small: *Parameter*

The given parameter is smaller than the lower limit of the valid range.

Obsolete module DUMMYDLL.OBJ is no longer required; do not link it

RTLoc encountered a reference to the DLL NODLL. Object file DUMMYDLL.OBJ and DLL NODLL.DLL should no longer be used with RTTarget-32. You can remove all references to them.

Option syntax error: *Option*

RTLoc encountered the given option either on the command line or in a configuration file and was unable to interpret it.

Out of memory to allocate *Name*

RTLoc ran out of heap space.

Page table located twice

RTLoc found two or more *Locate PageTable* commands. Only one such command is supported.

Page table must be page (4096 byte) aligned

The alignment of the page table must be at least 4k. This is required by the 386 architecture.

Page table must have at least SysRead access

The page table must have at least SysRead access, but NoAccess is specified, making it inaccessible for the boot code.

Page table required to support virtual regions

The configuration file defines virtual regions, but no *Locate PageTable* command.

Page table should have System access

A *Locate PageTable* command has specified an access value other than System. If you want to place the page table in ROM, SysRead is also possible. In this case, however, page attributes cannot be changed at run-time! As a consequence, the heap **must** have ReadWrite access.

If the page table is located in RAM, access System is recommended for maximum protection and flexibility.

Page table size must be multiple of 4k

The specified page table size is not a multiple of 4k. The 386 CPU does not support page tables that do not consist of a whole number of pages.

Page table too small, need at least Size bytes

RTLoc found that the required address space cannot be mapped onto the page table. This is a common error if command *FillRAM* or virtual regions are used, since RAM remapping can enlarge the address space **after** the page table's size has been determined. The problem is solved by explicitly setting the page table's size to the required value.

Parameter must be power of two, found: Number

The given numeric parameter in the given line of a configuration file must be a power of two.

Parts of section SectionName not covered by any hex or bin file

The given section needs data in a region of memory not covered by any hex or bin file. If the data is not placed in its location by some other means, the program will not be able to run.

Physical region name expected after virtual region name

A *Locate* command references a virtual region without also specifying a physical region name. Except for Heap and Stack, a physical region must be supplied.

Protected mode boot code does not support BIOS boot vector

A *Locate BIOSVector* command was specified with a protected mode boot code, which is not supported.

Recursive preprocessor symbol expansion for symbols Sym and Sym

A loop was encountered expanding preprocessor symbols.

Region RegionName exceeds 4gb address space

RTLoc has found a region definition with a region's end address beyond 4G. Check the parameters of the *Region* command.

Region RegionName must be virtual

A *Locate* command references a physical region name followed by another physical region name. Either use only a single physical region or use a virtual region.

Region RegionName not found for section SectionName

A *Locate* command references an undefined region. All regions must be defined before they can be used.

Region RegionName overflow

Too many program entities have been allocated to the given region. Allocate some of the sections or other entities to other regions. If the given region is a virtual region, make sure a *FillRAM* command remaps memory to it to accommodate the stack and heap.

Region RegionName overlaps region RegionName

The two given regions overlap. Please check the address and size settings for these regions in the configuration file(s).

Reserved app old image file. Rebuild ApplicationName with current version of RTTarget-32

RTLoc found that the program specified in a *Reserved* command was built with an older version of RTTarget-32 incompatible with the current version.

Reserved application built with different memory layout, rebuild AppName

RTLoc found that the program specified in a *Reserved* command was built with a different set of region commands. If two programs to be run simultaneously on one computer assume a different memory layout, mutual corruption can result. Therefore, it is recommended to build all programs (including the debug Monitor) with the same set of regions, preferably stored in a single file shared by all programs.

Reserved application contains remapped page at logical address: Address

RTLloc has detected that the application given in a *Reserve* command contains remapped pages, which is not supported. Remove the *FillRAM* command or any virtual regions from the reserved application's configuration file and re-run RTLloc for both applications.

Reserved application file invalid or different version, file name: FileName

RTLloc did not find the expected signature in the application file of an application given in a *Reserve* command. The file was either not produced by the current version of RTLloc or the file is corrupted.

Reserved application has no page table

The application specified in a *Reserve* command does not contain a page table, but the current application does. Either both or none of the two programs must use paging.

Reserved application uses page table, add Locate PageTable

The application specified in a *Reserve* command contains a page table, but the current application does not. Either both or none of the two programs must use paging.

ROM does not support write access in region: RegionName

A *Region* command specifies memory type ROM and access System or ReadWrite, which is not supported for ROM.

Section containing code not mapped, section: SectionName

RTLloc found a section in the PE file that contains code but is not included in the application image. This warning should only be ignored if it is guaranteed that the application will never try to access that section. Otherwise, add an appropriate *Locate Section* or *Locate NTSection* command to include the section.

Section SectionName access rights possibly too low

RTLloc has compared the access requirements specified by the PE file's section table with the access rights in the configuration file and found a discrepancy. For example, if you have located a data section as ReadOnly, but the PE file specifies that write access is required for this section, RTLloc issues this message.

RTLloc adheres to the access value specified in the *Locate Section* command. The warning can only be ignored if it is guaranteed that the actual access at run time does not violate the given value.

Section SectionName has no image, command Locate Copy has no effect

The given section has no data associated with it. Thus, a copied section which consists of only such data contains nothing. You should remove the corresponding *Locate Copy* command since it has no effect.

Section SectionName has no size and is not mapped

The given section has no size. The section was probably not found in the EXE or DLL file and can therefore be removed from the configuration file.

Section SectionName in ROM; application not loadable

RTLloc has determined that the application is loaded to RAM, but the given section resides in ROM. Map the section to a RAM region.

Section SectionName must be in RAM to be able to be copied

The section referenced in a *Locate Copy* command resides in ROM. At run-time the RTTarget-32 boot code would not be able to copy or expand the data to its destination.

Section SectionName not compressed, because no decompression code located

RTLloc has encountered a copied section which could not be compressed because no decompression code or data is present. Either add appropriate *Locate DecompCode* and/or *Locate DecompData* commands or disable compression with option -c-.

Section SectionName not in ROM; application not ROMable

This warning is only issued when option -o (ROMable) is specified or a *HexFile* command is present. It identifies a program entity that requires an image in RAM at boot time. To make the application ROMable, move the section to ROM or use the *Locate Copy* command for this section to place a copy of it into ROM.

Section *SectionName* overlaps section *SectionName*

Two sections overlap in the given region. This can happen if *Locate NTSection* and other *Locate* commands are mixed for the same region. RTLoc is forced to locate all but the first NTSection at specific addresses, even if these addresses are already allocated to other entities. To avoid this problem, all *Locate NTSection* commands should be grouped together in the configuration file in order of ascending addresses, without other interleaving *Locate* commands.

Section to copy not found, section: *SectionName*

A *Locate Copy* command references an undefined program entity. Entities to be copied must be located first.

Segment value in Intel hex start address record truncated from *Address* to *Address*

An Intel Hex file Start Address record has been requested with RTLoc option -s, but the target's boot vector is outside the real mode address space.

Size of section *SectionName* too small, need at least *Size* bytes

RTLoc has found that the size specified for a section is too small. Either increase the size in the *Locate* command or let RTLoc determine the size automatically.

Size specification for boot code/data ignored

RTLoc found a size specification for the boot code or data. RTLoc uses the size given in the boot code's .EXE file. This behavior cannot be overridden.

Size specification for decompression code/data ignored

RTLoc found a size specification for decompression code or data. RTLoc uses the size required by the decompression algorithm. This behavior cannot be overridden.

Size specification of application header ignored

RTLoc found a size specification for the application header. RTLoc calculates the size automatically. This behavior cannot be overridden.

Stack is suspiciously small, size: *Size* bytes

RTLoc found that the application stack is smaller than 512 bytes. Only very small applications without a run-time system will be able to run with such a small stack. Increase the stack size. Typically, a C/C++ or Pascal program will need 4k to 32k of stack space.

Stack located twice

RTLoc found two or more *Locate Stack* commands. Only one such command is supported.

Stack section must have ReadWrite access

The specified access in a *Locate Stack* was not ReadWrite, the only supported value.

Stack size is 0

The stack of the program has zero length. Probably all available memory has been allocated to other entities (such as the heap). Either reduce the heap's size or specify an explicit value for the stack size in the *Locate Stack* command.

Target of fixup not found, fixup: *Address*, target: *Address*. Fixup ignored!

RTLoc was unable to locate the target of a fixup in the application image. Both addresses given are virtual addresses of the PE file. This warning can only occur if *Locate Section* is used instead of *Locate NTSection*. There are two possible causes for this warning:

- The section containing the target of the fixup is not included. From the target address and the .EXE file report in the .LOC file, it can be inferred which section is required. If it is guaranteed that the application will never access that section, the warning can be ignored. Otherwise, add the section containing the target with a *Locate Section* command.
- The fixup target address does not belong to any section. This can happen if a fixup references a program entity with an offset **and** an index. Example:

```
int MyArray[10];
...
if ((i>=1000) && (i<1010))
    i = MyArray[i-1000];
```

The code produced for the last line with variable i in register eax could be:


```
mov  eax, [4*eax + offset MyArray - 4000]
```

The problem is that the compiler will already calculate the value *offset MyArray - 4000*. Therefore, RTLoc will assume the fixup to reference a memory location located 4000 bytes before variable MyArray. However, this location could well be located outside the section containing MyArray.

The only method to resolve this problem is to use *Locate NTSection* instead of *Locate Section* or to modify the above code so that the fixup references MyArray directly. Example:

```
int MyArray[10];
...
if ((i>=1000) && (i<1010))
{
    i -= 1000;
    i = MyArray[i];
}
```

TLS and TEB need Size bytes stack space. Stack possibly too small

The Thread Local Storage and Thread Environment Block are allocated from the program's stack at run-time. The required size exceeds 20% of the available stack. This warning can be ignored if the program does not need more than the remaining stack space.

TLS and TEB need Size bytes stack space. Stack too small

The Thread Local Storage and Task Environment Block are allocated from the program's stack at run-time. The required size exceeds the available stack. Increase the stack size or reduce the size of variables declared with `__thread`.

Too many #ifs

#if directives are nested too deeply. RTLoc supports at most 64 levels.

Too many graphics modes

Too many modes were specified in *GMode* commands. Note that when multiple *GMode* commands are used, each command adds modes. Up to 16 modes are supported.

Too many hex or bin files

The maximum number (64) of supported hex or bin files has been exceeded.

Too many lines in macro *MacroName*

The given macro has more than 256 lines, which is not supported.

Too many links

The maximum number (256) of supported *Link* commands has been exceeded.

Too many macro parameters: *MacroName*

The given macro has more than 8 parameters, which is not supported.

Too many messages

The maximum number of error and warning messages of 20 has been reached. If you do not want RTLoc to stop after 20 messages, use option -m.

Too many modules

RTLoc has encountered more than 31 DLL commands, which is not supported.

Too many parameters

The maximum number of supported parameters is exceeded in the given configuration file line.

Too many regions, region: *RegionName*

The maximum number of supported regions (64) has been exceeded. Combine adjacent regions or remove unneeded regions.

Too many sections, section: *SectionName*

The number of supported program entities (255) has been exceeded.

Two regions have the same name: *RegionName*

Two regions have been defined with the same name. Change the name of one of them.

Two sections have the same name: *SectionName*

RTLoc found two or more *Locate* commands defining program entities with identical names. The name of each entity must be unique.

Unmatched parenthesis

A preprocessor expression contains unmatched parenthesis.

Unable to create file *FileName*, errno: *ErrorMessage*

RTLoc is unable to create the given file for the given reason.

Unable to open file: *FileName*, errno: *ErrorMessage*

RTLoc is unable to open the given file for the given reason. RTLoc first searches for the file in the default directory and then in the directory RTLoc was loaded from.

Unable to rename .LOC file from *FileName* to *FileName*

Assigning the correct name to the .LOC file failed, possibly because the file is currently opened by some other process.

Undefined identifier: *Identifier*

The preprocessor was unable to convert *Identifier* to a number.

Unknown keyword: *Keyword*

RTLoc cannot interpret the given command in a configuration file. Please check the keyword's spelling.

Unknown operator

The given command contains an unknown numeric operator.

Unknown option *Option*: ignored

RTLoc encountered the given unknown option.

Unsupported boot vector type for protected mode boot code

RTLoc has detected that protected mode boot code is being used. Use a *Locate BootVector* command to invoke the boot code.

Unsupported fixup type: *Number* at *Address*

RTLoc encountered a fixup type it does not know. The fixup is ignored and the program can only run if the fixup location is never used at run-time. RTLoc can only process fixups of type IMAGE_REL_BASED_ABSOLUTE and IMAGE_REL_BASED_HIGHLOW. All others will lead to this warning.

VESA mode info block at physical address 0C00h clashes with section *SectionName*

A VESA mode has been specified in a *GMode* command, but the memory at address 0C00h serving to hold the VESA Info block has been allocated to other entities. RTPEG-32 or MetaWINDOW drivers will not be able to read the VESA Info block. Locate a *Nothing* entity to the first page of RAM to avoid this warning.

VESA mode info block at physical address 0C00h has been allocated to FillRAM

A VESA mode has been specified in a *GMode* command, but the memory at address 0C00h serving to hold the VESA Info block has been allocated to some other region through FillRAM. RTPEG-32 or MetaWINDOW drivers will not be able to read the VESA Info block. Locate a *Nothing* entity to the first page of RAM.

VideoRAM region *RegionName* not found

The region name specified in a *VideoRAM* command was not defined previously.

VideoRAM region *RegionName* should have at least 4k size

RTLoc encountered a suspiciously small video RAM. The RTTarget-32 run-time functions assume at least 4k of video RAM to be available. For non PC compatible displays, specify VideoRAM = None.

VideoRAM region *RegionName* should have ReadWrite access

Access right for the video RAM is too low. The RTTarget-32 run-time functions writes to the video RAM.

Virtual region must start at page boundary, region: *RegionName*

The starting address of a virtual region must be page aligned, which it is not.

Virtual regions not supported for section *SectionName*

An attempt was made to locate an unsupported entity to a virtual region. Only *Locate Section*, *NTSection*, *File*, *Nothing*, *Stack*, or *Heap* may be placed in virtual regions.

Part II

RTKernel-32

RTKernel-32 is a powerful real-time multitasking system. It was designed for software developers who wish to implement professional process control applications on 32-bit embedded systems. Every effort has been made to ensure easy usage and excellent run-time performance. RTKernel-32 is compact (about 16k of code, 6k of data) and provides the programmer with the basic tools needed to develop efficient real-time software.

RTKernel-32 is a library you can link to your application program. It offers a number of functions to manage tasks, semaphores, mailboxes, interrupts, etc. All RTKernel-32 threads run within a single program. An RTKernel-32 application consists of a single executable containing the kernel, the required drivers, and all threads.

The main features of RTKernel-32:

- **Unlimited number of tasks**
RTKernel-32 can handle as many tasks as will fit into the computer's available memory. One thread requires about 1k.
- **Task switch time of about 5 microseconds (80486/33)**
RTKernel-32 is a powerful system offering unequaled performance.
- **Task switch time remains constant for any number of tasks**
Many multitasking systems exhibit severely reduced performance when too many tasks are activated. RTKernel-32's performance is independent of the number of tasks activated.
- **64 priorities**
The behavior of programs may be fine-tuned precisely using different priorities.
- **Cooperative and preemptive scheduling**
With *preemptive scheduling*, task switches can take place at practically all times. Tasks can be activated directly out of interrupt handlers. Preemptions can be enabled or disabled.
- **Support of math coprocessor / emulator**
If a math coprocessor is installed, it can be utilized by any number of tasks. Its registers can be preserved by RTKernel-32 during task switches. 80387 software emulators are also supported.
- **Interrupt support**
Using RTKernel-32, it is possible to exchange data with other tasks and to suspend or activate tasks out of interrupt handlers. The interrupt priorities of the interrupt controller can be re-programmed to achieve the best possible interrupt response times. This qualifies RTKernel-32 for high-speed data acquisition, communications, process control, and similar tasks.
- **Time slicing**
RTKernel-32 can be used as a time sharing system, sharing CPU time evenly among a number of tasks.
- **Semaphores**
Semaphores can be used to exchange signals between tasks. RTKernel-32 supports counting, binary, event, resource, and mutex semaphores. Semaphores may also be used by interrupt handlers.
- **Mailboxes**
Mailboxes can be used to exchange data between tasks. Mailboxes may also be used by interrupt handlers (e.g., for data buffering).
- **Message passing**
Message passing can be used to exchange data between tasks directly. This synchronizes the tasks involved.

- **Real-Time Memory Management**

RTKernel-32's Memory Pools allow memory allocation and deallocation within guaranteed time limits and can even be used by interrupt handlers.

- **Portability**

RTKernel-32 is completely written in ANSI C; only some hardware or CPU dependent drivers contain assembler code. Since the portable kernel, which is written completely in ANSI C, has been implemented separately from hardware or systems dependent drivers, RTKernel-32 can easily be ported to other environments.

- **Three different APIs**

RTKernel-32's application interface has three different forms: RTKernel-32 native, RTKernel-C for DOS, and Win32. Depending on your requirements for compatibility with other systems, you can select the appropriate API. It is even possible to mix different APIs within the same program.

Chapter 1

Multitasking, Real-Time, and RTKernel-32

This chapter gives an introduction to the terms and concepts of RTKernel-32 and multitasking systems in general.

What is Multitasking?

A *thread* (also called a *task*) is a sequential path of execution. The discrete statements of a thread are processed sequentially according to the semantics of C, C++, or Pascal.

The term *multitasking* means that several sequential tasks are processed in parallel. However, on single processor systems, several tasks cannot run at the same time; therefore, task switches must be performed. This is the job of a multitasking system like RTKernel-32.

In many cases, tasks cannot run completely independently of each other, but are expected to cooperate. For example, it may be required that a certain task can only continue to run after another task has completed a certain operation. In such a case, the tasks involved must be synchronized, i.e., the parallelism of tasks is restricted again. Synchronization can be accomplished using inter-task communication.

For a good understanding of parallel programming, it is important to be aware of the different requirements of multitasking systems. The following two sections discuss the most important differences between time sharing and real-time systems.

Time Sharing

Time sharing utilizes multitasking for the purpose of sharing a high performance computer among several users (or batch jobs) at the same time. In general, tasks run largely independently of each other in time sharing systems. Therefore, inter-task communication is only offered in a simple fashion.

One of the most popular time sharing systems is Unix. It was developed when the processing power of the available computer was smaller than that needed for the job. Consequently, the multitasking system had to share the scarce resource "processing power" as "fairly" as possible among competing tasks. However, it should be noted that system throughput is usually degraded by multitasking in such systems, because the scheduling overhead is significant. Under Unix, for example, it is possible that two compute-bound programs each runs a minute when it is the only program running; however, running in parallel, they would need 2.5 minutes. Parallel processing may be more fair (User 1 and User 2 have to wait equally long for their job to finish), but is usually less efficient.

Real-Time Systems

Real-time systems satisfy a completely different set of requirements. A real-time system must never be overloaded. As soon as no more processing power is available, real-time response cannot be sustained. For example, a situation with real-time requirements might be: a meter generates data every second. The data must be collected, processed, and stored by the computer. If processing a data record requires **more** than a second on the target computer, the system is overloaded and real-time response cannot be sustained.

Real-time systems are not concerned about "fairness". Tasks have priorities which must be obeyed strictly. A task with a high priority can take away CPU time from another task with a lower priority at any time without "being fair" to the other task. Since overloading is ruled out, tasks having low priorities will sooner or later also receive CPU time.

Real-time systems are furthermore characterized by the requirement that they have to react to events within a predetermined - usually short - time span. External events are processed using interrupts whenever possible. Thus, the most stringent real-time requirements apply to interrupt handlers. Therefore, real-time systems must have a low interrupt latency (the time between the hardware interrupt signal and execution of the first statement of the interrupt handler).

For the task response time, the cases of cooperative and preemptive scheduling must be distinguished. In preemptive scheduling, the task response time to interrupts is mainly the interrupt latency and the task switch time. In cooperative scheduling, the maximum time span between two kernel calls is added.

Cooperative and Preemptive Multitasking

Preemptive multitasking means that task switches can be initiated directly out of interrupt handlers. With cooperative (non-preemptive) multitasking, a task switch is only performed when a task calls the kernel, i.e., it behaves "cooperatively" and voluntarily gives the kernel a chance to perform a task switch.

Example: a receive interrupt handler for a serial port writes data to a mailbox. If a task is waiting at the mailbox, it is immediately activated by the scheduler during preemptive scheduling. In cooperative scheduling, however, the task is only brought into the state "Ready". A task switch does not immediately take place; after the interrupt handler has completed, the task having been interrupted continues to run. Such a "pending" task switch is performed by the kernel at some later time, as soon as the active task calls the kernel.

RTKernel-32 supports both cooperative and preemptive scheduling. The preconfigured default is cooperative scheduling.

Real-Time

The term "real-time" is often used but seldom defined. One possible definition is given here:

Real-time software must deliver computation results under application specific time constraints. When a result is made available too late (or too early in some systems), the software has failed, even if the result is otherwise correct.

This definition says nothing about multitasking. Multitasking is not necessarily required to develop real-time software; however, usually multitasking will simplify real-time software development. Multitasking can achieve excellent response times even when other jobs must be performed in parallel to the real-time processing. Prerequisites for real-time processing are a sufficiently small interrupt latency, a constant task switch time (which should be as small as possible), and, in many cases, preemptive scheduling. Therefore, complex operations with non-deterministic run-time requirements (e.g., loading a process from disk) cannot be performed by a real-time system during a task switch.

RTKernel-32's Scheduler

The decision which task is to run is made by the *Scheduler*. Only task states and priorities are considered in this process.

RTKernel-32's scheduler is not - like in many other systems - primarily based on the timer interrupt; rather, RTKernel-32 is an event-driven system. An event is an inter-task communication that may be initiated by a task or an interrupt handler. Events can lead to task state changes of the tasks involved. RTKernel-32's timer interrupt handler is just one interrupt handler among others. It has the sole purpose of bringing tasks waiting for a certain point in time into the state "Ready". Some multitasking programs could run completely without the timer interrupt handler (e.g., demo program RTPPrimes).

The scheduler obeys the following rules:

1. Of all tasks in the state *Ready*, the task with the highest priority runs.
2. If the scheduler has to choose among several Ready tasks having the same priority, the task that hasn't run for the longest time is activated.
3. If several tasks are waiting for an event, they are activated at the occurrence of the respective event in sequence of their priorities.
4. Except for time slice task switches, task switches are only performed if otherwise rule 1 would be violated. The number of task switches is minimized.

Normally, the rules given above are obeyed strictly, i.e., they are never violated. Whenever a task's state changes, the scheduler checks whether a task switch becomes necessary according to the scheduling rules. Only during cooperative scheduling, rule 1 can be violated between the occurrence of an interrupt and the next call to the kernel.

Time slice task switches are performed if the following conditions are satisfied:

1. Time slicing must be on (default is off).
2. At least one task with the same priority as the current task must be in the state Ready.
3. The last task switch must have occurred at least *TimeSlice* timer ticks ago. The last task switch may have been caused by an event other than time slicing.

The value of *TimeSlice* is set using function `RTKTimeSlice`. If *TimeSlice* is 1, condition 3 is true in every timer interrupt. Please note that preemptions are **not** required for time slicing.

Time slicing plays a minor role and must not violate the scheduling rules given above.

Task Switches

RTKernel-32 distinguishes three principal types of task switches: *Blocking*, *Activating* and *Time Slice*. A task always continues exactly where it has been previously suspended by a task switch. The three types of task switches occur under the following conditions:

Blocking task switch	A blocking task switch takes place whenever a task blocks itself, i.e., cannot continue to run. This will happen, for example, if a task attempts to retrieve data from an empty mailbox, or if a task releases the CPU for a certain time by calling function <code>RTKDelay</code> . RTKernel-32 will in this case select the highest priority task with state Ready to run. Should none of the application's tasks be ready, the Idle Task is activated.
Activating task switch	Activating task switches are performed whenever a task having a higher priority than the current task becomes Ready. An activating task switch is carried out, for example, if a task with a high priority is waiting for data at a mailbox when data is stored in the mailbox. The formerly blocked task can now continue to run and is immediately activated.
Time slice task switch	Time slice task switches are performed by RTKernel-32's timer interrupt handler if the conditions given above are satisfied. Time slice task switches can also be triggered directly by a call to <code>RTKDelay(0)</code> .

In addition to the task switch types discussed above, *cooperative* and *preemptive* task switches are distinguished. Preemptive task switches are initiated by an interrupt handler; cooperative task switches are initiated by tasks. Blocking task switches are always cooperative, because they are not *allowed* to be initiated by interrupt handlers. A blocking preemptive task switch is considered an error by RTKernel-32. Activating task switches can occur in both forms using the same respective mechanisms. For example, a task or an interrupt handler storing data in a mailbox can lead to an activating task switch. In the first case, the task switch would be cooperative; in the latter, it would be preemptive.

RTKernel-32 can be configured for cooperative or preemptive scheduling. When preemptions are disabled, task switches which would be preemptive with preemptions enabled are postponed by the kernel until the currently running task calls a scheduler entrypoint (an RTKernel-32 API function). In this way, potentially preemptive task switches are converted to cooperative task switches if the application has not enabled preemptions.

RTKernel-32 Tasks

A *task* or *thread* is a C function with its own stack. A task has a priority between 1 and 64. A high priority means high urgency. Within a program, priorities are only meaningful relative to each other, e.g., the behavior of a program with two tasks is identical if the priorities are 1 and 2 or 10 and 50.

Tasks are referenced using *task handles*¹⁵. Task handles are similar to file handles. When a task is created, RTKernel-32 returns a unique handle to the creating task which can subsequently be used to reference the task (e.g., to send data to it).

¹⁵ RTKernel-32 task handles are **not** identical to Win32 handles.

All variables declared local to the task function are allocated on the task's stack (except static variables). The same is true for the local variables of all functions called by the task. Therefore, several tasks can be started using the same task function, and each would be allocated its own stack and thus its own local variables. A function may be called by different tasks; this results in the same code being executed, but no reentrance problems arise, because each task has its own stack - provided the called function accesses only its parameters and local (non-static) variables.

The common C/C++ visibility rules fully apply to multitasking programs. All tasks can access global data.

When RTKernel-32 is initialized, two tasks are created: the *Idle Task* and the *Main Task*. The Idle Task has a priority of 0. Since user tasks must have priorities between 1 and 16, it is ensured that the Idle Task has the lowest priority in the program. It runs whenever no other task can run. The Idle Task is required by the scheduler, which always needs at least one "Ready" task it can activate.

The Main Task's priority is specified when function `RTKernelInit` is called. The Main Task's stack is provided by the run-time system, i.e., RTKernel-32 does not modify the Main Task's stack.

A task is always in one of the following states:

Current	The currently active task is in the state Current. Under RTKernel-32, exactly one task is in this state at any time (possibly the Idle Task).
Ready	All tasks ready to run are in the state Ready. Usually, all Ready tasks have the same or lower priorities as the active task.
Suspended	Suspended tasks cannot run because they were stopped explicitly by the RTKernel-32 operation <code>RTKSuspend</code> . They can only be made Ready by calling the RTKernel-32 function <code>RTKResume</code> .
Blocked	Tasks in the state Blocked cannot run because they are waiting for an event (e.g., a semaphore signal or a message coming in at a mailbox). These tasks can only be made Ready by another task or an interrupt handler.
Delaying	These tasks have blocked themselves for a certain time span. They will be made Ready automatically by RTKernel-32's timer interrupt handler after their delay has elapsed.
Timed	Tasks waiting for the occurrence of an event with a timeout are in the state Timed. Such a task will become Ready either if the event occurs or if the timeout expires.

All tasks not currently running are maintained by RTKernel-32 in several queues. There is, for example, a queue for all Ready tasks. Another queue contains all tasks waiting for a certain point in time. Queues can also build up at semaphores or mailboxes if tasks are blocked on them.

Inter-Task Communications

The term inter-task communication comprises all mechanisms serving to exchange information among tasks. RTKernel-32 offers three different techniques: semaphores, mailboxes, and message passing.

Semaphores are offered by virtually all multitasking systems. They allow the exchange of signals for activating and suspending tasks. A semaphore is a variable signals can be stored in or read from. Task switches may take place whenever a semaphore containing 0 signals is accessed. RTKernel-32 defines five different types of semaphores: counting, binary, event, resource, and mutex.

Mailboxes extend the concept of semaphores. Instead of signals, data of any type can be stored in or read from a mailbox. A task switch occurs whenever an empty or a full mailbox is accessed. The number of data records a mailbox can store is configurable. Mailboxes are especially suited for data buffering between tasks or between interrupt handlers and tasks.

Message passing serves to exchange data directly between two tasks; specifically, no data or signals will be buffered. This represents the tightest coupling between tasks because the tasks involved must synchronize for the data exchange.

Reentrance

The term reentrance denotes the problems that can occur when the same code is simultaneously executed by several tasks or when global data is simultaneously accessed by several tasks. Reentrant code means that a task may "re-enter" the code before another task has left it. In a multitasking environment, care should be taken that as much code as possible is reentrant so that it may be used by several tasks.

The problems that can occur when global data is used shall be illustrated by a little example. Assuming that two tasks are supposed to count something and the program contains a global variable "Counter" of type int which is initialized with the value 0. Both tasks perform the statement

```
Counter = Counter + 1;
```

whenever an event to be counted occurs. This statement might be translated by the compiler as follows:

```
MOV    EAX, Counter                ; line 1
ADD     EAX, 1                     ; line 2
MOV     Counter, EAX               ; line 3
```

For the sake of simplicity, assume that both tasks have the same priority, preemptive scheduling is enabled, and time slicing is active.

Now the following could happen: Task 1 recognizes an event and begins to execute the machine language instructions given above. After line 1 has been executed (register EAX contains 0), a time slice task switch occurs and Task 2 is activated. Task 2 also recognizes an event and increments variable Counter to a value of 1 without being interrupted. At some later time, Task 1 resumes (at line 2), increments EAX from 0 to 1 and stores this value in the variable Counter. Even though Counter has been incremented twice, it still has a value of 1.

This example shows that two - or more - tasks must never access global data simultaneously if at least one of the tasks can modify the data. Actually, it is not the code which is non-reentrant, but rather, it is the data being manipulated by the code. Even when the statement "Counter = Counter + 1;" is included in both tasks separately, the reentrance problem is not solved.

As a consequence, shared global data should be avoided. The same is true for local variables declared as static. If this is impossible, access to global data should be protected by semaphores. Chapter 7, *Mutual Exclusion* discusses how to implement *mutual exclusion* areas in order to avoid access conflicts.

Unfortunately, some parts of the code are not under the programmer's control, e.g., the run-time system libraries. Please refer to Chapter 7, *Reentrance of the C/C++ Run-Time Systems* for a discussion of solving reentrance problems of the run-time library.

Chapter 2

Module RTKernel-32

Module RTKernel-32 contains all constants, types, and functions needed for multitasking operation. A multitasking program should `#include` file `RTK32.H`; it contains the declarations detailed in this chapter.

All identifiers declared `PUBLIC` in module RTKernel-32 start with the letters "RTK" in order to avoid naming conflicts with other modules. If naming conflicts with other identifiers (e.g., types or constants) do arise, the identifier concerned can be renamed in `RTK32.H`.

RTKernel-32 Configuration

Some of the kernel's configuration parameters are stored in the global structure `RTK32Config` with the following layout:

```
typedef struct {
    int    StructureSize;
    DWORD  DriverFlags;
    DWORD  UserDriverFlags;
    DWORD  Flags;
    DWORD  DefaultTaskStackSize;
    DWORD  DefaultIntStackSize;
    int    MainPriority;
    int    DefaultPriority;
    DWORD  HookedIRQs;
    DWORD  TaskStackOverhead;
    RTKDuration TimeSlice;
} RTK32Config;
```

A default version of this structure is included in the RTKernel-32 library. The source code of the default configuration is contained in file `MODULES\RTKCFG.C`. If you want to define your own configuration, include this structure in your program.

Example:

```
RTK32Config RTKConfig = {
    sizeof(RTK32Config),           // StructureSize
    0,                             // Driver flags
    0,                             // User driver flags
    RF_AUTOINIT | RF_TRACE |      // Flags
    RF_TCPU TIME | RF_STACKCHECKS, // ditto
    16*1024,                       // DefaultTaskStackSize
    512,                           // DefaultIntStackSize
    5,                             // MainPriority
    0,                             // DefaultPriority
    0x00000003,                   // HookedInterrupts (IRQ 0 and 1)
    256,                           // TaskStackOverhead
    0                             // Time slice (0==off)
};
```

The example given above corresponds to RTKernel-32's defaults.

Alternatively, you can modify parts of `RTKConfig` at run time. However, care must be taken not to change them after RTKernel-32 has read and used them. Most values should only be changed before `RTKernelInit` is called.

The various fields of `RTK32Config` are described below.

StructureSize

This field must always be equal to `sizeof(RTK32Config)`; if not, RTKernel-32 will generate a fatal error.

DriverFlags

DriverFlags defines 32 bits which may be enquired by the drivers supplied with RTKernel-32. Currently, the following values are defined:

DF_TIMER_CHAIN	If this flag is specified, the clock driver (which supplies timer interrupts to the kernel) will chain to the interrupt handler installed before RTKernel-32 was loaded. Specify this flag if you have other software components installed which require the timer interrupt. Setting this flag can degrade RTKernel-32's accuracy for activating tasks waiting for a specific point in time. Interrupt latency can also be affected adversely. By default, this flag is not set.
DF_IDLE_HALT	Some system drivers can execute the CPU instruction Halt in RTKernel-32's Idle Task, significantly reducing power consumption and heat generation. However, the target computer must be able to handle Halt bus cycles, which is frequently not the case for low-cost embedded systems. If this flag is set, Halt will be executed by the Idle Task if the program runs at CPL 0 and preemptions are enabled. By default, this flag is not set.

UserDriverFlags

Field UserDriverFlags is not used by RTKernel-32 or any drivers supplied with it. It is reserved for user-defined drivers. If you intend to write your own drivers requiring options definable at run-time, use this field to specify them.

Flags

Field Flags controls various options of the kernel itself. Currently, the following flags are available:

RF_TCPU TIME	If set, RTKernel-32 will accumulate the CPU time consumed by each task. This flag is only recognized by the Debug Version of RTKernel-32. The Standard Version ignores it. It is set by default.
RF_ICPU TIME	<p>If set, RTKernel-32 will accumulate the CPU time consumed by each interrupt handler. When RF_ICPU TIME is set, RTKernel-32 will also set RF_TCPU TIME automatically. RF_ICPU TIME is only recognized by the Debug Version of RTKernel-32. The Standard Version ignores it. It is not set by default.</p> <p>This flag can severely degrade interrupt latency, primarily depending on the speed of the high resolution timer driver being used. For example, the Pentium timer is very fast and thus will not degrade performance significantly. However, the default PC timer driver may be too slow for many systems. Use RF_ICPU TIME only for testing and performance analysis.</p>
RF_PREEMPTIVE	<p>Setting this flag instructs RTKernel-32 to start up in preemptive mode. It is not set by default, causing RTKernel-32 to run cooperatively.</p> <p>You can change the scheduling mode at any time using RTKPreemptionsON and RTKPreemptionsOFF.</p>
RF_AUTOINIT	<p>Specifies that RTKernel-32 should automatically initialize itself on the first call to RTKCreateThread. This flag is set by default. If you attempt to create tasks before the kernel is initialized and this flag is not set, RTKernel-32 will generate a fatal error message.</p> <p>With automatic initialization, RTKCreateThread calls</p> <pre>RTKernelInit(RTKConfig.MainPriority);</pre> <p>and there is no need for the application to call RTKernelInit explicitly.</p>
RF_TRACE	This flag is set by default and enables the kernel tracer. It is only recognized by RTKernel-32's Debug Version and is ignored by the Standard Version.
RF_FPCONTEXT	This flag is not set by default. If set, RTKernel-32 will by default maintain a floating point context for each task.

Note that `RF_FPCONTEXT` only specifies RTKernel-32's default behavior. You can specify a floating point context for each task in the call to `RTKCreateThread`, overriding the default set by `RF_FPCONTEXT`. In addition, floating point context swapping can be switched on and off dynamically at run time using `RTKProtect8087()` and `RTKFree8087()`.

`RF_STACKCHECKS` This flag is set by default and is only recognized by the kernel's Debug Version. When set, it causes RTKernel-32 to call `RTKStackCheck` on entry to every RTKernel-32 API function. This flag can be set and reset dynamically at run-time. It is checked on every kernel entry. You must reset this flag at least temporarily if the stack is set to some data area not allocated by RTKernel-32.

`RF_PULSWIN32` This flag is not set by default. If it is set, `RTKPulse` behaves just like the Win32 `PulseEvent` function. If no task is being released by the call to `RTKPulse`, the event is left in a signalled state. RTKernel-32's default behavior is to always reset the event, regardless of the number of tasks released.

You should set this flag if you prefer Win32's behavior (see Chapter 3, *Function PulseEvent*).

`RF_WIN32CS_MUTEX` If set, RTKernel-32 will use mutex semaphores to implement Win32 Critical Sections. By default, binary semaphores with recursion support are used.

`RF_WIN32MUTEX_MUTEX` If set, RTKernel-32 will use mutex semaphores to implement Win32 Mutex semaphores. By default, binary semaphores with recursion support are used.

DefaultTaskStackSize

This field specifies the default task stack size. This value is used if the stack size parameter for `RTKCreateThread` is zero. The default is 16k.

DefaultIntStackSize

This field specifies the default interrupt stack size. The default is 512. You can change the interrupt stack size for each IRQ with `RTKSetIRQStack`.

MainPriority

This field defines the priority of the main task if zero is specified in the call to `RTKKernelInit` or when the kernel is initialized through auto initialization. This value also defines the absolute value of Win32's relative priority `PRIORITY_NORMAL`. If this field is zero, `RTKConfig.DefaultPriority` is used instead; its default value is 5.

DefaultPriority

When the Priority parameter to `RTKCreateThread` is zero, this value is used instead. If `DefaultPriority` is also zero, the new task inherits its priority from the creating task.

HookedIRQs

This field defines a bit for each IRQ on which RTKernel-32 should install its low-level interrupt handlers even when no interrupt handler is installed through RTKernel-32's interrupt API. You should set these bits for all interrupt handlers which are installed before RTKernel-32 is initialized by software. This will ensure that these interrupt handlers are executed on their own interrupt stack and will protect them from preemptive task switches. By default, only bit 0 (for IRQ 0) is set.

TaskStackOverhead

The `TaskStackOverhead` is added to the stack size parameter given for each task to accommodate interrupts and stack space needed by RTKernel-32 internally. It defaults to 256.

TimeSlice

If this field has a value greater zero, RTKernel-32 will start up with time slicing enabled. The default value is zero (no time slicing). Please note that preemptions and time slicing are independent of each other.

RTKernel-32 Initialization

Before tasks can be created, RTKernel-32 must be initialized by function `RTKernelInit`. `RTKernelInit` must either be called explicitly by the application or implicitly by the first `RTKCreateThread` call of the program. Automatic initialization can be disabled by not setting the `RF_AUTOINIT` flag in `RTKConfig.Flags`.

Function `RTKernelInit`

Function `RTKernelInit` initializes RTKernel-32:

```
RTKTaskHandle RTKernelInit(unsigned MainPrio);
```

Parameter `MainPrio` defines the priority of the main task. If it is zero, `RTKConfig.MainPriority` is used. If `RTKConfig.MainPriority` also is zero, `RTKConfig.DefaultPriority` is used. A floating point context is maintained for the main task if `RF_FPCONTEXT` is set in `RTKConfig.Flags`.

`RTKernelInit` will call the initialization routines of all drivers, install its low-level interrupt handlers for all IRQs specified in `RTKConfig.HookedIRQs`, create the Idle Task, and install an exit function using `atexit`. If the kernel has already been initialized through a previous explicit or implicit call to `RTKernelInit`, only the priority of the main task is set and no other actions are performed.

The task handle of the main task is returned.

RTKernel-32 Exit Function

RTKernel-32's exit function is called by function `exit()` or after the return of function `main()`.

All tasks (except the active task and the Idle Task) are suspended.

Finally, all interrupt vectors modified by RTKernel-32 are restored to their original values.

Task Management

This section describes the RTKernel-32 operations for creating, terminating, and managing tasks.

Function `RTKCreateThread`

Any task in a program can create other tasks using this function:

```
RTKTaskHandle RTKCreateThread(RTKThreadFunction TaskCode,
                              unsigned           Priority,
                              unsigned           StackSize,
                              unsigned           Flags,
                              void              * Parameter,
                              const char        * Name);
```

Parameter `TaskCode` is a C function with `__cdecl` calling conventions and a single `void *` parameter. It contains the code to be executed by the new task. Any number of tasks can be created using the same function. All these tasks would execute the same code, but each would have its own local data. Methods of classes cannot be used for parameter `TaskCode`. Starting a task with a class method is discussed in Chapter 7, *Starting Objects' Methods as Tasks*.

Parameter `Priority` is the base priority of the new task as an integer between `MIN_PRIO` (1) and `MAX_PRIO` (64). Alternatively, zero may be specified to instruct RTKernel-32 to use the default priority given in `RTKConfig.DefaultPriority`. If this value also is zero, the new task is created with the same base priority as the creating task. The higher the priority, the higher is the urgency of the task. If the priority of the new task is higher than that of the creating task and the task is not created in suspended state, the new task is immediately activated.

RTKernel-32 distinguishes between the *base priority* and the *execution priority* of a task. For scheduling purposes, only the execution priority is considered. The execution priority of a task is the maximum of its base priority and the priorities of all resource or mutex semaphores occupied by the task. The priority of a resource or mutex semaphore is the highest priority of all tasks waiting at the semaphore. Please refer to section *Semaphores* for details on resource and mutex semaphores and priority inheritance.

Parameter `StackSize` is the net stack required by the new task (in bytes). RTKernel-32 adds the value `RTKConfig.TaskStackOverhead` to the value specified and allocates a memory block of corresponding size. If parameter `StackSize` is zero, `RTKConfig.DefaultStackSize` is used instead.

Please note that stack overflows are one of the most common and most evasive errors of multitasking programs. During program development, use functions `RTKGetTaskStack`, `RTKGetMinStack`, and `RTKTaskInfo` extensively to analyze actual stack usage of your tasks. At least during the program development phase, stacks should be dimensioned generously (at least 8k - 16k).

Parameter `Flags` can be used to select options for the new task. The following values are currently defined:

<code>TF_SUSPENDED</code>	If this flag is specified, the task is created in suspended state. It does not start running until <code>RTKResume</code> is called.
<code>TF_MATH_CONTEXT</code>	This flag instructs RTKernel-32 to maintain a floating point context for the new task. You cannot combine this flag with <code>TF_NO_MATH_CONTEXT</code> .
<code>TF_NO_MATH_CONTEXT</code>	This flag instructs RTKernel-32 not to maintain a floating point context for the new task. You cannot combine this flag with <code>TF_MATH_CONTEXT</code> .

If neither `TF_MATH_CONTEXT` nor `TF_NO_MATH_CONTEXT` are given, RTKernel-32 will set up a floating point context if `RF_FPCONTEXT` is set in `RTKConfig.Flags`.

Parameter `Parameter` is passed to the task's function. RTKernel-32 does not interpret this value; it is just passed on. You can use it to pass arbitrary information to the new task.

Parameter `Name` is a pointer to the name of the task. The name of the new task is only used for easy identification of the task and should not be longer than 15 characters. RTKernel-32 just copies the pointer to the name. Therefore, the name should not be modified after the task has been created.

The function value returned to the caller is an RTKernel-32 task handle (not a Win32 handle). It is a reference to the newly created task.

In addition to the stack, RTKernel-32 allocates a TCB (Task Control Block, used by RTKernel-32 internally) and - if required - a buffer for the floating point context. If RTKernel-32 is unable to allocate sufficient memory through its memory driver, the program is aborted with a corresponding error message. Please note that creating a task with a higher priority immediately leads to a task switch if the task is not created in suspended state.

Function `RTKRTLCreateThread`

Function `RTKRTLCreateThread` creates a thread with multithread run-time system library support:

```
RTKTaskHandle RTKRTLCreateThread(RTKThreadFunction TaskCode,
                                unsigned           Priority,
                                unsigned           StackSize,
                                unsigned           Flags,
                                void               * Parameter,
                                const char        * Name);
```

This function has the same parameters as function `RTKCreateThread` (see previous section). The only difference is that the program's C/C++ or Pascal run-time system is made aware of the new thread. This is required to avoid reentrance problems when several threads call non-reentrant run-time systems function.

`RTKRTLCreateThread` requires the compiler's multithread run-time system libraries to be linked.

Threads created with `RTKRTLCreateThread` should not be terminated with `RTKTerminateThread`, because resources allocated by the run-time system for the thread would not be deallocated. Instead, such threads should return from their thread function or they should call whatever terminating function is made available by the run-time system.

Function `RTKTerminateTask`

Function `RTKTerminateTask` normally does not need to be called. The natural termination of a task occurs when the task reaches the end of its task function. Only the Main Task terminates all tasks (and, consequently, the whole program) when it reaches the end of function `main`, or when function `exit` is called.

```
void RTKTerminateTask(RTKTaskHandle * Handle);
```

Parameter `Handle` is a pointer to a task handle identifying the task to be terminated. It must have been assigned a legal value by `RTKCreateThread` or `RTKCurrentTaskHandle`. If an illegal value is passed to `RTKTerminateTask`, the program is aborted with an error message. `RTKTerminateTask` initiates a task switch if the task to be terminated is the active task. `Handle` is assigned the value `RTK_NO_TASK`.

Prior to actually terminating the task, `RTKernel-32` makes sure the task is not occupying a resource or mutex semaphore (see section *Semaphores*). If this is the case, the task continues to run until all resources have been released. The task calling `RTKTerminateTask` does not wait until task termination has been completed, but continues to run immediately.

Stack and TCB of the task to be terminated are deallocated, provided that `*Handle` does not reference the current task. The memory of a task terminating itself is released by the next call to `RTKCreateThread` or `RTKDeallocTerminatedTasks`. Until then, the task still exists in the state `Terminated`; however, it cannot run any more.

Care should be taken that a task is not terminated twice (e.g., by itself and another task), because the task would not exist any more when `RTKTerminateTask` is called the second time. In this case, `RTKernel-32` would abort the program with an error message since the handle would have an illegal value.

Threads created with `RTKRTLCreateThread` should not be terminated with `RTKTerminateThread`, because resources allocated by the run-time system for the thread would not be deallocated. Instead, such threads should return from their thread function or they should call whatever terminating function is made available by the run-time system.

As in sequential programs, function `exit` can be called at any time to terminate the program (and consequently all tasks). However, function `abort` should be avoided because it does not restore the interrupt vectors.

Function `RTKSuspend`

Function `RTKSuspend` can be used to deactivate a task. Thereafter, the task can only be reactivated by calling function `RTKResume`.

```
void RTKSuspend(RTKTaskHandle Handle);
```

Parameter `Handle` references the task to deactivate. If the task is already suspended, `RTKSuspend` has no effect.

Prior to actually suspending the task, `RTKernel-32` makes sure the task is not occupying a resource or mutex semaphore. In this case, the task continues to run until all resources have been released. The suspending task does not wait until task termination has been completed, but continues to run immediately.

Function `RTKResume`

`RTKResume` is the counterpart of `RTKSuspend`. A task suspended by `RTKSuspend` can be reactivated using `RTKResume`.

```
void RTKResume(RTKTaskHandle Handle);
```

If task "`Handle`" is not suspended, `RTKResume` has no effect.

Function RTKSetPriority

Function RTKSetPriority serves to change a task's priority.

```
void RTKSetPriority(RTKTaskHandle Handle, unsigned Priority);
```

Parameter Handle references the task whose priority to change. Parameter Priority is the new base priority of the task. It must be in the range MIN_PRIO to MAX_PRIO. After a call to RTKSetPriority, the execution priority of the task is re-evaluated and the scheduler performs a task switch if required.

Function RTKProtect8087

In addition to the main processor, a task can use a math coprocessor, if installed, or an 80387 software emulator. In this case, the state of the coprocessor will also be saved and restored during a task switch. Coprocessor/emulator usage can be turned on or off separately for each task using RTKProtect8087 and RTKFree8087. The initial floating point usage of a task is controlled by the Flags parameter to RTKCreateThread or flag RF_FPCONTEXT in RTKConfig.Flags.

RTKProtect8087 turns on coprocessor protection:

```
void RTKProtect8087(void);
```

The coprocessor task switch is only performed if actually required.

The exact behavior of floating point context switching is determined by the floating point driver used by RTKernel-32. For further details, please refer to Chapter 5, *Floating Point*.

If tasks use the coprocessor without coprocessor protection enabled, calculations may yield wrong results or the program may crash due to an error exception of the coprocessor.

During cooperative scheduling, coprocessor protection is not required at all if it is guaranteed that task switches cannot occur during the evaluation of a floating point expression.

If RTKernel-32 finds that no floating point context switching is required by the floating point driver (for example, because the emulator used is reentrant), all calls to RTKProtect8087 are ignored.

Function RTKFree8087

Function RTKFree8087 can be used to turn off coprocessor/emulator protection:

```
void RTKFree8087(void);
```

This may result in significantly improved task switch times.

Function RTKAllocUserData

RTKernel-32 reserves up to 16 user data entries for each task. Each entry is a void pointer and can be used by the application for arbitrary purposes.

To reserve an entry for each task, the following function call can be used:

```
int RTKAllocUserData(void);
```

The function result is a valid index for every task, even those created at a later time. It can be used as a parameter for RTKGet/SetUserData.

If all 16 user data entries are already occupied, the program is aborted by RTKAllocUserdata with an error message.

RTKernel-32's user data corresponds to Win32's TLS data. However, unlike Win32, RTKernel-32 also allows access to the TLS of tasks other than the current task. However, task user data indexes and TLS indexes should not be mixed (e.g., do not call TlsGetValue with a task user data index).

Function RTKSetUserData

A task's user data entry can be set to a specific value with the following function call:

```
void RTKSetUserData(RTKTaskHandle Handle, int Index, void * UserData);
```


Parameter Index must be a value returned by RTKAllocUserData. Parameter UserData is stored in the TCB of the task referred to by parameter Handle and can be retrieved later using RTKGetUserData or RTKGetLocalData.

Function RTKGetUserData

A task's user data entry set by function RTKSetUserData can be retrieved using this function:

```
void * RTKGetUserData(RTKTaskHandle Handle, int Index);
```

If no entry has been set for the respective task using RTKSetUserData, NULL is returned.

Function RTKGetLocalData

This function returns a user data entry for the currently running task:

```
void * RTKGetLocalData(int Index);
```

This function returns the same result as

```
RTKGetUserData(RTKGetCurrentTaskHandle(), Index);
```

but is faster.

Enquiring Tasks

This section discusses all RTKernel-32 operations for enquiring the properties of tasks.

Function RTKCurrentTaskHandle

Function RTKCurrentTaskHandle returns the handle of the currently running task:

```
RTKTaskHandle RTKCurrentTaskHandle(void);
```

Function RTKGetTaskState

Function RTKGetTaskState returns the current state of a task:

```
RTKTaskState RTKGetTaskState(RTKTaskHandle Handle);
```

The result is of enumeration type RTKTaskState. One of the following values can be returned:

TS_READY	The task is ready to run.
TS_CURRENT	The task is running.
TS_SUSPENDED	The task has been suspended by a call to RTKSuspend.
TS_DELAYING	The task is waiting in a call to RTKDelay or RTKDelayUntil.
TS_BLOCKED_WAIT	The task is waiting in a call to RTKWait at a semaphore.
TS_TIMED_WAIT	The task is waiting in a call to RTKWaitTimed at a semaphore.
TS_BLOCKED_PUT	The task is waiting in a call to RTKPut at a full mailbox.
TS_BLOCKED_GET	The task is waiting in a call to RTKGet at an empty mailbox.
TS_TIMED_PUT	The task is waiting in a call to RTKPutTimed at a full mailbox.
TS_TIMED_GET	The task is waiting in a call to RTKGetTimed at an empty mailbox.
TS_BLOCKED_SEND	The task is waiting in a call to RTKSend for the receiver task.
TS_BLOCKED_RECEIVE	The task is waiting in a call to RTKReceive to receive data.
TS_TIMED_SEND	The task is waiting in a call to RTKSendTimed for the receiver task.
TS_TIMED_RECEIVE	The task is waiting in a call to RTKReceiveTimed to receive data.
TS_DEADLOCKED	The task is blocked in a send operation (message passing) and the receiver task has terminated in the meantime.

TS_ILLEGAL The handle passed does not reference an existing task. Every other RTKernel-32 operation expecting a task handle as a parameter will abort the program with an error message if an invalid handle is passed to it. You can use RTKGetTaskState to verify handles.

TS_TERMINATED The task has terminated itself by calling RTKTerminateTask with its own handle or by reaching the end of its task function. The task cannot run any more but it still exists because its memory has not yet been deallocated.

Function RTKGetTaskPrio

Function RTKGetTaskPrio returns the current execution priority of a task:

```
unsigned RTKGetTaskPrio(RTKTaskHandle Handle);
```

Parameter Handle references the task whose priority to enquire.

Function RTKGetTaskStack

Function RTKGetTaskStack returns the remaining free stack space of a task:

```
unsigned RTKGetTaskStack(RTKTaskHandle Handle);
```

Parameter Handle references the task whose stack to enquire. To enquire the stack of the current task, RTKGetTaskStack(RTKCurrentTaskHandle()) can be used. Please note that sufficient stack space for interrupts must be available at all times.

Depending on the system driver, RTKernel-32 may not be able to determine the stack limits of the main task. In this case, the value FFFFFFFFh is returned for it. The actual free stack space can never have this value.

Function RTKGetMinStack

Function RTKGetMinStack returns the least number of bytes that has been free on a task's stack since it was created:

```
unsigned RTKGetMinStack(RTKTaskHandle Handle);
```

Parameter Handle references the task whose stack to enquire.

RTKernel-32 initializes the stack memory of a task with the ASCII code of the letter 'S'. RTKGetMinStack searches the stack from bottom to top for a byte not having value 'S' to determine how far the stack has been used.

Depending on the system driver, RTKernel-32 may not be able to determine the stack limits of the main task. In this case, the value FFFFFFFFh is returned for it. The actual free stack space can never have this value.

Function RTKTaskInfo

RTKTaskInfo can write a list of all currently existing tasks to a string:

```
void RTKTaskInfo(char * Buffer, unsigned BufferSize, unsigned ListFlags);
```

Parameter Buffer points to the string to store the list in. BufferSize specifies the size of the buffer. RTKTaskInfo will write no more than BufferSize characters to *Buffer. Parameter ListFlags is a sum of individual list flags. Each list flag will produce a specific column in the task list.

Some of the list flags are only available in RTKernel-32's Debug Version. If they are used in the Standard Version, only a dash is displayed in the corresponding columns.

RTKTaskInfo is indispensable during the program development phase. You should always provide for interactively displaying a task list containing at least the tasks' names and states. Please try the commands 'TASKS1' and 'TASKS2' in program RTDemo for examples.

The discrete list flags are:

- LF_TASK_HANDLE** Heading: Handle. The task's handle in hexadecimal notation.
- LF_TASK_NAME** Heading: Name. The name of the task passed to RTKCreateThread.

LF_BASE_PRIO	Heading: BPrio. The task's base priority defined by RTKCreateThread. The base priority can be modified at run-time using RTKSetPriority.
LF_TASK_PRIO	Heading: Prio. The task's execution priority. Due to priority inheritance, the execution priority of a task can temporarily be higher than the base priority.
LF_TASK_STATE	Heading: State. The task's state.
LF_REL_DELAY	Heading: R.Del. For tasks in a timed kernel operation, this is the number of ticks the task must wait until the timeout occurs.
LF_ABS_DELAY	Heading: A.Del. For tasks in a timed kernel operation, this is the absolute time when the timeout expires.
LF_FREE_STACK	Heading: FStck. The current free stack of the task. If RTKernel-32 cannot determine the stack limits, only a dash is displayed (also refer to section <i>Function RTKGetTaskStack</i>).
LF_MIN_STACK	Heading: MStck (MinFreeStack). Number of bytes in the task's stack space that have never been used by the task. If RTKernel-32 cannot determine the stack limits, only a dash is displayed (also refer to section <i>Function RTKGetMinStack</i>).
LF_COPROCESSOR	Heading: 80387. Shows whether RTKernel-32 maintains a floating point context for the task.
LF_TASK_SWITCHES	Heading: Scheds. Number of task switches to the respective task.
LF_CPU_TIME	Heading: CTime. CPU time usage of the respective task in seconds. This column is only available in the Debug Version when flag RF_TCPU- TIME is set in RTKConfig.Flags.
LF_REL_CPU_TIME	Heading: CT%. Percentage of available CPU time allocated to the respective task. This column is only available in the Debug Version when flag RF_TCPU- TIME is set in RTKConfig.Flags.
LF_WAITING_AT_POS	Heading: CodePos. Address where the respective task will resume when activated by the kernel. If the task was interrupted by an interrupt, this address cannot be determined in most cases and only a dash is displayed. If the program uses a source code position driver and has loaded a symbol table, the name and line number of the respective source file are displayed instead of a hex address. For more information, please refer to Function RTKLoad- Symbols. This column is only available in the Debug Version.
LF_WAITING_AT_OBJ	Heading: WaitingAt. The object the task is waiting at. Depending on the task's state, this can be another task, a semaphore, a mailbox, or no object.
LF_RESOURCES	Heading: Resources. Displays a list of all resources (resource or mutex semaphores) currently occupied by the respective task. This information can be very useful in discovering deadlocks; however, this requires that resource management is indeed implemented using resource or mutex semaphores (which is strongly recommended).
LF_LIST_ALL	LF_LIST_ALL is the sum of all available list flags.

CPU time usage and the number of task switches are cumulated since program startup or the last call to RTKClearStatistic.

The width of the list's up to 16 columns is determined by the respective column headings. These are stored in array RTKListTitles and may be modified by the application at run-time to obtain a different list layout. If, for example, you find column LF_CPU_TIME too narrow, you can create a column of width 10 and a right-justified heading using this assignment:

```
RTKListTitles[11] = "    CPU Time";
```

Function RTKClearStatistic

Function RTKClearStatistic clears the task switch counters and CPU time usage statistics for all tasks and interrupts:

```
void RTKClearStatistic(void);
```

Function RTKLoadSymbols

With RTKernel-32's Debug Version, function RTKTaskInfo and error messages can display tasks' code positions at source level after a symbol table has been loaded using function RTKLoadSymbols:

```
int RTKLoadSymbols(const char * FileName);
```

The type of file accepted depends on the source code position driver being used. Currently, the only driver supplied with RTKernel-32 is SRCTDS. It requires a TDS or EXE file with Borland Debugger symbol tables, the same debug symbol table format supported by RTTarget-32's debugger RTD32.

The return value indicates the success of the operation. The following values can be returned:

- 0 Symbol table loaded successfully
- 1 File containing symbol table could not be opened
- 2 Invalid symbol table file type
- 3 Not enough memory for symbol table
- 4 No line numbering information found
- 6 Display of code positions not supported

Time

For a real-time system, time is of great importance. RTKernel-32 maintains an interrupt-driven clock which can be set and read. Furthermore, a task can block itself for a certain time span in order to make CPU time available to other tasks.

Time is expressed in *timer ticks*. RTKernel-32 uses type *RTKTime* to store times and type *RTKDuration* to store time intervals. Both Time and Duration have type signed long and thus can span a range of 2^{32} ticks.

RTKernel-32 gets its timer interrupts from the clock device driver. The clock driver is only required to generate periodic interrupts. RTKernel-32 does not (and need not) know how much real time (i.e., time expressed in seconds) elapses between two consecutive timer ticks. For information on translating timer ticks to real time, please refer to Chapter 4, *Module Clock*.

Since RTKernel-32's clock is limited to a resolution of 32 bits, an overflow will occur after 2^{31} timer ticks. For example, if the timer interrupt interval is set to 1 millisecond, the first overflow will occur after about 25 days, followed by periodic overflows every 50 days. Such an overflow is no problem for RTKernel-32; the behavior of tasks is not affected by clock overflows. RTKernel-32 increments the clock from 2147483647 to -2147483648. Application programs storing times in variables of type RTKTime or RTKDuration are compatible with RTKernel-32's behavior. However, times stored in variables that overflow differently (e.g., floats or doubles), must be corrected by the application when RTKernel-32's clock overflows.

Function RTKSetTime

Function RTKSetTime can be used to set RTKernel-32's internal clock:

```
void RTKSetTime(RTKTime NewTime);
```

Parameter NewTime is the new value for RTKernel-32's internal clock. Negative values are allowed. RTKernelInit calls RTKSetTime(0).

The behavior of tasks waiting in an RTKDelay, RTKDelayUntil, or a timed operation is not affected by RTKSetTime; i.e., the point in real-time when the timeout occurs stays the same.

Function RTKGetTime

Function RTKGetTime can be used to read RTKernel-32's clock.

```
RTKTime RTKGetTime(void);
```

The return value is the current time of RTKernel-32's internal clock. If RTKSetTime has not been called, RTKGetTime returns the number of timer ticks since program start (if no overflow has occurred).

Function RTKDelay

RTKDelay blocks the calling task for the specified time span and allows other tasks to run.

```
void RTKDelay(RTKDuration Ticks);
```

Parameter Ticks specifies the number of timer interrupts the task must be blocked. Please note that the actual time interval of suspension will lie in the range [(Ticks-1) .. Ticks], depending on how long ago the last timer interrupt occurred.

If RTKDelay is called with $Ticks \leq 0$, RTKernel-32 checks whether other tasks with the same or a higher priority are ready. If a ready task with higher priority is found (this can only happen during cooperative scheduling), it is activated. Otherwise, if tasks with the same priority are ready, the task not having run longest is activated. Thus, RTKDelay(0) can be used to implement *round-robin* scheduling, also known as *cooperative time slicing*. Time slicing need not be enabled for this purpose.

Function RTKDelayUntil

Tasks that want to continue running at a certain time can use RTKDelayUntil:

```
void RTKDelayUntil(RTKTime Ticks);
```

Parameter Ticks specifies the absolute time for the task to continue. RTKDelay, on the other hand, interprets its parameter as a time interval during which the task should not run. RTKDelayUntil can be used to implement cyclic tasks that run in a fixed time frame (see Chapter 7, *Cyclic Tasks (Timer)*).

Function RTKTimeSlice

The term time slicing denotes forced task switches after a fixed time interval if other tasks with the same priority as the active task are ready. Function RTKTimeSlice activates this algorithm:

```
void RTKTimeSlice(RTKDuration Ticks);
```

Ticks defines the maximum time interval a task is allowed to run before another task is activated. If $Ticks \leq 0$, time slicing is disabled (RTKernel-32's default).

Time slicing only takes effect when several tasks having the same priority are ready and no tasks having a higher priority can run. If, for example, three tasks of priority 7 are ready and another of priority 8, only the task of priority 8 will run. As soon as this task blocks itself, however, the other three are activated in turn after Ticks timer ticks.

It is not required to activate preemptions for time slicing. During cooperative scheduling, time slice task switches are not triggered by the timer interrupt handler. Instead, they are performed in the next kernel call, if required.

Semaphores

Semaphores are a popular mechanism for synchronizing tasks. A semaphore can be thought of as an event counter which can never become negative. A semaphore can be used by any number of tasks using the functions described in this section. A program can use any number of semaphores.

Function RTKSignal stores an event in a semaphore; RTKWait retrieves an event from the semaphore if one is available; otherwise, it waits until an event is signalled. RTKernel-32 distinguishes five types of semaphores: counting, binary, event, resource, and mutex semaphores. A semaphore's type is set when the semaphore is created.

Counting semaphores can store up to $2^{32}-1$ events. They are suitable for general synchronization purposes and comply with the definitions given by Dijkstra¹⁶ or Ben-Ari¹⁷.

Binary semaphores cannot count, they can only accept the values 0 and 1. Function `RTKSignal` will always set a binary semaphore to value 1, even if it already has a value of 1; `RTKWait` always sets its value to 0.

Event semaphores are similar to binary semaphores. However, executing `RTKWait` will not decrement the semaphore's count value. Thus, a single call to `RTKSignal` can release any number of tasks waiting at the event semaphore. To force an event semaphore back to zero, you must explicitly call function `RTKResetEvent` or `RTKPulse`, which are only available for event semaphores. Function `RTKSignal` will always set an event semaphore to value 1. `RTKWait` does not change the event semaphore value.

Resource semaphores are especially suited for resource management (also refer to Chapter 7, *Mutual Exclusion*). A resource semaphore guarantees the priority of a task occupying a resource to be at least the maximum of the priorities of all other tasks also requiring the respective resource. This technique is known as *priority inheritance*. A task requesting a resource using `RTKWait` hands down its priority to the task occupying the resource. This makes sure that a high priority task is never unnecessarily blocked by a task of lower priority.

Priority inheritance may be thought of as pseudo-priorities of resource semaphores. Assume that a task wants to occupy a resource using `RTKWait`, but is blocked because the resource is already occupied. In this case, the task passes its priority to the resource semaphore, which in turn passes its priority to the task currently occupying the resource. Priority inheritance only takes place, of course, if it leads to an increase of the respective priority. A task's priority can only be raised by priority inheritance; it can never become lower than its base priority.

Another property of resource semaphores facilitates the safe termination and suspension of tasks. A task can only be suspended or terminated immediately if it does not occupy any resources. Otherwise, it continues to run until it has released all resources and subsequently suspends or terminates itself. This mechanism serves to avoid deadlocks.

Compared to other semaphore types, two important restrictions apply for resource semaphores: a resource must always be released (using `RTKSignal`) by the task that has acquired it using `RTKWait`. A task can occupy any number of resources simultaneously. However, it must release the resources exactly in the reverse sequence as they have been acquired. Example:

```
RTKWait(R1);
RTKWait(R2);
...
RTKSignal(R2);
RTKSignal(R1);
```

If the sequence of `RTKSignal` calls were exchanged, `RTKernel-32's` Debug Version would abort the program with an error message. Since the Debug Version can check for compliance with the rules for resource semaphores, errors like those described in Chapter 8, *Deadlock* cannot occur. This is another important advantage of resource semaphores.

Due to these restrictions, resource semaphores are not suited for use by interrupt handlers.

A variation of resource semaphores are mutex semaphores. The only difference is that a task is allowed to acquire a resource it already owns. Example:

```
RTKWait(R);
RTKWait(R);
...
RTKSignal(R);
RTKSignal(R);
```

¹⁶ E. W. Dijkstra. Co-operating Sequential Processes. In F. Genuys (ed.) *Programming Languages*, Academic Press, New York, 1968

¹⁷ M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall International, Hemel Hemstead, 1990

If R were a resource semaphore, the second call to RTKWait would cause an error (a task cannot acquire a resource it already owns). However, for a mutex semaphore, such a construct is legal and the resource would not be released until the same number of RTKSignal calls as RTKWait calls have occurred.

The disadvantage of mutex semaphores compared to resource semaphores is that unmatched RTKWait/RTKSignal pairs are not detected. If a call to RTKSignal is missing, the resource would never be released and no error is reported, even when the task continues to call RTKWait and RTKSignal on the same mutex.

RTKernel-32 uses resource semaphores to implement its Automatic Library Protection.

The following example shows how to use a semaphore for general synchronization. Assume Task A should be activated when Task B has reached a certain point (Task B is the main program):

```
#include <stdio.h>
#include <rtk32h>

RTKSemaphore S;

void RTKAPI TaskA(void * P)
{
    printf("Task A: waiting on semaphore S\n");
    RTKWait(S);
    printf("Task A: continued\n");
}

int main(void)
{
    printf("\n");
    RTKKernelInit(3);
    S = RTKCreateSemaphore(ST_COUNTING, 0, "Semaphore S");
    printf("Main : creating task A\n");
    RTKCreateThread(TaskA, 4, 0, 0, NULL, "Task A");
    printf("Main : setting semaphore S\n");
    RTKSignal(S);
    printf("Main : done.\n");
    return 0;
}
```

The semaphore is initialized with 0 events. Task A has a higher priority and thus is started immediately after RTKCreateThread. Its second statement is RTKWait(S). Since there are no events stored, Task A is blocked and the main task runs until the semaphore is signalled. When the main task executes the statement RTKSignal(S), Task A is activated immediately and retrieves the event. Both tasks then run to completion and the program terminates.

Using semaphores to implement *mutual exclusion* is discussed in Chapter 7, *Mutual Exclusion*.

The discrete functions for semaphores shall now be introduced.

Function RTKCreateSemaphore

RTKCreateSemaphore creates and initializes a semaphore:

```
RTKSemaphore RTKCreateSemaphore(RTKSemaType  Type,
                                unsigned      InitialValue,
                                const char *  Name);
```

Parameter Type is the desired semaphore type; the values ST_COUNTING, ST_BINARY, ST_EVENT, ST_RESOURCE, and ST_MUTEX are valid. InitialValue must be valid for the chosen semaphore type. For counting semaphores, this is $0 \dots 2^{32}-1$, for binary and event semaphores 0 or 1, and for resource/mutex semaphores it must be 1. Parameter Name is a string of up to 15 characters length, which can be displayed by functions RTKTaskInfo, RTKSemaInfo, and by error messages.

RTKCreateSemaphore allocates the semaphore. The return value is a reference to the new semaphore.

RTKCreateSemaphore is a macro and expands to

```
RTKOpenSemaphore(Type, InitialValue, 0, Name)
```

Function RTKOpenSemaphore

Function RTKOpenSemaphore creates or locates an existing semaphore:

```
RTKSemaphore RTKOpenSemaphore(RTKSemaType  Type,
                               unsigned      InitialValue,
                               unsigned      Flags,
                               const char *  Name);
```

Parameter Flags specifies whether and how the function should search for an existing semaphore. It can take the following values:

- | | |
|-------------------|---|
| 0 | A new semaphore is created unconditionally. |
| SF_SEARCH | RTKOpenSemaphore attempts to locate a semaphore of the same Type and Name that was also created with SF_SEARCH specified. Unnamed semaphores are not considered and name comparison is case sensitive. If found, a reference to this existing semaphore is returned and no new semaphore is created. In this case, Parameter InitialValue is ignored. |
| SF_FAIL_NOT_FOUND | This flag may be specified in addition to SF_SEARCH. It prevents RTKOpenSemaphore from allocating a new semaphore if the search failed. Value RTK_NO_SEMAPHORE is returned in this case. |

Parameters Type, InitialValue, and Name have the same meaning as for RTKCreateSemaphore (see above).

Function RTKDeleteSemaphore

RTKDeleteSemaphore deallocates and invalidates an existing semaphore:

```
void RTKDeleteSemaphore(RTKSemaphore * S);
```

Parameter S must point to a valid semaphore. No task may be queued to access the semaphore. After successful deletion, RTKernel-32 assigns value RTK_NO_SEMAPHORE to *S.

The attempt to access a semaphore after its deletion will produce a fatal error under the Debug Version. In the Standard Version, the results are unpredictable.

Function RTKSemaInfo

RTKSemaInfo writes a list of all currently existing semaphores to a string buffer:

```
void RTKSemaInfo(char * Buffer, unsigned BufferLen);
```

Parameter Buffer is a pointer to the string to receive the list. BufferLen specifies the size of the buffer. It should have at least 120+n*60 bytes, where n is the number of currently existing semaphores.

RTKSemaInfo will display the semaphore's names, types, values, and a list of all tasks waiting at the respective semaphore. For resource and mutex semaphores, the name of the owning task (if any) is also given.

RTKSemaInfo is intended primarily for debugging purposes.

Function RTKSemaValue

Using function RTKSemaValue, the number of events stored in a semaphore can be enquired. It never leads to a task switch.

```
unsigned RTKSemaValue(RTKSemaphore S);
```

Parameter S references the semaphore to enquire. The number of events is returned.

Function RTKResourceOwner

Function RTKResourceOwner can be used to enquire which task currently occupies a resource or mutex semaphore:

```
RTKTaskHandle RTKResourceOwner(RTKSemaphore S);
```


Parameter *S* references the resource or mutex semaphore to enquire. If the semaphore is free, the function returns the value `RTK_NO_TASK`; otherwise, the task handle of the task occupying the resource is returned.

This function can only be used for resource and mutex semaphores.

Function `RTKSignal`

Function `RTKSignal` stores an event in a semaphore.

```
void RTKSignal(RTKSemaphore S);
```

S references the semaphore to store the event. If one or more tasks are waiting at the semaphore, the task with the highest priority is made Ready. If its priority is higher than that of the calling task, it is activated immediately. If the semaphore has type `ST_EVENT`, all waiting tasks are released.

If no task is waiting at the semaphore, the event is stored. A counting semaphore can store up to $2^{32}-1$ events. The application should make sure that this limit is not exceeded. Binary and event semaphores ignore additional events if they already contain one.

Attempting to set a resource or mutex semaphore to a value > 1 using `RTKSignal` is considered an error. In this case, the Debug Version will issue an error message and abort the program; in the Standard Version, the results are unpredictable.

If *S* is a resource or mutex semaphore, the priority of the active task is re-evaluated following the rules of priority inheritance.

Function `RTKPulse`

Function `RTKPulse` releases all tasks waiting at an event semaphore and immediately resets the semaphore.

```
void RTKPulse(RTKSemaphore S);
```

All tasks currently waiting at *S* are made ready. If one or more of these have a higher priority than the calling task, a task switch occurs.

If flag `RF_PULSWIN32` is set in `RTKConfig.Flags` (which is not the case by default), the semaphore is set to 0 if one or more tasks are actually released by this call. If no tasks are released, the semaphore is set to 1. If flag `RF_PULSWIN32` is not set in `RTKConfig.Flags`, `RTKernel-32` always sets the value to 0.

If *S* does not reference an event semaphore, a fatal error is generated by the Debug Version.

Function `RTKWait`

Function `RTKWait` retrieves an event from a semaphore:

```
void RTKWait(RTKSemaphore S);
```

S references the semaphore from which to retrieve the event. If no event is available, the calling task is blocked. It can only be reactivated by a different task calling `RTKSignal` for the respective semaphore.

Any number of tasks can wait for events at a semaphore. The tasks are made ready in sequence of their priorities.

If *S* is an occupied resource or mutex semaphore, the priority of the occupying task is set to the priority of the current task, if this is higher. After the call to `RTKWait`, the active task *occupies* or *owns* the resource or mutex semaphore. It cannot be suspended or terminated until it has released all its resources.

Except for event semaphores, `RTKWait` decrements the semaphore's value on completion of the operation.

Function `RTKWaitCond`

`RTKWaitCond` (wait conditional) retrieves an event from a semaphore under the condition that one is immediately available. `RTKWaitCond` never leads to a blocking task switch.

```
RTKBool RTKWaitCond(RTKSemaphore S);
```

S references the semaphore from which to retrieve the event. If the return value is TRUE, a signal was retrieved, otherwise not.

Except for event semaphores, RTKWaitCond has decremented the semaphore's value if it returns TRUE.

Function RTKWaitTimed

Using function RTKWaitTimed, a timeout for the occurrence of an event can be specified.

```
RTKBool RTKWaitTimed(RTKSemaphore S, RTKDuration Timeout);
```

S is the semaphore to wait at. Timeout is the maximum time (in timer ticks) to wait for the occurrence of an event. If the return value is TRUE, an event was retrieved; otherwise, no event was available and the timeout has expired.

If S is an occupied resource or mutex semaphore, the execution priority of the occupying task is re-evaluated according to the rules of priority inheritance. After successful completion of RTKWaitTimed, the active task occupies the resource or mutex semaphore. It cannot be suspended or terminated until it has released all its resources.

Except for event semaphores, RTKWaitTimed has decremented the semaphore's value if it returns TRUE.

Function RTKResetEvent

RTKResetEvent sets the value of an event semaphore to 0. Subsequent calls to RTKWait or RTKWaitTimed will block:

```
void RTKResetEvent(RTKSemaphore S);
```

To set an event semaphore to 1, use RTKSignal. To merely release all tasks waiting at the event semaphore without setting the semaphore, use RTKPulse.

Mailboxes

The previous section covered semaphores which can be employed to synchronize tasks, thus providing a mechanism allowing orderly inter-task communication using global data.

However, communication via global data is hard to keep track of and error-prone, since a task might "forget" the required semaphore operation before accessing the data. Moreover, no protocol has yet been introduced for a controlled exchange of data.

Mailboxes serve to close this gap. A mailbox is a data buffer that can store a fixed number of *messages*. As implemented by RTKernel-32, messages can have any size and the size of mailboxes can be freely configured.

Tasks can store messages in a mailbox. If the mailbox is full, the task is blocked until space becomes available. Of course, tasks can also retrieve messages from mailboxes. In this case, the task is blocked if no message is available in the mailbox. Any number of tasks can use the same mailbox for storing and retrieving messages.

Messages can be appended to a mailbox's message queue using functions RTKPut, RTKPutCond, and RTKPutTimed (see below). Moreover, messages can be inserted at the start of a message queue using functions RTKPutFront, RTKPutFrontCond, and RTKPutFrontTimed.

If messages are stored/retrieved in a mailbox using pairs of RTKPut.../RTKGet... function calls, the mailbox will behave as a *FIFO* buffer (first in, first out), i.e., messages are retrieved from a mailbox in the same sequence as they have been stored. However, if pairs of RTKPutFront.../RTKGet... function calls are used, the mailbox will behave as a *LIFO* buffer (last in, first out). For maximum flexibility, the RTKPut... and RTKPutFront... functions may be mixed freely, even for the same mailbox.

The following sections discuss how to use mailboxes with RTKernel-32. As an example, we can extend the little semaphore demo program for mailboxes. In addition to a signal, data can now be sent to another task:

```

#include <stdio.h>
#include <rtk32.h>

RTKMailbox Box;

void RTKAPI TaskA(void * P)
{
    int i;

    printf("Task A: waiting at mailbox\n");
    RTKGet(Box, &i);
    printf("Task A: have received number %i\n", i);
}

void main(void)
{
    int i;

    printf("\n");
    RTKKernelInit(3);
    Box = RTKCreateMailbox(sizeof(int), 1, "Test Box");
    printf("Main : creating task A\n");
    RTKCreateThread(TaskA, 4, 0, 0, NULL, "Task A");
    printf("Main : please enter a number: ");
    fflush(stdin);
    scanf("%i", &i);
    RTKPut(Box, &i);
    printf("Main : done.\n");
}

```

Function RTKCreateMailbox

Function RTKCreateMailbox creates and initializes a mailbox:

```

RTKMailbox RTKCreateMailbox(unsigned    DataLen,
                           unsigned    Slots,
                           const char * Name);

```

Parameter DataLen is the length of the mailboxes' messages in bytes. Slots is the maximum number of messages the mailbox can store. Parameter Name points to the name of the mailbox. The name can be displayed by RTKTaskInfo, RTKMailboxInfo, and error messages.

RTKCreateMailbox allocates and initializes the mailbox. The return value is a reference to the new mailbox.

Function RTKDeleteMailbox

Function RTKDeleteMailbox releases the storage used by a mailbox:

```

void RTKDeleteMailbox(RTKMailbox * Box);

```

RTKernel-32 checks whether a task is waiting at the mailbox. In this case, the program is aborted with an error message. The application must make sure that deleted mailboxes are not used any more. RTKernel-32 assigns the value RTK_NO_MAILBOX to *Box.

Function RTKClearMailbox

Function RTKClearMailbox clears the contents of a mailbox:

```

void RTKClearMailbox(RTKMailbox Box);

```

All data stored in the mailbox is discarded. If the mailbox was full and a task was waiting to write to the mailbox, the task is made ready by RTKClearMailbox.

Function RTKMessages

Function RTKMessages returns the number of messages currently stored in a mailbox.

```

unsigned RTKMessages(RTKMailbox Box);

```

Box specifies the mailbox to enquire. The return value will always lie between 0 and the value of parameter Slots passed to RTKCreateMailbox when the mailbox was created.

Function RTKPut

RTKPut stores a message in a mailbox.

```
void RTKPut(RTKMailbox Box, void * Data);
```

Parameter Box is the mailbox to store the message in. *Data is stored in the mailbox.

If the mailbox is full, the calling task is blocked until another task (or an interrupt handler) retrieves a message. If, however, the mailbox is empty and another task is waiting for a message at the mailbox, the waiting task is made ready. If it has a higher priority, it is activated immediately.

Function RTKPutFront

RTKPutFront corresponds to RTKPut, but inserts the message at the start of the mailbox queue rather than at the end:

```
void RTKPutFront(RTKMailbox Box, void * Data);
```

Function RTKGet

RTKGet retrieves a message from a mailbox.

```
void RTKGet(RTKMailbox Box, void * Data);
```

Parameter Box is the mailbox from which to retrieve the message. Data points to the variable to store the message in.

If the mailbox is empty, the calling task is blocked until another task (or an interrupt handler) stores a message. However, if the mailbox is full and another task is waiting to store a message in the mailbox, the waiting task is made ready. If it has a higher priority, it is activated immediately.

Function RTKPutCond

RTKPutCond stores a message in a mailbox under the condition that space is available in the mailbox. RTKPutCond never leads to a blocking task switch. If the mailbox is full, this is indicated by the return value and no data is transferred.

```
RTKBool RTKPutCond(RTKMailbox Box, void * Data);
```

Parameter Box is the mailbox to store the message in. *Data is stored in the mailbox.

If the return value is TRUE, the message has been successfully stored; otherwise, there was no space available in the mailbox.

Function RTKPutFrontCond

RTKPutFrontCond corresponds to RTKPutCond, but inserts the message at the start of the mailbox queue rather than at the end:

```
void RTKPutFrontCond(RTKMailbox Box, void * Data);
```

Function RTKGetCond

RTKGetCond retrieves a message from a mailbox under the condition that one is available immediately. RTKGetCond never leads to a blocking task switch. If the mailbox is empty, this is indicated by the return value and no data is transferred.

```
RTKBool RTKGetCond(RTKMailbox Box, void * Data);
```

Parameter Box is the mailbox from which to retrieve the message. Data points to the variable to store the message in.

If the return value is TRUE, the message has been successfully retrieved; otherwise, there was no message available in the mailbox.

Function RTKPutTimed

RTKPutTimed stores a message in a mailbox if space becomes available within a certain time span.

```
RTKBool RTKPutTimed(RTKMailbox Box,
                    void *      Data,
                    RTKDuration Timeout);
```

Parameter Box is the mailbox in which to store the message. *Data is stored in the mailbox. Timeout is the timeout for waiting until space becomes available in the mailbox (in timer ticks).

If the return value is TRUE, the message has been successfully stored; otherwise, there was no space available in the mailbox and the timeout has expired.

Function RTKPutFrontTimed

RTKPutFrontTimed corresponds to RTKPutTimed, but inserts the message at the start of the mailbox queue rather than at the end:

```
RTKBool RTKPutFrontTimed(RTKMailbox Box,
                        void *      Data,
                        RTKDuration Timeout);
```

Function RTKGetTimed

RTKGetTimed retrieves a message from a mailbox if a message becomes available within a certain time span.

```
RTKBool RTKGetTimed(RTKMailbox Box,
                    void *      Data,
                    RTKDuration Timeout);
```

Parameter Box is the mailbox from which to retrieve the message. Data points to the variable to store the message in. Timeout is the timeout for waiting until a message becomes available in the mailbox (in timer ticks).

If the return value is TRUE, the message has been successfully retrieved; otherwise, there was no message available in the mailbox and the timeout has expired.

Function RTKNextCond

Function RTKNextCond enquires the next message of a mailbox under the condition that at least one message is available. But unlike RTKGetCond, the message is not retrieved. RTKNextCond never leads to a task switch.

```
RTKBool RTKNextCond(RTKMailbox Box, void * Data);
```

Parameter Box is the mailbox from which to enquire the message. Data points to the variable to store the message in.

If the return value is TRUE, at least one message is available in the mailbox and *Data contains a copy of the message; otherwise, there was no message available in the mailbox.

Message Passing

In addition to mailboxes, RTKernel-32 offers message passing as a mechanism for inter-task communication. In message passing, no data objects like semaphores or mailboxes are required for intermediate data storage; data is copied directly between tasks. Message passing may be thought of as a mailbox of size 0.

A fundamental difference to mailboxes is that the sending task explicitly addresses the receiving task. A receiving task, on the other hand, can accept data from any other task. With mailboxes, any task can use any mailbox; a sending task cannot determine who is to receive the data, and a receiving task does not know who sent the data.

Another difference is the absence of a data buffer; both the sending and the receiving task must be ready for the transfer before data can be copied. Thus, if two tasks want to exchange data using message passing, the first task is blocked when it reaches its RTKSend or RTKReceive until the second task in turn reaches its respective RTKReceive or RTKSend. The data transfer then takes place and both tasks can continue. Mailboxes do not provide such a tight coupling, e.g., a task can immediately continue running after a Put operation, even if there was no task ready to receive the data.

Message passing can also be used solely for synchronization. In this case, the receiving task specifies a data length of 0 and the pointers to the data may assume the value NULL, since RTKernel-32 does not perform a data transfer.

The tight coupling between tasks and the lack of data buffering normally disqualifies this mechanism for use by interrupt handlers; however, it can also be used here.

The little mailbox demo program can now be modified for message passing. Instead of storing the data in a mailbox, it is passed directly to another task:

```
#include <stdio.h>
#include "RTK32.h"

void RTKAPI TaskA(void * p)
{
    int i;

    printf("Task A: waiting for data\n");
    RTKReceive(&i, sizeof(int));
    printf("Task A: have received number %i\n", i);
}

void main(void)
{
    int i;
    RTKTaskHandle HandleA;

    printf("\n");
    RTKKernelInit(3);
    printf("Main : creating Task A\n");
    HandleA = RTKCreateThread(TaskA, 4, 0, 0, NULL, "Task A");
    printf("Main : please enter a number: ");
    fflush(stdin);
    scanf("%i", &i);
    RTKSend(HandleA, &i);
    printf("Main : done.\n");
}
```

Please note that, with mailboxes, the mailbox concerned must always be passed as a parameter. In message passing, on the other hand, the sending task specifies the receiver as a parameter. The receiving task does not specify the source of data; it does not know who sent the data.

Function RTKSend

RTKSend sends data to another task:

```
void RTKSend(RTKTaskHandle Receiver, void * Data);
```

Parameter Receiver is the handle of the task to receive the data. Data points to the data to send.

If the receiving task is waiting in an RTKReceive or RTKReceiveTimed, the data transfer takes place immediately and the task with higher priority continues to run. Otherwise, the sending task is blocked until the receiving task is ready to accept the data.

Function RTKReceive

RTKReceive receives data from any other task.

```
void RTKReceive(void * Data, unsigned DataLen);
```

Parameter Data points to the variable to store the received data in. DataLen is the length of the expected data.

If a task sends data using `RTKSend` or `RTKSendTimed` to a receiving task blocked in `RTKReceive`, the data transfer takes place immediately and the task with higher priority continues to run. Otherwise, the receiving task is blocked until another task sends data.

Function `RTKSendCond`

`RTKSendCond` sends data to another task under the condition that the receiving task is immediately ready to accept data; otherwise, the return value indicates failure and no data is transferred. `RTKSendCond` never leads to a blocking task switch.

```
RTKBool RTKSendCond(RTKTaskHandle Receiver, void * Data);
```

Parameter `Receiver` is the handle of the receiving task. `Data` points to the data to send.

If the return value is `TRUE`, the data transfer has been successfully completed; otherwise, the receiving task was not ready to accept data.

Function `RTKReceiveCond`

`RTKReceiveCond` receives data from any task under the condition that a task is immediately ready to send; otherwise, the return value indicates failure and no data is transferred. `RTKReceiveCond` never leads to a blocking task switch.

```
RTKBool RTKReceiveCond(void * Data, unsigned DataLen);
```

Parameter `Data` points to the variable to store the received data in. `DataLen` is the length of the expected data.

If the return value is `TRUE`, the data transfer has been successfully completed; otherwise, no task was ready for immediate data transfer.

Function `RTKSendTimed`

`RTKSendTimed` sends data to another task if the receiving task becomes ready to accept data within the timeout; otherwise, the return value indicates failure and no data is transferred.

```
RTKBool RTKSendTimed(RTKTaskHandle Receiver,
                    void *          Data,
                    RTKDuration    Timeout);
```

Parameter `Receiver` is the handle of the receiving task. `Data` points to the data to send. `Timeout` is the timeout for waiting until the receiving task becomes ready to accept data (in timer ticks).

If the return value is `TRUE`, the data transfer has been successfully completed; otherwise, the receiving task was not ready to accept data and the timeout has expired.

Function `RTKReceiveTimed`

`RTKReceiveTimed` receives data from any other task if a task becomes ready to send data within the timeout; otherwise, the return value indicates failure and no data is transferred.

```
RTKBool RTKReceiveTimed(void *      Data,
                      unsigned      DataLen,
                      RTKDuration    Timeout);
```

Parameter `Data` points to the variable to store the received data in. `DataLen` is the length of the expected data. `Timeout` specifies the timeout for waiting until a task becomes ready to send data (in timer ticks).

If the return value is `TRUE`, the data transfer has been successfully completed; otherwise, no task was ready to send data and the timeout has expired.

Interrupt Handling

In order to simplify the processing of interrupts, `RTKernel-32` provides a number of supporting routines for interrupt handlers.

The word *interrupt* is used for three different types of events: hardware interrupts, software interrupts (also known as *traps*), and error interrupts or *exceptions*. In this manual, the word interrupt always means a hardware interrupt.

Hardware interrupts are identified by their IRQ (Interrupt **Re**Quest number). IRQs must not be confused with interrupt vectors. Each IRQ is assigned a vector; the mapping is performed by the interrupt controller. For example, IRQ 0 is assigned vector 0x08 under MS-DOS, but vector 0x40 under RTTarget-32. RTKernel-32 does not know about interrupt vectors; it deals exclusively with IRQs. The mapping of IRQs to vectors is performed by the interrupt driver used. RTKernel-32 can handle up to 32 different IRQs.

RTKernel-32 installs its own *low-level interrupt handlers* for all IRQs with a bit set in RTKConfig.HookedIRQs at initialization time and any other IRQs used at run-time. When an interrupt is triggered, the low-level handlers (located in RTKernel-32's CPU driver) perform the following actions:

- all registers that might be modified by compiler-generated code are saved on the stack. If required, addressability is established (i.e., segment registers are loaded);
- all further task switches are disabled;
- the stack is set to the stack area reserved for the respective IRQ;
- the corresponding *high-level handler* is called;
- the original stack is restored;
- task switching is reactivated;
- the registers are restored.

This mechanism guarantees that high-level handlers run with their own stack while the scheduler is disabled.

RTKernel-32 switches to its own interrupt stacks to avoid tasks' stack overflows.

In case an interrupt occurs recursively, RTKernel-32 uses a reserved *panic stack*, since an interrupt stack must not be used twice at the same time. If the panic stack is also occupied, the stack is not switched.

This manual's discussions of interrupt handling issues usually refer to the application's high-level handlers. Please refer to Chapter 7, *Interrupt Handling* for details on implementing high-level interrupt handlers.

Function RTKSetIRQHandler

RTKSetIRQHandler is used to install a high-level interrupt handler:

```
void RTKSetIRQHandler(int IRQ; RTKIRQHandler Handler);
```

IRQ specifies the desired interrupt; Handler is a pointer to an interrupt handler, which should be a normal C function without any parameters. Please refer to Chapter 7, *Interrupt Handling* for details on writing interrupt handlers.

Interrupt handlers should not be installed using a method other than RTKSetIRQHandler. If you must support third-party software that cannot be modified, make sure the respective handler is installed **before** RTKernelInit is called and that the respective bit in RTKConfig.HookedIRQs is set. Please note that interrupt handlers installed bypassing RTKernel-32 do not use their own interrupt stack and may be interrupted by preemptive task switches.

Function RTKGetIRQHandler

RTKGetIRQHandler is used to enquire the current high-level handler of an IRQ:

```
RTKIRQHandler RTKGetIRQHandler(int IRQ);
```

IRQ specifies the desired interrupt; the return value is a pointer to the respective handler. If the application has not set a high-level handler for the given IRQ, NULL is returned.

Function RTKSaveIRQHandlerFar

RTKSaveIRQHandlerFar saves the currently installed interrupt handler for a given IRQ:

```
void RTKSaveIRQHandlerFar(int          IRQ,
                          RTKIRQDescriptor * Handler);
```

The RTKIRQDescriptor structure pointed to by parameter Handler contains complete information about the handler. It can be a handler previously installed or a high-level handler of the application. Use this call to inquire a handler you would like to chain to using RTKCallIRQHandlerFar or restore using RTKRestoreIRQHandlerFar.

Function RTKRestoreIRQHandlerFar

RTKRestoreIRQHandlerFar can restore an interrupt handler previously saved using RTKSaveIRQHandlerFar:

```
void RTKRestoreIRQHandlerFar(int          IRQ,
                             const RTKIRQDescriptor * Handler);
```

Example: To insert your own handler into an existing interrupt chain, you should use the following code:

```
#define IRQ ??                // desired IRQ
RTKIRQDescriptor OrgHandler;
void MyHandler(void)         // application handler
{
    ...                      // the handler's processing
    RTKCallIRQHandlerFar(&OrgHandler); // call the old handler
                                   if required

void Install(void)           // install MyHandler
{
    RTKSaveIRQHandlerFar(IRQ, &OrgHandler);
    RTKSetIRQHandler(MyHandler);
}

void CleanUp(void)           // deinstall MyHandler
{
    RTKRestoreIRQHandlerFar(IRQ, &OrgHandler);
}
```

Please note that restoring interrupt handlers is not always required at program exit. RTKernel-32 restores all modified interrupt vectors when the program terminates.

Function RTKCallIRQHandlerFar

Using function RTKCallIRQHandlerFar, an interrupt handler read with RTKSaveIRQHandlerFar can be called:

```
RTKBool RTKCallIntFar(const RTKIRQDescriptor * Handler);
```

Calling such a handler is not always possible (e.g., when the target handler runs at a different privilege level). The return value indicates whether the handler was actually called.

Function RTKSetIRQStack

Using RTKSetIRQStack, the available stack space for an interrupt handler can be enlarged.

```
void RTKSetIRQStack(int IRQ, unsigned StackSize);
```

Parameter IRQ specifies the interrupt to which the new stack should be assigned. StackSize is the size of the new stack in bytes.

RTKernel-32 usually assigns RTKConfig.DefaultIntStackSize (default: 512) for each IRQ. This call can change the default value for a particular IRQ. If the new stack is larger than the current panic stack, the panic stack is also reallocated with StackSize bytes.

Function RTKIRQInfo

RTKIRQInfo writes interrupt statistics to a string variable:

```
void RTKIRQInfo(char * Buffer, unsigned BufferLen);
```

Parameter Buffer points to the string variable to store the returned list in. BufferLen is the length of the string pointed to by Buffer. The Buffer should have at least about 80 + (40 * # of IRQs) bytes.

Example:

IRQ	Calls	FreeStack	Doubles	Time
0	1194	188	0	0.103566
1	137	158	2	0.021934
3	6663	156	0	0.734534
Panic	2	158	0	0.004392

One line of information is issued for each IRQ hooked by RTKernel-32 on which interrupts have occurred since program start. The IRQ number, the number of interrupts that have occurred, unused interrupt stack, and the number of recursive interrupts (normally 0) are returned. The accumulated CPU time consumption of each IRQ is only returned by RTKernel-32's Debug Version if flag RF_ICPUTIME is set in RTConfig.Flags ('-' otherwise). If recursions have occurred (as on the keyboard interrupt in the example), statistics for the panic stack are also given. Recursions on the panic stack (column "Doubles" in line "Panic") mean danger and suggest an error in the interrupt handler (see Chapter 7, *Interrupt Handling*) or an interrupt overload.

Function RTKIRQTopPriority

Using function RTKIRQTopPriority, the interrupt controller's interrupt priorities can be reprogrammed. This call is only supported on systems with two Intel 8059A or compatible interrupt controllers.

```
void RTKIRQTopPriority(int Master, int Slave);
```

Parameter Master specifies the IRQ to have the highest priority for the master interrupt controller. Correspondingly, parameter Slave specifies the IRQ to have the highest priority for the slave interrupt controller.

Normally, IRQ 0 (Timer) has the highest priority, IRQ 1 (keyboard) the second highest, etc. The interrupt controller supports only the cyclic rotation of the priorities. Using RTKIRQTopPriority, you can determine which IRQ should have the highest priority for the respective interrupt controller. The other priorities then follow from the cyclic shift. Please note that the slave interrupt controller is connected to the master via IRQ 2. Thus, all slave IRQs are handled by the master with the priority of IRQ 2. The default corresponds to the call

```
RTKIRQTopPriority(0, 8);
```

Below, the table on the left shows the respective interrupt priorities (low interrupt priorities mean high urgency). The table on the right shows, as an example, the priorities after calling `RTKIRQTopPriority(3, 10)`:

IRQ	Priority
0	0
1	1
2	Slave
3	3
4	4
5	5
6	6
7	7
8	2.0
9	2.1
10	2.2
11	2.3
12	2.4
13	2.5
14	2.6
15	2.7

RTKIRQTopPriority(0, 8)

IRQ	Priority
3	0
4	1
5	2
6	3
7	4
0	5
1	6
2	Slave
10	7.0
11	7.1
12	7.2
13	7.3
14	7.4
15	7.5
8	7.6
9	7.7

RTKIRQTopPriority(3, 10)

Function `RTKEnableIRQ`

This function opens the interrupt controller's interrupt mask.

```
void RTKEnableIRQ(int IRQ);
```

Only after `RTKEnableIRQ` has been called, interrupts of the specified IRQ are passed on to the CPU.

Function `RTKDisableIRQ`

This function closes the interrupt controller's interrupt mask.

```
void RTKDisableIRQ(int IRQ);
```

After calling `RTKDisableIRQ`, interrupts of the specified IRQ are no longer passed on to the CPU. This is the default for unused IRQs.

Function `RTKIRQEnd`

This function disables interrupts and informs the interrupt controller that processing of an interrupt is completed, reenabling lower-priority interrupts:

```
void RTKIRQEnd(int IRQ);
```

Every interrupt handler must call `RTKIRQEnd` before it terminates.

Function `RTKDisableInterrupts`

This function disables interrupts at the CPU level:

```
void RTKDisableInterrupts(void);
```

Using `RTKDisableInterrupts`, a task can make sure that it is not disrupted by interrupts. It should be noted, however, that interrupt response time suffers when interrupts are disabled. Unlike preemptive task switches, cooperative task switches can take place even while interrupts are disabled.

Function RTKEnableInterrupts

This function enables interrupts at the CPU level:

```
void RTKEnableInterrupts(void);
```

Using RTKEnableInterrupts, interrupts can be re-enabled after previously having been disabled using RTKDisableInterrupts or through interrupt processing.

Real-Time Memory Management

In C/C++, memory management is normally performed using malloc, free, realloc, new, etc. The run-time system's heap offers great flexibility and efficiency, but it cannot fulfil real-time requirements. The run-time requirements are non-deterministic and all heap operations are non-reentrant. The latter problem is easily solved using RTKernel-32's Automatic Library Protection feature or multithreading run-time libraries; however, this necessitates blocking task switches in the heap manager, which makes the heap unusable for interrupt handlers.

RTKernel-32 offers memory management with real-time capabilities through *Memory Pools*. A Memory Pool is an isolated heap with data buffers of equal size. Any number of memory pools can exist simultaneously. A pool is initialized once and allocated a certain number of buffers. Thereafter, buffers can be allocated and deallocated from the pool under real-time conditions using the functions RTKGetBuffer and RTKPutBuffer. Both functions are completely reentrant (even for one and the same pool), can be used in both tasks and interrupt handlers and have very small execution times (about 2 microseconds on a 486/33).

Function RTKAllocMemPool

Function RTKAllocMemPool initializes a Memory Pool:

```
void RTKAllocMemPool(RTKMemoryPool * Pool, unsigned BlockSize, unsigned Blocks);
```

Parameter Pool points to the pool handle of the pool to initialize. The pool is allocated with Blocks data buffers of size BlockSize. The buffers are allocated using RTKAlloc, which in turn uses RTKernel-32's memory driver. Therefore, RTKAllocMemPool will generally **not** fulfil real-time requirements. Its time behavior is non-deterministic and it must not be called in interrupt handlers.

Function RTKGetBuffer

Function RTKGetBuffer retrieves a buffer from a Memory Pool:

```
void * RTKGetBuffer(RTKMemoryPool * Pool);
```

Parameter Pool points to the pool handle of the pool from which to retrieve the buffer. A pointer to the buffer is returned. If this was the last available buffer in the pool, *Pool is assigned the value RTK_EMPTY_POOL. If the Pool is empty, the return value is NULL.

Function RTKFreeBuffer

Function RTKFreeBuffer passes a buffer to a Memory Pool:

```
void * RTKFreeBuffer(RTKMemoryPool * Pool, void * Buffer);
```

Parameter Pool points to the pool handle of the pool to which to pass the buffer. In general, the buffer will have been retrieved previously from the pool; however, it can also have been allocated by a different method (e.g., malloc, RTKAlloc, or allocated as a static variable).

It must be ensured that the same buffer is not passed to a pool twice by RTKFreeBuffer, because this would destroy the Memory Pool. This condition is not recognized by RTKFreeBuffer and the program will crash in most cases.

The Kernel Tracer

The Kernel Tracer can be used to analyze the exact sequence of events within the kernel. About 32 different event types are written to a ring buffer by the kernel. Another 10 event types are reserved for the application; thus, you can also use the Tracer to analyze your application code.

A trace event consists of an event identification (type `RTKTraceEvent`, an enumeration type declared in header file `RTTRACE.H`), and, in many cases, some supplementary information. There is, for example, the trace event `tStateCurrent`. It occurs whenever a task enters the state `Current` (i.e., a task switch takes place). As supplementary information for this event type, the task handle of the task being activated is recorded in the trace buffer. The supplementary information of a trace event can be a task, semaphore, mailbox, an IRQ, a pointer, or a number.

Please refer to file `RTTRACE.H` for details on the trace events supported and the exact data structures used by the tracer. The variable `RTKTraceBuffer` contains a pointer to the trace buffer and can be inspected using a debugger.

The trace buffer is supported by `RTKernel-32`'s Debug and Standard Versions. However, the tracer will record kernel events in the trace buffer only in the Debug Version; in the Standard Version, the tracer only records user events. The default trace buffer size is 64 entries in the Debug Version and 0 entries in the Standard Version. The functions for handling the trace buffer are detailed below.

Function `RTKSetTraceBufferSize`

Function `RTKSetTraceBufferSize` sets the size of the trace buffer.

```
void RTKSetTraceBufferSize(unsigned Size);
```

`RTKSetTraceBufferSize` sets the number of events the trace buffer can store. `RTKernelInit` calls `RTKSetTraceBufferSize(64)` in the Debug Version, `RTKSetTraceBufferSize(0)` in the Standard Version. In complex applications, a larger buffer may be required. Each entry in the trace buffer uses 8 bytes of memory.

Function `RTKEnableTrace`

Function `RTKEnableTrace` enables tracing a specific event type.

```
void RTKEnableTrace(RTKTraceEvent E);
```

Parameter `E` is the event type to trace. The default is that all events are traced.

Function `RTKTraceAll`

Function `RTKTraceAll` enables tracing all event types.

```
void RTKTraceAll(void);
```

This is the Kernel Tracer's default in the Debug Version.

Function `RTKDisableTrace`

Function `RTKDisableTrace` disables tracing a specific event type.

```
void RTKDisableTrace(RTKTraceEvent E);
```

Parameter `E` is the event type for which to disable tracing. Subsequently, this event type will not be traced any more.

Function `RTKStopTracing`

Function `RTKStopTracing` disables the Kernel Tracer.

```
void RTKStopTracing(void);
```

This is the Kernel Tracer's default in the Standard Version.

Function `RTKClearTraceBuffer`

Function `RTKClearTraceBuffer` sets all entries in the trace buffer to type `tNoEvent`.

```
void RTKClearTraceBuffer(void);
```

Function RTKUserTrace

In addition to kernel events, user events can be recorded in the trace buffer.

```
void RTKUserTrace(RTKTraceEvent E, int Parameter);
```

RTKUserTrace writes a user event to the trace buffer. Events tUserEvent_1 to tUserEvent_10 are available. Parameter Parameter can be any number, which is displayed by RTKDumpTrace. If a text other than, for example, "User Event 3" is desired for the output from RTKDumpTrace, it may be modified in array RTKTraceName. Example:

```
#include "RTK32.H"
#define MyEvent tUserEvent_3
void main(void)
{
    RTKTraceName[MyEvent] = "My event";
    ...
    RTKUserTrace(MyEvent, 1234);
    ...
}
```

Please refer to file RTTRACE.H for the declarations of RTKTraceEvent, RTKTraceName, etc.

Function RTKTraceHeader

Function RTKTraceHeader copies a heading for the trace dump to a string:

```
char * RTKTraceHeader(char * Buffer);
```

Parameter Buffer should point to a string of at least 100 characters length. The return value points behind the end of the heading. It may be passed to RTKDumpTrace to dump the trace buffer.

Function RTKDumpTrace

RTKDumpTrace formats a trace entry as a text line and copies it to a string:

```
char * RTKDumpTrace(char * Buffer, int Entry);
```

Parameter Buffer should point to a string of at least 80 characters length. The return value points behind the end of the string. Thus, it can be passed to RTKDumpTrace again to dump additional trace entries. Parameter Entry specifies which entry to dump. "0" is the latest entry, "1" is the previous entry, etc. The Tracer should be disabled by a call to RTKStopTracing prior to dumping the buffer.

Although the trace buffer only contains two pieces of information per entry (the event and a parameter), RTKDumpTrace displays three columns next to the event index. The first column contains the current task. DumpTrace generates this column by searching for 'tStateCurrent' entries in the buffer and "memorizing" which task is the current one. Therefore, this column is only filled starting with the first 'tState-Current' event in the trace buffer.

To dump the last 10 trace buffer entries, the following code can be used:

```
char Buffer[2048];
char * B = Buffer;
int i;
B = RTKTraceHeader(B);
for (i=9; i >=0; i--)
    B = RTKDumpTrace(B, i);
printf(B);
```

Care must be taken to dimension the Buffer large enough to store the generated data. It should be at least $100 + N * 80$ bytes in size, where N is the number of trace entries required. Alternatively, each text line generated can be written to a file to free the buffer. Please refer to demo program RTDemo for examples of this technique (command 'TRACE').

The following output was generated by program RTDemo while data was being received on COM2. It reflects the reception of two characters (with the second one still in the mailbox), a keypress, and a user event.

Index	Current Task	Event	Object
183	CPU Monitor	Interrupt Start	IRQ: 3
184	CPU Monitor	Mailbox In	Mailbox: Receive COM2
185	CPU Monitor	State: Ready	Task: COM Receiver
186	CPU Monitor	Interrupt End	IRQ: 3
187	CPU Monitor	State: Ready	Task: CPU Monitor
188	CPU Monitor	State: Current	Task: COM Receiver
189	COM Receiver	Mailbox Out	Mailbox: Receive COM2
190	COM Receiver	State: TimedGet	Mailbox: Receive COM2
191	COM Receiver	State: Current	Task: CPU Monitor
192	CPU Monitor	Interrupt Start	IRQ: 1
193	CPU Monitor	Semaphore Inc	Semaphore: Keyboard
194	CPU Monitor	State: Ready	Task: Main Task
195	CPU Monitor	Interrupt End	IRQ: 1
196	CPU Monitor	State: Ready	Task: CPU Monitor
197	CPU Monitor	State: Current	Task: Main Task
198	Main Task	Semaphore Dec	Semaphore: Keyboard
199	Main Task	Interrupt Start	IRQ: 3
200	Main Task	Mailbox In	Mailbox: Receive COM2
201	Main Task	State: Ready	Task: COM Receiver
202	Main Task	Interrupt End	IRQ: 3
203	Main Task	User Event 3	Number: 1234

Miscellaneous RTKernel-32 Operations

Miscellaneous RTKernel-32 functions that have not yet been introduced are discussed in this section.

Function RTKDebugVersion

If an application wants to enquire whether it is using the RTKernel-32 Debug Version, this function can be used. In the Debug Version, it will return TRUE, otherwise FALSE.

```
RTKBool RTKDebugVersion(void);
```

During the program development phase, the Debug Version of RTKernel-32 should be used by all means. Please refer to Chapter 7, *RTKernel-32's Debug Version* for details about the Debug Version.

Function RTKStackCheck

RTKernel-32 offers its own stack-check mechanism. Unlike compiler-generated stack checks, function RTKStackCheck can also be used in interrupt handlers.

```
void RTKStackCheck(void);
```

The Debug Version of RTKernel-32 will call RTKStackCheck in every RTKernel-32 operation if flag RF_STACKCHECKS is set in RTKConfig.Flags. The program is aborted with an error message if less than 64 bytes are left on the stack. These 64 bytes are required by RTKernel-32 to issue the error message and abort the program. Error-free termination of the program cannot be expected due to the lack of stack space.

It should be noted that RTKernel-32 can only recognize stack overflows that occur in a kernel call. Stack overflows in the application code or the Standard Version may go unnoticed.

Function RTKCanPreempt

Using this function, the program can enquire whether the current environment supports preemptive multitasking:

```
RTKBool RTKCanPreempt(void);
```

If this function returns FALSE, a call to RTKPreemptionsON will produce a fatal error.

Function RTKPreemptionsON

RTKernel-32 supports both cooperative and preemptive multitasking. Function RTKPreemptionsON enables preemptive scheduling:

```
void RTKPreemptionsON(void);
```

After initialization of RTKernel-32, preemptions are enabled automatically only if RF_PREEMPTIVE is set in RTKConfig.Flags. When preemptions are enabled, RTKernel-32's Automatic Library Protection is also enabled (if installed).

Please refer to Chapter 1, *Cooperative and Preemptive Multitasking* and Chapter 7, *Preemptive or Cooperative Multitasking?* for a detailed discussion of preemptive and cooperative scheduling.

RTKPreemptionsON is only supported if the interrupt driver supports task switches inside interrupt handlers. Use function RTKCanPreempt to check whether preemptions are supported.

Function RTKPreemptionsOFF

This function disables preemptive scheduling:

```
void RTKPreemptionsOFF(void);
```

RTKernel-32's Automatic Library Protection is not deactivated.

Function RTKScheduler

To make task switches possible during cooperative scheduling, all tasks must regularly perform kernel calls. Function RTKScheduler has the sole purpose of performing task switches that may have become necessary:

```
void RTKScheduler(void);
```

RTKScheduler checks whether a task with a higher priority than the currently active task has become Ready; if so, a task switch to this task is performed.

Optionally, RTKDelay(0) can be called. This, however, will also trigger task switches to tasks of the same priority (see Chapter 2, *Function RTKDelay* and Chapter 7, *Avoid Time Slicing*).

RTKScheduler is usually not required in preemptive scheduling mode, but it does no harm. RTKScheduler is very fast and it ensures that a piece of code runs smoothly with cooperative and preemptive scheduling.

Function RTKSetMessageHandler

Function RTKSetMessageHandler allows the installation of an alternate handler to display fatal error messages:

```
typedef void (RTKAPI * RTKMessageHandler)(const char * Message);  
void RTKSetMessageHandler(RTKMessageHandler Handle);
```

By default, RTKernel-32 uses a function supplied by the system driver to display fatal error messages. Function RTKSetMessageHandler can be used to install a different handler. For example, this may become necessary when the program has switched to graphics mode, which the system driver cannot handle.

Function RTKSetTaskSwitchHook

RTKernel-32 can call a user-defined function in every task switch. Such a *task switch hook* is installed using function RTKSetTaskSwitchHook:

```
typedef void (RTKAPI * RTKTaskSwitchHook)(RTKTaskHandle OldTask,  
                                           RTKTaskHandle NewTask);  
  
void RTKSetTaskSwitchHook(RTKTaskSwitchHook Hook,  
                          RTKTaskSwitchHook * OldHook);
```

Parameter Hook is a function with two task handles as parameters, respectively referencing the task being suspended and the task being activated. Parameter OldHook should point to a variable to receive a pointer to the previously installed hook.

For an application task switch hook, some severe restrictions apply. Specifically, it must be considered that:

- Interrupts are disabled when the hook is called. Interrupts must never be enabled inside the hook.
- The stack context is undefined. There might be very little stack space available.
- Only the following RTKernel-32 functions may be used:

```
RTKGetUserData
RTKGetTaskState
RTKGetTaskPrio
RTKGetTime
RTKGetBuffer
RTKFreeBuffer
RTKUserTrace
```

- Before returning, the hook must call the previously installed task switch hook.

If any of these restrictions is violated, the program will very probably crash. Possible error messages may be misleading.

RTKGetTaskState(OldTask) returns the state of the task after the task switch. RTKGetTaskState(NewTask) is undefined (should be TS_CURRENT, of course). If Win32 emulation is used, all TLS-related functions will access OldTask's data.

Using a task switch hook is discouraged. In particular, it should not be attempted to solve reentrance problems by using a hook; resource semaphores are much better suited for this purpose. It should also be noted that a task switch hook can dramatically degrade RTKernel-32's performance.

The following example shows how the number of task switches can be counted for each task using a task switch hook:

```
#include <stdlib.h>
#include <stdio.h>
#include <rtk32.h>

int UserDataIndex;
RTKTaskSwitchHook OldHook;

void RTKAPI MyThread(void * p)
{
    while (1)
        RTKDelay((RTKDuration)p);
}

void RTKAPI TheHook(RTKTaskHandle Old, RTKTaskHandle New)
{
    int Count;

    Count = (int) RTKGetUserData(New, UserDataIndex);
    Count++;
    RTKSetUserData(New, UserDataIndex, (void*) Count);
    OldHook(Old, New);
}

int main(void)
{
    RTKTaskHandle H1, H2;

    RTKKernelInit(3);
    UserDataIndex = RTKAllocUserData();
    RTKSetTaskSwitchHook(TheHook, &OldHook);
    H1 = RTKCreateThread(MyThread, 2, 1024, 0, (void*)1, "ThreadA");
    H2 = RTKCreateThread(MyThread, 2, 1024, 0, (void*)3, "ThreadB");
    RTKDelay(20);
    printf("Task switches for ThreadA: %i, ThreadB: %i\n",
        RTKGetUserData(H1, UserDataIndex),
```

```
        RTKGetUserData(H2, UserDataIndex));  
    return 0;  
}
```

Function RTKSetTaskStartStopHook

Function RTKSetTaskStartStopHook allows the installation of a function to be called on every task creation and termination:

```
typedef void (RTKAPI * RTKTaskStartStopHook)(RTKTaskHandle Task, int Reason);  
void RTKSetTaskStartStopHook(RTKTaskStartStopHook Hook,  
                             RTKTaskStartStopHook * OldHook);
```

Parameter Hook is a function with the handle of the respective task and an integer as parameters. Parameter OldHook should point to a variable to receive a pointer to the previously installed hook.

The Reason parameter to the hook function indicates whether the hook was called due to a task creation (Reason == 0) or termination (Reason == 1). The following rules should be observed in task start/stop hooks:

- At task creation, the hook is called after the system driver's hook.
- At task creation, the hook is guaranteed to be called in the context of the task being created. The hook is executed immediately before the task function code.
- At task termination, the hook is called before the system driver's hook.
- At task termination, the hook is **not** guaranteed to be called in the context of the terminating task; it could be called by another task which has called RTKTerminateTask.
- The hook is not called for the creation of the main task. However, the hook can be called for termination if the main task is terminated.
- The hook is not called at program exit.
- Before returning, the hook must call the previously installed hook.

Function RTKFatalError

Application programs can make use of RTKernel-32's error handling mechanism using this function:

```
void RTKFatalError(const char * Message);
```

RTKFatalError will display the program location where RTKFatalError was called. If a source code position driver is used and a symbol table has been loaded, the position is displayed as source file name and line number. Otherwise, it is given in hex. The currently executing task or interrupt name and the message parameter are displayed. The currently active message handler is used for output. Finally, the fatal error exit handler of the system driver is called.

Function RTKAlloc

RTKAlloc is a high-level interface to RTKernel-32's memory allocation driver:

```
void * RTKAlloc(unsigned size, const char * Name);
```

Parameter size specifies the size in bytes of the memory block to allocate. Name is a pointer to a string with a descriptive name of the object to allocate. If the function succeeds, a pointer to the allocated block is returned. Otherwise, the program is aborted and the name of the object to be allocated is displayed using RTKernel-32's error handling mechanism (see RTKFatalError).

Function RTKDeallocTerminatedTasks

A task that terminates itself (e.g., by reaching the end of its task function) cannot deallocate its own data. Therefore, many unused memory blocks can accumulate if many tasks terminate themselves. To release these memory blocks, function RTKDeallocTerminatedTasks is provided:

```
void RTKDeallocTerminatedTasks(void);
```

Normally, this function need not be called. RTKCreateThread will always deallocate all terminated tasks before a new task is created.

Tasks that are terminated but have not yet been deallocated are in the state `TS_TERMINATED` and will be displayed by `RTKTaskInfo`.

Functions `RTIn`, `RTInW`, `RTInD`, `RTOut`, `RTOutW`, `RTOutD`

Some 32-bit compilers do not supply functions for port I/O. Therefore, `RTKernel-32` offers functions for this purpose:

```
BYTE  RTIn  (unsigned int addr);
WORD  RTInW (unsigned int addr);
DWORD RTInD (unsigned int addr);
void  RTOut (unsigned int addr, BYTE  val);
void  RTOutW(unsigned int addr, WORD  val);
void  RTOutD(unsigned int addr, DWORD val);
```

Chapter 3

Alternate APIs for RTKernel-32

RTKernel-32's native Application Program Interface (API) was described in Chapter 2. For compatibility with other multitasking systems, RTKernel-32 provides APIs compatible with RTKernel-C for DOS and 16-bit embedded systems and the Win32 thread API. You can use these alternate APIs if you already have application software based on these APIs, or if you want to develop software to run under several different systems.

RTKernel-C 4.5 for DOS Compatible API

RTKernel-C is a real-time multitasking system for DOS and 16-bit embedded systems, also available from On Time. Almost all of its features are also supported by RTKernel-32. Exceptions are DOS-specific features (e.g., automatic DOS protection, TSR programs, interrupt tasks, etc.).

Programs using the RTKernel-C API must include header file `RTKERNEL.H` instead of `RTK32.H`. `RTKERNEL.H` in turn includes `RTK32.H` and `RTKDOS.H`. `RTKDOS.H` contains preprocessor macros that map all differing names of constants, types, and functions to the corresponding names of RTKernel-32. There is no thunking layer required for RTKernel-C emulation. Thus, there is no performance or code size penalty for using the RTKernel-C API.

RTKernel-32 contains many new features not available in RTKernel-C. These new features can be accessed by directly using RTKernel-32's native API. RTKernel-C emulation does not restrict the set of available features in any way. When `RTKERNEL.H` is included, both APIs (RTKernel-C and RTKernel-32) are available and can be mixed within the same program.

The following RTKernel-C API functions are not available under RTKernel-32:

- `RTKDOSProtectionON` / `RTKDOSProtectionOFF`
- `RTKExec`
- `RTKDiskIntsON` / `RTKDiskIntsOFF` / `RTKSetDiskTimeout`
- `RTKCPU`
- `RTKUses8087`
- `RTKSwap32BitRegs`
- Interrupt 2Fh Interface

The following functions of RTKernel-C are available under RTKernel-32, but are not covered by `RTKDOS.H`. If your program uses any of these, minor source code modifications are required:

- `RTKGetIRQHandlerFar` / `RTKSetIRQHandlerFar` / `RTKCallIntFar`
- `RTKSetTaskSwitchHook`

The following supplemental modules supplied with RTKernel-C are not available for RTKernel-32:

- `RTTimer` / `RTClock` / `PCTimer` (see timer device drivers and modules `FineTime/Clock` instead)
- `KillKey`
- `RTVision`
- `Spooler`
- `DOSMem`
- `RTIPX`
- `IPC`
- `SmartDrv`
- `RTKDPMI`

By using conditional compilation, the same source file can be used with both RTKernel-C and RTKernel-32. Under RTKernel-32, the symbol `RTK32_VER` is defined.

Win32 Thread Compatible API

In addition to its native and RTKernel-C's API, RTKernel-32 also offers a Win32-compatible interface. This allows porting programs originally designed for Windows NT to run under RTKernel-32. The Win32 thread API is made available by including `windows.h` in the source code.

The documentation in this chapter does not completely describe all Win32 functions emulated by RTKernel-32. Rather, only differences from the original Win32 functions are detailed. Parameters or features of a function not mentioned here have the same functionality as under Win32. For a complete description, please refer to Microsoft's Win32 API documentation, which is included with all C/C++ compilers supported by RTKernel-32.

For all functions expecting a character string, only the ASCII versions are supplied. RTKernel-32 does not support Win32 emulation for Unicode programs. Any security attribute parameters are ignored by RTKernel-32; application programs may specify `NULL` (this is also supported by Win32).

Please note that Win32 emulation is only available if RTKernel-32 is used with RTTarget-32, On Time's 32-bit cross development system.

Win32 Priorities

Under Win32, all thread priorities are relative to the process priority. Under RTKernel-32, there is no process priority; instead, the value `RTKConfig.MainPriority` is used. Therefore, applications using the Win32 API should make sure that the main task uses this priority. The simplest method to guarantee this is to use RTKernel-32's auto initialization feature, which is enabled by default (see Chapter 2, *RTKernel-32 Configuration* for details). If you prefer to explicitly call `RTKernellnit`, you should supply `RTKConfig.MainPriority` as the parameter.

Win32 Handles

Handles are an important concept of the Win32 API. RTKernel-32 uses RTTarget-32's handle manager to implement handles for threads, semaphores, events, and mutex objects. The number of available Win32 handles is limited but can be changed (please refer to Part I or this manual for details). It is very important that all handles of objects with limited lifetime are closed to avoid errors caused by an out-of-handle situation. RTKernel-32 does not define functions `CloseHandle` and `DuplicateHandle`; however, RTTarget-32's corresponding functions can be used, even for handles created with RTKernel-32's Win32 emulation.

Win32 handles (type `HANDLE`) must not be confused with RTKernel-32's task handles (type `RTKTaskHandle`). Win32 handles indirectly reference objects, while RTKernel-32 handles reference them directly. Consequently, Win32 handles can be duplicated with function `DuplicateHandle`; the associated object is destroyed only after **all** handles have been closed. In contrast, RTKernel-32 task handles (like mailboxes or semaphores) cannot be duplicated; they can only be copied, which does not create a new handle. Any call to `RTKTerminateTask` (or `RTKDeleteSemaphore` / `RTKDeleteMailbox`) will immediately destroy the associated object. In particular, this is also true for multiple references to the same semaphore obtained from `RTKOpenSemaphore`.

Win32 also knows Thread IDs. These are unique (there is exactly one Thread ID for each thread) and cannot be duplicated. Under RTKernel-32, Win32 Thread IDs are identical to RTKernel-32 task handles.

Win32 and RTKernel-32 Error Handling

RTKernel-32's Win32 emulation is implemented by a set of Win32-compatible functions which reformat the given parameters and then call equivalent RTKernel-32 functions. One area difficult to translate is the different error-handling philosophy of the two systems. When a Win32 function fails, a return value indicating error is returned and the application must check for any errors by calls to `GetLastError()` after each single Win32 call. This method can severely increase code size and reduce readability. RTKernel-32, on the other hand, will not pass fatal errors to the application, but rather abort the application. This method is adequate since program bugs cannot be corrected dynamically at run time anyway.

Rather, it is desirable to get a clear error message with all relevant information (address of call, names of tasks, semaphores, or mailboxes involved, etc.). Software debugging is significantly simplified in this way.

RTKernel-32's Win32 emulation makes every effort to check all parameters at run-time to emulate Win32's error handling. `SetLastError()` is called whenever possible. However, there may be cases where not all application bugs are caught (e.g., when the application has corrupted RTKernel-32's internal data structures). In these rare instances, an RTKernel-32 error message could be issued when Win32 would indicate the error with a return code instead.

Mixing RTKernel-32 and Win32 APIs

The simultaneous use of both RTKernel-32's native and Win32 API is possible and quite typical, since the Win32 thread API lacks many features required by real-time systems (e.g., interrupt processing, real-time memory management, etc.). However, care must be taken not to confuse Win32 thread handles and RTKernel-32 task handles.

Every task running under RTKernel-32 has exactly one RTKernel-32 task handle. However, only those tasks started by Win32's `CreateThread` function have one or more Win32 handles. If you want to use the compiler's multithread run-time libraries, all threads using the run-time systems should be started using the run-time systems' function for this purpose (e.g., `_beginthread`).

Function `RTKWin32ToRTKHandle`

RTKernel-32 supplies this function to translate a Win32 handle to an RTKernel-32 task handle:

```
RTKTaskHandle RTKWin32ToRTKHandle(HANDLE H);
```

If `H` is a valid Win32 handle referencing an existing task, the function will return the desired RTKernel-32 task handle. Otherwise, the returned value is `RTK_NO_TASK`.

Function `RTKToWin32Handle`

RTKernel-32 supplies this function to translate an RTKernel-32 task handle to a Win32 handle:

```
HANDLE RTKToWin32Handle(RTKTaskHandle H);
```

If `H` references an existing task which was created by Win32's `CreateThread` function and the handle returned by `CreateThread` has not been closed, this function returns the original Win32 handle of the referenced task. Otherwise, the returned value is `INVALID_HANDLE_VALUE`.

Function `GetCurrentThreadId`

This function simply returns the RTKernel-32 task handle of the current task. The function always succeeds, even if the current task was not created using Win32 functions.

Function `CreateThread`

`CreateThread` allocates a thread object and associates a Win32 handle with it. The thread is created by calling `RTKCreateThread`. The name of the task is set to "Win32 Thread". Floating point context maintenance for the new thread is controlled by `RTKConfig.Flags`. The priority is set to `RTKConfig.MainPriority`. If this is 0, `RTKConfig.DefaultPriority` is used. If this value is also 0, the new thread is created with the priority of the creating thread. `*ThreadId` receives the RTKernel-32 task handle. In addition, an event semaphore is associated with the thread object to allow waiting for task termination.

If parameter `dwStackSize` is 0, `RTKConfig.DefaultTaskStackSize` is used.

The *thread object* created by this function exists until the last handle to the thread is closed using `CloseHandle`. This can happen before or after the thread terminates. In particular, if you plan to wait for thread completion with `WaitForSingleObject`, you must not close the handle before the call to `WaitForSingleObject` returns.

Usually you should not call `CreateThread` directly. Instead, run-time system routines provided for thread creation (like `_beginthread`) should be used.

All high-level thread creation functions of the C/C++ run-time systems call `CreateThread`. Please note that some of these will automatically close the returned Win32 handle, while others do not. If you need a handle with a longer life span, use `DuplicateHandle` to get a second handle. Please also note that some run-time system functions do not return the Win32 handle, but the thread ID.

Function `ExitThread`

This function terminates the current task, even if it was not created using `CreateThread`. If a Win32 handle is associated with the thread, it is **not** closed. If a task is waiting for the termination of the current task, it is released by this call.

Threads created with C/C++ run-time systems functions such as `_beginthread` should not use this function; `_endthread` should be used instead.

Function `TerminateThread`

If the given handle references an existing thread, the thread is terminated. Different from running under Win32, the thread continues to run until all resource and mutex semaphores have been released. The stack of the task is deallocated. Terminating the last thread of a program does not terminate the program (RTKernel-32's Idle Task would continue to run). The Win32 handle is **not** closed (this behavior is identical to Win32's). If one or more tasks are waiting for the termination of the task, they are released by this call.

Threads created with C/C++ run-time system functions such as `_beginthread` should not use this function; `_endthread` should be used instead.

Function `GetExitCodeThread`

This function behaves just like under Win32.

Function `GetCurrentThread`

This function returns the Win32 handle created by the call to `CreateThread` to create the current task. If the task was not created by `CreateThread`, `INVALID_HANDLE_VALUE` is returned. Unlike under Win32, the return value is not a pseudo handle but a real handle.

Function `Sleep`

This function uses module `Clock` to translate the parameter `dwMilliseconds` to RTKernel-32 timer ticks and calls `RTKDelay`. The accuracy of this operation is determined by the current timer interrupt frequency.

Function `GetTickCount`

This function calls `RTKGetTime` and translates the result to milliseconds using module `Clock`. The resolution of this operation is determined by the current timer interrupt frequency.

Function `SuspendThread`

This function suspends the specified thread. RTKernel-32 does not maintain a suspend count. Therefore, `SuspendThread` will always return 0 (or `0xFFFFFFFF` in the case of an error).

Function `ResumeThread`

This function resumes the specified thread. RTKernel-32 does not maintain a suspend count. Therefore, `ResumeThread` always resumes the thread and will always return 1 (or `0xFFFFFFFF` in the case of an error).

Function `SetThreadPriority`

`SetThreadPriority` sets the priority of the given thread relative to a *Base*. The *Base* is the value `RTKConfig.MainPriority`, `RTKConfig.DefaultPriority`, or the current task's priority (the first non-zero value is used). The priority is changed as follows:

Parameter nPriority	Priority
THREAD_PRIORITY_IDLE	Base - 3
THREAD_PRIORITY_LOWEST	Base - 2
THREAD_PRIORITY_BELOW_NORMAL	Base - 1
THREAD_PRIORITY_NORMAL	Base
THREAD_PRIORITY_ABOVE_NORMAL	Base + 1
THREAD_PRIORITY_HIGHEST	Base + 2
THREAD_PRIORITY_TIME_CRITICAL	Base + 3

If the resulting priority is below RTK_MIN_PRIO, RTK_MIN_PRIO is used instead. If the resulting priority is above RTK_MAX_PRIO, RTK_MAX_PRIO is used instead.

RTConfig.MainPriority defaults to 5. This makes the range of priorities 2 to 8 available to Win32 threads.

Function GetThreadPriority

GetThreadPriority uses the same Win32 <-> RTKernel-32 priority mapping as SetThreadPriority. The same *Base* as for SetThreadPriority is used. If a priority difference relative to the base greater than 3 is encountered, THREAD_PRIORITY_TIME_CRITICAL or THREAD_PRIORITY_IDLE is returned.

Function InitializeCriticalSection

InitializeCriticalSection creates an RTKernel-32 binary or mutex semaphore (see Chapter 2, section *RTKernel-32 Configuration*) named "Win32 CS" and associates it with the critical section object passed as the parameter.

Function EnterCriticalSection

EnterCriticalSection performs RTKWait on the RTKernel-32 semaphore associated with the given critical section.

Function LeaveCriticalSection

LeaveCriticalSection performs RTKSignal on the RTKernel-32 semaphore associated with the given critical section.

Function DeleteCriticalSection

DeleteCriticalSection performs RTKDeleteSemaphore on the RTKernel-32 semaphore associated with the given critical section.

Function CreateEvent

CreateEvent calls RTKOpenSemaphore to create or find a matching semaphore. If parameter bManualReset is TRUE, the semaphore has type ST_EVENT; otherwise, it has type ST_BINARY. CreateEvent also allocates a Win32 handle for the semaphore, which is returned to the caller.

Function CreateMutex

CreateMutex calls RTKOpenSemaphore to create or find a matching semaphore of type ST_BINARY or ST_MUTEX (see Chapter 2, section *RTKernel-32 Configuration*). CreateMutex also allocates a Win32 handle for the semaphore which is returned to the caller.

Function CreateSemaphore

CreateSemaphore calls RTKOpenSemaphore to create or find a matching semaphore of type ST_COUNTING. CreateSemaphore also allocates a Win32 handle for the semaphore which is returned to the caller.

Parameter lMaximumCount is ignored. RTKernel-32 counting semaphores can count up to $2^{32}-1$.

Function OpenEvent

OpenEvent calls RTKOpenSemaphore to find a matching semaphore of type ST_EVENT or ST_BINARY. If one is found, OpenEvent also allocates an additional Win32 handle for the existing semaphore, which is returned to the caller.

The parameter fdwAccess is ignored (EVENT_ALL_ACCESS is assumed).

Function OpenMutex

OpenMutex calls RTKOpenSemaphore to find a matching semaphore of type ST_BINARY or ST_MUTEX (see Chapter 2, section *RTKernel-32 Configuration*). If one is found, OpenMutex also allocates an additional Win32 handle for the existing semaphore, which is returned to the caller.

The parameter fdwAccess is ignored (MUTEX_ALL_ACCESS is assumed).

Function OpenSemaphore

OpenSemaphore calls RTKOpenSemaphore to find a matching semaphore of type ST_COUNTING. If one is found, OpenSemaphore also allocates an additional Win32 handle for the existing semaphore, which is returned to the caller.

The parameter fdwAccess is ignored (SEMAPHORE_ALL_ACCESS is assumed).

Function SetEvent

SetEvent calls RTKSignal for the specified event semaphore.

Function ResetEvent

ResetEvent functions just like under Win32.

Function PulseEvent

If the given handle references a manual reset event object, PulseEvent calls RTKPulse. The behavior of PulseEvent is completely Win32 compatible if the flag RF_PULSWIN32 is set in RTKConfig.Flags (by default, it is not set). Otherwise, the event semaphore is always left in an unset state by this call (also see Chapter 2, *Function RTKPulse*).

If the hEvent parameter specifies an auto-reset event, RTKSignal is called.

Function ReleaseMutex

This function calls RTKSignal on the given mutex semaphore.

Function ReleaseSemaphore

This function calls RTKSignal cReleaseCount times on the given counting semaphore. The value assigned to *lpPreviousCount is not reliable if other tasks are also using the semaphore at the time of this call.

Function WaitForSingleObject

WaitForSingleObject accepts thread, event, mutex, and semaphore handles. If a thread handle is passed, the wait operation will be performed on the event object created for the thread by CreateThread.

If the dwTimeout parameter is set to INFINITE, RTKWait is called for the respective object. If it is zero, RTKWaitCond is used. For all other values, the value is converted to timer ticks and RTKWaitTimed is called.

Chapter 4

Supplemental Modules

This chapter introduces the supplemental modules delivered with RTKernel-32 in source code. They offer various services that may be helpful in a real-time multitasking environment. These modules are delivered in source code for good reasons. For once, they offer some examples of using RTKernel-32 that you can study to improve your understanding of RTKernel-32. Secondly, it is expected that these modules will not be suited for all purposes in unmodified form.

Module FineTime

Module FineTime is a high-level interface to RTKernel-32's high resolution timer device driver. It allows measuring time intervals and converting high resolution times to real time (seconds, milliseconds, and microseconds). High resolution times are stored as 64-bit unsigned integer values. The resolution of time measurements depends on the timer device driver linked. For example, for the PC timer driver, the resolution is 0.838 microseconds. For the Pentium driver, the resolution depends on the clock rate of the CPU, but is typically much higher.

The application interface to module FineTime is contained and documented in include file FINETIME.H.

Function FTSetResolution

This function defines conversion constants required to convert fine times to seconds and vice versa:

```
void FTSetResolution (unsigned UnitsPerSecond, unsigned Divisor);
```

Times will be converted using the formula

```
FineTime = Seconds * UnitsPerSecond / Divisor;
```

Divisor should be less than 4295 to allow conversions to microseconds.

In most cases, the timer device driver will make a call to this function to supply the appropriate default conversion values. However, some drivers do not know the clock rate of the hardware they rely on (example: the Pentium driver). In these cases, the application must call FTSetResolution after RTKernel-Init to be able to convert fine times to seconds.

The PC Timer driver calls FTSetResolution(14318180, 12) at program startup. The Pentium driver calls FTSetResolution(120000000, 1), assuming a CPU speed of 120MHz. You can use FTCalibrate to calculate appropriate conversion constants based on the clock driver (see function FTCalibrate).

Function FTCalibrate

Function FTCalibrate can calibrate the high resolution timer using the clock driver:

```
unsigned FTCalibrate(RTKDuration Ticks);
```

Parameter Ticks specifies the calibration period in ticks. FTCalibrate calls FTSetResolution internally and returns the timer frequency or 0 if the calibration failed.

Fine Time Arithmetic Functions

Module FineTime contains some functions to simplify arithmetic with 64-bit high resolution times:

```
typedef unsigned int UINT;

UINT FTIntMultDiv(UINT Factor1, UINT Factor2, UINT Divisor);
void FTSubtract (RTKFineTime * Result, RTKFineTime * T1, RTKFineTime * T2);
void FTAdd      (RTKFineTime * Result, RTKFineTime * T1, RTKFineTime * T2);
UINT FTMultiply (RTKFineTime * Result, RTKFineTime * T,  UINT Factor);
UINT FTDivide   (RTKFineTime * T,  UINT Divisor);
```

FTIntMultDiv returns Factor1 * Factor2 / Divisor. The intermediate result of the multiplication is maintained as a 64-bit value.

FTSubtract calculates (*Result) = (*T1) - (*T2).

FTAdd calculates $(*Result) = (*T1) + (*T2)$.

FTMultiply calculates $(*Result) = (*T) * Factor$.

FTDivide calculates $(*T) / Divisor$ (rounded down).

Functions FTIntMultDiv and FTDivide can trigger an exception if the divisor is zero or the result would exceed 32 bits. None of the functions detect overflows; the most significant bits which do not fit into the result are discarded. This behavior corresponds to the C/C++ operators for unsigned integers such as $+$, $-$, $*$, $/$, etc.

Function FTReadTime

This function reads the current value of the high resolution timer:

```
void FTReadTime(RTKFineTime * T);
```

*T is assigned the current high resolution time.

Time Interval Measurements

The following functions calculate the time elapsed since a call to FTReadTime and convert the result to seconds, milliseconds, or microseconds:

```
unsigned FTElapsedSeconds (const RTKFineTime * T);
unsigned FTElapsedMilliSecs(const RTKFineTime * T);
unsigned FTElapsedMicroSecs(const RTKFineTime * T);
```

The parameters (*T) must have been initialized with a call to FTReadTime. These functions return the time elapsed since the respective call to FTReadTime. Any number of independent time intervals can be measured in this way.

The return values of all three functions are 32-bit unsigned integers. The results are always rounded down. Therefore, the range of these functions is limited. For example, FTElapsedMicroseconds is limited to about 72 minutes, FTElapsedMilliSecs to about 50 days, and FTElapsedSeconds to 136 years. When these limits are exceeded, the function results are undefined.

Time Conversions

The following functions can convert 64-bit high resolution times to seconds, milliseconds, microseconds, and vice versa:

```
unsigned FTTimeToMicroSecs (const RTKFineTime * T);
unsigned FTTimeToMilliSecs (const RTKFineTime * T);
unsigned FTTimeToSeconds   (const RTKFineTime * T);

void      FTMicroSecsToTime(RTKFineTime * T, unsigned Micros);
void      FTMilliSecsToTime(RTKFineTime * T, unsigned Millis);
void      FTSecondsToTime  (RTKFineTime * T, unsigned Secs);
```

All return values are rounded down. Again, the return values of the first three functions limit the available range to about 72 minutes, 50 days, and 136 years, respectively.

Module Clock

Clock is the high-level interface to RTKernel-32's timer interrupt device driver. It supplies functions to convert timer ticks (RTKernel-32's type RTKDuration) to real time (seconds, etc.) and vice versa. In addition, it allows modifying the timer interrupt rate (if supported by the underlying driver).

Module clock is primarily concerned with timer ticks. However, it also knows *timer units*. Timer units define the granularity with which the timer interrupt intervals can be set. For example, the PC timer driver uses the PC's 8253 timer chip, which counts in intervals of 0.838 microseconds. Timer interrupt intervals can be specified as multiples of 0.838 microseconds. Thus, for this driver, the *timer unit* is 0.838 microseconds.

The application interface to module Clock is given in include file CLOCK.H. The available functions are described below.

Function CLKSetResolution

This function defines conversion constants required to convert clock timer units to seconds and vice versa:

```
void CLKSetResolution (unsigned UnitsPerSecond, unsigned Divisor);
```

Times will be converted using the formula

```
Timer Unit = Seconds * UnitsPerSecond / Divisor;
```

Divisor should be less than 4295 to allow conversions to microseconds.

In most cases, the clock device driver will call this function to supply the appropriate default conversion values.

Function CLKSetTimerIntVal

This function changes the frequency of the timer interrupt:

```
void CLKSetTimerIntVal(unsigned Micros);
```

Parameter Micros should be supplied as microseconds. Module clock will convert it to timer units and pass that value on to the timer interrupt driver. The actual value used will depend on the capabilities of the driver. The driver will round the value to the nearest value it can support.

Alternatively, applications can also call the low-level routines `_rtkCLKSetTimerInterval` and `_rtkCLKCurrentTimerInterval` for a higher degree of control over the timer interrupt interval. However, this approach would not be portable between different timer interrupt drivers.

It is recommended to immediately execute an `RTKDelay(1)` statement after changing the timer interrupt frequency. Some timer drivers are not able to deliver accurate results in the time between changing the timer frequency and the first timer interrupt.

Time Conversions

Module clock provides the following functions to convert timer ticks to seconds, milliseconds, microseconds, and vice versa:

```
unsigned    CLKTicksToMicroSecs(RTKDuration T);
unsigned    CLKTicksToMilliSecs(RTKDuration T);
unsigned    CLKTicksToSeconds  (RTKDuration T);

RTKDuration CLKMicroSecsToTicks(unsigned Micros);
RTKDuration CLKMilliSecsToTicks(unsigned Millis);
RTKDuration CLKSecondsToTicks  (unsigned Secs);
```

All parameters and function results are limited to 32 bits. The function results are undefined if they would exceed this limit. In addition, all functions converting to `RTKDuration` can trigger an exception 0 if the result would exceed 32 bits.

All conversion functions round down.

Module Timer

Module Timer is a high-level floating point interface to the timer device drivers. The advantage of using Timer is that better rounding can yield more accurate results and overflow situations are handled more gracefully than by modules `FineTime` and `Clock`. The disadvantage is that Timer requires floating point. Thus, all tasks that want to use Timer must have their own floating point context and the program must run in an environment that supports floating point (either with a math coprocessor or with an emulator). The primary data type of module Timer is seconds (type `TISecds`, double).

The application interface to module Timer is given in include file `TIMER.H`. Its functions are described below.

Function TimerInit

Function `TimerInit` must be called once by the application before any of the functions defined by `Timer` can be used:

```
void TimerInit(void);
```

Function TIElapsedTime

This function calculates the time relative to a given time value:

```
TISeconds TIElapsedTime(const RTKFineTime * T);
```

Parameter `(*T)` must have been set by a call to `FTReadTime`. `TIElapsedTime` subtracts an overhead value from the measured time. Thus, a construct such as:

```
TISeconds Result;  
RTKFineTime T;  
  
FTReadTime(&T);  
Result = TIElapsedTime(&T);
```

will yield zero or a very small value. Please note that the time returned may be negative due to caching or other time-distorting effects.

Function TIElapsedAndMark

`TIElapsedAndMark` returns the elapsed time since `FTReadTime` and resets the Parameter `(*T)` to the current time:

```
TISeconds TIElapsedAndMark(RTKFineTime * T);
```

This function is useful to measure a set of consecutive time intervals. No overhead is subtracted from the returned value.

Function TISetTimerInterval

This function changes the timer interrupt interval:

```
void RTKAPI TISetTimerInterval(TISeconds Seconds);
```

Parameter `Seconds` is rounded to the nearest value supported by the underlying timer interrupt driver. To enquire the actual value, use `TITicksToSeconds(1)`.

It is recommended to immediately execute an `RTKDelay(1)` statement after changing the timer interrupt frequency. Some timer drivers are not able to deliver accurate results in the time between changing the timer frequency and the first timer interrupt.

Time Conversions

Module `Timer` provides the following functions for conversions between floating point seconds and `RTKernel-32` timer ticks:

```
TISeconds TIFineTimeToSeconds(const RTKFineTime * T);  
TISeconds TITicksToSeconds (RTKDuration T);  
RTKDuration TISecondsToTicks (TISeconds T);
```

The results are rounded to the nearest representable value. The result of `TISecondsToTicks` is undefined if it would exceed 32 bits.

These time conversion routines only work correctly if `TimerInit` has been called and all changes to the timer interrupt interval have been performed using function `TISetTimerInterval`.

Module RTCom

Module `RTCom` offers completely interrupt-driven communication through serial ports. Some of `RTCom`'s features:

- transmission at any baud rate supported by the hardware
- simultaneous support of up to 38 ports

- support of DigiBoard and Hostess boards
- configurable I/O addresses and IRQs
- protocols XOn/XOff, RTS/CTS, and DSR/DTR
- receive and send buffers of arbitrary size
- full error detection for each received byte
- full support of the 16550 UART's FIFO buffer

The interface for RTCom is given in include file RTCOM.H.

RTCom supports both polled and interrupt-driven communication. In general, interrupt-driven communication should be used for better performance. In interrupt-driven mode, data is buffered up to a user-defined limit before data is lost. In polling mode, no data is buffered. If a 16550A UART chip is used by a serial port, its internal 16-byte buffer is used.

If interrupt-driven communication is desired on a "standard" port (i.e., a port not on a DigiBoard, Hostess card, or other interrupt-sharing card), the following steps should be taken:

- If the port does not use a standard I/O port address as defined for the IBM PC, call COMSetIOBase. If it does not use a standard IRQ as defined for the IBM PC, call COMSetIRQ also.
- Initialize the port using function COMPortInit.
- If you need a protocol, call COMSetProtocol.
- Call COMEnableInterrupt. All incoming data is put into mailbox COMReceiveBuffer[x], where x is the corresponding port. The mailboxes contain records of type COMData containing the actual data received in the low byte and an error status byte (as defined by the Line Status Register) in the high byte.
- Tasks that want to receive data should perform RTKGet, RTKGetCond, or RTKGetTimed on the corresponding mailboxes. Tasks that want to send should use COMSendChar. COMSendChar writes the data to a send buffer which is emptied by an interrupt handler.

Function COMEnableInterrupt allocates both receive and send buffers with the same size. If you need different sizes, call COMAllocateBuffers first; COMEnableInterrupt will not allocate buffers again.

If polled communication is desired, the following steps should be taken:

- If the port does not use a standard I/O port address as defined for the IBM PC, call COMSetIOBase.
- Initialize the port using COMPortInit.
- To receive, call COMReceiveCharPolled.
- To send data, use COMSendCharPolled.

Protocols

By default, RTCom uses no protocol to prevent overflow errors. In this mode, it is assumed that all communication participants are ready to accept data. If a receive buffer of RTCom overflows, the error COM_BUFFER_FULL is reported.

In interrupt-driven mode, RTCom also supports protocols XOn/XOff, RTS/CTS, and DTR/DSR. These protocols can prevent buffer overflows. A receiver checks after each byte received whether the receive buffer has filled up to a dangerous level. If so, the sender is informed to stop sending. As soon as the receive buffer has been emptied to a safe level, the sender is allowed to continue sending.

The XOn/XOff protocol uses special ASCII control characters to send the necessary information to the sender. It can only be used if the actual data stream never contains XOn/XOff characters. Protocols RTS/CTS and DTR/DSR use handshake lines. However, they require that the corresponding lines are actually connected in the cable used.

RTCom supports these protocols in *Passive* and *Active* modes. In *Passive* mode, RTCom will understand and adhere to XOn/XOff characters or CTS or DSR signals, respectively. In this way, a buffer overflow error can be prevented on the remote device. The local receive buffer can still overflow if the receiving task does not collect the data fast enough. In *Active* mode, RTCom will also protect its own receive buffer. If a buffer fills to more than 70% of its size, an XOff is sent to the remote device (or signal RTS or DTR is reset). In this case, the remote device should stop sending. A protocol task in RTCom will then periodically check whether the buffer is filled to less than 30% of its size. If so, the remote device is allowed to continue sending (by sending an XOn to it, or by setting the RTS or DTR signal).

Hardware Configuration

RTCom expects the following IRQ settings for COM1 to COM4:

```
COM1  ->  IRQ4
COM2  ->  IRQ3
COM3  ->  IRQ4
COM4  ->  IRQ3
```

Please make sure that your serial I/O board actually supports interrupt sharing. If it does not, you cannot use COM1 and COM3 or COM2 and COM4 simultaneously in interrupt-driven mode. COMSetIRQ may be used to define different IRQs for ports COM1 .. COM6. Interrupt sharing can be defined for any IRQ, but only for port pairs COM1/COM3 and COM2/COM4. Best performance is achieved when every port has its own IRQ.

DigiBoard Cards (PC/4, PC/8, PC/16)

Before you can use one or more DigiBoard cards, you have to inform RTCom about the card type, the status register address, the interrupt request to be used, and which port number you would like to assign to the first port on the board. Please note that the card(s) must be configured to use only one interrupt request. If you have several cards installed, they should all use the same address for the status register.

For example, if you have set the status register to 0x140, the IRQ to 7, and you would like the ports on your card to start at COM5, you should include the following statement in your program before the card can be used:

```
COMSetBoardType(COM_BOARD_DIGIBOARD, 0x140, 7, 4);
```

Subsequently, the I/O address of each port must be individually defined using COMSetIOBase. If you have a card with four ports and you have left the I/O addresses at factory settings, the following statements would be required:

```
COMSetIOBase(4, 0x100);
COMSetIOBase(5, 0x108);
COMSetIOBase(6, 0x110);
COMSetIOBase(7, 0x118);
```

After this initialization, ports 4 through 7 can be used as explained above. Both interrupt-driven and polling modes are supported.

Hostess Cards (4, 8, or 16 Ports)

Before you can use a Hostess card, you have to inform RTCom about the card type, its base address, the interrupt request to be used, and which port number you would like to assign to the first port on the board. Cards with four or eight ports are identified by card type COM_BOARD_HOSTESS. Cards with 16 ports have card type COM_BOARD_HOSTESS_16. For example, if your board with four ports is installed at address 0x280 and uses IRQ 7, and you already have COM1 and COM2 in your computer, then you should include the following statement in your program before the card can be used:

```
COMSetBoardType(COM_BOARD_HOSTESS, 0x280, 7, 2);
```

Then, the I/O address of each port must be defined individually using COMSetIOBase. Using the example above, the following statements would be required:

```
COMSetIOBase(2, 0x280);
COMSetIOBase(3, 0x288);
COMSetIOBase(4, 0x290);
COMSetIOBase(5, 0x298);
```

After this initialization, ports 2 through 5 (COM3 .. COM6) can be used as explained above. Both interrupt-driven and polling modes are supported.

Other Interrupt Sharing Cards (2 to 32 Ports)

DigiBoard and Hostess cards have a special status register used to identify the port that has generated an interrupt. If your I/O card does not have such a register, you can use the `COM_BOARD_GENERIC` type. It checks all ports on the card at every interrupt. Before you can use a generic card, you have to inform RTCom about the card type, the interrupt request to be used, which port number you would like to assign to the first port on the board, and how many ports the board has. For example, if your board with four ports uses IRQ 7 and you already have COM1 and COM2 in your computer, then you should include the following statement in your program before the card can be used:

```
COMSetBoardType(COM_BOARD_GENERIC, 4, 7, 2);
```

Then, the I/O address of each port must be defined individually using `COMSetIOBase`. Assuming the first port uses I/O address 0x280, the following statements would be required:

```
COMSetIOBase(2, 0x280);
COMSetIOBase(3, 0x288);
COMSetIOBase(4, 0x290);
COMSetIOBase(5, 0x298);
```

After this initialization, ports 2 through 5 can be used as described above. Both interrupt-driven and polling modes are supported.

Please note that DigiBoards and Hostess cards show better performance than generic cards thanks to their hardware support for interrupt sharing.

Function `COMSetBoardType`

`COMSetBoardType` can define the properties of an interrupt sharing serial I/O board:

```
void COMSetBoardType(int Type, unsigned Address, int IRQ, int FirstPort);
```

If you intend to use a generic, Hostess, or DigiBoard card, its parameters must be defined using this function. Parameter `Type` may assume one of the values `COM_BOARD_GENERIC`, `COM_BOARD_DIGIBOARD`, `COM_BOARD_HOSTESS`, or `COM_BOARD_HOSTESS_16`. Parameter `Address` specifies the I/O address of the board's status or vector register. For the `COM_BOARD_GENERIC` type, parameter `Address` is interpreted as the number of ports on the card. `IRQ` specifies the interrupt the card will use to signal events. Parameter `FirstPort` is the port number you want to assign to the board's first port. If, for example, you want to address the ports on a board with four ports as COM5, COM6, COM7, and COM8, `FirstPort` should be set to COM5 (value 4).

For additional instructions on how to use COM boards with RTCom, please refer to section *Hardware Configuration* in this chapter.

Function `COMSetIOBase`

`COMSetIOBase` sets the port base address for a port:

```
void COMSetIOBase(int Port, unsigned int IOBase);
```

Any port not using standard I/O port addresses must be defined using this function. This is always necessary for port numbers greater than 3 (COM4).

Function `COMSetIRQ`

`COMSetIRQ` defines the interrupt triggered by a port:

```
void COMSetIRQ(int Port, int IRQ);
```

If no `IRQ` is specified in this call, RTCom assumes `IRQ 4` for COM1 and COM3 and `IRQ 3` for COM2 and COM4. You can share `IRQs` for port pairs COM1/COM3 and/or COM2/COM4. Best performance is achieved when each port uses its own `IRQ`. Only ports 0 .. 5 (COM1 .. COM6) are supported by this call. There are no default values for COM5 and COM6.

Function COMPortInit

COMPortInit initializes a port:

```
void COMPortInit(int Port, long Baud, int Parity, int StopBits, int WordLength);
```

COMPortInit sets the port parameters and disables interrupts for the port. If a 16550A UART is detected, its FIFO is automatically enabled at trigger level 8.

Parameter Port must be in the range 0 to COM_MAX_PORTS-1 (37). Supported baud rates range from 50 to 115200. Parity must be set to PARITY_NONE, PARITY_ODD, PARITY_EVEN, PARITY_MARK, or PARITY_SPACE. StopBits may be set to 1 or 2. WordLength must be in the range 5 to 8.

Please note that RTCom assumes the UART uses a clock input of 1.8432MHz (the value commonly used on PCs). If a different input clock frequency is used by the target hardware, the baud rate specified for COMPortInit must be adjusted according to the following formula:

$$B = (1,843,200 / \text{Clock Frequency}) * \text{Baudrate}$$

where *B* is the value to be supplied for COMPortInit and *Baudrate* is the effective baud rate to be used.

Example: The target computer has a 386EX CPU running at 25Mhz and you want to use COM2 (the second internal serial port of the 386EX) at 57600 baud. The input frequency for the serial ports is CLK2 (50Mhz in this example) divided by 4, giving 12.5 MHz. The value required for COMPortInit is:

$$(1.8432 / 12.5) * 57600 = 8493$$

Function COMHasFIFO

COMHasFIFO can detect UARTs equipped with an internal FIFO buffer:

```
RTKBool COMHasFIFO(int Port);
```

The function returns TRUE for all 16550A or compatible UARTs.

Function COMEnableFIFO

COMEnableFIFO can enable or disable the internal buffer of 16550A or compatible UARTs:

```
void COMEnableFIFO(int Port, int Trigger);
```

If the port uses a 16550A UART, this function can be used to set the interrupt trigger level. The trigger level specifies after how many received bytes an interrupt is generated. If no bytes are received during a time span sufficient to transmit four bytes, the UART will generate a time-out interrupt for all trigger levels. Thus, no data will stay in the FIFO buffer forever. By default, RTCom sets the trigger level to 8 if it detects a 16550A. You may change this value to 1, 4, 8, or 14. A value of 0 disables the FIFO buffer. Low values for the trigger level may result in a faster response of tasks waiting for data; however, overall system performance will suffer due to many interrupts. With high values (e.g., 8 or 14), you get less interrupts resulting in a higher system throughput. At 14, overrun errors may occur at very high baud rates. For most applications, trigger level 8 is a good compromise.

Function COMSetProtocol

COMSetProtocol can set the desired protocol for a particular port:

```
void COMSetProtocol(int Port, int Prot, UINT ProtTaskPrio, RTKDuration PollCycle);
```

Parameter Prot may have one of the following values: COM_PROT_NONE, COM_PROT_XON_XOFF, COM_PROT_RTS_CTS, COM_PROT_DTR_DSR. If parameter ProtTaskPrio is equal to COM_PASSIVE, a passive protocol is assumed and PollCycle is ignored. Otherwise, a protocol task is created with priority ProtTaskPrio. This task will run in PollCycle intervals and check whether a protocol action is required on any port. Only one such protocol task is created for all ports. The last values given for ProtTaskPrio and PollCycle apply.

If this function is never called for a specific port, no protocol is assumed. Protocols are only supported for interrupt-driven communication.

Function COMAllocateBuffers

COMAllocateBuffers allocates send and receive buffers for a port:

```
void COMAllocateBuffers(int P, UINT ReceiveBufferSize, UINT SendBufferSize);
```

If the buffers have been allocated already, this call has no effect. The call to this function is not mandatory; function COMEnableInterrupt will call

```
COMAllocateBuffers(P, BufferSize, BufferSize)
```

to ensure that buffers are available.

Function COMEnableInterrupt

COMEnableInterrupt enables interrupt-driven communication for a particular port:

```
void COMEnableInterrupt(int Port, unsigned BufferSize);
```

COMEnableInterrupt enables interrupts for sending/receiving and allocates send and receive mailboxes with BufferSize slots each if no buffers have been allocated yet. If you need different sizes for receive and send buffers, call COMAllocateBuffers first. If the port uses a 16550A UART, the interrupt trigger level defaults to 8 characters.

Function COMDisableInterrupt

COMDisableInterrupt disables interrupts previously enabled with COMEnableInterrupt.

```
void COMDisableInterrupt(int Port);
```

The send and receive buffers are not cleared or deleted.

Function COMSendChar

COMSendChar sends a character asynchronously using interrupts:

```
void COMSendChar(int Port, BYTE Data);
```

Parameter Data is placed in the send buffer to be transmitted by the interrupt handler as soon as the transmit register becomes empty. Note that this function will typically return before the data is actually transmitted.

Function COMSendCharTimed

COMSendCharTimed sends data with a timeout:

```
RTKBool COMSendCharTimed(int Port, BYTE Data, RTKDuration Timeout);
```

Same as COMSendChar, but this function returns FALSE if the data could not be placed in the send buffer within the time given in parameter Timeout. Please note that the data is only placed in the send buffer and is not guaranteed to have been sent even when this function returns TRUE.

Function COMSendBlock

COMSendBlock sends a block of data to a COM port:

```
void COMSendBlock(int Port, void * Data, UINT Length);
```

This function is equivalent to:

```
for(i=0; i<Length; i++)  
    COMSendChar(P, Data[i]);
```

but is faster.

Function COMSendBlockTimed

COMSendBlockTimed sends a block of data to a COM port with a timeout:

```
UINT COMSendBlockTimed(int Port, void * Data, UINT Length, RTKDuration Timeout);
```

COMSendBlockTimed is similar to COMSendBlock, but the latter function returns the number of bytes placed in the send buffer without getting a timeout. If the data was transferred to the send buffer completely, Length is returned. The timeout applies to the complete data block.

Function COMWaitSendBufferEmpty

COMWaitSendBufferEmpty waits for a send buffer to become empty:

```
RTKBool COMWaitSendBufferEmpty(int Port, RTKDuration Timeout);
```

This function waits until the send buffer of Port is empty, that is, all pending data has been sent to the UART. COMWaitSendBufferEmpty does not use polling; instead, the calling task is blocked until the send buffer is empty. If the function returns TRUE, the send buffer is empty. Otherwise, data may still be in the send buffer and the timeout has expired. Please note that the last byte may still be in the send shift register. To be sure all data has been sent out, check bit COM_TX_SHIFT_EMPTY in the Line Status register after a call to COMWaitSendBufferEmpty. Example:

```
if (!COMWaitSendBufferEmpty(P, 100))
    printf("Send buffer not empty after 100 timer ticks!");
else
    while (!(COMLineStatus(P) & COM_TX_SHIFT_EMPTY))
        RTKScheduler();
```

Function COMSetModemStatusHook

COMSetModemStatusHook installs an application function to be called on every modem status line change:

```
void COMSetModemStatusHook(int Port, void (* RTKAPI Hook)(int Port, BYTE Status));
```

Such a modem status hook may be used to trap the change of modem status signals, such as Data Carrier or Ring Indicator. Apart from the port number, the hook is called with the new modem status value. The hook is called in the context of an interrupt handler. COMEnableInterrupt must have been called for the hook to take effect.

Function COMReceiveCharPolled

This function receives one byte using polling:

```
COMData COMReceiveCharPolled(int Port);
```

The received character and any error bits are returned. Do not use this function together with function COMLineStatus. COMLineStatus will delete any errors that may have occurred.

Function COMSendCharPolled

COMSendCharPolled sends a byte using polling:

```
void COMSendCharPolled(int Port, BYTE Data);
```

The function polls the port until the transmit register is empty. Subsequently, the Data character is transmitted.

Function COMLineStatus

COMLineStatus enquires the line status register of a port:

```
BYTE COMLineStatus(int Port);
```

The contents of the Line Status Register is returned. The return value is bit-oriented; the following bits can be set:

COM_DATA_READY	A data byte has been received.
COM_OVERRUN	An overrun error has occurred.
COM_PARITY	A parity error has been detected.
COM_FRAME	A frame error has been detected.
COM_BREAK	A break condition has been detected.
COM_TXB_EMPTY	The transmit hold register is empty.
COM_TX_SHIFT_EMPTY	The transmit shift register is empty.

All error bits (COM_OVERRUN, COM_PARITY, COM_FRAME, and COM_BREAK) are cleared after they have been read once. Since RTCom's interrupt handler also reads the line status register, these can never be read with COMLineStatus if interrupt-driven communication is used.

Function COMModemStatus

COMModemStatus reads a port's Modem Status Register:

```
BYTE COMModemStatus(int Port);
```

The following bits are defined:

COM_DCTS	Signal CTS has changed.
COM_DDSDR	Signal DSR has changed.
COM_DRI	Signal RI has changed.
COM_DDCD	Signal DCD has changed.
COM_CTS	Clear To Send.
COM_DSR	Data Set Ready.
COM_RI	Ring Indicator.
COM_DCD	Data Carrier Detect.

Function COMModemControl

Function COMModemControl writes a value to the Modem Control Register:

```
void COMModemControl(int Port, int SetToOneZero, int NewValue);
```

If parameter SetToOneZero is one, the value is *ored* into the register; if it is zero, it is *not-anded*. The following bits are defined:

COM_DTR	Set Data Terminal Ready.
COM_RTS	Set Request To Send.
COM_OUT1	Set OUT 1.
COM_OUT2	Set OUT 2.
COM_LOOPBACK	Set Loop-Back Mode.

Function COMError

COMError can translate an RTCom error code to readable text:

```
char * COMError(COMData Data);
```

COMError returns a pointer to a string corresponding to the most severe error set in parameter Data.

Module RTKeybrd

RTKeybrd contains functions KBGetCh, KBKeyPressed, and KBPutChar (to insert keystrokes into the keyboard buffer by software). RTKeybrd releases the CPU for other tasks while waiting for keyboard input.

Multitasking programs requiring keyboard input should utilize RTKeybrd in order to make CPU time available to other tasks that would otherwise be wasted waiting for keyboard input.

RTKeybrd hooks into RTTarget-32's user input event manager and registers RTTarget-32's keyboard interrupt handler with RTKernel-32's interrupt system. All functions waiting for user input (e.g., RTGetCh(), Win32 console I/O functions or C/C++ run-time systems functions) will block the calling task, freeing CPU time for other tasks. A keyboard interrupt will then reactivate the respective task instantaneously.

RTKeybrd declares a binary semaphore which is set by every keyboard interrupt. Applications can wait on this semaphore (e.g., to be able to wait for keyboard input with a timeout). Please note that setting semaphore `KBKeyAvailable` does not guarantee that a key is available in the keyboard buffer. It only means that some kind of keyboard event has occurred (e.g., a key was released).

The functions of module `RTKeybrd` are declared in header file `RTKEYBRD.H`.

Function `KBInit`

`KBInit` initializes module `RTKeybrd`:

```
void KBInit(void);
```

`KBInit` creates semaphore `KBKeyAvail` and installs all required hooks for `RTTarget-32`'s user input event management.

Function `KBKeyPressed`

`KBKeyPressed` checks whether a key is available in the keyboard buffer.

```
RTKBool KBKeyPressed(void);
```

If characters are available in the keyboard buffer, `TRUE` is returned and `KBGetCh` would succeed immediately.

`KBKeyPressed` corresponds to the `kbhit` function supplied by most C/C++ compilers.

Function `KBGetCh`

`KBGetCh` waits for and retrieves keyboard input:

```
int KBGetCh(void);
```

As long as no keyboard input is available, the calling task is blocked. The return value is the ASCII value of the key pressed. If the key was an extended key without an associated ASCII value, 0 is returned and a second call to `KBGetCh()` retrieves the key's keyboard scan code.

`KBGetCh` corresponds to the `getch` function supplied by most C/C++ compilers. However, `getch()` can also be used.

Function `KBPutCh`

`KBPutCh` places a character in the keyboard buffer:

```
void KBPutCh(char C, char ScanCode);
```

Parameter `C` is the ASCII value of the key. For normal keys, `ScanCode` is ignored and may be zero. For extended keys without ASCII representation, parameter `C` should be zero and `ScanCode` is the keyboard scan code of that key.

Module `RTextIO`

Module `RTextIO` provides a simple windowing facility to share the screen among several tasks. Each task can reserve a window on the screen and perform text I/O in this window. Each window can have a title, a frame, and different colors for foreground and background.

Each window owns a structure created and initialized by function `WOpenTextWindow` or `WNewWindow`. This structure emulates the functions of a text file. `RTextIO` provides functions corresponding to `fprintf`, `fputs`, `fgets`, etc., for I/O in the window.

A task that wants to write to the screen should perform the following steps:

- If you are not using a color graphics display, call `WSetVideoRAMAddress` to define screen and video controller parameters.
- If user input is required, call `WSetUserInput()`.

- Execute `WNewWindow` or `WOpenWindow` to obtain a pointer to a `WWindow` structure. Make sure the window does not overlap any windows defined by other tasks. If `WOpenWindows` is used and a frame is desired, an extra line and two extra columns must be reserved above, below, and to the right and left of each window.
- Use `WPutC`, `WPutS`, `WGetS`, and `Wprintf` to access the window. If two tasks try to read simultaneously, an orderly distribution of the characters to the tasks cannot be expected.
- Use `WGotoXY` to position the (virtual) output cursor anywhere in the window. Please note that the upper left-hand corner is at (0,0). Since the display adapter supports only one physical cursor, the physical cursor is used only to mark the next character input position. However, each window has its own logical output cursor.
- If you want the physical cursor to appear at the logical cursor position of a window, call `WSetCursor`. `RTTextIO` will normally display the physical cursor only when user input is expected.
- If you want `RTTextIO` to expand function keys on the command line, use function `WDefineFunctionKey` to associate a key with a string. The string should not be longer than 15 characters. `WDefineFunctionKey` may be used to reassign any key that generates a zero followed by a scan code using `KBGetCh`.

If you want to use a non-standard video mode (e.g., to have more rows and columns on the screen), use function `WSetScreenSize` to inform `RTTextIO`. Up to 132 columns and 75 rows are supported.

The functions of module `RTTextIO` are declared in header file `RTTEXTIO.H`.

Function `WSetVideoRAMAddress`

`WSetVideoRAMAddress` defines the addresses of the video RAM and the video controller:

```
void WSetVideoRAMAddress(unsigned short Segment,
                        unsigned int   Offset,
                        unsigned short CRTController);
```

Parameter `Segment` is a selector used for screen access. If it is zero (default), the default data segment selector is used. The default offset is `0xB8000`. For monochrome displays, you must use this function to set the address to physical address `0xB0000` at program initialization. `CRTController` defaults to `W_CRT_COLOR`. For monochrome displays, you must set it to `W_CRT_MONO`. If zero is used, `RTTextIO` will not address the controller for cursor movement.

Function `WSetScreenSize`

`WSetScreenSize` defines the size of the physical screen:

```
void WSetScreenSize(int Cols, int Rows);
```

`RTTextIO` assumes the screen to have 80 columns and 25 rows. Calling this function is only required for other screen sizes. Up to 132 columns and 75 rows are supported.

Function `WSetUserInput`

If any task needs user input, you must install an input routine with this function:

```
void WSetUserInput(WUserInputFunction GetInput);
```

Parameter `GetInput` has no parameters and returns an integer. It should return an ASCII value or a 0 followed by a scan code for extended keys. Most applications will only use function `KBGetCh` to read from the keyboard.

The following example shows how to read input from serial port COM2 instead of the keyboard:

```
int RTKAPI UserKey(void)
{
    COMData Data;
    RTKGet(COMReceiveBuffer[COM2], &Data);
    COMSendChar(COM2, Data);
    if (Data == '\r')
```

```
        COMSendChar(COM2, '\n');
    return Data;
}

int main(void)
{
    COMPortInit(COM2, 9600, PARITY_NONE, 1, 8);
    COMEnableInterrupt(COM2, 100);
    WSetUserInput(UserKey);
    ...
}
```

Function WDefineFunctionKey

WDefineFunctionKey can be used to instruct RTextIO to expand function keys to a string:

```
void WDefineFunctionKey(char ScanCode, const char * S);
```

When the user input function returns zero, the following value is interpreted as a scan code. If this scan code matches a scan code specified in a previous call to WDefineFunctionKey, the corresponding string pointed to by S is used as the input instead.

Function WClearScreen

WClearScreen clears the entire screen with the desired color:

```
void WClearScreen(int Attr);
```

A blank character is written to the entire screen with attribute Attr. Usually, the attribute should be set to 0x7 for white foreground and black background.

Function WNewWindow

Function WNewWindow creates a new window on the screen and returns a pointer to the associated windows structure:

```
WWindow * WNewWindow(int FirstCol, int FirstRow,
                     int LastCol, int LastRow,
                     int BufferSize, int Attr,
                     const char * Title);
```

WNewWindow calls WOpenWindow and WFrame and clears the window.

Function WOpenWindow

WOpenWindow creates a new window on the screen, but does not draw a frame and does not clear the new window:

```
WWindow * WOpenWindow(int FirstCol, int FirstRow,
                     int LastCol, int LastRow,
                     int BufferSize);
```

Parameter BufferSize defines the size of a buffer which RTextIO allocates to store formatted strings in function Wprintf. If you don't need Wprintf, you can set BufferSize to 0 to save memory. The coordinates of the new window are relative to the upper left hand corner of the screen starting at (0,0).

Function WCloseWindow

WCloseWindow deallocates a window created by WOpenWindow or WNewWindow:

```
void WCloseWindow(WWindow * W);
```

Function WFrame

WFrame draws a frame around a window created by WOpenWindow.

```
void WFrame(WWindow * W, const char * Title);
```

You must reserve one character above and below and two characters on the left and right sides of the window for the frame. Adjacent frames are connected. The title is placed in the frame at the top of the window.

Function **WGotoXY**

WGotoXY positions the logical (invisible) output cursor:

```
void WGotoXY(WWindow * W, int Col, int Row);
```

Parameters Col and Row are relative to the window's upper left-hand corner.

Function **WCursorXY**

WCursorXY positions the visible cursor to the given absolute screen position:

```
void WCursorXY(int Col, int Row);
```

Parameters Col and Row are relative to the screen's upper left-hand corner.

Function **WCursorOFF**

WCursorOFF hides the physical cursor:

```
void WCursorOFF(void);
```

Function **WCursorON**

WCursorON displays the physical cursor:

```
void WCursorON(void);
```

Function **WSetCursor**

WSetCursor positions the physical cursor to the current position of the logical output cursor of the given window:

```
void WSetCursor(WWindow * W);
```

Function **WSetColor**

WSetColor defines a new screen attribute to be used for the given window:

```
void WSetColor(WWindow * W, int color);
```

Parameter color is the attribute to be written to the screen in subsequent output.

Function **WPutC**

WPutC writes a character to a window:

```
int WPutC(WWindow * W, int c);
```

Characters CR, LF, BS, and TAB will be interpreted as control characters. FF (form feed) will clear the window. All other characters are written to the video RAM and the logical output cursor position is advanced.

Function **WPutS**

WPutS writes a null-terminated ASCII string to a window:

```
int WPutS(WWindow * W, const char * s);
```

Function **WGetS**

WGetS reads a string into the given buffer:

```
char *WGetS(WWindow * W, char * s, int maxstrlen);
```

You must call WSetUserInput to define an input routine before this function can be used. The input read is placed in buffer *s until character CR is encountered. The trailing CR is not stored in *s. Parameter maxstrlen defines the length of the buffer pointed to by s. WGetS will discard any input after (maxstrlen-1) characters have been read.

Function Wprintf

Wprintf works just like printf, but the formatted output is displayed in the given window:

```
int Wprintf(WWindow * W, const char * format, ... );
```

The formatted string is first placed into the string buffer associated with the window. The size of this buffer is specified by the BufferSize parameter to WOpenWindow or WNewWindow.

Module CPUMoni

Module CPUMoni can be used to determine the current CPU load of a multitasking program. CPU load is defined as the portion of CPU time used by tasks having priorities greater than 0 or 1.

CPUMoni supports two different methods for calculating the average CPU load over a given time interval. The method used is determined by the Method parameter to CPUMonitorStart and the current configuration of RTKernel-32. The methods are:

CPU_IDLE_TASK: CPUMoni will determine the amount of CPU time allocated to RTKernel-32's Idle Task. The CPU load returned by CPURelativeLoad is calculated from the total real time and the CPU time used by the Idle Task since the last call to CPURelativeLoad. This method will yield slightly different results depending on flag RF_ICPUTIME in RTKConfig.Flags. If this flag is not set, the time consumed by interrupt handlers is assumed to be used by the task being interrupted.

CPU_COUNTER_TASK: With this method, a task with priority 1 that counts in an endless loop is created. CPUMonitorStart will measure how fast this task can count when it runs exclusively. Subsequent calls to CPURelativeLoad will evaluate how far the counting task has counted and calculate the amount of CPU time that was available to this task from the count rate. The CPU_COUNTER_TASK method only works if all other tasks (apart from RTKernel-32's Idle Task) have a priority greater than 1.

In general, the CPU_IDLE_TASK method is more accurate, since it is independent of cache and other CPU speed-distorting effects.

Function CPUMonitorStart

CPUMonitorStart must be called once at program initialization to start the CPU monitor:

```
int CPUMonitorStart(int Method);
```

Parameter Method can have value CPU_IDLE_TASK or CPU_COUNTER_TASK. If CPU_IDLE_TASK is specified, CPUMonitorStart will check whether RTKernel-32 measures the CPU time consumed by the Idle Task. If this is not the case, method CPU_COUNTER_TASK is used. The return value shows which method (CPU_IDLE_TASK or CPU_COUNTER_TASK) is actually being used.

Function CPURelativeLoad

CPURelativeLoad calculates the relative CPU load since the call to CPUMonitorStart or the last call to CPURelativeLoad:

```
unsigned CPURelativeLoad(unsigned Factor);
```

The function return value is in the range of 0 .. Factor. 0 means that no CPU time was available; Factor means that all CPU time was available. If Factor is set to 100, the return value is in percent.

The return value is rounded down to the next lower integer value.

Chapter 5

RTKernel-32 Drivers

RTKernel-32 is designed to be portable. It is completely written in ANSI C. All target system and hardware dependencies are encapsulated in a set of drivers described in this chapter. Various drivers are delivered with RTKernel-32 in full source code. Only the source code of the portable kernel and the Intel 386 CPU driver must be purchased separately, if required. Thus, it is possible to port RTKernel-32 to other platforms by supplying an adequate set of drivers.

The complete interfaces of the drivers are given in the respective header files (one for each driver). The following description of each driver details the header file describing the respective driver, which drivers are available, and for which platforms they are suitable. Section *Overview of all Drivers* gives an overview of all available drivers. Each driver is delivered as a library file to be linked to RTKernel-32 applications. Alternatively, the prebuilt driver library DRVRT32.LIB can be used for RTKernel-32 programs running under RTTarget-32 with the default drivers.

System Interface

The system driver contains some target system or run-time system dependencies of RTKernel-32. Specifically, the following functions are concerned:

- default error message handler
- fatal error handler
- idle handler (called by the Idle Task)
- query main task stack limits
- task initialization hook
- task termination hook

The complete interface of the system driver is documented in header file RTKSYS.H.

Driver SYSSTD

This driver is a generic system driver. The error message handler prints a string to stdout using function fputs. The fatal error handler only calls exit. The idle handler executes HLT if running at CPL 0 with flag DF_IDLE_HALT set in RTKConfig.Driverflag and otherwise returns immediately. DF_IDLE_HALT is not set by default. All other functions are dummy (i.e., do nothing and return).

This driver may be used as a template for custom system drivers. SYSSTD does not set up Win32 TEBs for the tasks and thus does not completely support Win32 emulation or TLS data segments (all tasks would share the same TLS data segment of the main task).

Requirements: C/C++ run-time system, file I/O for stdout.

Driver SYSRT32

SYSRT32 is intended for RTTarget-32. The message handler calls RTTarget-32's function RTDisplayString and the fatal error handler calls exit. The main task stack query function is fully implemented. The init/done task hooks will appropriately set up TEBs and TLS data using the local descriptor table. Therefore, this driver is suitable for Win32 emulation and supports TLS data segments and the TLS Win32 API functions. However, since each task requires a selector, this driver limits the number of tasks existing at any given time to 8192 (including the Main and Idle Tasks).

Requirements: RTTarget-32 1.1 or higher, C/C++ run-time system.

Interrupt Handling

The Interrupt driver is used to implement RTKernel-32's interrupt handling API. Specifically, it must provide functions to:

- enable and disable interrupts
- install and restore interrupt handlers
- chain interrupt handlers
- program the interrupt controller(s) (optional)

In addition, this driver is queried by RTKernel-32 for preemptive task switch support.

Actually, some of these functions are RTKernel-32 API functions, but are also used internally (e.g., RTKDisableInterrupts). The interrupt controller access functions RTKIRQEnd, RTKIRQTopPriority, RTKEnableIRQ, and RTKIRQDisableIRQ are implemented here, but are not used by the kernel itself. However, other drivers do rely on these functions.

The interrupt driver's API is defined in header file RTKIRQ.H.

Driver IRQRT32

IRQRT32 uses RTTarget-32's API for interrupt handling. It uses RTTarget-32's default interrupt request-to-interrupt vector mapping (IRQ0 .. 15 map to vectors 40h to 4Fh) and supports 16 IRQs.

Since RTTarget-32 supports running programs at CPL 0 or 3, this driver will replace any interrupt handler it finds at startup with a different privilege level than the current privilege level. This should usually not be a problem because RTTarget-32's boot code only installs dummy interrupt handlers.

Requirements: RTTarget-32 1.1 or higher, 8059A interrupt controller at address 20h, optionally second interrupt controller at A0h.

Kernel Clock

The clock driver supplies the kernel's interrupt-driven time base. RTKernel-32 does not know about *real time* in the sense of seconds, minutes, etc. All of its time services use a *timer tick* as their unit of time. RTKernel-32 is not aware of the actual time period between two timer ticks.

The Clock driver supplies functions for:

- timer interval programming
- installation of the RTKernel-32 timer callback.

The clock driver's interface is documented in file RTKCLK.H.

Driver CLKPC

This driver uses the 8253 timer chip, channel 0, to implement the clock interrupt. The maximum timer interrupt interval is 55 ms; the interval can be set at a resolution of 0.838 microseconds. The timer interrupt handler installed when RTKernelInit is called will continue to be called with the minimum 18.2 Hertz frequency if flag DF_TIMER_CHAIN is set in RTKConfig.DriverFlags, even when the timer interrupt rate is changed. By default, DF_TIMER_CHAIN is not set.

This driver calls RTKIRQTopPriority(1,8) at initialization to ensure that the timer interrupt is processed with the lowest interrupt priority. Applications requiring a different interrupt priority distribution should call RTKIRQTopPriority() again after RTKernelInit().

Requirements: 8253 timer chip at address 40h, timer generates IRQ 0. Incompatible with drivers CLKHRTPC and HRTPC.

Driver CLKHRTPC

This driver is actually a combined driver for the Clock and the High Resolution Timer. This driver must be combined since both times are implemented using the same hardware: the 8253 timer chip, channel 0.

This driver calls `RTKIRQTopPriority(1,8)` at initialization to ensure that the timer interrupt is processed with the lowest interrupt priority. Applications requiring a different interrupt priority distribution should call `RTKIRQTopPriority()` again after `RTKernellInit()`.

Requirements: 8253 timer chip at address 40h, interrupt controller 8059A at address 20h, timer generates IRQ 0. Incompatible with drivers `CLKPC` and `HRTPC`.

High Resolution Timer

The high resolution timer is an optional driver for RTKernel-32. If it is available, RTKernel-32's Debug Version can supply CPU time statistics for tasks and interrupt handlers. In addition, modules `FineTime` and `Timer` can be used for high resolution time measurements by the application. The driver must supply a function for reading a 64-bit integer time value.

The high resolution driver's interface is documented in file `RTKHRT.H`.

Driver `HRTNULL`

This is a dummy high resolution timer driver. The function to read the current high resolution time always returns zero. Use this driver if you don't need high resolution times or no suitable hardware is available.

Requirements: none.

Driver `HRTPC`

Driver `HRTPC` uses the 8253 timer chip, channel 0, to read high resolution times. It assumes that the timer chip's reload value is set to 2^{16} and is running at 1.193 Mhz (the PC's default). Times can be measured with a resolution of 0.838 microseconds.

Requirements: 8253 timer chip at address 40h, interrupt controller 8059A at address 20h, timer generates IRQ 0. Incompatible with drivers `CLKHRTPC` and `CLKPC`.

Driver `CLKHRTPC`

This driver is actually a combined driver for the Clock and the High Resolution Timer. This driver must be combined since both times are implemented using the same hardware: the 8253 timer chip, channel 0. This driver cannot guarantee accurate results between the time the timer interrupt frequency is changed and the first timer interrupt after the change. Also, times are only accurate as long as long as no recursive timer interrupts occur. Thus, the timer interrupt rate should not be set extremely high. The actual achievable interrupt rate will vary depending on the speed of the CPU used.

Requirements: 8253 timer chip at address 40h, interrupt controller 8059A at address 20h, timer generates IRQ 0. Incompatible with drivers `CLKPC` and `HRTPC`.

Driver `HRTPENT`

Applications that will always run on Intel Pentium or higher CPUs can use this driver for high resolution time values. Pentium and higher CPUs contain a 64-bit counter register which is incremented in every clock cycle. The resolution of the Pentium timer is far superior to the 8253 and reading the timer register is much faster. The only disadvantage is that the timer will run at different speeds, depending on the CPU's clock rate. You will have to calibrate the driver either manually by calling module `FineTime`'s function `FTSetTimeConverter` or with function `FTCalibrate`. The driver assumes a default clock rate of 120Mhz.

Requirements: Intel Pentium or higher CPU. Incompatible with clock driver `CLKHRTPC` (use `CLKPC` instead).

Driver `HRTSC520`

Driver `HRTSC520` uses the Software Timer of the AMD Élan SC520 CPU to deliver times with 1 microsecond resolution. By default, the driver assumes a system clock of 33.333MHz. If you use a 33.0MHz clock, be sure to include

```
MOVESB SWTMRCFG 1
```

in the Sc520ini.cfg file. This driver is incompatible with the default driver CLKHRTPC.LIB, which is also included in DRVRT32.LIB. To use it, link libraries HRTSC520.LIB, CLKPC.LIB, and DRVRT32.LIB in this order.

Note that the Software Timer of the Élan SC520 only covers a time period of about 65 seconds. Applications which need to measure time periods longer than 65 seconds must ensure that FTReadTime is called during the measurement at least every 65 seconds to allow the timer driver to update its internal time value in RAM. This is best achieved by including a call to FTReadTime in a low priority time cyclic task which has a cycle time of less than 65 seconds.

Floating Point

RTKernel-32 supports maintaining a floating point context for each task. This driver supplies information about the floating point implementation and functions to save and restore the floating point context.

The interface of the floating point driver is documented in file RTKFLT.H.

Driver FLTNUL

This is a dummy floating point driver. It informs RTKernel-32 that no floating point context is needed and no context swapping is required. You should use this driver either if your program does not require any floating point arithmetic or it will always use a completely reentrant floating point emulator. Please note that the 387 or internal math coprocessors are **not** reentrant and are not compatible with this driver.

Requirements: floating point free application or reentrant emulator.

Driver FLT387

This driver can manage the floating point context for the i387 math coprocessor or any external or internal math coprocessor compatible with it (e.g., 487, 486 internal, Pentium, etc.). In addition, it will also work for a completely 387 compatible emulator. In particular, it must support calls to FSAVE/FRSTOR while other FPU instructions are being executed, and it must be able to run with interrupts disabled.

Requirements: 387 compatible floating point hardware or emulator.

Driver FLTPII

This FPU driver uses the FXSAVE and FXRSTOR instructions introduced with the Intel Pentium II and higher CPUs. On Pentium III and higher CPUs, this driver will also save and restore all 128-bit SSE registers. This driver sets the OSFXSR bit in CR4 at program initialization. The FLTPII driver is slightly faster than FLT387, but it needs about 500 bytes more memory per thread. If used on CPUs which do not support the FXSAVE and FXRSTOR instructions (i.e. Pentium MMX or earlier or non-Intel), exception 6 (Invalid Opcode) will occur.

Requirements: Pentium II or higher CPU, RTTarget-32 3.0.

Driver FLTEMUMT

FLTEMUMT behaves differently depending on the presence of floating point hardware or an emulator. If floating point emulation is enabled, the driver will assume that a completely reentrant emulator is used and will not perform floating point context swapping. However, if a coprocessor is installed and not disabled, the driver behaves just like driver FLT387.

Requirements: 387 compatible floating point hardware or reentrant emulator, RTTarget-32 1.1 or higher.

Memory Management

Many of RTKernel-32's objects are allocated dynamically at run-time. These objects' memory is allocated by the memory driver, which provides functions to allocate and free memory.

The memory driver's interface is given in include file RTKMEM.H.

Driver MEMCHEAP

This driver maps memory allocation and deallocation functions to standard ANSI C functions malloc and free. The driver does not contain any reentrance protection. Therefore, malloc and free must be protected by some other means (e.g., RTKernel-32's Library Protection, the run-time system's Win32 multithreaded libraries, etc.). This driver should be used if you don't plan to use RTKernel-32's Win32 emulation and do not call any RTKernel-32 functions before function main has been invoked.

Requirements: C/C++ run-time system, no RTKernel-32 calls before main has been called.

Driver MEMSTH

MEMSTH (*Static Heap*) is implemented using a statically allocated uninitialized array in the program's data segment. The size of this heap is fixed and must be set at compile time. The default size is 2048 bytes. To change it, include the following statements in your program:

```
#define STHEAPSIZE (2*1024)
char STHeapData[STHEAPSIZE];
DWORD STHeapDataSize = STHEAPSIZE;
```

and adjust STHEAPSIZE to the desired value.

The static heap uses a semaphore to protect against recursive calls. The allocation strategy is "best fit" and will frequently use memory more efficiently than malloc. Allocations are supported even before the C/C++ run-time system is completely initialized.

To enquire the current state of the static heap, function MEMSTHeapInfo can be called:

```
typedef struct {
    DWORD TotalFree,           // available memory
           TotalUsed,         // total allocated memory
           FreeBlocks,        // number of free blocks
           UsedBlocks,        // number of allocated blocks
           LargestFree;       // largest free block
} MEMSTHeapInfoRec;

void RTKAPI MEMSTHeapInfo(MEMSTHeapInfoRec * Info);
```

These declarations are also contained in header file STHEAP.H.

Requirements: none.

Driver MEMSTCH

This driver combines MEMCHEAP and MEMSTH. All allocation requests are first routed to MEMSTH. If the allocation fails (e.g., because all available memory is already allocated), malloc is called.

This driver allows allocations (e.g., for Semaphores, critical sections, etc.) in the start-up phase of a program, but the available memory is not limited to the static heap. For critical sections required by multithreaded run-time systems, the default size of the static heap is sufficient.

Requirements: C/C++ run-time system.

Driver MEMW32

This driver uses Win32 heaps to satisfy allocation requests. Under RTTarget-32, the Win32 heaps are protected with a critical section and are also used by RTTHeap, RTTarget-32's alternate heap manager. This is the default memory driver.

Requirements: Emulation of Win32 heap functions.

Source Code Positions

The Debug Version of RTKernel-32 can display source code position information for each task in its function RTKTaskInfo or in its fatal error handler. The optional source code position driver is used to retrieve this information.

The source code position driver's interface is given in include file RTKSRC.H.

Driver SRCNULL

This is a dummy source code position driver which does not return any source code information to RTKernel-32. Use this driver if you don't need source code position information or if no suitable symbol table information is available.

Driver SRCTDS

The TDS (Turbo Debugger Symbols) driver can extract source code positions from symbol tables generated by Borland compilers or RTTarget-32's RTLoc. Function RTKLoadSymbols will accept .EXE files and .TDS files to read the required information.

For Microsoft and Watcom compilers, the driver must know the starting address of the code segment. This value must be assigned to global variable TDSCodeSegmentBase as in the following example:

```
extern char * TDSCodeSegmentBase;

#ifdef _MSC_VER
TDSCodeSegmentBase = (char*) Clock;
#endif

#ifdef __WATCOMC__
TDSCodeSegmentBase = (char*) Clock - 16;
#endif
```

In this example (taken from demo program RTKDEMO.C), it is assumed that Clock is the first function linked and thus marks the start of the program's code section.

Requirements: RTLoc option -g+, C/C++ run-time system, ANSI C file I/O.

CPU

RTKernel-32 requires a few low-level CPU-specific operations, which either cannot be implemented in C/C++ or are unique to the CPU. Examples are access to CPU registers, low-level interrupt handler stubs, etc.

Currently, RTKernel-32 is delivered with two such drivers for any CPU compatible with Intel 386 32-bit protected mode. The source code of the CPU drivers is only available with the source code of RTKernel-32.

Driver CPU386F

CPU386F is the flat address CPU driver. It assumes the program is running in a completely flat address environment. In particular, interrupt handlers do not save, restore, or reload any segment registers. While this driver delivers very good performance and low interrupt latencies, it can only be used if no code interruptible by hardware interrupts modifies any of the segment registers DS, ES, FS, or SS. This is the case for compiler-generated code and all assembler modules delivered with On Time RTOS-32. If this rule is violated by any application code, sporadic and difficult to debug program crashes can be expected.

Requirements: Intel 386 or compatible CPU (32-bit protected mode only), no segment register changes.

Driver CPU386

This driver does not enforce the flat address model. Registers DS, ES, and SS are saved, reloaded, and restored by the low-level interrupt handlers. Performance is slightly inferior to CPU386F, but segment register changing software is supported.

Overview of all Drivers

The following tables list all drivers delivered with RTKernel-32 by their respective names and hard- and software requirements:

Type	Name	Requirements	Comments
System	SYSSTD	C run-time system file I/O for stdout	No TLS support.
	SYSRT32	RTTarget-32 1.1 or higher C run-time system	TLS support, limits number of tasks to 8192.
Interrupt	IRQRT32	RTTarget-32 1.1 8059A at 20h	
Clock	CLKPC	8253 at 40h timer generates IRQ 0	Incompatible with drivers CLKHRTPC and HRTPC.
	CLKHRTPC	8253 at 40h 8059A at 20h timer generates IRQ 0	Combined Clock and high-resolution timer driver. Incompatible with all other high res. timer drivers.
Highres. Timer	HRTNULL	none	Does not implement any timer services.
	HRTPC	8253 at 40h 8059A at 20h timer generates IRQ0	Incompatible with drivers CLKHRTPC and CLKPC.
	CLKHRTPC	8253 at 40h 8059A at 20h timer generates IRQ0	Combined Clock and high-resolution timer driver. Incompatible with all other clock drivers.
	HRTPENT	Pentium or higher CPU	Must be calibrated at run-time. Incompatible with drivers CLKHRTPC.
	HRTSC520	AMD Élan SC520 CPU	Incompatible with drivers CLKHRTPC.
Floating Point	FLTNULL	No floating point or reentrant emulator	Does not swap floating point context.
	FLT387	387 compatible floating point hardware or emulator	
	FLTPII	Pentium II or high Intel CPU	Saves/restores 387 and SSE registers.
	FLTEMUMT	387 compatible floating point hardware or reentrant emulator RTTarget-32 1.1	
Memory	MEMCHEAP	C run-time system	Does not support RTKernel-32 calls before function main has been called.
	MEMSTH	none	Supports additional function STHeapInfo.
	MEMW32	Win32 Emulation for Win32 heaps	Shares heap space with RTTHEAP.
	MEMSTCH	C run-time system	
Source Code	SRCNULL	none	Does not implement source code position information.
	SRCTDS	file I/O, TDS file	
CPU	CPU386F	No segment register changes	
	CPU386		Saves/restores segment registers in low-level interrupt handlers

Preconfigured Driver Library DRVRT32.LIB

To successfully link an RTKernel-32 application, the main RTKernel-32 library RTK32S.LIB or RTK32.LIB and a complete set of drivers is required. Since the number of libraries can become quite large, RTKernel-32 comes with the preconfigured library DRVRT32.LIB, which incorporates the drivers you will most likely use with RTKernel-32 and RTTarget-32. It contains the following drivers:

- SYSRT32
- IRQRT32
- CLKHRTPC
- FLTNULL
- MEMW32
- SRCTDS
- CPU386F

Thus, an RTKernel-32 program to run under RTTarget-32 needs to link only DRVRT32.LIB and RTK32.LIB and the RTTarget-32 library RTT32.LIB. You can override the preconfigured drivers by linking the required driver library before DRVRT32.LIB. For example, if you want to use floating point driver FLTEMUMT instead of FLTNULL, specify library FLTEMUMT.LIB before DRVRT32.LIB on the linker command line.

Chapter 6

Demo Programs

All RTKernel-32 demo programs are configured the same way as the RTTarget-32 demo. Please refer to Part I of this manual for instruction on how to run the demos.

Program Threads

Program Threads is a very simple program which can be used to verify that RTKernel-32 is correctly configured and installed. The main thread creates a semaphore, a mailbox, and a second thread. The main thread then sends a signal, a mailbox message, and a direct message to the new thread. This program can be regarded as the minimal RTKernel-32 framework for applications using RTKernel-32's native API.

Program RTKDemo

Program RTKDemo is an interactive program showing some of the capabilities of RTKernel-32's Debug Version. RTKDemo creates several tasks and then waits for user input. Depending on the command entered (press F10 for a list of all available commands), RTKDemo can display the result of the following functions on the screen:

- RTKTaskInfo
- RTKMailboxInfo
- RTKSemaInfo
- RTKIRQInfo
- RTKDumpTrace

One of the threads created by RTKDemo is a high-priority cyclic task running exactly once every 10th of a second.

Program RTKInt

Program RTKInt shows how a hardware interrupt handler can be installed under RTKernel-32. A simple serial port receive handler is used. The handler reads the UART's receive registers on every interrupt and places the received byte in the receive buffer mailbox. A receive thread then retrieves the data and can process it.

Program COMDemo

COMDemo demonstrates serial communications using module RTCom. It can send and receive strings using a serial port. To use it, you must have another PC running a terminal program or a terminal attached to COM2 of the target computer. COMDemo will send and receive data at 9600 baud, 8 data bits, 1 stop bit, and no parity.

Program RTBench

RTBench is a benchmark program that measures the run-time performance of various RTKernel-32 operations. RTBench yields a number of run-times in microseconds. It shows that the overhead incurred by RTKernel-32 is very low and that the scheduler's operation is highly efficient.

Program RTBenchP

RTBenchP uses the same source code as RTBench, but a different set of drivers is linked, optimized for Pentium class CPUs. This program does not use the 387 FPU emulator but links hardware floating point support. In addition, the Pentium high resolution time driver is used.

Program RTBenchA

RTBenchA uses the same source code as RTBench, but a different set of drivers is linked, optimized for the AMD Élan SC520 CPU. This program does not use the 387 FPU emulator but links hardware floating point support. In addition, the SC520 high resolution time driver is used.

Program W32Bench

W32Bench is a small benchmark to compare the speed of multithreading under Windows 95/98/NT/2000 and RTKernel-32. Unlike most other demo programs, W32Bench is linked with the Standard Version (as opposed to the Debug Version) of RTKernel-32. Instead of linking the On Time RTOS-32 libraries directly, it uses a custom system DLL, and uses only Win32 API function. In this way the program can be executed under Win32 and under RTTarget-32.

W32Bench executes and measures the time for 200,000 task switches via semaphores using the Win32 API.

Chapter 7

Advanced Topics

This chapter introduces some advanced topics and discusses techniques of solving typical real-time programming tasks.

RTKernel-32's Debug Version

During the program development phase, the Debug Version of RTKernel-32 should be used by all means. It contains additional code to recognize a number of error conditions. If an error occurs, the program is aborted with an error message. Moreover, the Debug Version can determine at which source code position a task is suspended. Also, it can calculate the CPU time usage of tasks and interrupts. To locate persistent bugs, it features the real-time Kernel Tracer, which can also be used by the application.

Naturally, the Debug Version will not perform as fast as RTKernel-32's Standard Version. The only reason not to use the Debug Version during the testing phase could be the higher interrupt latency. To change the version used, a program need not be recompiled; relinking is sufficient.

In the Debug Version, function `RTKernelInit` will display the message

```
RTKernel-32 Debug Version
```

to inform the programmer that the Debug Version is being used.

The Debug Version's exit function will display the message

```
RTKernel-32 exit function
```

If this message is not displayed, execution of the exit function has been aborted, RTKernel-32 was unable to restore the interrupt vectors, and the system may be in an unstable state.

During execution of an RTKernel-32 application, the Debug Version principally checks the following conditions:

- It is checked whether every semaphore and mailbox used has been initialized and has not been corrupted.
- It is checked whether every task handle passed to RTKernel-32 references an existing task (except for function `RTKGetTaskState`).
- Every time RTKernel-32 is called, the stack of the active task or interrupt handler is checked for overflow. This behavior can be suppressed by resetting flag `RF_STACKCHECKS` in `RTKConfig.Flags`.
- In a blocking task switch, it is checked whether the task switch was triggered by an interrupt handler.
- Internal RTKernel-32 data structures are checked for consistency.

All error messages are listed in Appendix D.

Reentrance of the C/C++ Run-Time Systems

Parts of the C/C++ run-time systems may be non-reentrant. If preemptive multitasking is used, potential reentrance problems must be considered by the application.

The most critical non-reentrant part of the run-time system is the heap. With very few exceptions, all RTKernel-32 programs will need the heap. Since the heap can be regarded as a global resource, it can only be used simultaneously by several tasks if it is protected by a semaphore or critical section.

Other non-reentrant code parts may be the opening and closing of files and the use of variable `errno` or other global variables. If you do not intend to open and close files simultaneously in several tasks with preemptions enabled, you do not need to consider reentrance issues here.

Reentrant operations can only be used simultaneously by several tasks if different data (parameters) is processed. For example, all file read and write operations are reentrant, as long as no two tasks use the same file and don't attempt to write into the same buffer.

Non-reentrant parts of the run-time system can safely be used by **one** task in the program. If several tasks access such resources, a mutual exclusion algorithm (see Chapter 7, *Mutual Exclusion*) should be employed to make sure that no two tasks simultaneously perform the operation concerned. It must be considered that not the code is non-reentrant, but rather, the global data. For example, in general, graphics modules such as MetaWINDOW are completely non-reentrant, because virtually all routines use a global cursor position (among many other global parameters, e.g., color, line style, etc.). Simultaneous use of graphics routines by several tasks would not crash the program; however, the results wouldn't look as expected. Therefore, it is necessary to implement a mutual exclusion protocol for all routines of such a module.

RTKernel-32 offers different mechanisms for dealing with reentrance problems:

- Support for the multithreaded libraries
- Replacements for non-reentrant parts of the run-time system
- Automatic Library Protection

Which mechanism is best suited will depend on the particular application. A combination of the different methods is also possible. For best compatibility, the use of the multithread run-time libraries is recommended.

Multithreaded Libraries

Most 32-bit C/C++ compilers come with run-time libraries which address all reentrance issues. All non-reentrant parts are protected using Win32 Critical Sections or similar mechanisms. Even accesses to global variables such as `errno` are synchronized by duplicating `errno` for each thread.

The only disadvantage of the multithreaded libraries is their larger size and a small performance penalty, even for parts where no reentrance problems are to be expected. In addition, all threads of the application which might use the run-time system must be created with `RTKRTLCreateThread` or the functions of the run-time system provided for this purpose (e.g., `_beginthread`). It should also be noted that such threads might not support being terminated from a different thread or that such termination would cause a memory leak. Please consult the documentation of the multithread support of your run-time system for details.

Replacements for Non-Reentrant Parts of the Run-Time System

RTTarget-32 is delivered with a replacement for the C/C++ heap manager, which is protected using a Win32 Critical Section. If you link `RTTHEAP.LIB`, all reentrance problems of the heap are solved.

Automatic Library Protection

RTKernel-32 offers *Automatic Library Protection* to protect non-reentrant libraries in the Intel/Microsoft Relocatable Object File Format (OBJ)¹⁸. Using this feature, non-reentrant functions of arbitrary libraries can be protected from simultaneous calls from several tasks. Source code modifications are never required, neither in the libraries nor in programs using the libraries. Library Protection is fully implemented at the level of OBJ and LIB files. Library Protection can be used for the compilers' standard run-time libraries, and also for any third-party libraries.

The mechanism is quite simple: each function that must be protected is renamed in the library. A new function (the *Protector*) is created with the original name of the function to be protected. This new function occupies a resource semaphore, calls the original function, and subsequently releases the semaphore again. Thus, an application that uses the respective function automatically calls the Protector.

¹⁸ RTKernel-32's library protection currently does not support COFF libraries used by Microsoft Visual C++.

The utility program LIBPROT is responsible for renaming functions and creating Protectors. A *Library Protection Definition File* specifies which functions should be protected by which resource semaphores. These definition files are line-oriented text files. Each line contains the name of a semaphore to be used for protection, followed by one or more function names that are protected by the respective semaphore.

Consider the following example that could be used to protect the functions malloc and free:

```
Heap _malloc _free
```

This creates a resource semaphore named "Heap". Each call to malloc and free will be protected by this semaphore. Assuming that the definition file is named TEST.LP, and the library containing the functions malloc and free is named TEST.LIB in directory C:\COMPILER\LIB, Library Protection can be installed in RTKernel-32's LIB directory using the following command line:

```
LIBPROT C:\COMPILER\LIB\TEST.LIB TEST.LP
```

LIBPROT creates a copy of library TEST.LIB in RTKernel-32's LIB directory. The copy is searched for functions named _malloc and _free. If found, they are renamed to __clib__malloc and __clib__free, respectively. Also, two Protectors named _malloc and _free are created. Protector _malloc performs the following (simplified) pseudo-code:

```
#include <rtk32.h>

extern RTKSemaphore _lp_Heap;

ResultType malloc(Parameters)
{
    if (RTKProtectLibrary)
    {
        ResultType Result;
        RTKWait(_lp_Heap);
        Result = __clib__malloc(Parameters);
        RTKSignal(_lp_Heap);
        return Result;
    }
    else
        return __clib__malloc(Parameters);
}
```

Semaphore _lp_Heap is created in a separate module and is named "Heap".

Library Protection must be activated using variable RTKProtectLibrary. RTKernel-32 sets RTKProtectLibrary to TRUE in function RTKPreemptionsON. The application may also enquire or modify RTKProtectLibrary directly.

The semaphores used for Library Protection are created dynamically whenever the first call to a protected function occurs. The number of currently used semaphores is stored in global variable RTKLPSemas. Using this variable, an application can determine whether Library Protection has been linked successfully.

Automatic Library Protection is well-suited in situations where a set of functions manipulate a global resource which is always in a consistent state between function calls. It is **not** suited for systems requiring transactions consisting of several function calls. For example, a graphics system that maintains global color and cursor position may require three calls to draw a line with a particular color at a particular position (SetColor, SetCursor, DrawLine). Since none of the three calls may be interrupted by other graphics calls, Automatic Library Protection cannot help here. In this case, the application needs to implement mutual exclusion using a semaphore.

Automatic Library Protection also cannot solve access problems to global variables such as errno, since it can't prevent errno from changing its value after a run-time system call has completed.

RTKernel-32 comes with Library Protection Definition Files BC32.LP for Borland C/C++ and WATCOM.LP for Watcom C/C++. These files should be used for protecting the standard run-time libraries if you do not intend to use the multithread libraries supplied with the compilers. The files are extensively commented and contain a complete description of the definition file structure. Also, for each function, they contain a description of the reasons why the respective function is non-reentrant. It is recommended to study these files carefully to facilitate planning the functions' usage.

Please note that BC32.LP and WATCOM.LP only cover ANSI C compatible parts of the run-time libraries. Compiler-specific parts (such as C++ class libraries) are not covered and must be protected manually or by adding your own commands to the Library Protection Definition Files.

It should be noted that functions inherently non-reentrant due to their specifications are also not considered in the LP files. This comprises functions returning pointers to static variables (e.g., `asctime`).

On Time does not guarantee the LP files delivered with RTKernel-32 to be correct or complete. RTKernel-32 users are encouraged to point out any errors or omissions.

Automatic Library Protection can be installed for Borland C/C++ in the Libbc directory with command:

```
LIBPROT.EXE C:\BORLANDC\LIB\CW32.LIB BC32.LP
```

Please supply the appropriate path of CW32.LIB. It must be ensured the linker will link file LIBBC\CW32.LIB instead of C:\BORLANDC\LIB\CW32.LIB.

Automatic Library Protection can be installed for Watcom C/C++ in the Libwat directory contains with command:

```
LIBPROT.EXE C:\WATCOM\LIB386\NT\CLIB3R.LIB WATCOM.LP
```

Please supply the actual path of CLIB3R.LIB. It must be ensured the linker will link file LIBWAT\CLIB3R.LIB instead of C:\WATCOM\LIB386\NT\CLIB3R.LIB.

How to Create Threads

RTKernel-32 programs can choose among several different methods to create threads:

Level	Function	Comment
1	RTKCreateThread	RTKernel-32's native, low-level thread creation function.
2	CreateThread	Win32 thread creation function; calls RTKCreateThread internally.
3	_beginthreadNT, _beginthreadex, etc.	C/C++ or Pascal run-time system library functions. These functions call Win32 function CreateThread internally.
4	RTKRTLCreateThread	RTKernel-32's thread creation function with run-time system support. This function calls a run-time system thread creation function internally.

At each level, the preceding level is called.

Most applications should use level 4 or 3. Level 4 has the advantage that all RTKernel-32 parameters such as the thread's priority and flags are available. On the other hand, if portability between RTKernel-32 and Win32 is desired, level 3 might be better suited. Both level 3 and 4 use the multithread run-time system libraries, enabling such threads to call non-reentrant run-time system functions.

Threads created with level 1 and 2 cannot use C++ exceptions, and they must take their own precautions when non-reentrant run-time system functions are called (e.g., with Automatic Library Protection). Level 1 is well suited for simple tasks not requiring any non-reentrant run-time system services or exceptions. CreateThread does not support all RTKernel-32 thread parameters, but is Win32 compatible.

Interrupt Handling

One of the central tasks of real-time software is the processing of interrupts. As soon as several tasks run in a program, it is virtually impossible to achieve good response times by *polling* (continuous enquiry of an event). Continuous polling would prevent tasks with lower priorities from running and thus waste precious CPU time.

Therefore, it should always be attempted to replace polling by interrupts. This leads to the best response times and optimal use of the hardware available. RTKernel-32 uses the timer interrupt to activate tasks waiting for a certain point in time. Modules RTKeybrd and RTCom provide interrupt support for the keyboard and the serial ports. This chapter discusses how to implement a handler for any interrupt source.

An interrupt handler may be thought of as a task running with a priority higher than all other tasks. However, there are some important considerations to keep in mind.

Interrupt handlers usually have little stack space. Therefore, interrupt handlers should be very economical in their stack usage (e.g., refrain from using functions like `sprintf` by all means).

While an interrupt handler is active, no other interrupts with lower priorities can be processed. Therefore, it is important to minimize the execution times of interrupt handlers, because otherwise the interrupt response time for other interrupts might suffer. The handler should avoid any processing not immediately required and delegate it to a task.

Interrupt handlers under RTKernel-32 are never directly addressed by the hardware; instead, they are called by the low-level handlers of the kernel (see Chapter 2, *Interrupt Handling*). This incurs an overhead of only a few microseconds. While the handler is being executed, the scheduler is disabled; thus, the handler need not consider being disrupted by a task switch (it can, however, be interrupted by interrupts of higher priority). Since the scheduler is disabled, interrupt handlers must not force blocking task switches.

Writing interrupt handlers is not particularly difficult as long as the rules given in the following section are obeyed. Demo program `RTKInt` shows how hardware interrupts are handled with RTKernel-32. In addition, the respective handlers of supplemental modules `RTKeybrd` and `RTCom` may serve as further examples.

Structure of an Interrupt Handler

An interrupt handler should have the following structure:

- It should be declared as a void C function without parameters.
- While accepting the interrupt, the processor has masked all interrupts. To reenale interrupts of higher priorities, the handler should call `RTKEnableInterrupts()` as its first statement.
- Subsequently, the hardware that has generated the interrupt should be serviced, if required. The handler can communicate with other tasks using mechanisms that cannot lead to a blocking task switch.
- Thereafter, interrupts should be disabled again and an End-Of-Interrupt command should be sent to the interrupt controller. This is accomplished using statement:

```
RTKIRQEnd( IRQ );
```

An interrupt handler may:

- declare and use local variables,
- call other functions,
- call functions `RTKSignal` or `RTKWaitCond` to activate other tasks,
- call the **conditional** mailbox and message passing operations (`RTKPutCond`, `RTKGetCond`, `RTKSendCond`, `RTKReceiveCond`) to exchange data with tasks.

An interrupt handler **must not**:

- use the coprocessor or emulator without saving/restoring its state,
- use more than 512 bytes of stack (the less, the better),
- cause a blocking task switch.

The reason for the last restriction is that an interrupt handler runs in the context of the task it has interrupted. It follows that an interrupt handler that blocks itself (e.g., by calling `RTKDelay` or `RTKWait` at a semaphore with no events) actually blocks the task it has interrupted, even though this task could very well continue running. This results in a task with a low priority running, although another task with a higher priority is ready; this situation will sooner or later crash the scheduler.

For this reason, only the functions `RTKSignal` and `RTKWaitCond` (for semaphores), `RTKPutCond` and `RTKGetCond` (for mailboxes), and `RTKSendCond` and `RTKReceiveCond` (for message passing) can be used by interrupt handlers. These operations can lead to an activating but not to a blocking task switch.

RTKernel-32's Win32 emulation routines can also be used by interrupt handlers (although this is not possible under Win32 itself). The following functions may be used: `GetTickCount`, `SetEvent`, `ResetEvent`, `PulseEvent`, `ReleaseSemaphore`. It should be noted that a call to any of these functions can change the value returned by `GetLastError()` of the interrupted task. To avoid interfering with error handling, the error code should be saved and restored in the interrupt handler.

Avoid Polling

Polling is often used when a task has to wait for something. For this purpose, some condition is continually tested in a loop until the expected event is recognized. In sequential programs, this is the only way to wait since the processor can't simply be paused. On the other hand, in a multitasking environment, this is a quite uncooperative behavior of a task, because precious CPU time that might otherwise be used more efficiently by other tasks is wasted. While a task has nothing to do, it should not be in the state Ready or Current.

If the polled event is generated by another task, polling can be replaced by inter-task communications in all cases. The waiting task could, for example, block itself using `RTKReceive`. It can be reactivated when another task has recognized the event and transmits a corresponding message to the waiting task using `RTKSend`.

For hardware events, interrupts offer the desired functionality. A task can be blocked in order to be reactivated by an interrupt as soon as the expected event occurs. All of the waiting time is available to other tasks in this manner and response time is optimal. RTKernel-32 provides appropriate interrupt handlers (Timer, keyboard in `RTKeybrd`, COMx in `RTCom`) for frequently used devices. For other hardware you may be using, you should implement similar drivers if the hardware supports interrupts. This results in the best performance of your system by far.

Unfortunately, many I/O boards do not support interrupts (e.g., some digital I/O boards). In this case, polling cannot be avoided. However, polling should be as cooperative as possible, e.g., by providing CPU time to other tasks in each polling cycle using `RTKDelay`.

Preemptive or Cooperative Multitasking?

RTKernel-32 allows you to choose between preemptive and cooperative multitasking. The proper choice strongly depends, of course, on the requirements of your application. To make the right decision, it is necessary to exactly understand the differences between preemptive and cooperative multitasking (see Chapter 1, *Cooperative and Preemptive Multitasking*).

Advantages of Preemptive Scheduling

The main advantage of preemptive scheduling is real-time response on the task level. The task response time - i.e., the time required to activate a task waiting for an interrupt - largely depends only on the interrupt latency (the time span during which no other interrupts can be accepted). In cooperative scheduling, the task response time is the longest time span that can elapse between two calls to the kernel. Unfortunately, an upper limit for this time span cannot be defined in many cases. It is the responsibility of the programmer to ensure that the time spans between calls to the kernel or the scheduler are sufficiently small.

Advantages of Cooperative Scheduling

In cooperative scheduling, substantially fewer reentrance problems are encountered than in preemptive scheduling, because tasks cannot be interrupted arbitrarily by other tasks, but only at positions permitted by the programmer (i.e., in kernel calls). There are, for example, no reentrance problems with the heap manager, because there are no calls to the kernel from within it.

It should be noted that even though real-time response is impeded on the task level, it is fully preserved on the interrupt handler level. Interrupt handlers can continue to use semaphores and mailboxes in cooperative scheduling. Therefore, interrupt-driven modules like `RTCom` can run independently of the scheduling algorithm currently active. Also, these modules' performance is identical in both cases.

While RTKernel-32's Idle Task is running (or a task with a similar structure, e.g., the CPU Monitor Counter task), tasks' response times to interrupts are practically identical, with or without preemptive scheduling.

For programs that don't require a response time on the order of micro- or milliseconds at the task level, cooperative scheduling is recommended.

Waiting for Several Events

Even if a task must react to different types of events, polling should be avoided, if possible. The best strategy for this case is to split the task so that each task only has to wait for one event.

Alternatively, events can be bundled. Given that a task must react to two different data packets passed to it through mailboxes, one - not particularly elegant - solution could be:

```
typedef struct { int    a, b, c; } Rec1;
typedef struct { float a, b, c; } Rec2;

RTKMailbox Box1, Box2;

void RTKAPI PollTask(void * p)
{
    Rec1 S1;
    Rec2 S2;

    while (1)
    {
        if (RTKGetCond(Box1, &S1))
            /* process S1 */ ;
        if (RTKGetCond(Box2, &S2))
            /* process S2 */ ;
    }
}
```

Even if no data is ready for processing, this task would "gobble up" all CPU time.

A better solution would be to combine types Rec1 and Rec2, and use only one mailbox:

```
typedef enum { Rec_1, Rec_2 } DataKind;

typedef struct {
    DataKind Typ;
    union {
        Rec1 S1;
        Rec2 S2;
    } Data;
} MultiData;

RTKMailbox BigBox;

void RTKAPI MBWaitTask(void * p)
{
    MultiData S;

    while (1)
    {
        RTKGet(BigBox, &S);
        switch (S.Typ) {
            case Rec_1: /* process S.Data.S1 */ break;
            case Rec_2: /* process S.Data.S2 */ break;
        }
    }
}
```

In this example, no CPU time is wasted.

Avoid Large Data Types for Mailboxes and Message Passing

The length of the data packets for communication via mailboxes or message passing is not limited. However, large data packets can result in performance problems, since the data is copied in every inter-task communication. Instead of channelling the data itself through a mailbox, logical packet numbers or pointers to the data can be used.

In the following example, ten data buffers are allocated. Unused buffers are contained in mailbox Pool. ProducerTask retrieves buffers from Pool as required and passes them to ConsumerTask using mailbox NewData. Since only pointers - not the buffers themselves - are copied to the mailbox, performance is independent of the data packet size.

```
#define MaxBuffer 10
typedef struct {
    char x[5000];
    /* more fields... */
} BufferType;
RTKMailbox Pool, NewData;
void RTKAPI ProducerTask(void*p)
{
    BufferType * Ptr;
    while (1)
    {
        RTKGet(Pool, &Ptr);    /* get new buffer */
        /* Ptr-> is filled with data */
        RTKPut(NewData, &Ptr); /* pass on buffer */
    }
}
void RTKAPI ConsumerTask(void * p)
{
    BufferType * Ptr;
    while (1)
    {
        RTKGet(NewData, &Ptr); /* wait for new data */
        /* process Ptr-> */
        RTKPut(Pool, &Ptr);    /* release buffer */
    }
}
void main(void)
{
    int i;
    BufferType * Ptr;
    RTKKernelInit(7);
    Pool = RTKCreateMailbox(sizeof(BufferType*), MaxBuffer, "Pool");
    NewData = RTKCreateMailbox(sizeof(BufferType*), MaxBuffer, "New");
    /* allocate and make available all buffers */
    for (i = 1; i <= MaxBuffer; i++)
    {
        Ptr = RTKAlloc(sizeof(BufferType), "a buffer");
        RTKPut(Pool, &Ptr);
    }
    /* ... */
}
```

The same technique can be used for message passing.

Mutual Exclusion

Assume that two or more tasks have to manipulate the same global data object. In order to assure consistency of the data, care has to be taken that only one task accesses the data at any one time.

In this case, a semaphore can be utilized for coordination. The semaphore is initialized with one event. This event thereafter represents the permission to access the data. Before a task modifies the data, it calls `RTKWait` with the corresponding semaphore in order to get permission to access the data. When a task is finished accessing the data, it must call `RTKSignal` to store the access permission in the semaphore and thus enable other tasks to access the data. If all tasks adhere to this protocol, no more than one task will ever change the data at any one time. This algorithm can be used with counting, binary, resource, and mutex semaphores. Resource and mutex semaphores are especially suited for this purpose, because of priority inheritance, safe Suspend/Terminate operations, and - for resource semaphores - the Debug Version's error checking.

Avoid Time Slicing

Time slicing is a technique that originated in time sharing systems. In Chapter 1, the differences between time sharing systems and real-time systems were discussed briefly. Actually, time slicing only makes sense if your tasks don't have to meet real-time requirements, never have to wait for something, and can run largely independently of each other. In this case, however, you could let the tasks run sequentially and thus avoid the overhead incurred by `RTKernel-32`.

The fairness strived for by time sharing systems cannot be attained by `RTKernel-32` anyhow. The simplest definition of fairness would be that CPU time must be shared evenly among all tasks. (Systems like Unix use a much more elaborate definition of fairness.) The scheduling rules of `RTKernel-32` can only achieve this if all tasks of a program have the same priority, can run completely independently of each other, and never block themselves. If these conditions are not met, time slicing will only guarantee that each task receives CPU time in the long run - when and how much is, in general, indeterminate.

In many applications, `RTKDelay(0)` can be a feasible alternative to time slicing. Using `RTKDelay(0)`, round-robin scheduling (tasks are activated in turn) can be implemented easily. Assume the following problem shall be solved:

Two tasks must recognize an event using polling. The polling cycle must be as short as possible. A simple solution would be to switch among tasks using time slicing. In this case, however, the polling cycle would be 55 milliseconds in the worst case. The timer interrupt rate could be increased, thus achieving a polling cycle of a few milliseconds. However, it would be much more elegant to call `RTKDelay(0)` in each polling cycle. Each task involved could then process a polling cycle without being disrupted and afterwards allow the next task to run. It would not even be required to activate preemptions or time slicing. Assuming the polling itself requires only 10 microseconds, about 20,000 cycles per second could be achieved on an 80386/20. By the way, this algorithm would also ensure *fairness*, even if more than two tasks participated.

Cyclic Tasks (Timer)

Tasks that must run in a fixed time frame are frequently encountered in real-time applications. The structure of such a task could be as follows:

```
void RTKAPI CyclicTask(void * p)
{
    #define Cycle 5
    RTKTime NextActivation;

    NextActivation = RTKGetTime();
    while (1)
    {
        NextActivation += Cycle;
        RTKDelayUntil(NextActivation);
        /* the task's job is done here */
    }
}
```

This task would run exactly once every 5 timer ticks, provided the task's job takes no longer than 5 timer ticks. The actual cycle time can be adjusted using constant `Cycle` and the timer interrupt interval.

If the cycle time of a task is not an exact multiple of the timer interrupt interval, the following task can be used:

```
#include <timer.h>

void CyclicTask(void)
{
    #define Cycle 0.7    /* seconds */
    TISeconds NextActivation;

    NextActivation = TITicksToSeconds(RTKGetTime());
    while (True)
    {
        NextActivation += Cycle;
        RTKDelayUntil(TISecondsToTicks(NextActivation));
        /* the task's job is done here */
    }
}
```

Since `RTKernel-32` can only activate tasks waiting in an `RTKDelay` or `RTKDelayUntil` through a timer interrupt, these tasks would, on the average, conform to their cycle exactly. The start of a cycle can deviate from the desired point in time up to one tick. This round-off error arises from the conversion of a float point number to an integer in function `TISecondsToTicks` of module `Timer`. However, the error does not accumulate; it is the same in each cycle. It must be noted that rounding errors can occur when floats are used to store large numbers representing `RTKernel-32` times. Unlike the first example, an overflow of `RTKernel-32`'s clock must also be considered (see Chapter 2, *Time*).

Priorities

Priorities have the sole purpose of guaranteeing good response times of time-critical tasks, even when other, less critical tasks run or could run.

Priorities should **not** be used to synchronize tasks. It is never certain that no other task runs while the task with the highest priority appears to be ready for running. If, for example, this task performs a heap manager call such as `malloc` or `free`, it could be blocked by the kernel if some other task has not yet completed its call.

Synchronization among tasks can only be achieved using explicit inter-task communication.

`RTKernel-32` exhibits the best performance when the smallest possible range of the 64 priorities is used. Since the `Idle Task` has a priority of 0, few and small priorities should be used (see Appendix A). In general, 3 to 5 priorities will (and should) suffice.

Starting Objects' Methods as Tasks

The code of a task must always be a function with zero or one parameters. This condition cannot be met by methods of objects, because they always have the invisible parameter "this". If a method must be used as a task anyhow, two strategies are possible:

Encapsulate the method by a normal function and start the function as a task:

```
#include <rtk32.h>

class MyObject {
public:
    virtual void Task(void);
};

void RTKAPI RTKernelTask(void * p)
{
    MyObject Object1;
    Object1.Task();
}
```

```
void main(void)
{
    RTKernelInit(2);
    RTKRTLCreateThread(RTKernelTask, 7, 4096, 0, NULL, "Object-Task");
}
```

Alternatively, the task can be sent a pointer to the object to use:

```
#include <rtk32.h>
class MyObject {
public:
    virtual void Task(void);
};
void RTKAPI RTKernelTask(void * Object)
{
    (MyObject*) Object->Task();
    delete (MyObject*) Object;
}
void main(void)
{
    RTKernelInit(2);
    H = RTKRTLCreateThread(RTKernelTask, 7, 4096, 0, new MyObject, "Object-Task");
}
```

The second alternative has the advantage that any number of tasks can be started with a single task function; however, each task uses a different object. These tasks, in turn, can execute different code by using derived objects which redefine the method Task.

Creating and Terminating Tasks

Creating and terminating tasks frequently should be avoided. On the one hand, this can lead to heavy heap fragmentation, possibly making the creation of new tasks impossible. RTKCreateThread and RTKTerminateTask are among RTKernel-32's most involved operations. Their time requirements are not deterministically predictable.

Moreover, blocked tasks **do not** incur a performance penalty. Therefore, it is recommended to create all tasks of an application at program startup. If a task has nothing to do temporarily, it should wait in a blocked task state until it is reactivated.

Chapter 8

Typical Error Sources

Every effort has been made to make using RTKernel-32 as safe and simple as possible. However, RTKernel-32 cannot solve the general problems of parallel programming. Therefore, a few common sources of error shall be pointed out in the following.

Program Termination

The most critical phases of a program are initialization and termination. Please be aware that all tasks can run until RTKernel-32's exit function has been completed.

A program is terminated by a call to function `exit`. A call to `exit` is generated automatically by the compiler at the end of the main program. Therefore, the whole program terminates when the Main Task reaches its end. Function `exit` then calls all exit functions in reverse order as they have been inserted into the exit chain using `atexit`; meanwhile, task switches can still take place. Consequently, it can happen that a module is used after its exit function has been executed.

Termination can be initiated by any task and is carried out in this task's context (i.e., with its priority and stack).

Caution must also be exercised when one task terminates another. The terminating task generally does not know where the task to be terminated is suspended. Ideally, tasks should terminate themselves when they are in a defined state, e.g., when they reach the end of their task function. In this case, they are terminated automatically by RTKernel-32.

If a program is aborted using function `abort`, the computer must be rebooted to reinitialize the interrupt vectors. RTKernel-32's Debug Version reports the successful completion of its exit function.

Stack Errors

If the stack of a task overflows and the run-time system does not catch the error (because the routine concerned has been compiled with stack checking off), an error usually occurs much later in some other task whose data has been corrupted. Therefore, this type of error is very difficult to locate.

Consequently, it is recommended to dimension the stacks generously. For most - but not all - tasks, 8k bytes are sufficient. Use functions `RTKIRQInfo` and `RTKTaskInfo` with options `LF_FREE_STACK` and `LF_MIN_STACK` extensively to monitor actual stack usage at run time.

Resource Management

Many errors in multitasking applications have to do with the erroneous (or neglected) management of global resources. Global resources are global variables, files, the screen, the keyboard, etc.; all entities that may be used by more than one task but exist in the system only in one instance. Whenever a task accesses something accessible by other tasks, the programmer must consider what happens when another task simultaneously performs an access.

The only safe method to manage data in a controlled manner is the explicit transfer of data from one task (or interrupt handler) to another using mailboxes or message passing, as well as the protection of global resources using semaphores. Resource semaphores are especially suited for this purpose, because they can make sure the synchronization protocol is observed.

Even though RTKernel-32 can sequentialize function calls from different tasks using Automatic Library Protection, the application must make sure that, for example, a file used by more than one task is handled in a controlled manner.

Deadlock

The term deadlock denotes the mutual blocking of several tasks. The mutual exclusion algorithm entails the danger of a deadlock.

Assume that access to a global data object shall be protected using a semaphore. For this purpose, a semaphore is initialized with 1 event. If the mutual exclusion protocol is used, two typical errors can occur:

Case 1:

A task "forgets" the call to RTKSignal after accessing the data. All other tasks that want to access the data are blocked as soon as they try to access the data. The erroneous task will block itself forever, too, if it again tries to get at the data using a call to RTKWait.

Case 2:

The task calls RTKWait twice. Even if two calls to RTKSignal follow, the task has blocked itself and all other tasks requiring access to the data forever.

In both cases, using resource semaphores and RTKernel-32's Debug Version would lead to error messages, since the rules for using resource semaphores are violated.

The only unambiguous deadlock situation RTKernel-32 can recognize arises when a task terminates that is expected to receive data from an RTKSend operation by another task. The sending task will enter the state TS_DEADLOCKED.

Appendix A

Performance and Interrupt Response Times

In the design and implementation of RTKernel-32, great care has been taken to achieve excellent run-time performance. Especially an increase of task switch times with an increasing number of tasks has been avoided.

The time needed for a blocking switch increases proportionally with the difference of the old and the new tasks' priorities. Therefore, using far-apart priorities should be avoided. Since the Idle Task has priority 0, the best performance is achieved using few and small priorities.

For mailbox and message passing operations, it should be considered that the execution time requirements depend on the size of the data objects copied.

Interrupt response time is defined as the time that elapses between the interrupt signal coming in at the interrupt controller and the execution of the first statement in the corresponding interrupt handler. It primarily depends on the maximum time during which interrupts are disabled and the time required to process other interrupts. Since all RTKernel-32 operations that can lead to task switches are performed with interrupts disabled, the interrupt response time directly depends on the run-time requirements of the discrete RTKernel-32 operations. Interrupts also cannot be accepted while an interrupt is being processed and interrupts have not yet been reenabled. This is also true for the timer interrupt handler of RTKernel-32. Even though its run-time requirements are proportional to the number of tasks entering state `TS_READY` during the current interrupt, interrupts are reenabled after each task that has been processed, so that interrupt response time is only marginally impeded by the timer interrupt.

The demo program `RTBench` delivered with RTKernel-32 can be used to measure RTKernel-32's performance.

Appendix B

Task Switches in Cooperative Scheduling

During cooperative scheduling, each task must explicitly allow the kernel to perform a task switch. If a task fails to do so, it runs exclusively and other tasks will not be able to run.

If a task switch has become necessary during cooperative scheduling by a hardware interrupt, the pending task switch is done later during a kernel call of the currently running task. All kernel functions can be classified in three types: calls that will always perform pending task switches, calls that will only perform task switches if they have caused a task state change, and calls that will never lead to a task switch.

The following table lists all kernel functions of the first two types:

Guaranteed Task Switch	Task Switch only after a Task State Change
RTKCreateThread	RTKDelayUntil
RTKDelay	RTKGet
RTKDumpTrace	RTKGetCond
RTKIRQInfo	RTKGetTimed
RTKMailboxInfo	RTKPut
RTKPulse	RTKPutFront
RTKReceive	RTKPutCond
RTKReceiveTimed	RTKPutFrontCond
RTKScheduler	RTKPutTimed
RTKSemaInfo	RTKPutFrontTimed
RTKSend	RTKReceiveCond
RTKSendTimed	RTKResume
RTKSetPriority	RTKSendCond
RTKSignal	RTKSuspend
RTKTaskInfo	RTKWait
RTKTerminateTask	RTKWaitTimed

Other kernel calls do not lead to task switches during cooperative scheduling.

Naturally, all operations containing one of the above calls can lead to a task switch indirectly. In particular, some of these are:

- Screen I/O using module RTTextIO
- COMSendChar, COMSendCharPolled, and COMReceiveCharPolled in module RTCom
- KBKeyPressed and KBGetCh in module RTKeybrd

During cooperative scheduling, you should make sure that each endless loop in your program contains at least one call to the kernel, which allows it to perform pending task switches caused by interrupts.

Appendix C

Writing Custom Kernel Drivers

RTKernel-32 is delivered with drivers suitable for PC compatible or similar systems running under RTTarget-32. Nevertheless, thanks to its driver structure, it is easily possible to run RTKernel-32 under other systems. However, modifications to the existing drivers or new drivers might be required.

The documentation for all drivers in Chapter 5 details each driver's hard- and software requirements. If you cannot meet the requirements for a complete set of drivers, one or more drivers must be replaced. To accomplish this, the recommended method is to copy the source files of the driver most closely matching the desired functionality and then modifying the copy as required. Each driver's interface is documented in the respective driver header file. For example, the system driver's functionality is detailed in file RTKSYS.H.

The following table summarizes the source files required to build the drivers delivered with RTKernel-32. All driver source files are located in directory Driver\Rtk32:

Type	Name	Source Files	Comments
System	SYSSTD	SYSSTD.C HALT386.ASM	No TLS support Relies only on C/C++ RTL
	SYSRT32	SYSRTT32.C HALT386.ASM	Requires RTTarget-32. TLS and Win32 support limits number of tasks to 8192.
Interrupt	IRQRT32	IRQCALL.OBJ IRQRTT.C I8259.ASM DUMISRS.ASM IFLAG.ASM	Support for Preemptions. Requires RTTarget-32 and 8259A interrupt controller(s). Supports 16 IRQs.
Clock	CLKPC	CLKHRTPC.ASM	Requires 8253 timer. Incompatible with drivers CLKHRTPC and HRTPC.
	CLKHRTPC	CLKHRTPC.ASM	Requires 8253 timer. Combined Clock and high-resolution timer driver. Incompatible with drivers CLKPC and HRTPC.
Highres. Timer	HRTNULL	HRTNULL.ASM	Does not implement any timer services
	HRTPC	CLKHRTPC.ASM	Requires 8253 timer. Incompatible with drivers CLKHRTPC and CLKPC.
	CLKHRTPC	CLKHRTPC.ASM	Requires 8253 timer. Combined Clock and high-resolution timer driver. Incompatible with drivers CLKPC and HRTPC.
	HRTPENT	HRTPENT.ASM	Requires Pentium or higher. Must be calibrated at run-time.
	HRTSC520	HRTSC20.ASM	Requires AMD SC520 CPU.
Floating Point	FLTNULL	FLTNULL.ASM	Does not swap floating point context.
	FLT387	FLT387.ASM	Requires 387 or 100% compatible emulator.
	FLTPII	FLTPII.ASM	Requires Pentium II or higher CPU.
	FLTEMUMT	FLTEMUMT.ASM	Requires 387 or reentrant emulator.

Memory	MEMCHEAP	CHEAP.C	Uses malloc/free. Does not support RTKernel-32 calls before function main has been called.
	MEMSTH	STHEAP.C STHDATA.C STHCALL.C	Static heap in data segment. Supports additional function MEMSTHeapInfo
	MEMW32	MEMW32.C	Uses Win32 Heaps.
	MEMSTCH	STHEAP.C STHDATA.C STCHEAP.C	Combined MEMCHEAP and MEMSTH.
Source Code	SRCNULL	SRCNULL.C	Does not implement source code position information.
	SRCTDS	READTDS.C TDSGETP.C	Requires read-only access to a file.

Subsequently, the new driver should be linked after library RTK32[S].LIB and before any driver libraries delivered with RTKernel-32.

Appendix D

Error and Information Messages

RTKernel-32 error messages have the following format:

```
RTKernel-32 error : Message
Current task      : CurrentTaskName
Int handler IRQ   : IRQ
Error location    : CodePosition (hex or source position)
Error in task     : ErrorTaskName
Semaphore        : SemaphoreName
```

Only the lines relevant to the respective error are displayed. *Message* is a meaningful text issued by RTKernel-32. If the error has occurred in a task, its name is given by *CurrentTaskName*. If the error has occurred in an interrupt handler, *IRQ* is the IRQ number. If RTKernel-32 can determine the code position where the error has occurred, this is displayed as *CodePosition*. If a symbol table is loaded, the source code position is also displayed (file name and line number). If a different task or a semaphore was involved in the error, its name is displayed also (*ErrorTaskName* or *SemaphoreName*, respectively).

RTKernel-32 messages are issued by an installable message handler. Please refer to Chapter 2, *Function RTKSetMessageHandler* for details.

Error Messages

All possible error messages follow in alphabetical order. Messages issued only by the Debug Version are marked "(dbg)".

AllocUserData: Too many user data entries used

RTKAllocUserData is called without any more user data entries available. The number of user data entries is limited to 16.

Block size must be at least 'sizeof(void*)'

RTKAllocMemPool was called with a block size less than 4. Buffers allocated through Memory Pools must have at least 4 bytes each.

Configuration structure size error

RTKernel-32 found that RTKConfig's Size field does not have the expected value. Please check the initialization of RTKConfig.

Conflicting math context flags

A call to RTKCreateThread specified both TF_MATH_CONTEXT and TF_NO_MATH_CONTEXT, which is contradicting.

DeleteMailbox: Task queued to get/put from/into mailbox

An attempt was made to delete a mailbox using RTKDeleteMailbox, although it is being used by other tasks.

Initial resource value must be 1

An attempt was made to initialize a resource or mutex semaphore with a value other than 1.

Initial semaphore value out of range

An attempt was made to initialize a binary or event semaphore with a value greater than 1.

Internal XXXX: Message (dbg)

An internal error has occurred in RTKernel-32. XXXX are 2 to 4 letters; *Message* provides more information about the error. Normally, such errors should never occur. They suggest that internal data structures of RTKernel-32 have been corrupted.

Interrupts enabled by task switch hook (dbg)

A task switch hook of an application has enabled interrupts which is not supported.

Invalid mailbox (dbg)

A mailbox was passed to RTKernel-32 that does not reference an existing or valid mailbox structure. It is also possible that internal data structures of RTKernel-32 have been corrupted, leading to a mailbox not being recognized as valid. Unnoticed stack overflows or dangling pointers can cause such errors.

Invalid semaphore (dbg)

A semaphore was passed to RTKernel-32 that does not reference an existing or valid semaphore structure. It is also possible that internal data structures of RTKernel-32 have been corrupted, leading to a semaphore not being recognized as valid. Unnoticed stack overflows or dangling pointers can cause such errors.

Invalid task handle (dbg)

A task handle was passed to RTKernel-32 that does not reference an existing or valid task. It is also possible that internal data structures of RTKernel-32 have been corrupted, leading to a task not being recognized as valid. Unnoticed stack overflows or dangling pointers can cause such errors.

IRQ out of range (dbg)

An invalid IRQ value was passed to one of the IRQ functions. RTKernel-32 supports IRQs 0 to 31, but the interrupt driver used can further limit the number of available IRQs.

Library Protection: negative recursion level

Functions protected by a single Automatic Library Protection semaphore have released the semaphore more often than it has been acquired.

Library Protection: recursion overflow

Functions protected by a single Automatic Library Protection semaphore have called each other recursively nested by more than three levels. If this recursion is intentional, your Library Protection must be reconfigured. In this case, please contact On Time's technical support.

Out of memory to allocate ...

An attempt was made to create a mailbox, a semaphore, a task, a trace buffer, a Memory Pool, or an interrupt stack, and not enough memory was available.

Preemptions not supported

RTKPreemptionsON was called although the interrupt driver does not support preemptions.

Priority out of range

RTKCreateThread or RTKKernelInit was passed an invalid priority. The Debug Version also checks the new priority in function RTKSetPriority.

Resource owned by task

Function RTKDeleteSemaphore was called for a resource or mutex semaphore currently occupied by a task.

ResourceOwner called on non-resource semaphore (dbg)

Function RTKResourceOwner was called with a semaphore that was not initialized as a resource or mutex semaphore.

RTKKernelInit() not called

An attempt was made to use the kernel before it has been initialized and automatic initialization was disabled in RTKConfig.

RTKIRQInfo: buffer too small

The buffer passed to RTKIRQInfo was too small. The buffer size should be at least 200 bytes.

RTKMailboxInfo: buffer too small

The buffer passed to RTKMailboxInfo was too small. The buffer size should be at least 120 bytes.

RTKPulse on non-event semaphores (dbg)

RTKPulse is only supported for event semaphores.

RTKResetEvent called on non-event semaphore (dbg)

Function RTKResetEvent is only supported for event semaphores.

RTKSemaInfo: buffer too small

The buffer passed to RTKSemaInfo was too small. The buffer size should be at least 130 bytes.

RTKTaskInfo: buffer too small

The buffer passed to RTKTaskInfo was too small to accept even one task. The buffer size should be at least 200 bytes.

RTKWait called in interrupt (dbg)

RTKWait or RTKWaitTimed was called inside an interrupt handler which is not supported.

Semaphore overflow (dbg)

An attempt was made to store more than $2^{32}-1$ events in a counting semaphore.

Semaphore type unknown

RTKOpenSemaphore was passed an unknown semaphore type.

Signal: Current task does not own resource (dbg)

A resource or mutex was released using RTKSignal by a task not occupying the resource or mutex.

Signal: Resource released out of sequence (dbg)

Resources or mutexes were released by RTKSignal in an improper sequence (see Chapter 2, *Semaphores*).

Stack overflow (dbg)

RTKernel-32 has recognized a stack overflow; the task had less than 64 bytes left on its stack. The Debug Version checks for stack overflow in each kernel call. This message is also issued by an explicit call to RTKStackCheck if a stack overflow has occurred.

Task already owns resource (dbg)

RTKWait, RTKWaitTimed, or RTKWaitCond were called by a task that already occupies the respective resource or mutex.

Task blocked in interrupt (dbg)

An attempt was made to perform a blocking task switch in an interrupt handler.

Task waiting on semaphore

RTKDeleteSemaphore was called although another task was waiting on the given semaphore.

User data index out of range

One of the user data functions has been called with an invalid user data index. The index must be allocated using RTKAllocUserData.

Informational Messages

In some cases, RTKernel-32 issues informational messages without aborting the program:

RTKernel-32 Debug Version (dbg)

This message is issued for informational purposes by RTKernelInit in the Debug Version.

RTKernel-32 exit function (dbg)

This message is issued for informational purposes by RTKernel-32's exit function in the Debug Version.

Part III

RTFiles-32

RTFiles-32 is a FAT-12/16/32 compatible file system for embedded systems. It allows embedded systems to access files on mass storage devices such as floppy disks and hard disks which can be shared with MS-DOS, Windows, or any other operating system which supports FAT disk volumes.

RTFiles-32's documented device driver structure allows it to be ported to many different environments and storage media. Device drivers for commonly used devices such as floppy disks and IDE drives are included. Custom drivers can easily be incorporated.

The main features of RTFiles-32 are:

- **Unlimited Number of Files**
RTFiles-32 can handle as many open files as required by the application. Only about 100 bytes of RAM is required for each open file.
- **Up to 32 Logical Drives**
RTFiles-32 easily handles many physical or logical drives.
- **Supports FAT-12, FAT-16, and FAT-32**
RTFiles-32 supports the same FAT (File Allocation Table) formats as MS-DOS, Windows, and many other systems. Media can be shared between RTFiles-32 and other operating systems to exchange data.
- **Supports Diskettes, Flash, SRAM Cards, and Hard Disks**
RTFiles-32 supports diskette drives and media with 360k, 1.2M, 720k, 1.44M, and 2.88M capacity. IDE disks with CHS (Cylinder, Head, Sector) or LBA (Logical Block Addressing) interfaces as well as IDE and DiskOnChip flash disks are supported. Adding custom drivers is easily accomplished through RTFiles-32's documented device driver interface.
- **Supports Removable Devices and Hot Swapping**
Floppy disks and removable hard disks (e.g., disks in PCMCIA sockets) can be removed and reinserted while an application is running. Devices are automatically remounted as needed. Critical error handling support is available to handle disk removal of devices in use.
- **Supports Hard Disks up to 2 Terabytes Size**
The latest standards (such as LBA) for extended sector addressing are supported if also supported by the disk controller.
- **Support for Contiguous Files**
RTFiles-32 can preallocate files to reside in a single contiguous chain of sectors. In this way, it can be guaranteed that no extra seek operations are required for contiguous read or write operations. Thus, the application can rely on achieving the maximum possible data throughput for such files.
- **Efficient Cache Support**
RTFiles-32 will cache frequently accessed data such as a volume's FAT or directories. The application has full control over the size of the cache and how delayed write operations should be handled. Large data blocks being read or written contiguously are not cached to avoid cache thrashing.
- **Extensive Diagnostics Support**
RTFiles-32 has functions to query the state of its cache in great detail. For each file or for a complete volume, the degree of fragmentation can be enquired.
- **Native API**
RTFiles-32 has its own native API consisting of about 45 functions. The native API is best suited for optimal performance and control.

- **Win32 Compatible API**
If used with RTTarget-32, RTFiles-32 emulates about 30 file I/O related Win32 functions. Thus, existing Win32 applications accessing files can be supported without source code modifications.
- **C/C++ and Pascal Run-Time System Support**
Through RTFiles-32's Win32 API emulation, all file I/O functions of the respective run-time systems are fully supported. C functions such as fopen, fread, fprintf, etc., work unmodified with RTFiles-32. The same is true for C++ classes such as iostream.
- **Multitasking Support**
When RTFiles-32 is used with a multitasking system such as On Time's RTKernel-32, all RTFiles-32 operations contain appropriate locking to support simultaneous calls from several tasks. Simultaneous access to different devices is fully supported (e.g., one task reads from a hard disk while another writes to a diskette). I/O device wait times are made available to other tasks by blocking the waiting tasks at a semaphore until the device I/O completion is signalled by an interrupt.
- **Low Interrupt Latencies**
Although RTFiles-32's device drivers are interrupt-driven, they never disable interrupts and do not process data transfers in interrupt handlers.
- **Installable Device Drivers**
RTFiles-32's device driver interface is very simple. Only three non-trivial functions are required to access a device: MountDevice, ReadSectors, and WriteSectors. All drivers shipped with RTFiles-32 come with complete source code which can be used as a model to implement custom drivers.
- **Installable System Drivers**
RTFiles-32 needs only a few functions of the underlying system (e.g., to install interrupt handlers or to set/reset semaphores, etc.). RTFiles-32 comes with system drivers for RTTarget-32 and/or RTKernel-32. These drivers are shipped with complete source code, allowing the implementation of alternate drivers.

Terms and Definitions

The following terms will be used throughout this manual:

Sector	The smallest storage unit which can be read from or written to a mass storage device. For most diskettes and hard disks, one sector has 512 bytes.
Cluster	A contiguous set of sectors used to allocate file space. RTFiles-32 supports cluster sizes of 1, 2, 4, 8, 16, 32, 64, and 128 sectors, as well as extended cluster sizes of up to 32768 sectors per cluster. Extended cluster sizes are not DOS/Windows compatible.
Physical Device	A physical mass storage device used to store files. Examples are hard disks or floppy disks.
Device	Alias for Physical Device.
Logical Drive	A portion (or possibly all) of a device holding a file system. Hard disks support several logical drives on a single device through partitions.
Drive	Alias for Logical Drive.
Volume	The media containing one or more logical drives such as a diskette or hard disk.
Partition	A physical portion of a hard disk. Hard disks can be divided into up to four primary partitions. A special type of partition (the extended partition) can be subdivided into any number of logical drives. Non-extended partitions can hold exactly one logical drive each.
Partition Table	A table with four entries located in the first physical sector of a hard disk. It describes the primary and extended partitions of the disk.
FAT	File Allocation Table. A data structure used by FAT file systems to record which clusters are used by each file.
CHS	Cylinder Head Sector. This abbreviation describes the traditional method of addressing sectors on diskettes and hard disks.
LBA	Logical Block Addressing. LBA is a newer method to address sectors on hard disks using a single 32-bit sector index. The translation to a physical CHS value is performed by the disk controller or device driver.
Sector Aligned Access	Reading and writing file data on sector boundaries. The file pointer is always a multiple of the sector size and data is read or written in multiples of the sector size. It does not mean that the data buffer supplied by the application must be aligned in any way.
Device List	A global data structure used by RTFiles-32 to locate all available devices.
File Instance	The RTFiles-32 internal data associated with an open file. When the same disk file is opened several times, several instances of a single file exist simultaneously.

Chapter 1

The FAT File System Structure

The FAT (File Allocation Table) file system was first introduced with MS-DOS in the early 1980s and then extended several times to accommodate for larger disk volumes. The term FAT refers to the allocation housekeeping method used. FAT file systems use a single table (the FAT) to record which areas of a disk are occupied.

This chapter describes the structure of FAT volumes.

Sectors, Sector Addressing, and Clusters

Mass storage devices such as diskettes or hard disks are frequently referred to as block devices. Software cannot access the device in arbitrarily small units of data. Rather, read and write operations have to be carried out in multiples of a *sector*. Most disks use a sector size of 512 bytes; however, other sector sizes would be possible.

Depending on the physical geometry of a device, addressing a sector may be required in different forms. On a single-sided diskette, a sector can be identified by its track and sector index. On a double-sided diskette, the head must also be known. Most hard disks are implemented as a stack of disks, so the disk must also be specified. On the other hand, a tape device would only require a linear sector index.

16-bit real-mode operating systems such as MS-DOS will normally use the BIOS to access a disk. The BIOS uses the traditional CHS (Cylinder-Head-Sector) addressing scheme, which requires three numbers to specify a sector. Unfortunately, the number of bits allocated for each number was chosen rather small, limiting the accessible disk size to about 8 GB. Newer BIOSes have addressed this deficiency and have defined a set of new, extended BIOS disk services which no longer have these limitations.

Because the traditional CHS sector addressing is device-dependent, limiting, and does not match the physical drive geometry on modern disk drives anyway, the LBA (Logical Block Addressing) scheme was introduced. It uses a single 32-bit value to address a sector. The first sector has index 0, the second 1, etc. The mapping of these LBA values onto the physical disk (i.e., the track, head, etc., the sector actually resides on) is performed by the device driver or even the device itself. With a sector size of 512 bytes and 2^{32} available sector addresses, disks with a capacity of up to 2 terabytes (2^{41}) are supported.

Internally, RTFiles-32 uses LBA values exclusively, so it can handle physical disks of up to 2 terabytes size. Depending on the device driver used, LBA values may be translated to CHS values, limiting the capacity available with a particular driver. RTFiles-32's floppy driver uses CHS addressing, which is good enough to address floppy disks with only a few MB capacity. Its IDE driver will query the IDE controller for LBA support. LBA is then used if available.

Keeping track of a very large number of sectors can be inefficient for a file system. Thus, the FAT file system introduces the concept of an allocation unit or *Cluster*. A cluster is simply a set of contiguous sectors which will be used for file space allocation. For example, if a disk has 1000 sectors, and it is formatted to have a cluster size of 2 sectors, only 500 clusters are available and must be managed by the FAT.

Through clusters, the size and number of available allocation units can be adjusted independently of the sector size. The advantage of few large clusters is that the file system needs less overhead for allocation unit housekeeping. The disadvantage is that disk space can be wasted because it is allocated to files in integral multiples of the cluster size. For example, on a volume with a cluster size of 32k, a file of 33k would occupy 2 clusters. Only 1k of the second cluster is actually used, wasting 31k. If the same file is copied to a drive with 4k cluster size, it would occupy 9 clusters, but only 3k of the last cluster would be wasted.

Logical Drives and Partition Tables

The basic unit of a FAT file system is a logical drive. A logical drive is a contiguous portion (or possibly all) of a disk consisting of the following components in the given order:

- boot record
- one or several FATs
- root directory
- data area

Diskettes, and all devices formatted as diskettes, will contain exactly these components.

With the introduction of MS-DOS 2.0, hard disks with much larger capacity than diskettes needed to be supported. Since MS-DOS was not able to handle such large volumes with a single logical drive, the concept of a partition table was added. The partition table only appears on hard disks. This table can contain up to 4 entries, each entry describing a single logical drive in a portion of the disk. A special partition type, the *Extended Partition*, can implement a linked list of any number of logical drives. Each logical drive described by a partition has exactly the format described above: boot record, FAT(s), root directory, and data area.

For backward compatibility, the first sector of a hard disk still has the basic structure of a boot record; it just adds the partition table to the end of the sector. Such a boot record with a partition table is called a *Master Boot Record*. If a hard disk is booted, the master boot record is executed which in turn will scan the partition table for a bootable logical drive. If one is found, the boot record of that logical drive is loaded and executed.

The Boot Record

The boot record is the first sector of a logical drive. It can (but does not have to) contain a boot loader needed to boot an operating system. The boot record always contains a structure called the *BIOS Parameter Block* describing some properties of the logical drive. For example, this information includes:

- size of the logical drive
- cluster size
- number and size of FATs
- size of root directory
- a disk serial number

The File Allocation Table and Cluster Sizes

The FAT follows the boot record. The FAT is a linked list of clusters, where each cluster chain represents a file. For example, if file C:\SOMEDIR\SOMEFILE.DAT resides in cluster 100, 101, 102, and 110, then the 100th integer value in the FAT has value 101. This means that the next cluster of this file is cluster number 101. The value found at 101 will be 102. At 102, value 110 is found. Cluster 110 is the last cluster of file, so an end-of-file marker is found there.

A volume's unused space is not linked in the FAT. Instead, its space is marked with value 0.

Microsoft has defined three different FAT types: FAT-12, FAT-16, and FAT-32. With FAT-12 (used mostly on diskettes), each FAT entry has a size of 12 bits. Thus, it can handle only about 4096 (minus a few reserved values) clusters. FAT-16 uses 16 bits to represent a cluster and consequently supports up to about 65536 clusters, while FAT-32 allocates 24 bits per cluster value. Since the number of available clusters is limited, large disks require larger cluster sizes. A cluster size must be a power of two. The maximum cluster size supported by DOS/Windows is 64 sectors or 32k bytes. Windows NT supports cluster sizes up to 128 sectors.

Some formatting programs allow the cluster size to be specified. In this case, the user must decide whether he prefers large clusters (and thus a small number of total clusters) or a large number of small clusters. If a volume will host many small files, a small cluster size should be selected to minimize the amount of space wasted in unused portions of the last cluster allocated to each file. Formatting with few large clusters is advantageous for volumes that should hold only a few, large files. The total size of the FAT depends on the number of available clusters and will therefore be small on such a volume, reducing the amount of housekeeping data the file system software has to maintain.

MS-DOS will usually format a drive to hold two copies of the FAT. The second copy is maintained as a backup copy for disk repair utilities. If the primary FAT gets corrupted, a disk repair program can copy the secondary FAT onto the primary FAT. The file system software itself will never actually use the second FAT. Thus, the second FAT can improve security, but it always incurs a performance penalty because each FAT update requires twice as many disk write operations as it would for a single FAT volume. It should also be noted that the added security is limited. Many errors will cause both FATs to be corrupted (example: the program terminates abnormally while unflushed data is cached). To improve performance, RTFiles-32 has an option not to maintain the second copy of the FAT. However, by default, both FATs are updated. RTFiles-32 can also format disks to hold only a single FAT.

Directories and Files

Information about individual files is maintained in directories. Each directory entry contains the name of a file (max. 8 characters), the file name extension (max. 3 characters), the date and time of the last file update, the current file size, file attributes, and the first cluster number holding the file's data. Subsequent clusters of the file must be looked up in the FAT.

Two types of directories must be distinguished: the root directory and subdirectories. On FAT-12 and FAT-16 drives, the root directory resides in a fixed location immediately following the last copy of the FAT. It has a fixed size determined at format time; its size is recorded in the boot record. The root directory cannot be extended after formatting. Subdirectories are stored like regular files. Unlike the root directory, each subdirectory has a directory entry in its parent directory. This directory entry contains the same information as for files and the space allocated to the subdirectory is recorded in the FAT. Thus, subdirectories can be extended dynamically and their size is not limited. Subdirectories can also be fragmented, just like files. On FAT-32 drives, the root directory is also maintained in a file chain and can thus be extended like any other directory.

Chapter 2

RTFiles-32 in Embedded Applications

This chapter gives an overview of RTFiles-32 and how it is integrated in an embedded program. Some of RTFiles-32's features are introduced.

Structure of an RTFiles-32 Program

Unlike file systems of most operating systems, RTFiles-32 is linked as a library into embedded systems application programs. This approach has several advantages: only those parts of the file system actually used will be linked and the file system can be accessed using function calls as opposed to software interrupts, dynamically linked entrypoints, or some other complicated access method.

RTFiles-32 consists of the following components:

- **Portable File System Core**
The file system core is contained in the RTFiles-32 library. It is completely written in ANSI C. Its source code is available as a separate add-on product and contains no device specific code.
- **RTFiles-32 Data Tables**
A default set of the data tables is also contained in the RTFiles-32 library, but it can be replaced by the application (see Chapter 5, section *RTFiles-32 Data Tables*). The configuration of these tables determines how many logical drives, simultaneously open files, and cache buffers RTFiles-32 can maintain.
- **Device List**
RTFiles-32 can use one or more devices with one or more device drivers. Most device drivers are capable of handling several physical devices. The RTFiles-32 library contains all device drivers shipped with RTFiles-32 as well as a default configuration of these drivers to be linked into an application. An application can override the default (e.g., to add a custom driver or to remove an unneeded driver) by defining a suitable data structure (see Chapter 5, section *Device List*).
- **System Driver**
The system driver enables RTFiles-32 and the device drivers to access system dependent services such as installing interrupt handlers, obtaining DMA buffers, etc. There is no default system driver in the RTFiles-32 library. Instead, each system driver is supplied as an additional library file, which must also be linked to the application. Further details are given in Chapter 5, section *The System Driver*.

When RTFiles-32's default configuration is used, no source code modifications are required in existing programs using files. Existing file I/O operations using one or more of RTFiles-32's alternate APIs work unmodified. Thus, it is very easy to port existing programs with file I/O to RTFiles-32.

RTFiles-32 APIs

The file system core implements RTFiles-32's native API, which consists of approximately 45 functions. Their prototypes are given in header file `RTFILES.H` and in `RTFILES.PAS` for Pascal. This API is completely documented in Chapter 3 and supports access to all of RTFiles-32's features.

As an option, RTFiles-32 can emulate the Win32 API. The Win32 emulation is available by default if RTFiles-32 is used with `RTTarget-32`. With the Win32 API emulation, all run-time system functions for file I/O are made available automatically since they are implemented using operating system calls. Thus, both C++ and C style file I/O operations such as `iostreams` or `printf` can be used.

Mounting Devices and Logical Drives

RTFiles-32 must assign drive letters and mount all drives to be used. Mounting involves determining the file system type (FAT-12/16/32), the drive's size, etc. RTFiles-32 mounts drives in an MS-DOS and Windows compatible order to achieve the same drive letter assignment. During the first RTFiles-32 API

call which needs access to a drive, drive letters are assigned according to the rules given below. Please note that removable devices are not accessed until they are actually needed. However, the partition tables of all fixed hard disks are read at program startup.

- Drive letters are assigned to devices in the order they appear in the device list.
- Drive letter assignment starts with letter 'A'. When a hard disk is mounted, the driver letter is set to 'C' or higher.
- Floppy disks and non-removable hard disks are mounted before removable hard disks.
- A single drive letter is reserved for every floppy and removable hard disk, even if they are not present or never accessed. For non-removable hard disks, drive letters are only assigned for existing FAT partitions.
- For non-removable hard disks, drive letters are assigned in this order:
 - the first primary partition of all devices, then
 - all logical drives in extended partitions of all drives, then
 - all other primary partitions of all drives.

While this mounting algorithm may seem complicated, it corresponds exactly to the method used by MS-DOS and should, therefore, result in the same drive letter assignment. Please note that you can utilize a much simpler scheme by setting flag `RTF_DEVICE_MOUNT_CONTIGUOUS` for all devices. If used, all partitions of a device will be assigned consecutive drive letters.

The only drive letters automatically skipped by RTFiles-32 are 'A' and 'B' if no or only one floppy disk is present before the first hard disk is found. However, you can force RTFiles-32 not to use drive letters by inserting dummy devices into the device list using the `RTFDrvNULL` driver.

RTFiles-32 Buffers

RTFiles-32's most important data structure is its buffers. Buffers are used to hold FAT and directory data, as well as application file data when data is read or written at file offsets which are not integral multiples of the sector size.

The buffers also serve as RTFiles-32's cache. Data is read into the buffers as needed and is kept as long as possible. If the same data is needed again later, RTFiles-32 does not need to reread it from disk. RTFiles-32's algorithm used to determine which buffers are flushed or discarded at which time has been developed based on statistics from many tests. The application has some control over this algorithm both for individual files (RTFOpen flags `RTF_COMMITTED`, `RTF_CACHE_DATA`, and `RTF_LAZY_DATA`) or complete devices (device flag `RTF_DEVICE_LAZY_WRITE`). In addition, the application can control the number of available buffers and function `RTFBufferInfo()` can be used to analyze buffer utilization and cache efficiency.

Frequently, the caching strategy is a trade-off between throughput, real-time performance, and data security. RTFiles-32's default behavior corresponds to that of MS-DOS when no disk cache program is loaded:

- Buffers containing application data are flushed and discarded when the file pointer leaves the sector of the buffer.
- When a file is close, and the file's allocated size has changed, the FAT of the drive holding the file is updated.
- A file's directory entry is written to disk when the file is created and, if the file was modified, when the file is closed.

Various options are available to override these defaults (see Chapter 5 and Chapter 7 for details).

It should be noted that RTFiles-32's buffers are not used as an application data cache. When the application reads or writes large data blocks which completely span one or more sectors, the device driver will read/write directly from/to the buffer supplied by the application, completely bypassing RTFiles-32's buffers. This is done to avoid large application I/O requests displacing FAT and directory data in the buffer cache, since FAT and directory data are more likely to be accessed again.

Another level of caching can be performed by device drivers. For example, the IDE and Floppy drivers can use a read-ahead buffer (which can also be configured). When the application reads a sector, the driver will actually read 4 sectors. If the following sectors are read in subsequent reads, entire disk accesses can be eliminated.

File Types

RTFiles-32 supports four different types of files: data files, directories, logical drives, and physical devices. Logical and physical drive files are also supported by Windows NT with the same naming conventions as under RTFiles-32. All four file types are accessed using the same file I/O functions such as RTFOpen, RTFRead, RTWrite, etc., or their API emulation equivalents. However, some operations may not be available for all file types. For example, you cannot rename a device file or set the file date and time of a logical drive file, since such files do not have a standard directory entry.

Data Files

A data file has a name, attributes, date and time of last update, an allocated file size, and a current file size. Unlike DOS, RTFiles-32 supports allocated file sizes exceeding the current file size by more than one cluster to support contiguous files (see Chapter 3, section *Function RTFExtend* for details).

If the file is open, it also has a current file pointer and a set of flags. The file pointer marks the offset within the file's data where the next read or write operation will start. The file pointer is advanced automatically. The file's flags define some options for the file, such as whether the file is open for read-only or read/write access, etc.

Data files are used by application programs to store data. Their size is limited to $2^{32}-1$ (4G) bytes, even on FAT-32 volumes.

Directory Files

Unlike DOS, RTFiles-32 allows opening directories as files. There are only a few differences to data files. Directories must be opened in read-only mode. The name of a root directory is 'X:\' (with 'X' being the drive letter) while the name of any subdirectory is the subdirectory's filename, and directory files do not have a current file size. Instead, their allocated file size is used.

Directory files can be useful for scanning directories for entries not accessible through RTFFind-First/RTFFindNext (e.g., deleted file entries, etc.).

Logical Drive Files

RTFiles-32 allows a logical drive to be opened as a file. Both read-only and read/write access are supported. The file name of a drive file is '\\.\X:', where 'X' must be replaced with the desired drive letter.

A drive file spans the complete logical drive, starting at the first sector of the drive (the boot sector). Access to drive files must be sector size aligned. This means you can only read or write data in complete sector size multiples. The same restriction applies to seek operations. Logical drive files bypass RTFiles-32's buffers. Read and write requests are passed directly to the device drivers. However, RTFiles-32 will ensure that the buffer contents remains consistent with the drive's contents when a device or drive file accesses data also present in the buffer cache.

Logical drive file I/O is possible even on unformatted drives as long as low-level sector I/O is possible. Logical drive files can be used to high-level format a volume.

Physical Device Files

Physical device files are similar to logical drive files. However, device files span a complete hard disk, not just a single logical drive of that disk. Since device files would be identical to logical drive files on floppy disks, they are not required here.

The file name of a device file is '\\.\'PHYSICALDRIVEx' where 'x' is a digit representing the 'x'th hard disk. Thus, file name '\\.\'PHYSICALDRIVE0' would open the first hard disk, '\\.\'PHYSICALDRIVE1' the second, etc. Hard disks are numbered starting with 0 in each RTF_DEVICE_FDISK entry in RTFiles-32's Device List.

For obvious reasons, using logical drive files and physical device files with read/write access is dangerous. Writing incorrect data to critical areas of a device such as the partition table, a boot record, the FAT or directory can completely destroy a file system.

Raw I/O

An even lower level of access to a device is available through RTFiles-32's raw I/O functions. Raw I/O functions call the device driver directly and bypass RTFiles-32's file system core. However, consistency with the buffer cache is maintained.

Chapter 3

RTFiles-32 Native API

This chapter describes RTFiles-32's native API. All functions documented here are declared in header file `RTFILES.H`, which must be included in every module using these functions.

Many applications will not use RTFiles-32's native API directly, but will prefer to use standard C or C++ library functions/operators instead. The use of alternate APIs (C++, C, and Win32 API emulation) is detailed in Chapter 4. However, some of RTFiles-32's advanced features are only accessible through its native API. Therefore, it is recommended to read this chapter to get a complete overview of RTFiles-32's functionality. Chapter 4 also describes how to use an alternate API primarily, but also use RTFiles-32's native API for functions only available through this interface.

The results of some API functions depend on RTFiles-32's configuration parameters described in Chapter 5. The documentation in this chapter assumes that you are using the default configuration:

- up to 8 logical drives,
- up to 8 simultaneously open files,
- 32 buffers,
- Floppy driver for floppies 0 and 1, IDE driver for the master and slave on the first IDE controller,
- all device flags are 0.

Return Codes and File Handles

Most RTFiles-32 functions return an integer value. If the documentation for the respective function does not explicitly state something different, negative return values indicate an error. Appendix A lists all possible error return codes. Values greater than or equal to 0 (constant `RTF_NO_ERROR`) indicate success. The meaning of positive return values is documented for each function.

`RTFILES.H` defines type `RTFHANDLE`, which is also an integer. `RTFOpen` and `RTFFindFirst` return values of this type. Such values are file handles (if greater than or equal to 0) or error codes if less than 0. File handles can be used in subsequent RTFiles-32 API calls to reference an open file. Handles returned by `RTFFindFirst` are actually file handles for a directory. Care must be taken to close all file handles to ensure that all data is flushed to the disk and subsequent `RTFOpen` calls will not fail because the number of available file handles is exhausted.

The documentation of each RTFiles-32 function does not explicitly enumerate all possible errors. In most cases, almost any error is possible. For example, each function that may access a device could return any device-specific error.

General File I/O

This section describes the most common file I/O functions to open, read, write, and close files.

Function `RTFOpen`

`RTFOpen` opens and possibly creates a file for subsequent read and/or write access:

```
RTFHANDLE RTFOpen(const char * FileName, DWORD Flags);
```

Parameter `FileName` must point to the name of the file to open/create. File names are not case sensitive; they will be converted to upper case. The file name can have one of the following formats:

[Drive:][\][Path\]Name[.Ext]	A data file or directory file name with optional drive and path information.
[Drive:]\	A root directory.
\\Drive:	A logical drive.
\\.\PHYSICALDRIVEx	A physical hard disk. 'x' must be a digit (starting at '0') specifying the desired hard disk.

Drive must be replaced with a single letter greater or equal to 'A'.

For data and directory files, the same rules for file name syntax apply as under MS-DOS. For logical drive and physical disk file names, the same rules as under Windows NT apply. For more information about logical drive and physical disk files, please refer to Chapter 1, section *File Types*.

Parameter Flags can be a combination of the following flags:

RTF_READ_WRITE	The file is opened for read and write access.
RTF_READ_ONLY	The file is opened for read only access.
RTF_OPEN_SHARED	Opening the file multiple times should not generate an "access denied" error. By default, RTFiles-32 will allow the same file to be opened several times only if all instances of the file are opened with read only access. However, if one or more instances also require write access and all instances specify this flag, the call succeeds. Please note that a drive file or device file spans one or more logical drives and thus conflicts with any other file on the respective drive or device.
RTF_OPEN_NO_DIR	Do not open directories. Use this flag to avoid accidentally opening directories.
RTF_OPEN_DIR	Overrides flag RTF_OPEN_NO_DIR and forces support for opening a directory. Please note that directories can only be opened with read only access.
RTF_CREATE	Instructs RTFOpen to create the file if it does not exist.
RTF_CREATE_ALWAYS	Instructs RTFOpen to create the file even if it already exists.
RTF_COMMITTED	Specifies that all updates to the file should be written to the physical device immediately. This includes the directory entry for the file as well as the drive's FAT when the file's size changes. Use this flag with care, since the performance penalty can be severe. By default, RTFiles-32 will flush data buffers when the file pointer leaves a sector and it will flush FAT and directory data when the file is closed.
RTF_CACHE_DATA	Instructs RTFiles-32 not to discard data buffers for this file. This option is useful during random access where small blocks are read with frequent inter-leaving seek operations. In this case, RTFiles-32's internal buffers will serve as a cache for the file. By default, RTFiles-32 assumes that files are read or written sequentially and will therefore discard data buffers when the file pointer leaves a sector. In this way, it is avoided that data sectors displace FAT and directory data in the buffers.
RTF_LAZY_DATA	Instructs RTFiles-32 not to flush dirty (modified) data buffers when the file pointer leaves a sector. This flag automatically also sets RTF_CACHE_DATA. This flag can improve performance when the same data is written several times, since data which would get overwritten will never actually be written to the disk. Such unflushed (lazy) data buffers will be flushed when the file is closed, or when the last file on the same drive is closed on drives with device option RTF_DEVICE_LAZY_WRITE.
RTF_ATTR_HIDDEN RTF_ATTR_SYSTEM RTF_ATTR_ARCHIVE	If RTFOpen creates the file, any combination of these file attributes may be set. The file will be created with all specified attributes set.

If the function succeeds, the return value is a file handle for the opened file and the file pointer of the file is set to 0. If the return value is less than 0, the function has failed and the return value is the error code. All possible error codes are given in Appendix A. Function RTFErrorMessage can be used to display a meaningful message string for the error code.

Function RTFClose

RTFClose closes an open file:

```
int RTFClose(RTFHANDLE File);
```

Parameter File must have been assigned in a previous successful call to RTFOpen. RTFClose will write any unflushed file data to the disk (except for RTF_DEVICE_LAZY_WRITE devices) and release all resources associated with the file handle.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code.

Function RTFRead

RTFRead reads data from a file:

```
int RTFRead(RTFHANDLE File, void * DataPtr, UINT Length, UINT * Read);
```

Parameter File references the open file from which to read.

DataPtr specifies the address for the data to be read.

Length specifies the number of bytes to read.

Parameter Read points to an unsigned integer to receive the number of bytes actually read. Usually, *Read will contain Length after the call. However, in case of an error or if the end of file is encountered during the read, the value may be less. Read may be set to NULL if this information is not required by an application.

Reading past the end of file is not regarded as an error. If RTFRead returns RTF_NO_ERROR, but *Read is less than Length, the end of file has been encountered.

This function advances the file's file pointer by the amount given in *Read. In case of an error and if parameter Read is NULL, the new file pointer is undefined.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code.

Function RTFWrite

RTFWrite writes data to a file:

```
int RTFWrite(RTFHANDLE File, void * DataPtr, UINT Length, UINT * Written);
```

Parameter File references the open file to write to.

DataPtr specifies the address of the data to be written.

Length specifies the number of bytes to write.

Parameter Written points to an unsigned integer to receive the number of bytes actually written. Usually, *Written will contain Length after the call. However, in case of an error, the returned value may be less. Written may be set to NULL if this information is not required by an application.

Writing past the current end of file will automatically extend the file. When the file size exceeds the current allocated file size, new clusters are allocated for the file. RTFiles-32 will allocate new clusters immediately following the current last cluster, if possible.

This function advances the file's file pointer by the amount given in *Written. In case of an error and if parameter Written is NULL, the new file pointer is undefined.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code.

Function RTFSeek

RTFSeek repositions a file pointer and possibly extends a file:

```
long RTFSeek(RTFHANDLE File, long Offset, int Whence);
```

Parameter File references the open file for which to reposition the file pointer.

Offset specifies how far the file pointer should be moved. Please note that Offset is a signed long value.

Parameter Whence specifies Offset's meaning. The following values are allowed:

RTF_FILE_BEGIN Offset is an absolute file pointer value.

RTF_FILE_CURRENT Offset should be added to the current file pointer value.

RTF_FILE_END Offset should be added to the current file size.

Moving the file pointer before the beginning of the file is an error. However, moving it beyond the current file size is supported. In this case, the file is extended. The data between the previous file size and the new file size is undefined. This method to extend a file is much faster than actually writing data to it.

Please note that this function cannot move the file pointer by more than $2^{31}-1$. If you want to set the file pointer to a larger value, at least two subsequent calls to RTFSeek with Whence set to RTF_FILE_CURRENT are required.

If the function succeeds, the return value is the new file pointer value, or, if the file pointer is larger than $2^{31}-1$, RTF_LONG_FILE_POS is returned. In this case, the actual file pointer can be queried using function RTFGetFileInfo. If the function fails, the return value is some other negative error code.

An alternate method to extend a file is to use function RTFExtend described below.

Function RTFExtend

RTFExtend changes the allocated file size, but not the current file size, of a file:

```
int RTFExtend(RTFHANDLE File, DWORD Length);
```

Parameter File references the open file to be extended.

Parameter Length specifies by how many bytes the file is to be extended relative to the current file pointer.

RTFExtend will add parameter Length to the current file pointer value and round the result up to the next multiple of the cluster size. If fewer clusters are currently allocated for the file, new clusters are allocated for the file. The allocation is guaranteed to consist of a single cluster chain. If the file has a current allocated file size of 0, the file will be unfragmented after this call. If, however, clusters are already allocated, RTFExtend will attempt to allocate the new cluster chain immediately following the existing last cluster chain. If this fails, the new cluster chain is allocated somewhere else and the file will not be contiguous.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code. If RTFExtend cannot find a chain of free clusters long enough to satisfy the request, error RTF_DISK_FULL is returned. However, it may still be possible to extend the file by some other means (e.g., RTFSeek or RTFWrite); only contiguous extension of the file is not possible.

Files allocated contiguously may exhibit improved and possibly even deterministic access times (see Chapter 7 for a detailed discussion on file I/O performance considerations). However, RTFExtend itself can take substantially longer than RTFSeek to extend a file. In particular on volumes with only little free space, reading the complete FAT may be required.

RTFExtend does not alter the current file size. As a consequence, the difference between the allocated file size and the current file size may become greater than or equal to a cluster size. This situation cannot occur under MS-DOS or other FAT file systems. If the extra allocated file space is not used by subsequent write or seek operations, disk analysis utilities such as SCANDISK will report either lost clusters (the clusters allocated but currently not in use) or an invalid file size in the directory entry. However, this is not a problem. MS-DOS is still able to use the file. Fixing the problem with SCANDISK does not loose any data.

Function RTFCommit

RTFCommit immediately flushes all buffers associated with a file to disk:

```
int RTFCommit(RTFHANDLE File);
```

Parameter File references the open file to be committed.

RTFCommit guarantees that all of the file's data is flushed. If the file was opened with flag `RTF_LAZY_DATA`, all dirty buffers of the file's drive are flushed. Otherwise, the file's data buffer, directory entry, and the complete FAT are flushed.

If the function succeeds, the return value is `RTF_NO_ERROR`. If the function fails, the return value is a negative error code.

Function RTFTruncate

RTFTruncate sets the current file size to the current file pointer position:

```
int RTFTruncate(RTFHANDLE File);
```

Parameter `File` references the open file to be truncated.

RTFTruncate sets the file size to the current file pointer and frees any allocated file space beyond the new file size.

If the function succeeds, the return value is `RTF_NO_ERROR`. If the function fails, the return value is a negative error code.

Information about Files

This section describes functions to get and set information about open files.

Function RTFGetFileInfo

RTFGetFileInfo returns information about an open file:

```
int RTFGetFileInfo(RTFHANDLE File, RTFFileInfo * FileInfo);
```

Parameter `File` references the open file for which information is requested.

Parameter `FileInfo` must point to a structure `RTFFileInfo`, declared in `RTFILES.H`:

```
typedef struct {
    RTFDOSDirEntry * DirEntry;
    DWORD           FilePos;
    DWORD           AllocatedSize;
    DWORD           ClusterChains;
    DWORD           VolumeSerialNumber;
    char            * FullName;
} RTFFileInfo;
```

When the function returns `RTF_NO_ERROR`, the structure is filled with the following information.

`DirEntry` points to the file's directory entry:

```
typedef struct {
    char      FileName[8];
    char      Extension[3];
    BYTE      Attributes;
    char      Reserved[8];
    WORD      FirstClusterHi;    // FAT-32 only
    RTFDOSDateTime DateTime;
    WORD      FirstCluster;
    DWORD     FileSize;
} RTFDOSDirEntry;
```

`DirEntry` is only valid while the file is open. If you need the information in the directory entry after the file has been closed, you must copy it.

`FilePos` has the current file pointer value.

`AllocatedSize` is the currently allocated file size.

`ClusterChains` indicates how many separate cluster chains are allocated for the file. For unfragmented files, this value will be 1.

`VolumeSerialNumber` holds the serial number of the volume the file resides on.

FullName points to the complete path name of the file. This field is only valid while the file is open. If you need the file's full name after the file has been closed, you must copy it.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code.

Function RTFGetFileSize

RTFGetFileSize retrieves the current size of an open file:

```
int RTFGetFileSize(RTFHANDLE File, DWORD * Size);
```

Parameter File references the open file for which to retrieve the file size.

Parameter Size must point to DWORD to receive the file's size.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code.

Function RTFSetFileTime

RTFSetFileTime sets the date and time information in a file's directory entry:

```
int RTFSetFileTime(RTFHANDLE File, const RTFDOSDateTime * Time);
```

Parameter File references the open file for which to set the time and date.

Parameter Time must point to a filled structure RTFDOSDateTime declared in RTFILES.H:

```
typedef struct {  
    WORD Second2:5;  
    WORD Minute:6;  
    WORD Hour:5;  
    WORD Day:5;  
    WORD Month:4;  
    WORD Year1980:7;  
} RTFDOSDateTime;
```

Seconds2 holds the seconds part of the desired time divided by 2. Year1980 holds the year part of the desired date minus 1980. The meaning of all other fields is self-explanatory.

RTFiles-32 will update the file's date and time on every write access. Thus, function RTFSetFileTime should not be followed by a call to RTFWrite, since this would overwrite the date and time set by RTFSetFileTime.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code.

File Attributes

The functions in this section can be used to query and change file attributes of closed files.

Function RTFGetAttributes

RTFGetAttributes returns the attributes set for a specific file:

```
int RTFGetAttributes(const char * FileName);
```

Parameter FileName is the name of the file for which to retrieve the attributes.

If the function return value is positive, it contains the file's attributes, which can be any combination of the following values:

RTF_ATTR_READ_ONLY	The file is read only. Any attempt to open it with read/write access will return error "access denied".
RTF_ATTR_HIDDEN	The file is marked as hidden. This attribute has no effect on any RTFiles-32 function.
RTF_ATTR_SYSTEM	The file is marked as being a system file. This attribute has no effect on any RTFiles-32 function.

RTF_ATTR_VOLUME	The file is a volume label. Only a directory entry on a logical drive in the root directory can have this attribute set.
RTF_ATTR_DIR	The file is a directory.
RTF_ATTR_ARCHIVE	The file is marked to be backed up. This attribute has no effect on any RTFiles-32 function, but it is set on every write operation.

If the function fails, the return value is a negative error code.

Function RTFSetAttributes

RTFSetAttributes assigns a new set of attributes to a given file:

```
int RTFSetAttributes(const char * FileName, BYTE Attributes);
```

Parameter FileName is the name of the file for which to set the new attributes.

Parameter Attributes can be a combination of the following values:

RTF_ATTR_READ_ONLY

RTF_ATTR_HIDDEN

RTF_ATTR_SYSTEM

RTF_ATTR_ARCHIVE

Please refer to the previous section for the meaning of the attributes. An attempt to set attributes RTF_ATTR_VOLUME or RTF_ATTR_DIR will cause this function to fail.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code.

Directories

Functions described in this section are used to create, remove, and change directories.

Function RTFGetCurrentDir

RTFGetCurrentDir returns the current drive and directory:

```
int RTFGetCurrentDir(const char * DirName, UINT MaxLength);
```

Parameter DirName must point to a string buffer to receive the full path of the current directory, including a drive letter.

Parameter MaxLength specifies the size of the buffer passed in DirName. It is recommended to use a buffer with RTF_MAX_PATH (80) characters length. If the buffer is too small to hold the current path, the function fails.

If the function succeeds, the return value is RTF_NO_ERROR and the current path has been copied to the specified buffer. If the function fails, the return value is a negative error code.

Function RTFSetCurrentDir

RTFSetCurrentDir changes the current directory and drive:

```
int RTFSetCurrentDir(const char * DirName);
```

Parameter DirName must point to the name of the new current directory. The new directory can have any legal file name syntax. If no drive is given, the current drive is not changed. If only a drive, but no file name is given, only the drive is changed, and the current directory last used on the new drive is used.

RTFiles-32 maintains a default directory for each drive, but only one current directory can be active at any one time.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code.

Function RTFCreateDir

RTFCreateDir creates a new directory:

```
int RTFCreateDir(const char * DirName);
```

Parameter DirName must point to the name of the directory to create. The directory can have any legal file name syntax.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code.

Function RTFRemoveDir

RTFRemoveDir removes a directory:

```
int RTFRemoveDir(const char * DirName);
```

Parameter DirName must point to the name of the directory to be removed. The directory can have any legal file name syntax.

This function fails on an attempt to remove a current directory, a directory which is not empty, or a root directory.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code.

Finding Files

The functions in this section are used to scan directories for files.

Function RTFFindFirst

RTFFindFirst searches a directory for a file satisfying certain criteria:

```
RTFHANDLE RTFFindFirst(const char * NamePattern,
                       BYTE Attr1, BYTE Attr2,
                       RTFDOSDirEntry * FileInfo,
                       char * FileName);
```

Parameter NamePattern must point to a file name, which can contain wildcard characters '*' and/or '?', and can optionally be preceded by a path. If a path is present, it must not contain any wildcard characters.

Parameter Attr1 specifies a set of all file attributes a file must have to match the request. Parameter Attr2 are exclude attributes. Files having any one of these attributes set do not satisfy the request. Please note that specifying the same attributes for both Attr1 and Attr2 will find no files. Any combination of the following flags can be specified for Attr1 and Attr2:

```
RTF_ATTR_READ_ONLY
RTF_ATTR_HIDDEN
RTF_ATTR_SYSTEM
RTF_ATTR_VOLUME
RTF_ATTR_DIR
RTF_ATTR_ARCHIVE
```

Parameter FileInfo must point to an RTFDOSDirEntry structure. If the function succeeds, this structure will be filled with the directory entry of the file found.

Parameter FileName points to a string buffer which must be at least 13 characters long. This buffer will receive the file name if a file is found. This parameter may be NULL.

If the function succeeds, at least one file satisfies the search criteria. *FileInfo contains the directory entry of the first file and the function return value is greater than or equal to 0. The return value is a file handle which may be passed to subsequent calls to RTFFindNext. It is important to close the handle using RTFFindClose when no longer needed. Failing to do so will quickly exhaust the available file handles.

If the function fails, the return value is a negative error code. In this case, no handle is allocated and RTFFindClose need not be called.

Function RTFFindNext

RTFFindNext finds more files with the same search criteria as a preceding call to RTFFindFirst:

```
int RTFFindNext(RTFHANDLE File, RTFDOSDirEntry * FileInfo, char * FileName);
```

Parameter File must be a valid handle returned by a previous call to RTFFindFirst.

Parameter FileInfo must point to a RTFDOSDirEntry structure.

Parameter FileName is an optional parameter (may be NULL) which points to a string buffer of at least 13 characters to receive a file name.

If the function succeeds, this structure will be filled with the directory entry of the file found.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails or no more files are found, the return value is a negative error code.

Function RTFFindClose

RTFFindClose closes a handle created by RTFFindFirst:

```
int RTFFindClose(RTFHANDLE File);
```

Parameter File must be a valid handle returned by a previous call to RTFFindFirst.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code.

File Name Operations

Functions in this category are used to rename, delete, or create unique files.

Function RTFRename

RTFRename renames a file:

```
int RTFRename(const char * FileName, const char * NewName);
```

Parameter FileName must point to the name of the file to be renamed. Parameter NewName points to the new name of the file. Both file names must not contain wildcards and must reference the same logical drive. However, they may reference different directories. Data files and directory files (except root directories) may be renamed or moved with this call.

This function fails if a file with the name specified by NewName already exists, on an attempt to rename the current directory or a parent of the current directory, or on an attempt to rename a volume label.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code.

Function RTFDelete

RTFDelete deletes a file:

```
int RTFDelete(const char * FileName);
```

Parameter FileName must point to the name of the file to be deleted and may not contain wildcards. This function cannot delete directories.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code.

Function RTFMakeTempFileName

RTFMakeTempFileName creates a file with a unique name:

```
int RTFMakeTempFileName(const char * DirName, char * FileName, UINT MaxLength);
```

Parameter DirName must point to the name of the desired directory. For the current directory, use ".".

Parameter `FileName` must point to a buffer to receive the full path and file name of the file to be created. `MaxLength` specifies the size of the buffer referenced by `FileName`. It is recommended to use a buffer with `RTF_MAX_PATH` (80) characters length.

This function will generate a file name consisting of a hexadecimal string derived from the current date and time and with extension `".TMP"`. If a file with this name already exists, the file name is modified until a unique name is found. This file is then created with current file size and allocated file size zero and no attributes set.

If the function succeeds, the return value is `RTF_NO_ERROR` and the created file's name has been copied to the supplied buffer. If the function fails, the return value is a negative error code.

Function `RTFMakeFileName`

`RTFMakeFileName` builds a valid file name from information found in a directory entry:

```
int RTFMakeFileName(const RTFDOSDirEntry * FileInfo, char * FileName);
```

Parameter `FileInfo` must point to a valid directory entry, which, for example, could have been read from a directory. Parameter `FileName` must point to a string buffer of at least 13 characters length to receive the file name.

If the `RTF_ATTR_VOLUME` attribute is set in `FileInfo->Attributes`, the file name built does not contain a period to separate name and extension. For other files, the period is present only if the extension is not blank. This function does not access any physical disks.

If the function succeeds, the return value is `RTF_NO_ERROR` and the file name is copied to `*FileName`. If the function fails, the return value is a negative error code.

Function `RTFExpandName`

`RTFExpandName` will translate an arbitrary file name to a complete absolute path with drive, path, and file name components:

```
int RTFExpandName(char * FileName, UINT MaxLength);
```

Parameter `FileName` must point to the file name to be expanded.

`MaxLength` specifies the size of the buffer pointed to by `FileName`. The result is written to `*FileName`, overwriting the original name.

`RTFExpandName` will access the drive only to determine the current directory. The function does not check whether the file or any of its parent directories exist.

If the function succeeds, the return value is `RTF_NO_ERROR` and the expanded file name is copied to `*FileName`. If the function fails, the return value is a negative error code.

Disk and Volume Management

Function `RTFResetDisk`

`RTFResetDisk` unmounts a single or all mounted devices:

```
int RTFResetDisk(const char * DriveName);
```

If parameter `DriveName` is `NULL`, all devices are unmounted; otherwise, only the device hosting the specified drive is unmounted.

Unmounting a device makes `RTFiles-32` discard all information it has about a device. The next time any drive on the device is accessed, `RTFiles-32` will reread the drive's boot sector and reinitialize all internal data structures for this drive.

Please note that this function affects drive letter assignment only if parameter `DriveName` is `NULL`. In that case, the next disk I/O function will rescan all partition tables, which may cause driver letters to change.

If the function succeeds, the return value is `RTF_NO_ERROR`. If the function fails, the return value is a negative error code. Note that all files on the respective drive(s) must be closed for this function to succeed. If any files are open, `RTF_ACCESS_DENIED` is returned.

Function RTFGetDiskInfoEx

RTFGetDiskInfoEx returns information about a logical drive:

```
int RTFGetDiskInfoEx(const char * DriveName, RTFDiskInfo * DiskInfo, int Flags);
```

Parameter DriveName must be a valid file name, e.g., a root directory name. Only the drive information (possibly the current default drive) is determined.

Parameter DiskInfo must point to a structure RTFDiskInfo declared in Rtf files.h:

```
typedef struct {
    char    Label[12];
    char    DriveLetter;
    char    Reserved[3];
    DWORD   SerialNumber;
    DWORD   FirstPhysicalSector;
    UINT    FATType;
    UINT    FATCount;
    UINT    MaxDirEntries;
    UINT    BytesPerSector;
    UINT    SectorsPerCluster;
    UINT    TotalClusters;
    UINT    BadClusters;
    UINT    FreeClusters;
    UINT    Files;
    UINT    FileChains;
    UINT    FreeChains;
    UINT    LargestFreeChain;
} RTFDiskInfo;
```

Parameter Flags can be any combination of:

- | | |
|-----------------------|---|
| RTF_DI_BASIC_INFO | Returns all fields in structure RTFDiskInfo except FreeClusters, BadClusters, Files, FileChains, FreeChains, LargestFreeChain. This flag never requires a FAT scan. |
| RTF_DI_FREE_SPACE | Returns field FreeClusters. This flag may require RTFiles-32 to scan the complete FAT if the amount of free space is not known. In this case, RTF_DI_FAT_STATISTICS is returned in addition to RTF_DI_FREE_SPACE. |
| RTF_DI_FAT_STATISTICS | Return fields BadClusters, Files, FileChains, FreeChains, LargestFreeChain. This flag will always cause RTFiles-32 to scan the complete FAT. |

If the function succeeds, the return value is the Flags value corresponding to the returned information, or a negative error code. RTFGetDiskInfoEx will complete significantly faster if no FAT scan is necessary. On large FAT-32 volumes, a complete FAT scan can take up to about one minute. When the FAT has been scanned, the number of free clusters is cached (on disk for FAT-32 and in RAM for all disk formats), allowing subsequent calls with RTF_DI_FREE_SPACE to return much faster.

The various fields of structure RTFDiskInfo are:

- | | |
|---------------------|--|
| Label | A zero-terminated string with the volume's label. The string is empty if the volume has no label. |
| DriveLetter | The drive letter in upper case. |
| SerialNumber | The volume's serial number. |
| FirstPhysicalSector | The LBA address of the logical drive's boot record. For diskettes, this value will be 0. |
| FATType | The type of file system found. It may have values 12, 16, or 32 for FAT-12, FAT-16, or FAT-32 volumes. |
| FATCount | The number of FATs on the volume. |
| MaxDirEntries | The size of the root directory. This value is 0 for FAT-32. |

BytesPerSector	The sector size. This value will usually be 512.
SectorsPerCluster	Specifies the size of the smallest unit of storage that can be allocated to a file in sectors.
TotalClusters	Number of clusters for file storage on the volume.
BadClusters	The number of clusters which are marked bad and are unavailable for file storage.
FreeClusters	The number of clusters currently available.
Files	The number of files on the volume including directories, but not counting the root directory and files with an allocated file size of 0.
FileChains	The number of contiguous cluster chains. On a completely unfragmented volume, this value would be identical to Files.
FreeChains	The number of contiguous cluster chains of free clusters. On a completely unfragmented volume, this value would be 1.
LargestFreeChain	The maximum allocated file size for a newly allocated contiguous file in clusters. On a completely unfragmented volume, this value would be identical to FreeClusters.

Fields FileChains and FreeChains are good indicators for the degree of fragmentation of the volume; the example program RTFCMD defines a *Fragmentation Percentage* as follows: if all files and all free clusters each reside in single cluster chains (the ideal case), the fragmentation is 0%. If the average number of file chains per file is 2 (where free space is counted as one file), fragmentation is 100%. Thus, the fragmentation percentage can be calculated as follows:

```
100 * (FileChains + FreeChains - (Files+1)) / (Files+1)
```

Of course, this is not a true percentage since the value can exceed 100%.

Function RTFGetPartitionInfo

RTFGetPartitionInfo returns information about a partition or physical disk:

```
int RTFGetPartitionInfo(const char * DriveName, RTFPartitionInfo * PartitionInfo);
```

Parameter DriveName must be a valid file name (for example, a root directory named "C:\"), a logical drive name ("\\.\C:"), or a physical disk name ("\\.\PHYSICALDRIVE0"). If the name refers to a physical disk file, the returned partition information applies to the whole disk, not just a single logical drive.

Parameter PartitionInfo must point to a structure RTFPartitionInfo declared in RTFILES.H:

```
typedef struct {
    RTFPartitionRecord Partition;
    DWORD               PartitionSector;
    int                 PhysicalDiskIndex;
    UINT                BytesPerSector;
    BYTE                MediaDescriptor;
    BYTE                Reserved;
    short               DeviceListIndex;
} RTFPartitionInfo;
```

Structure RTFPartitionRecord is defined as:

```
typedef struct {
    BYTE  BootIndicator, // 0x80 for bootable, 0 otherwise
    StartHead,           // 0 based
    StartSector,         // 1 based, bits 0-5,
    StartTrack,          // 0 based, bits 0-7, take bits 8,9 from StartSector
    OSType,              // FAT-12: 1, FAT-16: 4, 6, 14, FAT-32: 11, 12
    EndHead,             // see StartHead
    EndSector,           // see StartSector
    EndTrack;            // see StartTrack
    DWORD RelativeSector, // offset to first sector of partition data
                          // for primary partitions, this is the absolute
```

```
                // LBA of the boot sector
    Sectors;      // number of sectors in partition
} RTFPartitionRecord;
```

Please note that actual values must be calculated as follows:

```
ActualStartSector    = StartSector & 63;
ActualStartTrack     = StartTrack | ((StartSector & 0xC0) << 2);
ActualEndSector      = EndSector & 63;
ActualEndTrack       = EndTrack | ((EndSector & 0xC0) << 2);
```

For large hard disks, the EndTrack value may be incorrect, because it is limited to 1024 cylinders. If you need to calculate the size of a disk or partition, rely on field Sectors.

If the function succeeds, the return value is RTF_NO_ERROR and the structure is filled as described below. If the function fails, the return value is a negative error code.

Partition	This field contains a physical copy of the partition record read from the partition table to mount this partition. For floppy disks and physical disks, RTFiles-32 will create a fake partition record describing the whole disk.
PartitionSector	The physical sector number of the hosting device containing the partition table containing the above partition record. This value will be 0 for all primary partitions.
PhysicalDiskIndex	The index of the physical disk device hosting this partition, or -1 for a floppy disk. The first hard disk in the system has index 0.
BytesPerSector	The sector size of the disk.
MediaDescriptor	The media byte of the hosting device. This value will be F8h for hard disks and some other value read from the boot sector for floppies. If this value is 0, RTFiles-32 was unable to determine the media descriptor. This situation can occur when the boot sector of a floppy disk is unreadable.
DeviceListIndex	The zero-based index of the device hosting the partition in RTFiles-32's device list.

Function RTFSetVolumeLabel

RTFSetVolumeLabel writes or removes a volume label to/from a drive:

```
int RTFSetVolumeLabel(const char * DriveName, const char * Label);
```

Parameter DriveName must be a valid file name (e.g., a root directory named "C:\"). Only the name's drive information is evaluated. Parameter Label must point to the new label of up to 11 characters length, or it must be NULL to remove any existing label.

If the function succeeds, the return value is RTF_NO_ERROR. If it fails, the return value is a negative error code.

Function RTFFormat

Function RTFFormat formats a logical drive:

```
int RTFFormat(const char * DriveName,
              UINT        MinSectorsPerCluster,
              RTFFormatCallback Progress,
              DWORD        Flags);
```

Parameter DriveName must be a logical drive file name in the form "\\X:", where "X" is replaced with the drive letter of the logical drive to be formatted.

Parameter MinSectorsPerCluster specifies the minimum number of sectors per cluster RTFFormat should set up. This parameter should be 0 to use a default cluster size, 1 to always use the smallest possible cluster size, or any other power of two. Cluster sizes up to 64 sectors are compatible with other operating systems, 128 is compatible with Windows NT. RTFiles-32 supports cluster sizes up to 32768 sectors (extended cluster sizes), but such volumes cannot be mounted by any other OS. If the specified cluster size is too small, RTFiles-32 will automatically adjust it. This may become necessary on FAT-12 or FAT-16 drives to avoid exceeding the maximum supported number of clusters. For most applications, the default cluster size is recommended (set MinSectorsPerCluster to 0).

Parameter Progress is a callback to supply progress information:

```
typedef void (RTFAPI * RTFFormatCallback)(const char * DeviceName,
                                         int          Action,
                                         DWORD        Total,
                                         DWORD        Completed);
```

It will be called periodically by RTFFormat. Parameter Action can have one of the following values:

RTF_FMT_PGS_LOW_FMT Low-level formatting is in progress.

RTF_FMT_PGS_HIGH_FMT High-level formatting is in progress.

RTF_FMT_PGS_CLEAR_MEDIUM The data area of the drive is being deleted.

Parameters Total and Completed indicate how many sectors must be processed and how many have been processed successfully.

The Progress parameter for RTFFormat is optional and may be set to NULL.

Parameter Flags controls various options about how to format the device. It can have any combination of the following flags:

RTF_FPLY_DRIVE_360 These flags specify the type of floppy disk that should be formatted. If none of these flags is specified, the maximum capacity of the floppy disk drive is assumed. For example, if you want to format a 720k diskette in a 1.44M diskette drive, you must specify flag RTF_FPLY_DRIVE_720. If it is omitted, RTFFormat would attempt to format the diskette for 1.44M.

RTF_FMT_SINGLE_FAT RTFFormat should create only a single FAT. By default, two FATs are created. This flag is not compatible with all other operating systems. However, drives formatted with a single FAT have a larger capacity and better write performance. In particular, this flag should be used for Flash or RAM disks to improve performance and efficiency.

RTF_FMT_FORCE-
_LOW_LEVEL This flag forces RTFFormat to do a low-level format. By default, RTFFormat will do a low-level format only if writing to the device fails. Hard disks are never low-level formatted by RTFFormat. If you need to low-level format a hard disk, use the respective raw I/O functions.

RTF_FMT_NO_LOW_LEVEL Prevent RTFFormat from low-level formatting a device. If writing to the device fails, the function returns with an error.

RTF_FMT_FAT_12 Set up a FAT-12 file system.

RTF_FMT_FAT_16 Set up a FAT-16 file system.

RTF_FMT_FAT_32 Set up a FAT-32 file system.

RTF_FMT_NO_FAT_32 Set up a FAT-12 or FAT-16 file system, depending on the drive's size.

If none of the last four flags is specified, RTFFormat will select a FAT type automatically. Drives with less than 16M size will be formatted as FAT-12 and drives with less than 512M will be formatted as FAT-16. Larger drives are formatted as FAT-32.

Please note that some combinations of MinSectorsPerCluster and Flags can prevent RTFFormat from setting up a valid file system. For example, a FAT-16 drive must have at least 4085 clusters. However, if a drive has only 10000 sectors, MinSectorsPerCluster is set to 4, and flag RTF_FMT_FAT_16 is specified, RTFFormat will return error RTF_INVALID_FILE_SYSTEM.

RTFFormat does not perform a surface scan on the volume. Only if writing to any system portion of the drive fails, the function returns the respective error code.

If the function succeeds, the return value is the FAT type set up (12 for FAT-12, 16 for FAT-16, or 32 for FAT-32). If it fails, the return value is a negative error code. Demo programs RTFCMD and PartDemo use this function.

Miscellaneous File Functions

This section documents various other functions of RTFiles-32.

Function RTFCommitAll

RTFCommitAll will flush all currently dirty buffers to disk:

```
int RTFCommitAll(const char * DriveName);
```

Parameter DriveName must either be a valid file name (e.g., a root directory name) or NULL. If DriveName is not NULL, only the drive information (possibly the current default drive) is determined. All of that drive's hosting device's dirty buffers are flushed. If the parameter is NULL, the buffers of all devices are flushed.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code.

Function RTFCloseAll

RTFCloseAll closes all currently open files:

```
int RTFCloseAll(void);
```

This function also invalidates all of RTFiles-32's file handles which may be in use by the application. Subsequent use of such handles will fail.

Please note that this function does not close handles other than RTFiles-32's file handles. For example, the Win32 file I/O API emulation uses Win32 handles to access files. Open Win32 handles are not closed by this call.

If the function succeeds, the return value is the number of open files found and closed. If the function fails, the return value is a negative error code. In case of failure, some files may have been closed successfully, while others could not be closed.

Function RTFShutDown

RTFShutDown closes all currently open files and uninstalls all device drivers:

```
void RTFShutDown(void);
```

This function calls RTFCloseAll and then the ShutDown entrypoint for all devices in the device list.

No RTFiles-32 function calls will succeed after this call. RTFShutDown should only be called at program termination.

Not all embedded systems will have to use this function. It is only required if you must ensure that any changed interrupt vectors are restored after the program terminates.

This function does not return any information. Any errors encountered are ignored.

Function RTFErrorMessage

RTFErrorMessage returns a pointer to a message describing an RTFiles-32 error code:

```
char * RTFErrorMessage(int ErrorCode);
```

Parameter ErrorCode must be a negative value returned by a previous RTFiles-32 API call.

This function always succeeds. If the parameter specifies an invalid error code or a positive value, the returned string will indicate this.

Example:

```
RTFHANDLE Handle;
Handle = RTFOpen("somefile", 0);
if (Handle < RTF_NO_ERROR)
    printf("Unable to open file, reason: %s\n", RTFErrorMessage(Handle));
else
{
    RTFRead(Handle, ...);
}
```



```

    ...
    RTFClose(Handle);
}

```

Function RTFSetDefaultOpenFlags

RTFSetDefaultOpenFlags can be used to define flags to be applied to all subsequent calls to RTFOpen:

```
int RTFSetDefaultOpenFlags(DWORD GlobalFlags, DWORD LocalFlags);
```

Parameters GlobalFlags and LocalFlags can be any combination of flags specified for RTFOpen. GlobalFlags are ored into the Flags parameter of all subsequent RTFOpen calls. LocalFlags are only applied to RTFOpen calls from the task calling RTFSetDefaultOpenFlags. In a single-threaded environment, both flags are always ored into the Flags parameter.

RTFSetDefaultOpenFlags is useful for two purposes: globally changing open flags which affect caching/committing (e.g., RTF_COMMITTED, RTF_CACHE_DATA, RTF_LAZY_DATA) or to set flags for indirect calls to RTFOpen, such as through an emulated API. For example, ANSI C function fopen() does not support opening a committed file, but the same effect can be achieved with:

```

RTFSetDefaultOpenFlags(0, RTF_COMMITTED);
f = fopen("somefile.txt", "w");
RTFSetDefaultOpenFlags(0, 0);

```

Another use could be to protect directories from being opened accidentally:

```
RTFSetDefaultOpenFlags(RTF_OPEN_NO_DIR, 0);
```

If you want to explicitly open a directory, just specify RTF_OPEN_DIR to override RTF_OPEN_NO_DIR:

```
f = RTFOpen("c:\\adir", RTF_READ_ONLY | RTF_OPEN_DIR);
```

If RTFSetDefaultOpenFlags succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code.

Function RTFSetCriticalErrorHandler

RTFSetCriticalErrorHandler installs a critical error handler to be called by RTFiles-32 in case of a device error:

```
void RTFSetCriticalErrorHandler(RTFCriticalErrorHandler Handler);
```

Parameter Handler must be a function of the following type:

```

typedef enum { RTFRetry, RTFFail } RTFErrorAction;

typedef RTFErrorAction (RTFAPI * RTFCriticalErrorHandler)(char Drive,
                                                         DWORD SerialNumber,
                                                         int ErrorCode);

```

If the application never calls RTFSetCriticalErrorHandler, RTFiles-32 will fail all device errors. In other words, the error will be passed to the application as a return code of the RTFiles-32 API function in which the error occurred; if media has been removed or replaced, all files of that media are closed. This behavior is implemented with the default critical error handler RTFDefaultCriticalErrorHandler.

The critical error handler will be called by RTFiles-32 on every error reported by a device driver in a read or write operation. The handler is not called when the device is being mounted. Errors in this phase simply cause the device not to be mounted. Also, device error RTF_MEDIA_CHANGED detected while no files are open on a device simply cause the device to be remounted without calling the critical error handler.

When the critical error handler is called, RTFiles-32 will probably have unflushed data in its buffer cache. If the error cannot be recovered, data loss must be expected and the volume could be left in an inconsistent state.

The error handler receives parameters Drive, SerialNumber, and ErrorCode, which will contain the drive letter of the drive on which the error occurred, the serial number of the volume being accessed, and the RTFiles-32 error code which was reported by the device. The only valid return values are RTFFail or RTFRetry.

If the handler returns `RTFRetry`, RTFiles-32 will attempt to perform the I/O operation again. If it fails again (for the same or some other reason), the error handler will be called again. The only way to get out of this situation is by failing the operation (the handler would have to return `RTFFail`) or a retry succeeds. RTFiles-32 does not support ignoring errors as other file systems may.

If the handler decides to fail the operation by returning `RTFFail`, control passes back to the application. The current RTFiles-32 API function will return with the return code last passed to the critical error handler. If the error is one of the following,

```
RTF_MEDIA_CHANGED
RTF_WRONG_MEDIA
RTF_DRIVE_NOT_FOUND
RTF_DRIVE_NOT_READY
```

RTFiles-32 will assume that the media is no longer available and all data for this drive in the internal buffer cache is discarded. This may include dirty buffers; their data will be lost. In addition, all open files on the drive are closed without flushing any dirty data and the drive is unmounted. Please note that files closed in this way do not need to be closed by the application, but this is nevertheless recommended. In particular, if the files have been opened through a non-RTFiles-32 API, high level API file references may still need to be closed (e.g., Win32 file handles, FILE pointers, stream objects, etc.).

A critical error handler must be programmed very carefully. Some rules to be observed are given below:

- A critical error handler must not itself access the drive referenced by parameter `Drive`. It can access other drives, but this is not recommended to avoid recursive critical error situations.
- The critical error handler **must** return either value `RTFFail` or `RTFRetry`. It may not use any other mechanism to relinquish control. For example, it may not throw an exception which it does not catch within the context of the handler and it may not call `longjmp` to transfer to a scope built before the RTFiles-32 API function call.
- Typically, a critical error handler will prompt the user to make sure the correct diskette is in the drive, etc. In a multithreaded environment, it must be considered that the error handler can be called by any task performing file I/O. It can even be invoked by several tasks simultaneously. The application must make sure that the current task can communicate with the user safely (if required).

When a critical error handler is called in a multithreaded environment, the device on which the error occurred is locked and cannot be accessed by other tasks. Access to other devices, however, is not limited in any way.

Below is an example of a critical error handler, taken from the `RTFCMD` demo program. It shows how a critical error handler might look like in a multitasking environment:

```
RTFErrorAction RTFAPI MyCriticalErrorHandler(char Drive,
                                              DWORD SerialNumber,
                                              int ErrorCode)
{
    char Temp[4];
    if (RTKCurrentTaskHandle() != MainHandle)
        return RTFFail; // can't talk to the user in other tasks
    switch (ErrorCode)
    {
        case RTF_WRONG_MEDIA:           // we only want to
        case RTF_BAD_SECTOR:             // handle these errors
        case RTF_DATA_ERROR:
        case RTF_MEDIA_CHANGED:
        case RTF_SECTOR_NOT_FOUND:
        case RTF_ADDRESS_MARK_NOT_FOUND:
        case RTF_CRC_ERROR:
```

```

Wprintf(MainWindow, "RTFiles-32 Error on Drive %c:, "
           "Volume: %p: %s\r\n",
           Drive,
           SerialNumber,
           RTFErrorMessage(ErrorCode));
while (1)
{
    Wprintf(MainWindow, "Please enter 'F'ail or 'R'etry:");
    Temp[0] = '\0';
    WGets(MainWindow, Temp, 4);
    switch (toupper(Temp[0]))
    {
        case 'F':
            return RTFFail;
        case 'R':
            return RTFRetry;
    }
}
default:
    return RTFFail; // can't do anything about other errors
}
}

```

Function RTFDefaultCriticalErrorHandler

RTFDefaultCriticalErrorHandler is RTFiles-32's default critical error handler. It merely returns RTFFail. The application should never call this function. It should only be used by RTFSetCriticalErrorHandler to restore the default handler.

Function RTFCreateMasterBootRecord

RTFCreateMasterBootRecord creates a valid master boot record with a partition table for a hard disk:

```

int RTFCreateMasterBootRecord(void *                SectorBuffer,
                             const RTFPartitionRecord * DiskGeometry);

```

RTFCreateMasterBootRecord does not access any physical disk. Parameter SectorBuffer must point to a buffer of at least 512 bytes size. It will receive the new master boot record. Parameter DiskGeometry points to a partition record describing the complete hard disk. This information will usually be supplied by a device driver.

RTFCreateMasterBootRecord fills the supplied buffer with a master boot loader and a partition table containing exactly one primary active partition spanning the complete disk.

Please refer to function RTFGetPartitionInfo for more information about structure RTFPartitionRecord.

If the function succeeds, the return value is RTF_NO_ERROR. If the function fails, the return value is a negative error code. Demo program PartDemo uses this function.

Function RTFSplitPartition

RTFSplitPartition splits the last partition in a partition table into two partitions:

```

int RTFSplitPartition(void * MasterBootRecord, RTFSector Sectors);

```

RTFSplitPartition does not access any physical disk. Parameter MasterBootRecord must point to a buffer of at least 512 bytes size containing a valid master boot record with a partition table. Parameter Sectors specifies how many sectors to allocate to the currently last partition. It must be less than the number of sectors it currently has. All remaining sectors are then allocated to a new partition.

Please note that partition size is rounded down to full cylinders, so the actual number of sectors can be less than the specified value.

If the partition table already contains four partitions, the function will fail. If the function succeeds, the return value is a zero-based index of the created partition (a value between 1 and 3). If the function fails, the return value is a negative error code. Demo program PartDemo uses this function.

Function RTFCreateBootSector

RTFCreateBootSector creates a valid boot sector for a logical drive:

```
int RTFCreateBootSector(void          * BootSector,
                           const RTFPartitionRecord * Partition,
                           BYTE        MediaDescriptor,
                           UINT        MinSectorsPerCluster,
                           DWORD       Flags);
```

RTFCreateBootSector does not access any physical disk. Parameter `BootSector` must point to a buffer of at least 512 bytes. It will receive the boot sector data. Parameter `Partition` must point to a partition record describing the partition for which the boot sector is intended. If the boot record is to be written to a hard disk, this data must be identical to the respective partition record in the partition table. Parameter `MediaDescriptor` is the value to be written into the BIOS parameter block. RTFiles-32 never uses this value, but other operating systems may require it to mount a volume. MediaDescriptors are usually supplied by the device driver. Parameter `MinSectorsPerCluster` specifies the minimum number of clusters. Parameter `Flags` controls additional properties of the file system to set up. Please refer to the documentation of function `RTFFormat` for details about parameters `MinSectorsPerCluster` and `Flags`.

If the function succeeds, it returns the FAT type set up (12, 16, or 32). If it fails, the return value is a negative error code.

Raw I/O Functions

Raw I/O functions allow the application to directly call device driver functions. The first parameter of all of these functions is the zero-based index in RTFiles-32's device list. It specifies the device on which this operation should be performed. In this way, even devices which have not been mounted successfully or which contain no valid partition can be accessed.

Raw I/O functions can be used interleaved and in parallel with other, high-level I/O functions, but this is not recommended. Since no files are opened to use the raw I/O functions, no share protection is applied.

Function RTFRawMount

Function RTFRawMount calls the driver's `MountDevice` function:

```
int RTFRawMount(int DeviceIndex);
```

If the function succeeds, it returns the device's sector size. If it fails, the return value is a negative error code.

Function RTFRawSetMedia

RTFRawSetMedia informs RTFiles-32 whether a specific device is available or not:

```
int RTFRawSetMedia(int DeviceIndex, int Media);
```

When parameter `Media` is zero, all subsequent calls to any device driver function for this device will return `RTF_MEDIA_CHANGED` without attempting to actually access the device. Parameter `Media` unequal to 0 allows RTFiles-32 to physically access the device.

If the function succeeds, it returns `RTF_NO_ERROR`. If it fails, the return value is a negative error code.

Function RTFRawShutDown

Function RTFRawShutDown calls the driver's `ShutDown` function:

```
int RTFRawShutDown(int DeviceIndex);
```

If the function succeeds, it returns `RTF_NO_ERROR`. If it fails, the return value is a negative error code.

Function RTFRawRead

Function RTFRawRead calls the driver's `ReadSectors` function:

```
int RTFRawRead(int DeviceIndex, void * Data, RTFSector Sector, UINT Sectors);
```

If the function succeeds, it returns `RTF_NO_ERROR`. If it fails, the return value is a negative error code.

Function RTFRawWrite

Function RTFRawWrite calls the driver's WriteSectors function:

```
int RTFRawWrite(int DeviceIndex, void * Data, RTFSector Sector, UINT Sectors);
```

If the function succeeds, it returns RTF_NO_ERROR. If it fails, the return value is a negative error code.

Function RTFRawMediaChanged

Function RTFRawMediaChanged calls the driver's MediaChanged function:

```
int RTFRawMediaChanged(int DeviceIndex);
```

If the function succeeds, it returns RTF_NO_ERROR. If it fails, the return value is a negative error code.

Function RTFRawDiscardSectors

Function RTFRawDiscardSectors calls the driver's DiscardSectors function:

```
int RTFRawDiscardSectors(int DeviceIndex, RTFSector Sector, UINT Sectors);
```

If the function succeeds, it returns RTF_NO_ERROR. If it fails, the return value is a negative error code.

Function RTFRawGetDiskGeometry

Function RTFRawGetDiskGeometry calls the driver's GetDiskGeometry function:

```
int RTFRawGetDiskGeometry(int DeviceIndex,
                           RTFPartitionRecord * DiskGeometry,
                           BYTE * MediaDescriptor);
```

If the function succeeds, it returns RTF_NO_ERROR. If it fails, the return value is a negative error code.

Function RTFRawLowLevelFormat

Function RTFRawLowLevelFormat calls the driver's LowLevelFormat function:

```
int RTFRawLowLevelFormat(int DeviceIndex,
                          const char * DeviceName,
                          RTFFormatCallback Progress,
                          DWORD Flags);
```

If the function succeeds, it returns RTF_NO_ERROR. If it fails, the return value is a negative error code.

Functions for Debugging

This section describes functions provided for debugging and diagnostics purposes.

Function RTFBufferInfo

RTFBufferInfo returns information about RTFiles-32's internal buffer cache:

```
void RTFBufferInfo(RTFBufferStatistic * BufferInfo);
```

Parameter BufferInfo must point to a structure RTFBufferStatistic defined in RTFILES.H:

```
typedef struct {  
    DWORD TotalBuffers,  
          ValidBuffers,  
          DirtyBuffers,  
          MaxDirtyBuffers,  
  
          PhysicalBufferReads,  
          CachedBufferReads,  
          BuffersDiscarded,  
          CacheHits,  
  
          PhysicalBufferWrites,  
          CachedBufferWrites,  
          AsynchBufferFlushs,  
  
          UnbufferedReads,  
          UnbufferedWrites;  
} RTFBufferStatistic;
```

The counters contain the following values:

TotalBuffers	The number of available buffers. The default value is 32. See Chapter 5, <i>RTFiles-32 Data Tables</i> for information on how to change the number of buffers.
ValidBuffers	The number of buffers currently holding valid data.
DirtyBuffers	The number of buffers currently holding data which has not yet been written to disk. This value must be 0 when all files are closed.
MaxDirtyBuffers	<p>The maximum number of dirty buffers at any one time since program startup. This value should always be less than TotalBuffers. If it is not, consider increasing the number of buffers or avoid excessive use of device flag RTF_DEVICE_LAZY_WRITE or RTOpen flag RTF_LAZY_DATA.</p> <p>When all buffers are dirty, RTFiles-32 may fail file I/O operations with error RTF_OUT_OF_BUFFERS, since it is not always possible to flush buffers when a new buffer is required.</p>
PhysicalBufferReads	The number of sectors read from disk into a buffer since program start.
CachedBufferReads	The number of times RTFiles-32 required a sector of data and the required sector was found in the buffer cache (cache hits).
BuffersDiscarded	The number of times RTFiles-32 had to discard a valid buffer to accommodate a new sector. When a discarded sector is required at a later time, it must be reread from disk.
CacheHits	A percentage value indicating buffer cache efficiency. This value is calculated as $100 * \text{CachedBufferReads} / (\text{PhysicalBufferReads} + \text{CachedBufferReads})$.
PhysicalBufferWrites	The number of times RTFiles-32 has flushed a buffer to disk.
CachedBufferWrites	The number of times RTFiles-32 has modified the data in a dirty buffer. Such buffers are either still present in the cache as dirty buffers or they have been flushed at a later time.

AsynchBufferFlushs	The number of times a buffer was flushed to disk even though it had been modified in an earlier RTFiles-32 API function call. Such asynchronous buffer flushes will not occur if all files are opened with flag RTF_COMMITTED.
UnbufferedReads	The number of sectors read from disk, bypassing RTFiles-32's buffer cache. Such reads occur when the application reads large blocks of data containing complete sectors.
UnbufferedWrites	The number of sectors written to disk, bypassing RTFiles-32's buffer cache. Such writes occur when the application writes large blocks of data containing complete sectors.

Function RTFBufferInfo never fails and does not return a value. Please note that all counters start with 0 at program start.

Function RTFDumpFileTable

RTFDumpFileTable writes information about all currently open files as text to a string buffer:

```
void RTFDumpFileTable(char * Buffer, UINT BufferLen);
```

Parameter Buffer must point to a string buffer to receive the data. BufferLen specifies the size of the buffer. The buffer should have a size of at least 100 + n * 100, where n is the number of open files.

The buffer filled by this function can be displayed using printf to view the status of open files. Example:

Index	Handle	Flags	FilePos	Name
3	00070003	W_S	1234	C:\ADIR\SOMEFILE.DAT
4	00030004	RD_	256	C:\ADIR

Column Index contains the file slot the file occupies in RTFiles-32's internal file table. Handle is the hexadecimal representation of the file's handle. The Flags column may show a combination of the letters R, W, D, and S, representing read only access, read/write access, directory, and shared, respectively. FilePos gives the current file pointer value of the file. Name is the full path name of the file.

Use RTFDumpTable to analyze which files are open if you suspect that a program does not properly close all files.

Device Dependent Functions

The following functions interface directly with RTFiles-32 device drivers.

Function RTFFLPYTurnMotorOFF

This function should be called by an application at least once per second if the floppy driver is used with device flag RTF_CUSTOM_TIMER (see Chapter 5, *Floppy Disk Driver* for details):

```
void RTFFLPYTurnMotorOff(void);
```

It ensures that the floppy driver receives control to turn off the floppy motors when the motor timeout has expired.

Function RTFDrvFlashInfo

Function RTFDrvFlashInfo can be used to gather statistics about a flash disk:

```
typedef struct {
    DWORD   Blocks;
    DWORD   BlockSize;
    DWORD   EraseCountMax;
    DWORD   EraseCountMin;
    DWORD   EraseCountAverage;
    DWORD   SectorsInUse;
    DWORD   SectorsDeleted;
    DWORD   SectorsAvail;
} RTFFlashInfo;

int RTFDrvFlashInfo(const RTFDrvFlashData * D, RTFFlashInfo * FlashInfo);
```

Parameter D must point to the linear flash driver's data for the specific flash disk used in the device list. Parameter FlashInfo points to a structure to receive the results.

If the function succeeds, the return value is RTF_NO_ERROR and *FlashInfo is filled with statistics about the volume. Otherwise, a negative error code is returned.

Note that function RTFDrvFlashInfo directly calls the flash disk device driver and bypass RTFiles-32's file system locking. This function must never be executed while other threads are accessing the same flash disk volume.

Function RTFDrvFlashCompact

This function can be used to execute the flash disk driver's garbage collection without actually writing data to the disk:

```
int RTFDrvFlashCompact(RTFDrvFlashData * D, int RequestedSectors);
```

Parameter D must point to the linear flash driver's data for the specific flash disk used in the device list. Parameter RequestedSectors specifies the minimum number of sectors that should be available when the function returns. If the function succeeds, the return value is the number of available sectors on the flash disk. If more sectors are requested than can be freed, the maximum possible number of sectors is made available and returned, but no error is reported.

It is never necessary to explicitly call this function since the flash disk driver will execute its garbage collection automatically whenever data is written to the flash disk and not enough space is available. However, RTFDrvFlashCompact can be called before a write operation to speed up the write operation. If RTFDrvFlashCompact returns at least as many sectors as are to be written, RTFWrite (or any other disk write operation) is guaranteed not to erase any flash blocks during the write, allowing near real-time write performance.

Note that freeing the last few sectors of a flash volume may take an excessive amount of time and is therefore not recommended. The number of sectors that can be made available can be determined with function RTFDrvFlashInfo (return values SectorsDeleted + SectorsAvailable). For performance reasons, it is recommended to leave at least one erase block worth of sectors unused.

Note that function RTFDrvFlashCompact directly calls the flash disk device driver and bypasses RTFiles-32's file system locking. This function must never be executed while other threads are accessing the same flash disk volume.

Chapter 4

Alternate APIs for RTFiles-32

Many applications will prefer not to use RTFiles-32's native API directly, but instead use functions of the C/C++ run-time system or the Win32 API. In this way, programs can maintain compatibility with other operating system environments.

Win32 Emulation

When RTFiles-32 is used with RTTarget-32 2.1 or higher, RTFiles-32's installable file system feature is used to redirect all Win32 file I/O API calls to RTFiles-32. More information about configuring the installable file system for use with RTTarget-32 is given in Chapter 7, section *Win32 API Emulation with RTFiles-32 and RTTarget-32* and in Part I of this manual.

RTFiles-32 supports the following Win32 API functions:

CloseHandle	SetFileTime
FindClose	DeleteFileA
CreateFileA	MoveFileA
ReadFile	GetTempFileNameA
WriteFile	GetCurrentDirectoryA
SetFilePointer	SetCurrentDirectoryA
SetEndOfFile	CreateDirectoryA
FlushFileBuffers	RemoveDirectoryA
GetFileSize	GetDriveTypeA
GetFileTime	GetDiskFreeSpaceA
GetFullPathNameA	GetVolumeInformationA
GetFileInformationByHandle	SetVolumeLabelA
GetFileAttributesA	FindFirstFileA
SetFileAttributesA	FindNextFileA

RTTarget-32 Win32 Handles

One important concept of the Win32 API are handles. The number of available Win32 handles is limited but can be changed (please refer to Part I of this manual for details). It is very important that all handles are closed when no longer needed to avoid errors caused by an out-of-handle situation.

Win32 handles (type `HANDLE`) must not be confused with RTFiles-32's file handles (type `RTFHANDLE` or `int`). Win32 handles reference objects indirectly, while RTFiles-32 handles reference them directly. Consequently, Win32 handles can be duplicated with function `DuplicateHandle`; the associated file is closed only after **all** handles have been closed. In contrast, RTFiles-32 file handles cannot be duplicated; they can only be copied, which does not create a new handle. Any call to `RTFClose` will immediately close the file.

RTTarget-32 Flag `RT_CLOSE_FIND_HANDLES`

RTTarget-32 has a flag to work around a problem in some C/C++ run-time systems or older programs ported from DOS. The Win32 API functions to search for files are `FindFirstFile`, `FindNextFile`, and `FindClose`. `FindFirstFile` returns a handle which can be used in subsequent `FindNextFile` calls. This handle must be closed with `FindClose` to avoid a Win32 handle leak. However, under DOS, the corresponding DOS functions do not require closing such a handle. For this reason, some older programs never close find handles. RTTarget-32 allocates a Win32 handle in each `FindFirstFile` call and RTFiles-32 allocates a file handle in each call to `RTFFindFirst` (which is called by `FindFirstFile`). Thus, programs that do not close find handles will quickly exhaust all available Win32 and RTFiles-32 handles.

To work around this problem, RTTarget-32 can be instructed to automatically close find handles once no more files are found in `FindNextFile`. To use this feature, call

```
RTSetFlag(RT_CLOSE_FIND_HANDLES, 1);
```

during the startup phase of the program.

If this feature is used, programs which **do** close the find handle after no more files are found will receive an error from FindClose since the handle has been closed already. However, this should normally not cause any problems.

You should use RT_CLOSE_FIND_HANDLES only to support software with this bug which cannot be modified. Well-behaved programs should always close find handles (either Win32 handles with FindClose or the RTFiles-32 handle with RTFFindClose, if RTFiles-32's native API is used).

ANSI C Run-Time System Functions

Thanks to RTFiles-32's Win32 API emulation, standard ANSI C file I/O functions can be used. At least the following functions are available:

clearerr	fgetchar	fputc	fseek	remove	ungetc
fclose	fgetpos	fputchar	fsetpos	rename	unlink
feof	fgets	fputs	ftell	rewind	vfprintf
ferror	fileno	fread	fwrite	setbuf	vfscanf
fflush	fopen	freopen	getc	setvbuf	
fgetc	fprintf	fscanf	putc	tmpfile	

Many compiler vendors extend the number of available file I/O functions. Usually, the compiler-specific extensions will also work.

C++ iostreams

The iostream class library is built on the ANSI C file I/O functions. The only class which interfaces to the file system is filebuf. It is fully supported.

Mixing Different APIs

In principle, it is no problem to use functions from different API groups within the same program simultaneously or even to access a single file. However, great care must be taken not to mix the various methods to reference a file. For example, the ANSI C FILE stream functions use a pointer to a FILE structure to reference files, while RTFiles-32 uses integer handles.

Since all possible APIs are built upon one another, a higher level API must always remember the file reference of the API it builds on. The following table shows how to translate a file reference of one API to a file reference of the next lower API level.

Destination	Source	Operations
file descriptor	class filebuf	filebuf.fd()
file descriptor	struct FILE *	fileno(FILE *)
Win32 Handle	file descriptor	_get_osfhandle(file descriptor) (_os_handle() for Watcom C/C++)
RTFiles-32 Handle	Win32 Handle	RTParseHandle(Win32Handle, &Type, &RTFHandle);

Demo program FAPIDemo.CPP uses several different APIs to access a single file. It demonstrates how to translate file references between different APIs.

Chapter 5

Configuring RTFiles-32

This chapter describes how to configure RTFiles-32. This includes defining how much memory RTFiles-32 should use, which device drivers are available, and which system driver should be used to interface with the underlying operating system.

RTFiles-32 supplies defaults for the data tables and the device drivers, so many applications do not have to be concerned about these. However, you must always select the correct system driver depending on the platform the program will run on.

To change RTFiles-32's default configuration, replacement modules must be linked to the application as described below. It is important that these replacements are linked into the .EXE or .DLL file containing the RTFiles-32 library. Otherwise, RTFiles-32 will not see the replacement and continue to use its defaults.

RTFiles-32 Data Tables

RTFiles-32 needs memory to store information about currently available drives, open files, and for its buffers. RTFiles-32 does not contain any dynamic memory allocation internally, so these data areas must be statically defined in the program. By default, RTFiles-32 allocates enough space for up to eight open files, eight logical drives, and 32 sector buffers.

To change these defaults, include the following lines in one of the application's source files:

```
#define RTF_MAX_DRIVES    8
#define RTF_MAX_FILES     8
#define RTF_MAX_BUFFERS  32

#include <rtfdata.c>
```

and edit the respective constants to match your needs. If any of the given symbols is not defined, RTFiles-32 will apply its respective default.

Example: The application will need to open up to 30 files simultaneously. At least two file handles should be reserved, because RTFiles-32 may need them internally. To improve performance with so many open files, the number of buffers is increased to 256:

```
#define RTF_MAX_FILES     32
#define RTF_MAX_BUFFERS  256

#include <rtfdata.c>
```

To analyze an application's data needs, use functions RTFGetBufferInfo and RTFDumpFileTable.

The System Driver

RTFiles-32 needs a few system services to perform actions such as setting interrupt vectors, managing semaphores, etc. RTFiles-32 is portable and not tied to any specific system. Rather, a system driver must be supplied which maps RTFiles-32's system calls to the corresponding operating system's calls.

The system driver merely consists of a set of functions declared in INCLUDE\RTFSYS.H. To use a specific system driver, the driver's library file must be linked to the application. Please note that there is **no** default driver. You must link exactly one of the drivers listed below.

The following drivers are available for RTFiles-32:

RTFSRTT.LIB RTTarget-32 driver. This driver should be used with RTTarget-32 for single-threaded applications.

The time and date enquiry function is implemented using Win32 API function GetLocalTime. Please ensure that SetSystemTime is called at program startup; otherwise, the returned times will be undefined.

Time measurements and the delay function use function `GetTickCount`. The timer callback is implemented by hooking timer interrupt IRQ 0. Mutexes and semaphores are implemented as byte counters. When the driver must wait (e.g., at a semaphore or during a delay), function `RTWait` is called. For interrupt handling, functions `RTSetIntHandler`, `RTEnableIRQ`, `RTEnableInterrupts`, and `RTIRQEnd` are used. For TLS, the Win32 API is used. Fatal errors are displayed through `MessageBox` and the program is aborted with `ExitProcess`. DMA buffers are allocated by searching for a section name specified by the calling device driver.

The driver is supplied with complete source code in file `DRIVER\SYSRTT32.C`.

RTFSK32.LIB RTKernel-32 driver. This driver should be used in `RTTarget-32/RTKernel-32` applications.

The time and date enquiry function is implemented using Win32 API function `GetLocalTime`. Please ensure that `SetSystemTime` is called at program startup; otherwise, the returned times will be undefined.

Time measurements use `RTKGetTime()`. The delay function is implemented with `RTKDelay` for times larger than one tick. For shorter delays, a software loop timed with the high resolution timer is used instead. `RTKDelay(0)` is called in each loop iteration. The timer callback is implemented by a cyclic task running at priority 2. Mutexes and semaphores are implemented as RTKernel-32 mutexes and counting semaphores. For interrupt handling, functions `RTKSetIRQHandler`, `RTKEnableIRQ`, `RTKEnableInterrupts`, and `RTKIRQEnd` are used. For TLS, RTKernel-32's task user data is used. Fatal errors are displayed through `RTKFatalError`. DMA buffers are allocated by searching for a section name specified by the calling device driver.

The driver is supplied with complete source code in file `DRIVER\SYSRTK32.C`.

Device List

RTFiles-32 needs a list of devices with their associated drivers. Each available driver is implemented as a structure of function pointers to all driver entrypoints. To access a specific device, the drivers need a data structure for each device they should handle. In addition, RTFiles-32 must know whether the device is a floppy or hard disk and which options apply for the device.

RTFiles-32's driver list is a zero-terminated array of structures. Each structure defines everything RTFiles-32 needs to know about the device:

```
typedef struct {
    int         DeviceType;
    int         DeviceNumber;
    DWORD       DeviceFlags;
    RTFDriver    * Driver;
    void        * DriverData;
    RTFDeviceData DevData; // reserved for RTFiles-32 internal use
} RTFDevice;
```

The device list array used by RTFiles-32 is a global symbol:

```
extern RTFDevice RTFDeviceList[];
```

`DeviceType` can be either `RTF_DEVICE_FLOPPY` or `RTF_DEVICE_FDISK`. This information may be required by the device driver if the driver can support different device types. It is also used to determine whether the device has a partition table or not. Thus, you **must** use `RTF_DEVICE_FLOPPY` for all devices without a partition table, even if they are not floppies (e.g., a RAM disk).

`DeviceNumber` is a zero-based index of the device within its class. For example, floppy disk A has device number 0 and floppy B is device number 1. The exact meaning of the device number can depend on the device driver.

`DeviceFlags` can be used to select special options to be applied to the device. These options can either be general RTFiles-32 flags or device driver specific flags. The following device-independent flags are available and can be used with any device driver:

RTF_DEVICE_SINGLE_FAT	RTFiles-32 should use only the first FAT. Many devices are formatted to have two or even more copies of the FAT. Both DOS and RTFiles-32 will use only the first FAT, but all other copies are always updated on every change of the FAT. To improve performance, you can instruct RTFiles-32 not to update redundant FAT copies.
RTF_DEVICE_LAZY_WRITE	Usually, RTFiles-32 will flush all cached buffers of a file to the disk when the file is closed. However, when this flag is set, RTFiles-32 will wait until the last file of this drive is closed. This option can improve write performance when many files are created/deleted and/or extended simultaneously. However, the drive may be left in an inconsistent state if not all files are closed before the program terminates.
RTF_DEVICE_MOUNT_CONTIGUOUS	This flag instructs RTFiles-32 not to assign drive letters to logical drives the same way MS-DOS would. Instead, all of the device's drives are assigned after all other devices not specifying this flag, and all drives are assigned contiguous drive letters. The algorithm for assigning drive letters is given in Chapter 2, section <i>Mounting Devices and Logical Drives</i> .
RTF_DEVICE_REMOVABLE	This flag informs RTFiles-32 that the device is removable. It is not necessary to specify RTF_DEVICE_REMOVABLE for floppies; RTFiles-32 always assumes them to be removable. Removable devices are always assigned a drive letter and are not accessed until actually needed. RTFiles-32 will automatically remount such devices when they are hot swapped while no files are open on the device.
RTF_DEVICE_NO_MEDIA	This flag instructs RTFiles-32 to assume that this device is initially not present. RTFiles-32 will not attempt to access the device until RTFRawSetMedia(..., 1) has been called. You should specify this flag only for removable devices when the device driver is not able to detect media insertion by itself. For example, this flag is required for PCMCIA disks.
RTF_DEVICE_NEW_LOCK	RTFiles-32 maintains semaphores to lock devices to prevent two threads accessing the same device simultaneously. To support non-reentrant device drivers, no parallel access to any two devices managed by the same driver is allowed by default. Each driver (not device) is allocated its own lock. However, each device with this flag set will get a new lock, allowing parallel access with other devices of the same driver. Please consult section <i>Device Drivers</i> for details on which drivers support this flag.

Please refer to section *Device Drivers* for details about device driver specific options.

The Driver member of structure RTFDevice is a pointer to a disk device driver. The following device drivers are shipped with RTFiles-32; they are documented in detail later in this chapter:

RTFDrvFloppy	Device driver for 3.5" and 5.25" floppy disk drives. Media with 360k, 1.2M, 720k, 1.44M, and 2.88M are supported. DeviceType must be RTF_DEVICE_FLOPPY.
RTFDrvIDE	Device driver for standard IDE hard disks and IDE flash disks, including PCMCIA ATA and CompactFlash disks. DeviceType must be RTF_DEVICE_FDISK.
RTFDrvDOC	Device driver for M-Systems DiskOnChip 2000 or DiskOnChip Millennium flash disks. DeviceType must be RTF_DEVICE_FDISK.
RTFDrvSRAM	Device driver for PCMCIA SRAM memory cards. DeviceType must be RTF_DEVICE_FLOPPY.
RTFDrvFlash	Generic linear flash disk driver. DeviceType may be either RTF_DEVICE_FLOPPY or RTF_DEVICE_FDISK.
RTFDrvRAM	RAM disk driver. DeviceType may be either RTF_DEVICE_FLOPPY or RTF_DEVICE_FDISK.

RTFDrvNULL This is a dummy driver used to skip drive letters. DeviceType must be RTF_DEVICE_FLOPPY.

Member DriverData is device driver specific. Each driver defines a data structure, a pointer to which must be stored in this field. For many devices, the device data structure can be initialized to 0 to get the driver's default behavior. Please refer to the following sections for details about each driver's specific data.

RTFiles-32's default driver configuration looks like this:

```
static RTFDrvFLPYData FLPYDriveAData = {0};
static RTFDrvFLPYData FLPYDriveBData = {0};
static RTFDrvIDEData IDEDriveCData = {0};
static RTFDrvIDEData IDEDriveDData = {0};

RTFDevice RTFDeviceList[] = {
    { RTF_DEVICE_FLOPPY, 0, 0, &RTFDrvFloppy, &FLPYDriveAData },
    { RTF_DEVICE_FLOPPY, 1, 0, &RTFDrvFloppy, &FLPYDriveBData },
    { RTF_DEVICE_FDISK , 0, 0, &RTFDrvIDE,    &IDEDriveCData  },
    { RTF_DEVICE_FDISK , 1, 0, &RTFDrvIDE,    &IDEDriveDData  },
    { 0 }
};
```

Please note that this device list assumes that the RTTarget-32 configuration defines a Nothing section named FloppyDMA (see section *Floppy Disk Driver* below for details). Example:

```
Locate Nothing FloppyDMA LowMem 18k 64k ReadWrite
```

You can supply your own RTFDeviceList array to override RTFiles-32's default configuration. The following example would be used if you want to use only a single DiskOnChip:

```
static RTFDrvDOCData Disk0Data = {0};

RTFDevice RTFDeviceList[] = {
    { RTF_DEVICE_FDISK , 0, 0, &RTFDrvDOC, &Disk0Data },
    { 0 }
};
```

This example would also require an entry in your RTTarget-32 configuration file. Example:

```
Region DiskOnChip D0000h 8k Device ReadWrite
```

with a suitable address of the DOC's memory window.

The following example supports one diskette drive, one IDE hard disk on the motherboard's IDE controller, a second IDE hard disk to be mapped as the master drive in a PCMCIA slot, and a RAM disk of 8M maximum size:

```
static RTFDrvFLPYData A = {0};
static RTFDrvIDEData C = {0};
static RTFDrvIDEData D = {0};
static RTFDrvRAMData E = {8*1024*1024/512};

RTFDevice RTFDeviceList[] = {
    { RTF_DEVICE_FLOPPY, 0, 0, &RTFDrvFloppy, &A },
    { RTF_DEVICE_FDISK , 0, 0, &RTFDrvIDE,    &C },
    { RTF_DEVICE_FDISK , 2, RTF_DEVICE_REMOVABLE |
      RTF_DEVICE_NO_MEDIA |
      RTF_DEVICE_NEW_LOCK, &RTFDrvIDE,    &D },
    { RTF_DEVICE_FLOPPY, 0, 0, &RTFDrvRAM,    &E },
    { 0 }
};
```

This example also requires that a DMA buffer is allocated in RTTarget-32's configuration file.

The following example uses a single flash disk using RTFiles-32's proprietary linear flash disk driver:

```
static RTFDrvMTDFileData MTDDData = {"FLASH.BIN", 4*1024*1024};
static RTFDrvFlashData FlashDisk = { &RTF_MTD_File, &MTDDData };
```

```
RTFDevice RTFDeviceList[] = {
    { RTF_DEVICE_FDISK , 0, 0, &RTFDrvFlash, &FlashDisk },
    { 0 }
};
```

Please note that this last example is only intended to show the usage of an MTD driver. It cannot execute under On Time RTOS-32.

Device Drivers

This section describes characteristics and configuration options of the device drivers included with RTFiles-32.

RTFiles-32 uses a device list to find all available devices. The device list is an array of structures (type RTFDevice) named RTFDeviceList, where each structure describes a single physical device. Structure RTFDevice has the following layout:

```
typedef struct {
    int             DeviceType;
    int             DeviceNumber;
    DWORD           DeviceFlags;
    RTFDriver       * Driver;
    void            * DriverData;
    RTFDeviceData   DevData; // reserved for RTFiles-32's internal use
} RTFDevice;
```

This chapter describes how this structure must be filled for each driver, respectively. Device driver independent device flags in RTFDriver.DeviceFlags have already been discussed in section *Device List*, which also explains how to define a custom device configuration.

RTFDevice.DriverData is a void pointer; however, each driver defines a unique structure to which this field must point. Most drivers support this structure to be initialized to 0 to get the driver's default behavior. Other drivers may require additional configuration data. All these structures are declared in header file RTFILES.DRV in the Include directory and are explained in this section.

Some drivers may require various timeout values. Timeouts are always specified in milliseconds.

Floppy Disk Driver

The floppy disk driver requires a NEC uPD765A compatible diskette controller used in PC compatible systems at port address 03F0h - 03F7h connected to IRQ 6 and a 8237 compatible DMA controller, whose channel 2 is used for the floppy controller.

If the driver should automatically detect the installed drives and drive types, a BIOS CMOS RAM with valid diskette configuration data is also required.

DMA Buffer

The floppy driver uses DMA to transfer data between the computer's memory and the diskette controller. Some severe restrictions apply to such a buffer on PC compatible systems:

- The buffer must be located at a physical address below 16M.
- It may not span a 64k address boundary in the physical address space.
- The DMA hardware bypasses the CPU's MMU. Thus, the hardware accesses physical addresses while the driver must use virtual addresses.

The floppy driver will call a system driver function to allocate a DMA buffer. With RTTarget-32's and RTKernel-32's system drivers, the DMA buffer must be located explicitly in the application's RTTarget-32 configuration file with an entry such as:

```
Locate Nothing FloppyDMA <Region> 18k 32k ReadWrite
```

The name of this Nothing section must be "FloppyDMA". <Region> must be replaced with a physical (not virtual) RAM region located below 16M. The section must have a size of at least 512 bytes; however, for best performance, a larger buffer (such as 18k for a complete cylinder on a 1.44M diskette in the above example) is recommended. The buffer may not span a 64k boundary, which is achieved by specifying an alignment of at least the size of the buffer or 64k. The DMA hardware bypasses any CPU memory protection, but ReadWrite access is required to enable the driver to access the buffer. If you only need read-only access to floppies, you can also set it to ReadOnly.

The DMA buffer is used for all read and write operations. A large buffer allows more sectors to be transferred in a single operation, possibly improving performance. At most, a complete cylinder (2 tracks) of data can be transferred. Thus, it does not make sense to allocate a DMA buffer larger than 2 times the maximum number of sectors per track. For example, if you will be using 1.44M diskettes with 18 sectors per track, the maximum usable DMA buffer size would be $18 \times 2 \times 512 = 18k$. For 2.88M diskettes, it would be 36k.

The DMA buffer is also used for read-ahead operations. When only a single sector is read, the floppy driver will automatically round up the number of sectors to 4. If and how many sectors are read ahead can be controlled with device flags (see below). When sectors are read which are already present in the DMA buffer, disk access can be avoided. However, whether a read-ahead buffer improves performance depends on the application's characteristics.

If the driver does not find the "FloppyDMA" section at run time, any diskette access will return error RTF_DEVICE_RESOURCE_ERROR.

RTFDevice.DeviceType

RTFDevice.DeviceType must be set to RTF_DEVICE_FLOPPY for this driver.

RTFDevice.DeviceNumber

This field supports values 0, 1, 2, and 3, for drives A, B, etc. Please note that most PC compatible systems will only support 0 and 1 for drives A and B.

RTFDevice.DeviceFlags

Apart from device-independent device flags, the following flags can be specified for RTFDevice.DeviceFlags:

RTF_CUSTOM_TIMER This flag instructs the driver not to call the system driver to install a timer callback. The timer callback is required to turn the diskette motor(s) off after a timeout. The use of this flag is recommended when you already have a timer interrupt running or you have a low-priority cyclic task in a multitasking system. In this case, the application can periodically call function RTFFLPYTurnMotorOff(). This function should typically be called once per second. It requires very little CPU time and it never blocks.

When RTF_CUSTOM_TIMER is not set, the system driver will supply the callback. Under RTTarget-32, the timer interrupt is used for this purpose; for example, an Idle Handler called by RTWait would be a better solution (if supported by the application). The RTKernel-32 system driver will create a task with priority 2 for this purpose. The RTKernel-32 task handle is the public symbol RTFTimerTaskHandle. A dedicated task for this purpose is wasteful. If you already have a low priority cyclic task, use that instead.

RTF_MOTOR_TIMEOUT_1 This flag instructs the driver to turn the diskette motor(s) off after one second of no access. The default value is two seconds. Large values can improve performance, because turning the motor on requires an extra delay in the driver. On the other hand, head wear can be higher with long timeouts.

RTF_MOTOR_TIMEOUT_5	This flag instructs the driver to turn the diskette motors off after five seconds of no access.
RTF_MOTOR_TIMEOUT_10	This flag instructs the driver to turn the diskette motors off after 10 seconds of no access.
RTF_READ_AHEAD_0	Do not use the DMA buffer to read ahead sectors. By default, a read-ahead value of 4 is used (i.e., when the application reads only one sector, four contiguous sectors will be read instead).
RTF_READ_AHEAD_2	Use a read-ahead value of 2 sectors.
RTF_READ_AHEAD_8	Use a read-ahead value of 8 sectors.
RTF_READ_AHEAD_16	Use a read-ahead value of 16 sectors.

The floppy driver is not reentrant (this cannot be supported by the diskette controller) and therefore does not support device flag RTF_DEVICE_NEW_LOCK.

RTFDevice.Driver

This field must point to RTFDrvFloppy.

RTFDevice.DriverData

The floppy disk driver requires a pointer to a structure of type RTFDrvFLPYData for RTFDevice.DriverData with the following layout:

```
typedef struct {
    UINT DeviceType;
    RTF_FPLY_BIOS_Disk_Parameter * DPT;
    UINT DiskTimeout;
    UINT ContollerTimeout;
    UINT Retries;
    ...
} RTFDrvFLPYData;
```

DeviceType specifies the drive type. It can be initialized to 0 or any of the following values:

```
RTF_FPLY_DRIVE_360
RTF_FPLY_DRIVE_1200
RTF_FPLY_DRIVE_720
RTF_FPLY_DRIVE_1440
RTF_FPLY_DRIVE_2880
```

If set to 0, the driver will enquire the drive type from the BIOS CMOS RAM. Please note that this is only supported if the computer has a CMOS RAM and the drive has been correctly defined in the BIOS setup. You must **not** specify 0 for device numbers 2 and 3, since the BIOS only has information about diskette drives 0 and 1 (A and B).

DPT may be NULL or must point to a valid BIOS Parameter Block for the device. The floppy driver will supply a default BIOS Parameter Block which will work for most drives. If you know that your drives require special values here, please supply an alternate BIOS Parameter Block.

DiskTimeout, ContollerTimeout, and Retries default to 2000, 500, and 3 if set to 0. You can override the defaults by supplying non-zero values.

The other fields of RTFDrvFLPYData are for the driver's internal use and should be left uninitialized or initialized to 0.

IDE Hard Disk Driver

The IDE driver requires 1, 2, 3, or 4 IDE controllers (channels), each of which may have one or two devices attached. By default, the controllers are assumed to use the following resources:

Controller	I/O Addresses	IRQ
0	1F0h-1F7h, 2F6h-2F7h	14
1	170h-177h, 276h-277h	15
2	0F0h-0F7h, 2F6h-2F7h	11
3	070h-077h, 276h-277h	10

If the driver should automatically detect the installed drives and drive types, a BIOS CMOS RAM with valid hard disk configuration data is also required. BIOS CMOS RAM disk type look up is only supported for IDE disk numbers 0 and 1.

RTFDevice.DeviceType

RTFDevice.DeviceType must be set to RTF_DEVICE_FDISK for this driver.

RTFDevice.DeviceNumber

This field can assume values 0 to 7. Device 0 is the master device on the first IDE controller, device 1 is the slave on the first IDE controller, device 2 is the master device on the second IDE controller, etc. Please note that most PC compatible systems will only support 0 and 1 for a single IDE controller.

RTFDevice.DeviceFlags

Apart from device-independent device flags, the following flags can be specified for RTFDevice.DeviceFlags:

RTF_NO_CMOS_RAM	Instructs the driver not to attempt to query the BIOS CMOS RAM for the presence of the device. By default, the driver will check the CMOS RAM for device numbers 0 and 1 and return RTF_DRIVE_NOT_FOUND, if the device is not defined. For other device numbers or when this flag is set, the driver will attempt to access the device. If the device is not present, the operation will fail due to a timeout which may take several seconds (see InitTimeout in the driver's data).
RTF_16_BIT_IO	Instructs the driver to use 16-bit I/O instructions to communicate with the device (default).
RTF_32_BIT_IO	Instructs the driver to use 32-bit I/O instructions to communicate with the device.
RTF_NO_MULTI_SECTOR	By default, the IDE driver will transfer several sectors (typically 8 or 16) per command to/from the IDE device, depending on the disk's capabilities. If you experience compatibility problems, you can set this flag to disable multi-sector I/O.

The IDE driver is not reentrant within one channel/controller, but is reentrant for separate channels/controllers. Thus, device flag RTF_DEVICE_NEW_LOCK can be specified on every **even** device number.

RTFDevice.Driver

This field must point to RTFDrvIDE.

RTFDevice.DriverData

The IDE disk driver requires a pointer to a structure of type RTFDrvIDEData for RTFDevice.DriverData with the following layout:

```
typedef struct {
    void * ReadAheadBuffer;
    UINT ReadAheadBufferSize;
    UINT PortBase;
    UINT InitTimeout;
    UINT DiskTimeout;
    UINT ControllerTimeout;
    UINT IRQ;
    ...
} RTFDrvIDEData;
```

ReadAheadBuffer may be set to NULL if you do not want to use the driver's read-ahead feature. If it should be used, it should point to a buffer which has a size of 4 plus an integral multiple of the device's sector size (512). For example, if the read-ahead buffer should read up to 4 sectors, the size should be $4 + 4 * 512$. Field ReadAheadBufferSize contains the read-ahead buffer's size in bytes. Modern IDE disks will read ahead automatically, so RTFiles-32's read ahead buffer is not required. On older IDE controllers without cache, using it is recommended.

PortBase is the base I/O address of the IDE controller. If set to 0, the driver will use addresses 01F0h, 0170h, 00F0h, and 0070h for IDE controllers 1, 2, 3, and 4, respectively.

InitTimeout, DiskTimeout, and ControllerTimeout default to 10000, 5000, and 100 if set to 0. InitTimeout specifies the timeout for the Recalibrate command sent to the controller when the device is mounted. The IDE specification states that recalibration may take up to 2 minutes; however, we have not yet encountered any drives which will need more than a few seconds.

IRQ defaults to 14, 15, 11, and 10 (for controllers 0, 1, 2, and 3) if set to 0. If you want to specify IRQ 0, set this field to -1.

The other fields of RTFDrvIDEData are for the driver's internal use and should be left uninitialized or initialized to 0.

M-Systems DiskOnChip Driver

The M-Systems DiskOnChip flash disk driver supports up to two DiskOnChip 2000 or DiskOnChip Millennium devices with a maximum capacity of 166M each. Parts of the driver have been supplied by M-Systems. The driver's source code is not included with RTFiles-32, but it can be licensed from M-Systems (please contact On Time for details).

The DiskOnChip driver is not included in library RTFILES.LIB. Rather, library DRVDOC.LIB must be linked to be able to use this driver.

Memory Windows

DiskOnChips are configured to map a memory window of 8k or 32k into the host computer's address space. Typically, this address will be C8000h, D0000h, or D8000h. The DiskOnChip must know the address of this memory window and must have read/write access to it.

The driver will call a system driver function to find the memory window. With RTTarget-32's and RTKernel-32's system drivers, the memory window must be located explicitly in the application's RTTarget-32 configuration file with an entry such as:

```
Region DiskOnChip D0000h 8k Device ReadWrite
```

If a second DiskOnChip must be supported, another such region with name DiskOnChip1 is required. Example:

```
Region DiskOnChip1 D8000h 8k Device ReadWrite
```

The driver will search for regions with these names at program startup to find the installed devices. If the driver does not find the "DiskOnChip" region at run-time, any access to the device will return error RTF_DEVICE_RESOURCE_ERROR.

RTFDevice.DeviceType

RTFDevice.DeviceType must be set to RTF_DEVICE_FDISK for this driver.

RTFDevice.DeviceNumber

The device number of the first DiskOnChip device is 0, while the second device is numbered 1. Currently, only two DiskOnChip devices are supported simultaneously.

RTFDevice.DeviceFlags

This driver does not define any device-specific flags.

The DiskOnChip driver is reentrant and supports device flag `RTF_DEVICE_NEW_LOCK` for every device entry.

RTFDevice.Driver

This field must point to `RTFDrvDOC`.

RTFDevice.DriverData

This field must point to a unique structure of type `RTFDrvDOCData`. This structure should be initialized to 0 or be left uninitialized.

PCMCIA SRAM Card Driver

The SRAM driver implements a floppy disk like drive compatible with Microsoft Windows on a PCMCIA SRAM card. Up to two cards are supported in PCMCIA sockets 0 and 1. SRAM cards are memory-mapped into the target's address space using 4k aligned windows of at least 4k size. The addresses to be used for this mapping are determined by regions named *SRAMCard* and *SRAMCard1* for PCMCIA sockets 0 and 1, respectively. You must include such region declarations in your configuration file to use this driver. Example (also including the declarations needed for the PCMCIA controller):

Region	PCMCIA	D4000h	4k	Device	NoAccess
Region	CARDBUS	D5000h	8k	Device	NoAccess
Region	SRAMCard	DA000h	4k	Device	NoAccess
Region	SRAMCard1	DC000h	4k	Device	NoAccess

Note that PCMCIA SRAM cards do not contain a *CIS* (PCMCIA Configuration Information Space). Thus, the PCMCIA driver is not able to read any information from the card (e.g., vendor, card type, etc.). RTTarget-32 demo programs `PCCard` and `PCCardMT` assume that an SRAM card has been inserted whenever a card without a card function identification is found.

This driver directly interfaces with the PCMCIA driver of RTTarget-32 and can therefore only be ported to other operating systems with modifications.

An example device list which uses the SRAM driver is:

```
static RTFDrvSRAMData SRAM0 = {0};
static RTFDrvSRAMData SRAM1 = {0};

RTFDevice RTFDeviceList[] = {
    { RTF_DEVICE_FLOPPY, 0, RTF_32_BIT_IO |
      RTF_DEVICE_NO_MEDIA, &RTFDrvSRAM, &SRAM0 },
    { RTF_DEVICE_FLOPPY, 1, RTF_32_BIT_IO |
      RTF_DEVICE_NO_MEDIA |
      RTF_DEVICE_NEW_LOCK, &RTFDrvSRAM, &SRAM1 },
    { 0 }
}
```

Entries in the RTFiles-32 device list must be filled as follows for the SRAM driver.

RTFDevice.DeviceType

RTFDevice.DeviceType must be set to `RTF_DEVICE_FLOPPY` for this driver.

RTFDevice.DeviceNumber

The device number must be 0 for PCMCIA socket 0 and 1 for PCMCIA socket 1.

RTFDevice.DeviceFlags

This driver accepts flag `RTF_32_BIT_IO` or `RTF_16_BIT_IO` to instruct the driver to transfer data to/from the SRAM card using 16- or 32-bit instructions (*rep movsw* or *rep movsd*). By default, 8-bit (*rep movsb*) is used. Most PCI-PCMCIA controllers will support 32-bit access.

RTFDevice.Driver

This field must point to `RTFDrvSRAM`.

RTFDevice.DriverData

This field must point to a unique structure of type `RTFDrvSRAMData`:

```
typedef struct {
    RTFSector  Sectors; // card's capacity in 512 byte sectors
    ...
} RTFDrvSRAMData;
```

If `Sectors` is zero, the driver will attempt to determine the card's capacity automatically by reading the boot sector, or, for unformatted cards, through a memory test.

RAM Disk Driver

The RAM disk driver does not require any hardware. It uses function `malloc()` to allocate space to store files. The device configuration specifies the maximum size the RAM disk should have, but the driver will only allocate sectors as they are used. Thus, even if you request a RAM disk of 256M, for example, but write files with only 3M size to the RAM disk, only those 3M (plus some overhead for FAT and directories) will be allocated. When files are deleted, the space previously allocated to the files is deallocated with `free()`.

RAM disks format themselves automatically as an FAT-12 or FAT-16 volume when the disk is mounted. The application can reformat RAM disks as FAT-32, if desired. Any number of RAM disks can be installed.

The `RAMDisk` driver is reentrant as long as `malloc` and `free` can be called simultaneously from several tasks. Device flag `RTF_DEVICE_NEW_LOCK` is supported for each RAM disk.

RTFDevice.DeviceType

`RTFDevice.DeviceType` may be set to either `RTF_DEVICE_FLOPPY` or `RTF_DEVICE_FDISK`.

RTFDevice.DeviceNumber

This field is ignored and should be set to 0, even if several RAM disks are used.

RTFDevice.DeviceFlags

The RAM disk driver does not define any device dependent device flags. Please note that device-independent flag `RTF_SINGLE_FAT` is not required, because the RAM disk driver will always format the RAM disk with a file system with a single FAT.

RTFDevice.Driver

This field must point to `RTFDrvRAM`.

RTFDevice.DriverData

The RAM disk driver requires a pointer to a structure of type `RTFDrvRAMData` for `RTFDevice.DriverData` with the following layout:

```
typedef struct {
    DWORD Sectors;
    ...
} RTFDrvRAMData;
```

`Sectors` specifies the maximum size of the RAM disk, in sectors of 512 bytes. If set to 0, a RAM disk with a maximum size of approximately 4M is set up.

The other fields of `RTFDrvRAMData` are for the driver's internal use and should be left uninitialized or initialized to 0.

Linear Flash Driver

The linear flash disk driver implements a block device driver on directly addressable flash chips. The driver consists of two layers: The flash chip independent flash driver and the MTD (Memory Technology Driver), which is responsible for handling the flash memory (e.g., erasing, programming, mapping, etc.)

The linear flash disk driver maps 512 byte sectors onto flash blocks (also called *erase units*). Each block has a header, which is followed by the actual storage used for data sectors. The driver can handle flash devices with the following characteristics:

- Each erase unit has a size of at least 1024 bytes.
- At least two blocks are available to implement a disk device.
- At least 512 bytes and at least 1/128th of a block (whichever is larger) must be mappable into the computer's address space. Ideally, all of the flash memory is always directly accessible.
- When a block is erased, all bits are set to 1.
- Individual bits of the flash can be programmed from 1 to 0 without affecting other bits. However, changing bits from 0 to 1 is only possible by erasing a complete block (erase unit).

The linear flash disk driver performs active wear leveling and minimizes the number of erase and write operations. The driver records the number of erase cycles in the header of each block and it will not allow the difference between the highest and lowest erase count to become larger than 1000.

The number of sectors the driver can handle is limited to 2^{24} (16777216 sectors or 8 GB). The number of erase cycles the driver can record is also 2^{24} . Once all blocks of a volume have been erased 2^{24} times, no further wear leveling takes place, but the disk continues to function normally.

The driver has been designed to recover from interrupted write operations. For example, if the device is powered off while a sector is written to the flash, the previous sector's content is restored the next time the device is switched on. Every operation on the flash device can be rolled back to a consistent state. Note, however, that higher level FAT data structures might still be corrupted (e.g., an FAT update was completed successfully, but the respective directory entry was not written).

The data structures written to the flash memory by this driver are proprietary. Flash disks formatted with this driver cannot be used with any other operating system.

RTFDevice.DeviceType

`RTFDevice.DeviceType` may be set to either `RTF_DEVICE_FDISK` (recommended) or `RTF_DEVICE_FLOPPY`. However, once the flash disk has been formatted, this value must not be changed. If it is changed, the disk must be repartitioned and reformatted.

RTFDevice.DeviceNumber

The device number is ignored and should be set to 0.

RTFDevice.DeviceFlags

This driver defines the following device dependent flags:

- | | |
|--------------------------------|---|
| RTF_FLASH_NO_SECTOR_MAP | By default, the driver will allocate a sector map at program initialization using <code>malloc</code> . This sector map allows the driver to locate the physical location of a logical sector very quickly, but it does require 4 bytes of RAM per sector. For example, for a flash disk with a capacity of 4M, the sector map would require $4\text{M}/512 \times 4 = 32\text{k}$ of RAM. If low memory overhead is more important than disk I/O performance, specify this flag. |
| RTF_FLASH_NO_LOW_FMT | By default, the driver will automatically low-level format flash memory which contains no or invalid block headers. Use this flag to prevent such formatting. |

RTF_FLASH_NO_HIGH_FMT By default, the driver will automatically high-level format flash memory which contains no valid boot sector. Use this flag to prevent such formatting. The default formatting uses format flags **RTF_FMT_SINGLE_FAT** and **RTF_FMT_NO_FAT_32** with a default cluster size.

As long as no two flash disks share the same MTD or the MTDs used are reentrant, the flash disk driver is reentrant and supports device flag **RTF_DEVICE_NEW_LOCK** for every device entry.

RTFDevice.Driver

This field must point to **RTFDrvFlash**.

RTFDevice.DriverData

This field must point to a unique structure of type **RTFDrvFlashData**:

```
typedef struct {
    RTF_MTD * MTDDriver;
    void * MTDDData;
    ...
} RTFDrvFlashData;
```

Member **MTDDriver** must point to an MTD driver for the flash memory to be used. **MTDDData** points to the MTD driver's data. All other fields of this structure should be initialized to 0 or left uninitialized.

More Information about MTDs is available in Chapter 7, section *Flash Memory Technology Drivers (MTDs)*. The following section has a complete flash disk driver configuration example.

MTD Drivers **RTFMtdCFI2_8**, **RTFMtdCFI2_16**, and **RTFMtdCFI2_32**

These MTD drivers for the linear flash disk driver **RTFDrvFlash** support flash chips with the following characteristics:

- CFI compliant (Common Flash Memory Interface, a standard published by AMD),
- use CFI command set 2,
- 8 bits wide.

1, 2, or 4 such flash chips can be operated in parallel to form an 8, 16, or 32 bit wide flash bank. Driver **RTFMtdCFI2_8** can handle single chip 8 bit wide banks, **RTFMtdCFI2_16** handles 2 chip 16 bit wide banks, and **RTFMtdCFI2_32** handles 32 bit wide banks implemented by 4 chips. Each driver can handle any number of banks located in adjacent address regions of the target.

All three drivers share the same data type **RTFMtdCFI2Data** for their device data:

```
typedef struct {
    const char * RegionName;
    DWORD ReservedBlocks;
    DWORD EraseTimeout;
    DWORD EraseDelay;
    ...
} RTFMtdCFI2Data;
```

All fields may be initialized to zero for the driver's default behavior. **RegionName** can be set to the name of a Nothing entity declared in the program's configuration file. If no **RegionName** (NULL) is specified, name "FlashDisk" is assumed. The drivers determine the address and number of banks from the "Flash-Disk" Nothing entity. **ReservedBlocks** is the number of erase blocks at the end of the flash disk which will not be used. Such reserved blocks can be used as a boot device. However, no code can be executed from such a boot device while the flash disk is being written to. **EraseTimeout** and **EraseDelay** specify after how many milliseconds the driver should abort a block erase operation, or respectively wait until the block erase status is polled. If zero is specified, **EraseTimeout** is set to the value found in the CFI data structure (typically 10-15 seconds). The default value for **EraseDelay** is half of the typical erase time given in the CFI data (typically 0.5 seconds).

If the drivers find flash chips with erase block regions of varying block size, only the erase block region with the largest number of blocks is used.

Example: The AMD Élan SC520 Evaluation Board is equipped with eight AM29LV017D flash chips (2Mbits, 8 bits wide each), which satisfy the characteristics specified above. The chips are organized in two banks with four chips in each 32-bit wide bank.

The SC520 demos initialize these two flash banks to reside at addresses 30000000h and 30800000h in configuration file Sc520ini.cfg. The hardware declaration configuration file Sc520.cfg declares a region for the flash memory:

```
Region Flash 1G-256M 16M Device
```

The FlashDemo program's configuration file locates the required Nothing entity named "FlashDisk" into this flash region:

```
Locate Nothing FlashDisk Flash Flash.Size
```

The RTFiles-32 device list must include the linear flash disk driver which in turn must reference a suitable MTD. This configuration requires the RTFMtdCFI2_32 MTD driver:

```
#include <rtfiles.h>

// custom RTFiles-32 Device List
// we just want a single flash disk device configured as a hard disk
// the Eval Board uses 4 Flash (8 bits wide each) to implement a 32-bit
// wide device, so we use the RTFMtdCFI2_32 MTD driver.

// MT data
static RTFMtdCFI2Data MTDData = {0};

// Flash driver data, links to MTD driver
static RTFDrvFlashData FlashDisk = { &RTFMtdCFI2_32, &MTDData };

// Device List, pulls in flash driver
RTFDevice RTFDeviceList[] = {
    { RTF_DEVICE_FDISK, 0, 0, &RTFDrvFlash, &FlashDisk },
    { 0 }
};
```

The complete example executable on the AMD Élan SC520 Evaluation Board is called FlashDemo and is included with RTFiles-32.

NULL Device Driver

The NULL device driver does not implement access to any mass storage device. It is a dummy driver used to create gaps in the drive letter assignment. Any number of dummy drives using the NULL device driver can be used.

RTFDevice.DeviceType

RTFDevice.DeviceType must be set to RTF_DEVICE_FLOPPY.

RTFDevice.DeviceNumber

This field is ignored and should be set to 0, even if several NULL devices are used.

RTFDevice.DeviceFlags

The NULL device driver does not define any device dependent device flags. This field should be set to 0.

RTFDevice.Driver

This field must point to RTFDrvNULL.

RTFDevice.DriverData

The NULL device driver does not define any data structure. This field should be set to NULL.

Chapter 6

Demo Programs

This chapter briefly introduces the example programs included with RTFiles-32. They are intended to run under RTTarget-32 and are configured like the RTTarget-32 demos.

All RTFiles-32 demos are linked with module INIT.C, which contains an Init function to set some run-time options. For example, the Init function adds all installed memory to the program's heap and reads the date and time from the CMOS real-time clock. INIT.C is also a convenient place to add a custom device list or other configuration options.

Program HelloFiles

This very simple program shows how programs without any RTFiles-32-specific source code can use RTFiles-32. It uses the ANSI C file functions through the OS API emulation layer.

Program FAPIDemo

FAPIDemo shows how to use several different APIs within a single program. FAPIDemo is written in C++.

Program RTFCmd

The RTFiles-32 Command Processor is a fairly large program which uses most of RTFiles-32's native API. It implements a subset of the commands supported by Windows NT's command processor CMD or the MS-DOS shell COMMAND.COM (e.g., DIR, COPY, REN, CD, etc.). Type "HELP" or "?" to get a list of available commands.

Program RTFCmdMT

RTFCmdMT is a multithreaded version of RTFCmdMT and requires RTKernel-32. It allows spawning commands to be executed by separate threads in the background. RTFCmdMT displays the CPU load during all file I/O operations. Additional commands such as TASKS and INTS allow analyzing how much CPU time RTFiles-32's drivers need in interrupt handlers and in threads.

Program FlashDemo

This program demonstrates the use of the flash disk driver and the RTFMtdCFI2_32 Memory Technology Driver. This demo must be executed on the AMD Élan SC520 Evaluation Board.

Program DrvDemo

Program DrvDemo shows how to use a custom driver with RTFiles-32. The custom driver is included in source files DRVSIMPL.H and DRVSIMPL.C. It implements a simple RAM disk located in a contiguous region of memory on the target. Function RTFFormat is used to demonstrate formatting the device.

Chapter 7

Advanced Topics

This chapter discusses how to use various RTFiles-32 configuration options for different application requirements. RTFiles-32's default configuration is a compromise between good data security and good performance. Usually, improving one of them will compromise the other. Another aspect discussed here is real-time behavior, which may in turn require other configurations.

Optimizing for Best Throughput

The following parameters can have an impact on the average data throughput for file I/O.

- **A large number of buffers**
Ideally, all FAT data and all directory data required should fit into the buffer cache. For example, a FAT-16 partition with 40000 clusters, 256 root directory entries, and 5 medium-size directories (32 entries each) would require about $157 + 16 + 10 = 183$ buffers to hold this data. To have a few spare buffers for application files, 200 buffers would be a good value. For applications frequently accessing several logical drives, enough buffers for all drives should be allocated.

Please note that the only disadvantage of many buffers is the large memory requirement (approximately 512 bytes plus 40 bytes per buffer). RTFiles-32's buffer management is very efficient. Searching a large number of buffers incurs a negligible performance penalty.
- **Reading/Writing on sector boundaries**
Applications that keep file pointers aligned on sector boundaries and read and write full sectors bypass the RTFiles-32 internal buffer cache. The data is transferred directly between the device and the application's data area.
- **Reading/Writing large data blocks**
Most devices will exhibit superior performance when many sectors are read in a single driver call. For example, reading 18 contiguous sectors from a single track on a 1.44M diskette may take approximately 18 times as much time if RTFRead is called 18 times to read 512 bytes instead of a single call to read 9k. (However, if the diskette driver's read-ahead buffer feature is used, the performance penalty of single sector reads is not quite as high).
- **Avoid many seeks**
Seek operations are slow. Seeks can be avoided by reading/writing sequentially (i.e., avoiding calls to RTFSeek), using unfragmented files (see function RTFExtend), and not accessing several different files interleaved or simultaneously on the same device.
- **Driver read-ahead buffers**
Some drivers may support a read-ahead buffer. Such read-ahead buffers are not used when large data blocks are read, so they have no performance effect in such cases. If the application reads data in small blocks, but the reads are mostly contiguous in the file, the read-ahead buffers can significantly improve performance. The best size for such buffers is very difficult to predict and is best determined by tests.
- **RTFOpen flag RTF_CACHE_DATA**
This flag will have a positive effect on performance if file read accesses use small block sizes and frequent seek operations are performed. For sector-aligned accesses, it has no effect. If the file is read sequentially, performance may suffer if this flag is set as FAT and directory data will be pushed out of the buffer cache by application data which will never be used again.
- **RTFOpen flag RTF_LAZY_DATA**
This flag will have a positive effect on performance if file write accesses use small block sizes and frequent seek operations are performed. For sector-aligned accesses, it has no effect. If the file is written sequentially, performance may suffer if this flag is set.

- **Device flag RTF_DEVICE_LAZY_WRITE**
This flag will improve performance if many small files are frequently created, renamed, deleted, extended, etc. It does, however, compromise data security. If the program terminates abnormally, not all data may be flushed to the disk, leaving the volume in an inconsistent state.
- **Device flag RTF_DEVICE_SINGLE_FAT**
This flag will improve performance if many files are frequently created, renamed, deleted, extended, etc. However, the volume will be left with an invalid second copy of the FAT, causing disk checking utilities such as SCANDISK or CHKDSK to report errors. Nevertheless, the volume will be in a consistent state. The only disadvantage is that disk repair utilities will not be able to recover a volume when the primary copy of the FAT has been corrupted (something that only rarely succeeds in practice anyway).

Optimizing for Best Data Security

RTFiles-32's default configuration ensures that all data associated with a file is flushed to disk at the latest when the file is closed. The use of RTFOpen flag RTF_LAZY_DATA does not compromise this behavior, but device flag RTF_DEVICE_LAZY_WRITE does. Device flag RTF_DEVICE_SINGLE_FAT may prevent a corrupted primary FAT to be restored.

RTFOpen flag RTF_COMMITTED guarantees that a file is in a consistent state on disk after the successful completion of any RTFiles-32 API call, even when the file has not been closed. If files are written sector-aligned with large blocks, its performance penalty is acceptable. However, for write operations with small unaligned blocks, performance may suffer dramatically.

Real-Time File I/O

Real-Time file I/O means that the time needed by a read or write operation can be predicted, or at least an upper bound for this time can be guaranteed. It does not necessarily imply that the I/O operation must be fast, although many real-world applications will also require high speed.

To achieve real-time performance, a number of prerequisites must be satisfied:

- **Single file operation**
To avoid the need for the drive's head to seek to different areas on the disk, other files on the same device must not be accessed while real-time file I/O is in progress. The only exception could be files on physically different devices which do not share any resources with the device hosting the real-time file. For example, during a real-time read from an IDE drive, access to a diskette would be allowed.
- **Use of contiguous files**
Again, to avoid extra seeks, the file must reside in a single chain of clusters. Such files can be allocated using function RTFExtend. Please note, however, that RTFExtend itself is **not** real-time. Its time requirement depends heavily on the degree of fragmentation of the volume and the number and current state of the buffers.
- **FAT and directory data must be in the buffer cache**
RTFiles-32 must keep track of where the next read or write operation for a file will occur. The directory data and FAT data for the file is required for this. To avoid extra seek and read operations during real-time file I/O, this data must be in the buffer cache. The buffer cache must be large enough to hold this data and the data must have been explicitly loaded before the real-time access starts. RTFiles-32 API function RTFGetFileInfo guarantees to load this data into the buffers.
- **Access should be sector-aligned**
To avoid displacing sectors from the buffer cache and to best utilize the device's performance, data should be read and written sector-aligned (i.e., the file pointer should always be a multiple of the sector size) and the size of data blocks should also be multiples of the sector size.

- **The CPU must be fast enough**
The device driver must be able to keep up with the device to make best use of the device's throughput. This may not be possible for systems using a slow CPU and a fast hard disk. When a device driver misses a deadline (e.g., reacts too late when the device requests processing with an interrupt), a complete rotation of the disk may be required to restart the operation. Performance may suffer dramatically in such a situation.
- **I/O must execute at highest task priority**
If the real-time file I/O is performed in a multithreaded program, it must run at the highest priority of all tasks. If this is not the case, another task might get the CPU time when the device driver would need it, causing it to miss a deadline.
- **Device access must be error-free**
Most devices or device drivers will perform retries when read or write errors occur on a device. Usually, each retry will require at least one additional disk rotation.
- **The device must always be ready**
Some devices may periodically be unavailable. For example, a diskette needs an extra delay when its motor(s) must be started or a hard disk might perform internal recalibration at unknown and uncontrollable times.
- **Device access must be available**
If a device shares any resources with some other device, that other device should not be used in parallel. For example, a hard disk connected to a SCSI bus cannot sustain real-time performance if a tape streamer connected to the same bus is used at the same time.

If all of the conditions listed above are met, the worst case time for a read or write operation will require the following times:

The disk controller has to seek to the desired track. Since we are assuming contiguous access, a maximum of one track must be seeked, so we will assume the drive's track-to-track seek time.

The disk controller must wait for a specific mark on the disk to pass the read/write head. For example, diskettes have an index hole near the innermost track for this purpose. This operation can take up to one disk rotation.

The disk controller has to count passing sectors until the sector on the track at which the access should start reaches the read/write head. This operation can take up to one rotation.

Contiguous reading or writing is now possible until the head must move to the next cylinder. In a worst-case scenario, this will be only a single sector. However, a second seek operation will occur only after a complete cylinder is processed.

To seek to the next cylinder, the drive's track-to-track seek time plus 1 rotation is required to find the index hole.

Thus, the complete time can be calculated as:

StartupTime Time to find the first sector

Worst case: track-to-track seek time plus two rotations

SeekTime Number of seeks during the transfer

Worst case: $\text{Integer of } (\text{CylinderSize} + \text{DataSize} - 2 * \text{SectorSize}) / \text{CylinderSize}$

The time requirement per seek is the maximum of one rotation and the drive's track-to-track seek time.

TransferTime Time to read or write the data

$\text{DataSize} / \text{BytesPerTrack} / \text{RotPerSec}$ (rotations per second)

Example: A block of 4k size must be written to a 1.44M floppy disk. This diskette type rotates at six rotations per second, stores 9k per track, and has two tracks per cylinder. The track-to-track seek time is assumed to be equal to the time of one disk rotation (1/6 seconds). It is assumed that the drive's motor is still turned on from a previous disk access and that the read/write head is positioned one track before the track the first sector is written to:

StartupTime	$1/6 + 2 * 1/6$	0.50
Seek Steps	$(18k + 4k - 2 * 512) / 18k = 1$	
SeekTime	$1 * 1/6$	0.17
TransferTime	$4k / 9k / 6$	0.08
Total:		0.75

If 1M of data is written, the following times would apply:

StartupTime	$1/6 + 2 * 1/6$	0.50
Seek Steps	$(18k + 1024k - 2 * 512) / 18k = 57$	
SeekTime	$57 * 1/6$	9.50
TransferTime	$1024k / 9k / 6$	19.00
Total:		29.00

Using RTFiles-32 with RTTarget-32

Please ensure that the following conditions are met if you intend to use RTFiles-32 with RTTarget-32:

- The RTFiles-32 system driver to link is RTFSRTT.LIB. If you are also using RTKernel-32, use RTFSK32.LIB instead.
- If you are using floppy disks, be sure to allocate a DMA buffer for the floppy driver in your configuration file with:

```
Locate Nothing FloppyDMA <SomeRegion> 18k 32k ReadWrite
```
- If you are using M-Systems DiskOnChips, be sure to link library DRVDOC.LIB and to declare the DOC's memory window in your configuration file. Example:

```
Region DiskOnChip D0000h 8k ReadWrite
```
- You should link RTFILES.LIB and RTFiles-32's system driver library before RTTarget-32's library RTT32.LIB.
- The RTTarget-32 and RTFiles-32 libraries should be linked into the same module (.EXE or .DLL).

Win32 API Emulation with RTFiles-32 and RTTarget-32

When RTFiles-32 is used with RTTarget-32 2.1 or higher, about 30 Win32 file I/O functions are emulated by RTFiles-32. RTTarget-32 supports installable file systems and several file systems can be active at the same time. When RTTarget-32 is used without RTFiles-32, it will by default use its file system for consoles, LPTs, and RAM files. RTFiles-32, however, replaces RTTarget-32's default configuration to use console files, LPT files, and RTFiles-32's file system, but **not** RAM files. Here is the RTTarget-32 file I/O configuration used by RTFiles-32:

```
#include <rttarget.h>
#include <rtfiles.h>

static RTFileSystem Console =
{ RT_FS_CONSOLE, 0, 0, &RTConsoleFileSystem };

static RTFileSystem LPTFiles =
{ RT_FS_LPT_DEVICE, 0, 0, &RTLPTFileSystem };

static RTFileSystem FATFiles =
{ RT_FS_FILE | RT_FS_IS_DEFAULT, 0x7FFFFFFF, 0x00000FFF, &RTFilesFileSystem };

RTFileSystem * RTFileSystemList[] =
{
    &Console,
    &LPTFiles,
    &FATFiles,
    NULL
};
```

If, for example, you want to retain RTTarget-32's RAM files but exclude the LPT file system, you could include the following configuration in your program:

```
#include <rttarget.h>
#include <rtfiles.h>

RTFileSystem Console =
{ RT_FS_CONSOLE, 0, 0, &RTConsoleFileSystem };

RTFileSystem RAMFiles =
{ RT_FS_FILE, 1 << ('R'-'A'), 0, &RTRAMFileSystem };

RTFileSystem FATFiles =
{ RT_FS_FILE | RT_FS_IS_DEFAULT, 0x0F, 0x03, &RTFilesFileSystem };

RTFileSystem * RTFileSystemList[] =
{
    &Console,
    &RAMFiles,
    &FATFiles,
    (void*) 0
} ;
```

With this configuration, logical drive 'R' can be used to access the RAM files. Logical drives 'A' through 'D' are reserved for RTFiles-32.

For additional information about RTTarget-32's installable file systems, please refer to Part I of this manual.

Using RTFiles-32 with RTKernel-32

If you intend to use RTFiles-32 with RTTarget-32 and RTKernel-32, the following issues should be considered:

- The RTFiles-32 system driver to link is RTFSK32.LIB.
- Library RTFILES.LIB and RTFSK32.LIB should be linked before any RTKernel-32 and RTTarget-32 libraries.
- The CPU time required for file I/O with blocking device drivers (such as the floppy and IDE drivers) is very small. However, the achievable throughput can strongly depend on the priority of the task performing the I/O operation.
- The file I/O functions of the run-time systems may not be reentrant. Thus, you should either use the multithreaded run-time libraries or use RTKernel-32's library protection feature to protect these functions.

Implementing Custom Device Drivers

Each device driver must define three items:

- a structure of type RTFDriver containing the driver's functions,
- a unique structure type defining the data required for each device to be handled by the driver,
- and optionally, device-specific flags to control device options.

Custom drivers are used just like the drivers included with RTFiles-32. References to a custom driver can be placed in the RTFiles-32 device list to become effective.

Structure RTFDriver

Pointers to the driver's functions must be contained in a global variable of structure type RTFDriver declared in INCLUDE\RTFILES.DRV:

```
typedef struct RTFDriver {
    UINT Version;

    int  (RTFAPI * MountDevice)    (void * DriveData,
                                   int   DeviceNumber,
                                   int   DeviceType,
                                   DWORD  Flags);

    int  (RTFAPI * ShutDown)      (void * DriveData);

    int  (RTFAPI * ReadSectors)   (void * DriveData,
                                   RTFSector Sector,
                                   UINT  Sectors,
                                   void * Buffer);

    int  (RTFAPI * WriteSectors)  (void * DriveData,
                                   RTFSector Sector,
                                   UINT  Sectors,
                                   void * Buffer);

    int  (RTFAPI * MediaChanged)  (void * DriveData);

    int  (RTFAPI * DiscardSectors)(void * DriveData,
                                   RTFSector Sector,
                                   UINT  Sectors);

    int  (RTFAPI * GetDiskGeometry)(void * DriveData,
                                   RTFPartitionRecord * DiskGeometry,
                                   BYTE * MediaDescriptor);

    int  (RTFAPI * LowLevelFormat)(void * DriveData,
                                   const char * DeviceName,
                                   RTFFormatCallback Progress,
                                   DWORD  Flags);

} RTFDriver;
```

RTFAPI is defined as `__cdecl`.

Field Version must contain the RTFiles-32 version number (symbol RTFILE_VER). It is used to ensure that all drivers are recompiled when new RTFiles-32 versions are released. RTFiles-32 will issue a fatal error if it finds an out-of-date driver.

Function MountDevice is called when a device is mounted. This function may be called several times, in particular for removable devices. This function should perform all necessary initialization. The fields DriveData, DeviceNumber, DeviceType, and DeviceFlags from the device list are passed in parameters DriveData, DeviceNumber, DeviceType, and Flags. If successful, MountDevice should return the sector size of the device. If it fails, one of the negative errors codes defined in RTFILES.H should be returned.

Function ShutDown is called by RTFShutDown. It should bring the system into the state it had before any devices were mounted (e.g., restore interrupt handlers). Please note that ShutDown will be called unconditionally for all devices in the reverse order they appear in the device list, even for devices that were never mounted. The return code of ShutDown is currently not used by RTFiles-32. It should return RTF_NO_ERROR.

Function ReadSectors receives the device's device data, a zero-based sector index (LBA value), the number of sectors to read, and a pointer to the buffer to receive the data as parameters. ReadSectors should indicate success by returning value RTF_NO_ERROR or failure with a negative error code as defined in RTFILES.H.

Function WriteSectors receives the device's device data, a zero-based sector index (LBA value), the number of sectors to write, and a pointer to the buffer containing the data to be written as parameters. WriteSectors should indicate success by returning value RTF_NO_ERROR or failure with a negative error code as defined in RTFILES.H.

Function `MediaChanged` should check whether the volume in the drive has been removed or exchanged, if such a check is supported by the hardware. If the driver implements access to a fixed media device or the hardware is unable to detect media changes, the function should return `RTF_NO_ERROR`. If, however, a media change is detected, error code `RTF_MEDIA_CHANGED` should be returned. Please note that function `MediaChanged` is called only occasionally by `RTFiles-32`. `ReadSectors` and `WriteSectors` should check for media changes and return `RTF_MEDIA_CHANGED`, if detected.

Function `DiscardSectors` is called by `RTFiles-32` whenever a file is deleted or truncated to free any sectors formerly occupied by the file. This driver function may be used by RAM or flash disks to improve performance. `DiscardSectors` is optional and may be set to `NULL`.

Function `GetDiskGeometry` is called by `RTFiles-32` to inquire the size and other characteristics of a disk. This function is optional and may be set to `NULL`. However, if it is missing, `RTFiles-32` is not able to format such devices.

Function `LowLevelFormat` is called when a device must be low-level formatted. Low-level formatting should bring the volume into a state allowing sector-level reading and writing. Devices which do not require any low-level formatting should return `RTF_NO_ERROR`. If low-level formatting is required by a device type, but the driver does not support it, `RTF_UNSUPPORTED_DRIVER_FUNCTION` should be returned. This function is optional and may be set to `NULL`. Even if it is `NULL`, `RTFiles-32` can still high-level format devices handled by the driver.

To implement an `RTFDriver` structure, you must:

- implement the eight functions given above as static functions in a source file,
- declare a global variable of type `RTFDriver` and initialize it with pointers to these functions.

Please refer to the source code of the supplied drivers for examples. In particular, the example `DrvDemo` contains a very simple driver example in source files `DRVSIMPL.H` and `DRVSIMPL.C`. It is recommended to use this sample driver as a starting point to implement custom drivers.

Driver Device Data Structure

Most drivers will need some data to manage a device. Since a driver must be able to handle several devices, a unique data area is reserved for each device. The data must be statically allocated by the application and a pointer to the data must be placed in the `DriverData` field of the device list. It is passed in parameter `DriveData` to all driver functions.

The layout of the device data is arbitrary, but must be defined by the driver. Please see the documentation in the previous sections of this chapter and in include file `INCLUDE\RTFILES.DRV` for device data defined for drivers included with `RTFiles-32`.

Device-Specific Flags

A driver may define device flags to control options for the device. Such flags may be specified in the `DeviceFlags` field in the device list; they will be passed to the `Flags` parameter of `MountDevice`.

The lower 16 bits of the device flags are reserved for `RTFiles-32`'s device-independent flags. If a custom driver defines its own flags, bits 16 to 31 must be used.

Implementing Custom System Drivers

To implement a custom system driver, it is recommended to copy the source code of one of the drivers included with `RTFiles-32` and modify the copy for the intended system. As a starting point, you should select the existing driver most closely resembling your system. For example, if the target system supports multithreading and is 32-bit, you may want to start with the `Win32` driver or the `RTKernel-32` driver. For single-threaded systems, the `RTTarget-32` driver is better suited.

Header file `INCLUDE\RTFSYS.H` contains detailed documentation about the system driver and each function it should support. Please study it carefully before implementing your own driver.

The device drivers included with RTFiles-32 are intended to be portable. They will access the system driver for all system-dependent functions. Thus, they should be able to run on a new system without modification.

Flash Memory Technology Drivers (MTDs)

The linear flash disk driver does not know how to actually handle flash memory. Rather, it interfaces to an MTD driver to manipulate the flash. Thus, the same flash disk driver can support many different types of flash memory through different MTDs.

MTDs have a similar structure as RTFiles-32 device drivers. They consist of a structure with function pointers to the driver's various entrypoints:

```
typedef struct {
    DWORD   TotalBlocks;
    DWORD   BlockSize;
    UINT    WindowSize;
} RTF_MTD_FlashInfo;

typedef struct {
    UINT    Version;
    int     (RTFAPI * MountDevice)(void * DriveData,
                                   RTF_MTD_FlashInfo * FlashInfo);

    int     (RTFAPI * ShutDown)   (void * DriveData);
    void *  (RTFAPI * MapWindow)  (void * DriveData, DWORD BlockIndex,
                                   DWORD WindowIndex);

    int     (RTFAPI * EraseBlock) (void * DriveData, DWORD BlockIndex);
    int     (RTFAPI * ProgramData)(void * DriveData, void * Address,
                                   void * Data, UINT Length);
} RTF_MTD;
```

RTFAPI is defined as `__cdecl`.

Field Version must contain the RTFiles-32 version number (symbol `RTFILE_VER`).

For all driver functions, parameter DriveData is taken from the linear flash disk's data member `MTDDData`. The MTD must define its own data structure for its data. It should store all of its housekeeping data in this structure. The application should statically allocate such a structure for each flash disk using this MTD.

If nothing else is specified below, each MTD driver function should return `RTF_NO_ERROR` on success or a suitable error code on failure.

Function `MountDevice` is called when a device is mounted by the linear flash disk driver's `MountDevice` function. This function may be called several times. This function should perform all necessary initialization including locating the flash device. Parameter `FlashInfo` points to structure `RTF_MTD_FlashInfo`, which must be filled by `MountDevice`. `TotalBlocks` is the number of erase units to be used for this disk device. `BlockSize` is the size of each erase unit in bytes. `WindowSize` is the size of the memory window used to map portions of the flash memory into the computer's address space. The value must be at least 512 bytes and at least 1/128th of `BlockSize`. It must not be larger than `BlockSize`. For flash devices which are completely mapped into the computer's address space, specify `BlockSize` as the window size.

Function `ShutDown` is called by the linear flash driver's `ShutDown` function.

Function `MapWindow` is called whenever the linear flash disk driver needs to access portions of the flash memory. Parameter `BlockIndex` is in the range 0 .. `TotalBlocks-1`; `WindowIndex` is in the range 0 .. (`BlockSize/WindowSize - 1`). The return value must point to the mapped window and must allow read access to the mapped flash memory.

Function `EraseBlock` must set all bits of a complete erase unit to 1. Parameter `BlockIndex` is in the range 0 .. `TotalBlocks-1`. Window 0 of the block to erase is mapped when this function is called. The function should verify that erasing was successful and return an error if it was not.

Function `ProgramData` writes data to the flash. The linear flash disk driver will ensure that no bits are set from 0 to 1. Parameter `Address` is an address in the currently mapped memory window and indicates the address in the flash where the written data should appear. Parameter `Data` points to the data to write. Parameter `Length` specifies the number of bytes to write. Again, the function should ensure that writing was successful and return an error if it was not.

To implement an MTD driver structure, you must:

- implement the five functions given above as static functions in a source file,
- declare a global variable of type `RTF_MTD` and initialize it with pointers to these functions.

Please refer to the source code of the supplied drivers for examples. File `Driver\Rtf32\Mtdfile.c` contains a simple MTD driver simulating flash memory in a memory mapped file under Windows NT or Windows 2000.

Appendix A

RTFiles-32 Error Codes

This appendix lists and explains all error codes that can be returned by RTFiles-32.

The errors are sorted in numerical order. For each error, the numeric value, symbolic constant, and a description are given.

- 0 **RTF_NO_ERROR**
Not an error. This value indicates success of an operation.
- 1 **RTF_ERROR_RESERVED**
This error code is reserved and will not be returned by RTFiles-32.
- 2 **RTF_PARAM_ERROR**
The parameters passed to an RTFiles-32 function are invalid. For example, the flags passed to RTFOpen are contradictory or the size of a string buffer is too small.
- 3 **RTF_INVALID_FILENAME**
A device, directory, or file name passed to RTFiles-32 has an invalid syntax, contains illegal characters, or exceeds RTF_MAX_PATH (80) characters.
- 4 **RTF_DRIVE_NOT_FOUND**
The program attempted to access a logical drive which is not mounted.
- 5 **RTF_TOO_MANY_FILES**
The program attempted to open more files than slots were available in the file table. Changing the size of the file table is described in Chapter 5, *RTFiles-32 Data Tables*. Please note that this error can also be returned by functions other than RTFOpen and RTFFindFirst, since many other RTFiles-32 API functions need one (or two in case of RTFRename) file slots internally.
- 6 **RTF_NO_MORE_FILES**
This value is returned by RTFFindFirst and RTFFindNext when no more files satisfy the search criteria.
- 7 **RTF_WRONG_MEDIA**
A diskette has been replaced in a diskette drive, and the serial number of the new disk does not match the serial number of the original disk. To correct this error, the original diskette must be inserted or the operation must be failed.
- 8 **RTF_INVALID_FILE_SYSTEM**
The information found in the boot record of a logical drive is inconsistent and probably corrupted. The drive cannot be mounted.
- 9 **RTF_FILE_NOT_FOUND**
The requested file name was not found on the disk.
- 10 **RTF_INVALID_FILE_HANDLE**
A file handle passed to an RTFiles-32 API function is invalid. Either it has been closed already or it was not returned by a previous successful call to RTFOpen or RTFFindFirst, or it was closed automatically due to the removal of the media hosting the file.
- 11 **RTF_UNSUPPORTED_DEVICE**
A device in the device list specified a device other than RTF_DEVICE_DISKETTE or RTF_DEVICE_FDISK in the DeviceType field.
- 12 **RTF_UNSUPPORTED_DRIVER_FUNCTION**
This error is returned by device drivers or system drivers. For example, a device driver for read-only devices would return this error on attempts to write to the device.
- 13 **RTF_CORRUPTED_PARTITION_TABLE**
RTFiles-32 has found inconsistent values in a device's partition table. The device cannot be mounted.

-14 RTF_TOO_MANY_DRIVES

The number of logical drives found on all devices given in the device list exceeds RTFiles-32's internal drive table. Changing the size of the drive table is described in Chapter 5, *RTFiles-32 Data Tables*.

-15 RTF_INVALID_FILE_POS

A call to RTFSeek attempted to position the file pointer before the start of the file.

-16 RTF_ACCESS_DENIED

This error is returned whenever a security check fails. A few such checks are:

- Attempt to open a file for read/write access, but the read-only attribute is set.
- Attempt to open an already open file, and at least one of the file instances requires write access, and RTF_SHARED is not specified for all instances of the file.
- Attempt to create a file and the specified file already exists with the read-only attribute set.
- Attempt to open a volume label.
- Attempt to open a directory with read/write access.
- Attempt to open a directory with flag RTF_OPEN_NO_DIR.
- Attempt to delete a file with attribute read-only, volume label, or directory set.
- Attempt to rename a file across drives.
- Attempt to rename a volume label.
- Attempt to rename a directory to one of its child directories.
- Attempt to rename the current directory.
- Attempt to set attributes, date and time, or file size of a root directory.
- Attempt to change a volume label or directory attributes.
- Attempt to write to, truncate, or extend a file open for read-only access.
- Attempt to remove a directory which is not empty.
- Attempt to remove a directory with the read-only attribute set.
- Attempt to remove the root or the current directory.
- RTFResetDisk was called while files are open on the target drive.

-17 RTF_STRING_BUFFER_TOO_SMALL

The size of a string buffer passed to an RTFiles-32 function is too small to hold the result.

-18 RTF_GENERAL_FAILURE

A device driver reported an error without supplying additional information about the cause. For example, non-existing hardware or fatal hardware failures could generate this error.

-19 RTF_PATH_NOT_FOUND

The path for a file or a directory passed to RTFiles-32 was not found.

-20 RTF_FAT_ALLOC_ERROR

RTFiles-32 has found invalid values in a partition's FAT. The FAT is most likely corrupted and the partition must be reformatted.

-21 RTF_ROOT_DIR_FULL

It was attempted to create a new file in a root directory, but the directory is full. Unlike subdirectories, root directories have a fixed size and cannot be extended.

-22 RTF_DISK_FULL

It was attempted to extend a file or directory, but not enough free clusters to satisfy the request were found. For function RTFExtend, this error is returned when not enough contiguous clusters are found.

- 23 **RTF_TIMEOUT**
This device error is reported when a device fails to respond within a reasonable period of time.
- 24 **RTF_BAD_SECTOR**
A device driver has reported that a sector on the disk could not be read or written.
- 25 **RTF_DATA_ERROR**
A device driver has reported that a sector read from disk has failed a data integrity check. Typically, data is stored with CRC check sums which are used for such checks.
- 26 **RTF_MEDIA_CHANGED**
During a device access, the driver has detected that the media in the drive has been removed or exchanged.
- 27 **RTF_SECTOR_NOT_FOUND**
A device driver was not able to locate a requested sector. Either a corrupted boot sector or corrupted low-level formatting can cause this error.
- 28 **RTF_ADDRESS_MARK_NOT_FOUND**
The address mark normally written during a low-level format was not found during a disk read or write operation.
- 29 **RTF_DRIVE_NOT_READY**
A disk device does not respond, either because it is not present, the media is not inserted, or because of a hardware failure.
- 30 **RTF_WRITE_PROTECTION**
It was attempted to write to a write-protected media.
- 31 **RTF_DMA_OVERRUN**
A DMA controller has reported this error. This can happen when a DMA buffer spans a 64k address boundary.
- 32 **RTF_CRC_ERROR**
A CRC check failed during a device read or write operation.
- 33 **RTF_DEVICE_RESOURCE_ERROR**
A device driver has reported that some resource it requires is not available. For example, the floppy driver was unable to allocate a DMA buffer or the RAM disk driver was unable to allocate space for new sectors.
- 34 **RTF_INVALID_SECTOR_SIZE**
A device reported itself to be formatted with a non-standard sector size. Usually, FAT volumes should use sectors of 512 bytes size. Please contact On Time for information about how RTFiles-32 can support other sector sizes.
- 35 **RTF_OUT_OF_BUFFERS**
RTFiles-32 was unable to allocate a new buffer in its internal buffer cache. This situation can only occur if all buffers are dirty and none of the dirty buffers reside on the same physical device for which a new buffer is required. To avoid this error, increase the number of buffers, close some files, or flush buffers before the failing function is called.
- 36 **RTF_FILE_EXISTS**
This error is reported on an attempt to rename a file to an existing file name or create a directory with the name of an existing directory or file.
- 37 **RTF_LONG_FILE_POS**
This return code does not constitute an error. It is returned by RTFSeek when the new file pointer value exceeds $2^{31}-1$ (2G minus one). However, the function has succeeded when this value is returned. Use function RTFGetFileInfo to retrieve the actual file pointer.
- 38 **RTF_FILE_TOO_LARGE**
The application has attempted to extend a file to contain 4G or more bytes. However, FAT file systems only support file sizes up to FFFFFFFFh bytes. This restriction also applies to FAT-32 partitions.

Part IV

RTPEG-32

RTPEG-32 (Real-Time Portable Embedded Graphics) provides the building blocks for a powerful and extensible graphical user interface. Advanced clipping techniques, font support, graphic image support, and smart object methodologies are incorporated in RTPEG-32. In addition to the class library itself, RTPEG-32 provides all of the other tools, documentation, and support needed to construct custom graphical user interfaces.

RTPEG-32 is Swell Software Inc.'s product PEG ported to On Time RTOS-32.

This manual only contains a general introduction and programming tips for using RTPEG-32. The RTPEG-32 API class reference is supplied in HTML form for online viewing.

The main features of RTPEG-32 are:

- **True Windows 95 Look-and-Feel**
RTPEG-32 includes a full set of controls which look and behave the way users expect. Predefined classes include buttons, bitmaps, check boxes, scroll bars, menus, progress bars, radio buttons, prompts, combo boxes, dialog boxes, lists, tree views, etc.
- **Event Driven Programming Model**
User interface objects of an application are typically C++ classes derived from a predefined RTPEG-32 class. Such a derived classes can override methods such as Draw() to implement a custom appearance, or they can override Message() to catch messages sent by a user input device, separate threads, or other GUI controls.
- **Rich Set of Drivers**
A driver for VGA, 16 color mode, is included, as well as high performance drivers for 8-bit, 16-bit, 24-bit, and 32-bit color depth with arbitrary resolution. These drivers require VGA compatible hardware or VESA BIOS support. The source code of these drivers is included, allowing easy adaptation for custom video hardware.
- **Keyboard Support**
RTPEG-32 applications can be navigated using only the keyboard, if no mouse or other pointing device is available.
- **Mouse (MS Serial and PS/2) Support**
Standard Microsoft compatible mice are supported.
- **Never Disables Interrupts**
All RTPEG-32 operations are fully interruptable. Real-time performance is never affected.
- **Supports (But Does Not Require) RTKernel-32**
Several threads can perform screen output simultaneously. RTPEG-32 performs all required locking. Even the simultaneous execution of several modal windows is supported (e.g., to signal error conditions while other threads continue to run).
- **GUI Design Tools**
Program WindowBuilder is a rapid prototyping and design tool used to quickly create RTPEG-32 graphical objects such as bitmaps, fonts, windows, etc.
- **Win32 Emulation Library**
The RTPEG-32 Win32 Emulation Library allows executing RTPEG-32 GUI programs under Windows 95/98/ME/NT/2000.
- **Unicode Support (RTPEG-32 Source Code Required)**
The Unicode version of RTPEG-32 allows using fonts with up to 65535 characters, for example, to support Far East languages and multi-language programs.

Chapter 1

Overview

This chapter describes the relationships between objects that constitute a graphical user interface. Understanding these relationships is vital to making a user interface work efficiently.

Windowing Interface Terminology

This section introduces some terms used throughout this manual.

Window and Control

These terms are very loosely defined. They are only used for convenience when describing program operation as an alternative to itemizing long lists of actual class names. It is sometimes convenient to group the RTPEG-32 classes into two broad areas, the Window classes and the Control classes. This does not always imply that a Window class is derived from `PegWindow` or that a Control class is not derived from `PegWindow`, although that is often the case. The term Window implies only a background object that contains other objects. The term Control is used to refer to an object that is normally a child of a Window, or an object that the end user may interact with directly.

In RTPEG-32, there is actually very little difference between a Window and a Control. A Window can be a child of a Control, and a Control can contain a Window. Certain features, such as scrolling, are normally associated with Windows, while other features such as notification messages are normally associated with Controls. This does not mean that a control cannot scroll, or that a Window cannot send a notification message. These terms are only used to describe the general case. An application can modify and extend this general case at will.

The term Window refers to any PEG object that is normally used as a background container for other objects. Likewise, the term Control denotes the group of RTPEG-32 classes that most often do not contain other objects, and are generally used directly by the end user.

Parent, Child, Sibling

These terms refer to the relationships between the windows, controls, and other items that are all part the user interface. A control that is attached to a window is termed a Child of that window. Likewise, the window that contains the control is termed the Parent window. If there are several controls attached to the same window, those controls refer to each other as siblings.

While this is the most common case, there is nothing internal to RTPEG-32 that prevents a window class such as `PegWindow` from being the child of a control, such as a `PegButton`. In fact, it is often very useful to construct custom objects using exactly this type of parent-child relationship.

RTPEG-32 imposes no restrictions on the number of parent-child generations, nor will anything prevent an object that is a parent object in one case from becoming a child of another object in a different case. This allows reusing custom objects that are created in a variety of different ways.

Base, Derived, Inherited

The parent-child relationship described above is often confused with the class hierarchy, which describes a completely different relationship among the classes composing the graphical interface. Some of this confusion results from sloppy terminology, in that people often use the terms parent and child when what they are really referring to is base and derived.

The term Base or base class is a relative term, indicating the named class is the foundation for a class that is derived from it. A class that is called a base class in one case could easily also be a derived class, inheriting data and methods from an even more fundamental object.

Modal Execution

A window is said to be executed modally when that window must be closed or completed by the end user before other windows are allowed to receive any user input. This is most often used for executing a *Modal Dialog*, which is a dialog window that must be closed before any other open windows can receive user input such as a mouse click.

In RTPEG-32, any window can be executed modally. In fact, there can be several modal windows operating at one time in certain multitasking environments. Modal windows capture all input devices, preventing other windows and controls from being active while the modal window is executing.

Screen Coordinates

RTPEG-32 screen coordinates are in pixels relative to the upper left corner of the screen, which is 0,0. Screen coordinates are not relative to the client area of an object or its parent object.

Custom objects with custom drawing routines need to insure that they always use some corner of the object's client rectangle as the reference point for drawing routines.

Palettes and Colors

When running with a video configuration of 16 colors or less, RTPEG-32 always defines a fixed system palette that is programmed into the video controller palette registers. PegImageConvert supports advanced dithering techniques allowing you to make the best possible use of the limited number of colors when displaying high color images.

For 256 color systems, RTPEG-32 operates with a pre-defined fixed palette, or you may optionally generate and use a custom palette. The default system palette is defined such that the first 16 colors in this palette are identical to the fixed 16-color palette of VGA systems. The next 216 entries in the system palette are equally spaced color values covering the spectrum of RGB values from black (0, 0, 0) to white (256, 256, 256). The remaining 24 palette entries are reserved for future use.

For systems with more than 256 colors, *direct color* mapping is used. The color values used by an application are written directly into the video memory. The actual color to appear on the screen depends on how a pixel value is mapped to RGB values. For example, 16-bit color systems frequently use a 5:6:5 mapping (the upper 5 bits are red, the next 6 bits are green, and the low bits are blue). 24-bit and 32-bit color both use the 8:8:8 true color mapping. 32-bit color modes usually do not use the upper 8 bits of a pixel (0:8:8:8 color mapping).

Include file Peg.hpp defines a few predefined color values which will produce the same colors on any system with 16 or more colors:

BLACK	DARKGRAY
RED	LIGHTRED
GREEN	LIGHTGREEN
BROWN	YELLOW
BLUE	LIGHTBLUE
MAGENTA	LIGHTMAGENTA
CYAN	LIGHTCYAN
LIGHTGRAY	WHITE

Class PegScreen

Class PegScreen is the RTPEG-32 abstract device driver class that provides the drawing primitives used by the individual PEG objects to draw themselves on the display device. PegScreen provides a layer of isolation between the video hardware and the rest of the RTPEG-32 library.

PegScreen is an abstract class that defines the functions and function parameters instantiable PegScreen derived target specific interface classes must provide. Abstract class means the class contains one or more pure virtual functions which are placeholders in the class definition. They tell the compiler that all classes derived from the PegScreen base class must provide working versions of the virtual functions. In addition to the virtual functions, PegScreen also provides functionality that is common to all implementations.

RTPEG-32 is shipped with the following instantiatable classes derived from class PegScreen:

Driver Name	Creating Function	Comment
vga_4	CreatePegScreen_VGA_4	Supports graphics modes VGA 12h and VESA 102h (16 colors, 640x480 and 800x600).
vesa_8	CreatePegScreen_VGA_8	Supports all VGA and VESA 256 color modes. Linear frame buffer support is required if the required video RAM is larger than 64k.
vesa_16	CreatePegScreen_VGA_16	Supports all VESA 15/16 bit (32k/64k) color modes. Linear frame buffer support is required.
vesa_24	CreatePegScreen_VGA_24	Supports all VESA 24-bit (16M) true color modes. Linear frame buffer support is required.
vesa_32	CreatePegScreen_VGA_32	Supports all VESA 32-bit true color modes. Linear frame buffer support is required.
vgascrn	CreatePegScreen	Supports VGA mode 640x480, 16 colors on any VGA compatible graphics controller by programming this mode. This is the only driver which does not need a BIOS.

Class PegMessageQueue

PegMessageQueue is a simple encapsulated FIFO message queue with member functions for queue management. PegMessageQueue also performs timer maintenance and miscellaneous housekeeping duties.

Messages are placed in the message queue from one of three sources:

- An input device, such as a mouse, touch screen, or keyboard.
- Any other task in the multitasking system (via PegTask).
- A PEG object.

The messages placed in PegMessageQueue are the driving force behind the graphical interface. These messages contain notifications and commands which cause the graphical elements to redraw themselves, remove themselves from the screen, resize themselves, or perform any number of various other tasks. Messages can also be user defined, allowing you to send and receive a nearly unlimited number of messages whose meaning is defined by you. For example, it would be very common to have a task send data for display to a graphical element.

Messages

Messages are defined as simple structures containing fields indicating the source, target, and content of the message:

```
struct PegMessage
{
    PegMessage(void)          { Next = NULL; pTarget = NULL; }
    PegMessage(WORD wVal)    { Next = NULL; pTarget = NULL; wType = wVal; }
    WORD                    wType;
    SIGNED                  iData;
    PegThing                * pTarget;
    PegThing                * pSource;
    PegMessage              * Next;
    union
    {
        LONG                lData;
        PegRect             Rect;
        SIGNED              iUserData[4];
        WORD                wUserData[4];
        PegPoint            Point;
    }
};
```

```
void      * pData;  
};  
};
```

Messages are identified by the member field `wType`. Currently RTPEG-32 reserves the first 4000h message `wType` values for internal messages, which leaves message values 4000h through FFFFh available for user definition. `Peg.hpp` provides a `#define` indicating the first message value which is available for user definition. This `#define` is called `FIRST_USER_MESSAGE`.

Message Flow and Routing

RTPEG-32 follows a bottom-up message flow philosophy. This means that messages are sent directly to the lowest level object that should receive the message. If the object does not want to process the message, it passes it on to the message handler of its base class. Since all visible objects are derived from `PegThing`, unhandled messages will eventually arrive at the handler of `PegThing`, which passes the message to the parent of the current object. Again, the message can travel down the object's hierarchy, and eventually to the parent's parent, etc. Once `PegThing`'s handler finds it has no parent, the message is discarded.

Many messages, especially user defined messages, may be directed towards a particular object by the `pTarget` message field or by the message `iData` field. If `pTarget` is anything other than `NULL`, the message is always sent directly to the object pointed to by `pTarget`. Other messages do not have a particular object as their target. Examples of these message include mouse, touch screen, and keyboard messages. In these cases the `pTarget` member of the message is set to `NULL`, and it is the responsibility of `PegPresentationManager` to determine which object should receive the message.

When a user defined message is pulled from the message queue and it has a `pTarget` value of `NULL`, the message routing functions assume that the message `iData` field contains the ID of the object that should receive the message. This means there are two ways of directing user defined messages to particular objects. The `pTarget` field can contain an actual pointer to the destination object, which always takes precedence, or `pTarget` can be `NULL` and `PegPresentationManager` will route the message to the first object found with an ID value matching the message `iData` member. If you want to route user defined messages using object ID values, those objects should have globally defined object IDs to insure that there are never multiple objects visible with duplicate ID values.

Whenever an object sends a system-defined message to its parent window, the message contains a pointer to the object that sent the message. This pointer is contained in the message field `pSource`. This makes it very easy to identify the sender of the message and perform operations such as modifying the appearance of the object, interrogating the object for additional information, etc.

System Messages

System messages are used internally to command objects to perform certain operations. The definition of these messages is determined by RTPEG-32, and RTPEG-32 objects understand what to do when they receive various system messages.

In addition to defining custom messages, it is very common to want to receive and process the system messages that are generated internally. This is sometimes called *intercepting* a message, because it catches a message that RTPEG-32 has sent to an object and change the interpretation of the message, or even cause the object to ignore the message entirely.

User Defined Messages

A user interface will be composed of any combination of PEG windows, buttons, strings, etc., along with custom objects. At some point you will want to perform an action based on the user selecting a button, or typing into a string field. You are notified of this user input via messages sent from the PEG control to the parent window. When you create a control object, you tell the object what message to send back to the parent window when the object is modified by the user by defining the object ID value. Once you have constructed and displayed the control, you simply wait for the arrival of the message which indicates the control has been modified.

As an example, suppose the interface has two separate but related windows visible on the screen, Window A and Window B. Window A displays several data values, in alphanumeric format, that can be modified by the user. Window B displays these same data values as a line chart. When the user modifies a data value in Window A, we want Window B to update the line chart to reflect the new value. One way of accomplishing this is to define a new message which contains the altered data value. When Window A is notified by one of its child controls that a value has been changed, it builds an instance of the newly defined message, places the data value into the message, and sends the message to Window B. When Window B receives the message, it realizes that the line chart should be redrawn using the new data value.

When defining custom messages, using enumerations is recommended with the first enumeration equal to `FIRST_USER_MESSAGE`. For example, the following class declaration includes the typical method of defining custom messages:

```
Class MyWindow::public PegWindow
{
    public:
        enum ThisWindowMessages // my user defined messages
        {
            TURN_BLUE = FIRST_USER_MESSAGE,
            TURN_GREEN,
            TURN_INVISIBLE
        };
}
```

The above example illustrates a common way to define your own messages. We have implied in this example that you can re-use message values over and over again since the message number enumeration is a member of the class, rather than a global enumeration. This is in fact the case. As long as the object receiving the message can clearly identify what the message means you don't have to worry about reusing the same message numbers at various points in your application.

There are three ways to send a message from one window to another. First, you can either call the destination window's message handling function directly, passing your message as a parameter. Second, you can load the message's `pTarget` field with the address of the window (or any object) that should receive the message and push the message into `PegMessageQueue`. Finally, you can load the message's `pTarget` field with `NULL`, the message `iData` member with the ID of the target window, and push the message into `PegMessageQueue`. The second or third methods are generally preferred, because they adhere to the encapsulation philosophy.

With `pTarget` pointing to an application object, it must be ensured that the object is not deleted before the message arrives. When a user defined message contains a non-`NULL` `pTarget` value, there is no verification that the `pTarget` field of the message is a valid object pointer. For this reason, in some situations it is better to use `NULL` `pTarget` values, and route messages using object IDs. If `PegPresentationManager` is unable to locate an object with the indicated ID, the message is simply discarded.

There are also differences between these methods in terms of the order in which things are done. When a message is sent, the sending object immediately continues processing, and the target window will receive and process the new message after the sending window returns from message processing. If the receiving window's message handling function is called directly, it will immediately receive and process the message, in effect pre-empting the current execution thread. While these differences are generally inconsequential for user defined messages, they can be very important for RTPEG-32 system messages.

Signals

Messages are used to issue commands or send other information between objects that are part of the user interface. The previous section explained how a user defined messages notifies a parent window that a child control has been modified. This usage is so common, in fact, that RTPEG-32 has defined a simplified method for defining these messages and a corresponding syntax for receiving them. This method is called Signalling, and the messages sent and received via Signalling are called Signals.

Signals are designed to simplify programming by reducing the complexity associated with windows and dialogs containing a large number of child controls. Signals are also defined to solve some common problems associated with other less friendly messaging systems:

- Very often a single window, such as a modal dialog window, will have a large number of child objects. It can be very difficult to remember all of the unique messages associated with each of these objects.
- Complex control types, such as PegString or PegComboBox, can be modified in several different ways. The result of this is that either multiple message types must be sent by the control to the parent window, or the receiver of a single notification message would have to further interrogate the control to determine exactly why it sent a message.
- Although a control may define several different types of modification, you may not be interested in every type of control modification that can occur. In that case, you do not want the control to waste processing time by generating messages you are not interested in.
- To facilitate the implementation of a RAD window prototyping tool such as PegWindowBuilder, a consistent, simple, and robust message definition method must be in place.

Basically, signalling is nothing more than allowing an object to automatically generate and use multiple user defined message types based on a single *object ID* value. Every object using signals must have an object 'ID' value, which can and should be an enumerated ID name.

RTPEG-32 defines many different signals that can be monitored for each control. Whenever the control is modified by the user, the control checks to see if it is configured to notify its parent of the modification. If so, the control generates a unique message number based on the control ID and the type of notification. The message source pointer is loaded to point to the control, and the message is then sent to the parent window or dialog.

To receive a signal, RTPEG-32 defines the SIGNAL macro, which is used in the parent window message processing function. The parameters to the SIGNAL macro are the object ID and the notification message number. The SIGNAL macro is a shorthand method for determining the exact message number sent by a control with a given ID and corresponding to one of the 32 possible notification types.

The example below illustrates the use of signals in the definition and run-time processing of a typical dialog window. Since we have not yet discussed all of the information you need to know to fully understand this example, we don't want you to be concerned with the details (in fact, a lot of the details have been left out). Instead, the syntax for defining the control object IDs for each of the dialog controls and the message processing statements corresponding to each control is shown.

A few object IDs are reserved to facilitate the proper operation of modal dialog and modal message windows. The reserved object ID values occupy the upper range of the possible ID values. Custom ID values should start at 1.

```
Class MyDialog : public PegDialogWindow
{
    private:
        enum MyChildControls
        {
            USER_NAME = 1, // string control ID
            HAS_EMAIL,      // check box ID
            EMAIL_ADDRESS,  // email address string ID
        };
};

SIGNED MyDialog::Message(const PegMessage &Mesg)
{
    switch (Mesg.wType)
    {
        case SIGNAL(USER_NAME, PSF_TEXT_EDITDONE):
            // add code for user name modification here:
            break;
        case SIGNAL(USER_NAME, PSF_FOCUS_RECEIVED):
            // add code here to bring up help for user name:
```

```

        break;
    case SIGNAL(EMAIL_ADDRESS, PSF_TEXT_EDITDONE):
        // add code for email address change here:
        break;
    case SIGNAL(HAS_EMAIL, PSF_CHECK_ON):
        // add code for checkbox turned on:
        break;
    case SIGNAL(HAS_EMAIL, PSF_CHECK_OFF):
        // add code for checkbox turned off:
        break;
    default:
        return PegDialogWindow::Message(Mesg);
    }
    return 0;
}

```

Class PegPresentationManager

PegPresentationManager keeps track of all of the windows and sub-objects present on the display device. In addition, it maintains which object has the input focus (i.e. which object should receive user input such as keyboard input), and which objects are on top of other objects. There is no limit to the number of windows and controls and other objects that may be present on the screen at any one time.

PegPresentationManager keeps track of all of those windows and their children and grandchildren through the use of tree structured lists. Intrinsic to the design of RTPEG-32, all objects that can be displayed are derived from the common base class PegThing, described further below. Two important members of PegThing are a pointer to each PegThing's first child object, and a pointer to each PegThing's next sibling. Using these two pointers, PegPresentationManager maintains all objects in lists.

PegPresentationManager is also derived from PegWindow, which is derived from PegThing. This means that PegPresentationManager is more or less just another window, although in this case the window has no border, can often appear invisible, and always fills the entire screen or display. In essence, PegPresentationManager is the great-great-grandfather of all widows, dialogs, and controls.

Event Driven Programming

RTPEG-32 is message-driven, which may also be called event-driven. This means that real processing is only done in response to messages received from the outside world. No callbacks are used. One PEG object can communicate easily with another without worrying about how to physically address that object.

PegPresentationManager supplies the overall control of RTPEG-32 applications. PegPresentationManager::Execute() (called by PegExecute()) is the main execution loop for the GUI interface. In multi-threaded programs, other threads can run in parallel to PegPresentationManager::Execute(). PegPresentationManager::Execute() blocks itself on the internal message queue when no further messages are available for processing.

Input Focus Tree

An additional task of PegPresentationManager is message routing. Many system messages, such as mouse and keyboard input messages, are not directed to any particular object. For this reason, PegPresentationManager internally maintains a pointer to the object that was last selected by the user using the mouse or other input means. This object is called the *current* or *input* object, meaning that by default this object will receive input messages.

PEG views each displayed window and child objects of each window as branches in a tree. When input focus moves from object to object, PegPresentationManager insures that the entire branch of the tree up to the actual input object has input focus. Whether an object is a member of the input focus branch of the presentation tree can be detected at run-time time by testing the PSF_CURRENT system status flag:

```
if (StatusIs(PSF_CURRENT))
{
    // this object is in the branch
    // of the display tree that has input focus.
}
```

Just because an object is a member of the input focus tree does not mean the object is the end or leaf of the input focus branch. A pointer to the final input object is returned by `PegPresentationManager::GetCurrentThing()`.

An application can override the user's input selection and manually command `PegPresentationManager` to move the input focus by calling the `PegPresentationManager::MoveFocusTree()` function. `MoveFocusTree()` will set input focus to the indicated object by sending `PM_NONCURRENT` messages to objects that are no longer members of the input focus branch, and `PM_CURRENT` messages to objects that are members of the new input focus branch. The effect is that non-directed input messages will be sent to the newly designated input object.

When a new window is added to `PegPresentationManager`, that window automatically receives input focus. Likewise, if that window has any child objects, the first child object of the window receives focus. This continues until a leaf node (an object with no children) is found. The `Add()` function is most commonly used to add child objects to a window. Since the last object added to a window becomes the first child of the window (unless `AddToEnd()` is used to add the object), the last object added to a window will have input focus when the window is first displayed.

Keyboard Input Methods

Keyboard input arrives in the form of `PM_KEY` messages, meaning that the `Message.wType` field == `PM_KEY`. The actual key value is passed in the `Mesg.lData` field, and the key flags such as shift key state, control key state, etc., are passed in the `Mesg.lData` parameter.

As described above, when a window is added to `PegPresentationManager`, the first client-area child object of the window gains input focus. The end user can fully navigate through a PEG application by sending a very small number of `PM_KEY` messages to PEG. The list below describes the key values that PEG objects monitor to allow the user to navigate through the graphical interface.

PK_TAB The focus is moved to the next child of the current window. If the current child is the last child, focus wraps back to the first child of the current window. If the Shift key is pressed (i.e. the `KF_SHIFT` flag is set in the `PM_KEY` message `lData` member), the tab direction is reversed.

<Ctrl> PK_TAB Cycles the focus through top level windows.

F1 Moves focus to the first menu item of the menu bar of the current window.

PK_LNUP,
PK_LNDN,
PK_LEFT,
PK_RIGHT The arrow keys move focus from sibling to sibling, and are also used to navigate through `PegMenu` items.

PK_CR Carriage return is used to select the item which has focus.

PK_ESC Escape closes an open menu or cancels a `PegString` edit operation.

<Ctrl> F4 Closes the current window.

Class PegThing

Class `PegThing` is the most fundamental and important class of RTPEG-32. It is the base class from which all viewable objects are derived. You will be using the public functions of `PegThing` often when programming with RTPEG-32. Some basic properties of `PegThing` include: whether or not the object is visible; whether the object has a parent and who is the parent; whether the object has children and who are the children; whether the user should be allowed to interact with an object; etc.

The complete API of class `PegThing` is given in the online RTPEG-32 Reference.

The Class Hierarchy

Below is a list of all RTPEG-32 classes. Derived classes are indented below their base classes. *Italic* classes have two base classes and thus appear twice.

PegThing	PegMessageQueue
PegIcon	PegScreen
PegSlider	PegTextThing
PegHScroll	<i>PegTextButton</i>
PegVScroll	<i>PegRadioButton</i>
<i>PegGroup</i>	<i>PegCheckBox</i>
<i>PegTitle</i>	<i>PegGroup</i>
PegStatusBar	<i>PegMenuButton</i>
PegProgressBar	<i>PegTitle</i>
<i>PegPrompt</i>	<i>PegMessageWindow</i>
PegVPrompt	<i>PegPrompt</i>
<i>PegString</i>	<i>PegString</i>
PegSpinButton	<i>PegTextBox</i>
PegMenu	<i>PegDecoratedButton</i>
PegMenuBar	PegTreeNode
<i>PegMenuButton</i>	PegImageConvert
PegButton	PegBmpConvert
<i>PegTextButton</i>	PegGifConvert
PegMLTextButton	PegJpgConvert
PegBitmapButton	PegQuant
<i>PegCheckBox</i>	
<i>PegRadioButton</i>	
<i>PegDecoratedButton</i>	
PegWindow	
PegDecoratedWindow	
PegDialog	
<i>PegMessageWindow</i>	
PegProgressWindow	
PegTable	
<i>PegTextBox</i>	
PegEditBox	
PegTerminalWin	
PegList	
PegHorzList	
PegVertList	
PegComboBox	
PegNotebook	
PegPresentationManager	
PegSpreadSheet	
PegAnimatedWindow	
PegTreeView	
PegChart	
PegLineChart	
PegMultiLineChart	
PegStripChart	
PegToolBar	
PegToolBarPanel	

Unicode

Unicode support is available for owners of an RTPEG-32 source code license. Currently, Unicode support is not available for Watcom C/C++. If you need Unicode support with Watcom C/C+, please contact On Time for additional information.

An application that wants to use the Unicode version of RTPEG-32 must define C/C++ preprocessor symbol `PEG_UNICODE`. This will cause type `PEGCHAR` to resolve to type `wchar_t`. In addition, a few symbols and functions are defined to ease string handling of the application. Macro `PTEXT()` can be used to define a string with the correct type, depending on symbol `PEG_UNICODE`. The following string manipulation functions declared in `Include\Pegtypes.hpp` can be used as a replacement of corresponding run-time system functions: `PegStrCat`, `PegStrnCmp`, `PegStrCpy`, `PegStrnCat`, `PegStrnCpy`, `PegStrCmp`, `PegStrnCmp`, `PegStrLen`, `PegAtoL`, `PegAtol`.

The Unicode version of RTPEG-32 processes Unicode keyboard input and does not use input code page 1252 as its Ascii counterpart. For example, the Euro symbols (AltCar-E in most national keyboard drivers) is returned as hex value 20ACh in Unicode apps and as 80h in Ascii apps.

RTPEG-32 Unicode applications must link library `Pegu.lib` instead of `Peg.lib`. The Win32 emulation library is contained in `Peguw32.lib` and `Pegu.lib`. Demo program `PegDemo` compiles unmodified as either an Ascii or Unicode application. Demo program `Unicode` shows how a multi-language application can be implemented. Please refer to their respective makefiles for further details.

Win32 Emulation Library

RTPEG-32 programs to run under On Time RTOS-32 must link library `PEG.LIB` (or `PEG11.LIB` for Watcom C/C++ 11.0). However, library `PEGW32.LIB` (and `PEGW3211.LIB` for Watcom C/C++ 11.0) allows executing RTPEG-32 GUI programs under Windows 95/98/ME/NT/2000 as long as they do not need any other On Time RTOS-32 functions not available under Windows. No source code modifications are required, but RTPEG-32 programs for Windows must be linked with libraries `PEGW32.LIB` and `PEG.LIB` (in this order), no other On Time RTOS-32 libraries, but with standard Win32 API libraries such as `KERNEL32`, `GDI32`, `USER32`, etc. In addition, the linker must be instructed to produce a Win32 GUI instead of a Win32 Console program.

Examples for the Dialog demo (Microsoft, Borland, and Watcom):

```
cl -Zi -Zp4 Dialog.cpp pegw32.lib peg.lib user32.lib gdi32.lib
/link /subsystem:windows

bcc32 -W -v -a4 Dialog.cpp pegw32.lib peg.lib

wcl386 -s -zp4 -d2 -hc -l=nt_win Dialog.cpp pegw32.lib peg.lib
```

Library `PEGW32.LIB` contains Win32 emulation versions of all screen device drivers and dummy versions (do nothing, return 0) of the following RTTarget-32 native API functions:

```
RTSetFlags()
RTCMOSSetSystemTime()
RTCMOSExtendHeap()
RTInitMouse()
RTSetMousePos()
RTMouseDone()
RTGetGMode()
RTEmuInit()
```

These functions are not needed under Windows but allow a program to link successfully even if it references them.

Chapter 2

Programming with RTPEG-32

This chapter presents more in depth information and a few examples for using the RTPEG-32 library effectively.

Application Program Structure

An RTPEG-32 program's main function should perform the following steps:

- If RTKernel-32 is used, RTKernelInit() should be called first.
- A screen device driver (a PegScreen object) must be created.
- Objects PegMessageQueue and PegPresentationManager must be created. This is done by calling function PegInitialize().
- If mouse support is desired, the RTTarget-32 mouse driver must be initialized.
- All initial UI objects such as windows and control must be added to the PegPresentationManager. This will typically be done by function PegAppInitialize().
- Multithreaded applications can create additional threads at this point.
- The RTPEG-32 message loop is executed by calling PegExecute(). Function PegExecute() will not return before the application has terminated.

Example:

```
int main(void)
{
    PegScreen * pScreen = CreatePegScreen_VESA_8();
    PegPresentationManager * pPresent =
        PegInitialize(pScreen, sizeof(class PegScreen));

    RTInitMouse(-1, 12, 0, 5, 5); // this is for a PS2 mouse
    RTSetMousePos(pScreen->GetXRes() / 2, pScreen->GetYRes() / 2);

    PegAppInitialize(pPresent);

    PegExecute(pPresent);

    RTMouseDone();
    return 0;
}

void PegAppInitialize(PegPresentationManager * pPresentation)
{
    PegRect Rect;
    Rect.Set(0, 0, 399, 319);
    RobotFrame * pFrame = new RobotFrame(Rect);
    pPresentation->Center(pFrame);
    pPresentation->Add(pFrame);
}
```

Function PegInitialize

The RTPEG-32 library must be initialized with a call to PegInitialize:

```
PegPresentationManager * PegInitialize(PegScreen * pScreen,
                                       int PegScreenClassSize);
```

Parameter pScreen must point to an instantiated screen driver. Parameter PegScreenClassSize must be the size in bytes of the abstract class PegScreen. This value is used to verify that the application and the PEG.LIB library have been compiled with the same structure alignment (minimum 4) and options specified in header file pconfig.hpp.

The function's return value is a pointer to the PegPresentationManager of the application.

Function PegExecute

PegExecute executes the main message loop of the application:

```
void PegExecute(PegPresentationManager * pPresent);
```

Parameter pPresent must be the value returned by function PegInitialize. PegExecute will return when the application terminates through message PM_EXIT.

Rules Of Memory Ownership

When an RTPEG-32 object is added (i.e. attached) to another object, RTPEG-32 owns that object. You do not have to worry about deleting that object as long as you have passed it on to RTPEG-32. RTPEG-32 ensures that all children of an object, along with the object itself, are deleted when the parent object is closed.

For example, suppose you create a dialog window using the new memory allocation operator. After creating the dialog, you also create a dozen or so controls and add them to the dialog. At this point all of the controls are owned by the dialog. All that you need to do to delete all of the allocated memory is delete the dialog.

Next, assume that you add the dialog to PegPresentationManager (i.e. the dialog is now visible). At this point, you have given up all ownership of the dialog and the dialog's child controls. RTPEG-32 is now responsible for insuring that the dialog and its child controls are deleted from memory when the dialog is closed.

Finally, assume that you manually Remove() the dialog from PegPresentationManager, without allowing the dialog to close itself in response to user input. In this case, you again own the dialog, because PegPresentationManager no longer has any knowledge of the dialog's existence. However, the controls which were added to the dialog are still owned by the dialog, so once again all that needs to be done to delete all memory associated with the dialog and its controls is to delete the dialog.

Creating PegThings

PegThing contains information about the physical location of the objects on the screen, the client area of an object, the clipping area of an object, the system status flags for the object (selected, sizable, etc.), and pointers used to maintain the object's position in the presentation tree.

PegThing provides the member function Add(PegThing * what), which is used to add controls or windows to another. With Add(PegThing * what), parameter what is inserted into the current object's child list. If the current object is visible, the newly added object also becomes visible. The best way to create a complex window is to create the window, create all of the window's child objects and add them to the window, and finally add the window to PegPresentationManager. In this way, the window and all of the child objects become visible at the same time.

A further result of the class hierarchy is that it is perfectly reasonable to create an object that one would normally consider to be a self-contained bottom level object, such as a PegPrompt, and add another object, such as a PegButton, to the PegPrompt. The result is a PegPrompt that first displays the text associated with the prompt, and then allows its child objects to draw themselves. In this example, the PegButton would appear next to or over the prompt text, depending on the Prompt dimensions and text justification flags. While this result may not appear very useful, you should be able to see that by deriving your own version of PegPrompt specifically for this purpose, you could easily create a powerful new object type simply by combining these two predefined objects.

The following code fragment illustrates the ease of creating and displaying new windows. The newly created window will have a title, a menu bar, and a status bar:

```
PegRect WinSize;
WinSize.Set(10, 10, 120, 200);
Presentation()->Add(new AppWindow(WinSize, "My First Window"));
...
AppWindow::AppWindow(PegRect Rect, PEGCHAR * Title):
    PegDecoratedWindow(Rect)
{
    Add(new PegTitle(Title));           // add a title to myself
```

```
Add(new PegMenuBar(MainMenu));    // and a menu bar
PegStatusBar * pStat = new PegStatusBar();
pStat->AddTextField(80, "Hello");
pStat->AddTextField(20, "How are you today?");
Add(pStat);                        // and a status bar
}
```

Deleting/Removing PegThings

Removing a PegThing (i.e. a window, button, dialog, or other object derived from PegThing) from its parent is not the same as deleting it. Removing an object takes it out of the active display tree. After being removed, the object no longer has a parent, and it will not be visible. It is possible, even common, to later re-add the object to a visible PegThing and use it over again.

A PegThing is removed by calling the PegThing member function `Remove(PegThing * What)`. It doesn't matter if the parent object removes a child, or if a child removes itself, because the `Remove()` function properly handles either case. That is, it is perfectly acceptable to use the following statement:

```
Remove(this);
```

when an object decides based on some message input that it is time to go away.

While `Remove()` can be useful, it is more common to want to both remove the object from its parent, as well as delete the object from memory. There are three acceptable ways to remove and delete an object:

- Sending a `PM_DESTROY` message to `PegPresentationManager`. The `pSource` member of the `PM_DESTROY` message should point to the object which is to be destroyed. This method is most often used when deleting objects from tasks outside of RTPEG-32.
- Calling the PegThing member function `Destroy(PegThing * Who)`. Any PegThing can destroy any other PegThing, including itself. This does not mean that the `Destroy()` function will end up executing a `delete(this)` statement. The `Destroy` function checks to see if `Who == this`, and in this case automatically sends a `PM_DESTROY` message to `PegPresentationManager` to finish the job.
- If a PegThing is already removed from its parent through use of the `Remove()` function, and the object to be deleted is not this, nor is this a child of the object being deleted, it is fine to simply delete the object.

One should never execute `delete(this)`. When in doubt, it is always safe to call `Destroy()`. It is not necessary to manually delete the individual children of a PegThing, in fact it will cause errors if this is attempted.

Obtaining a Pointer to PegPresentationManager

The PegThing member function `Presentation()` is provided for this purpose. For example, the following code segment could be used to determine whether a window is a top-level window (i.e. a child of `PegPresentationManager`):

```
If (Presentation() == Parent())
{
    // Current object is a top-level object
}
```

Finding an Object's Parent

The PegThing member function `Parent()` returns a pointer to the parent of the current object. If the object has no parent, the `Parent()` function returns `NULL`.

Finding an Object's Children

The first child of any object can be found using the function `First()`. This returns a pointer to the head of a linked-list of child objects. The linked list can be traversed using the `Next()` function.

For example, an object could count the number of siblings (i.e. object's with the same parent) it has using the following code sequence:

```
PegThing * pTest = Parent()->First(); // first child of my parent
int iSiblings = 0;
while (pTest)
{
    if (pTest != this)
        iSiblings++;
    pTest = pTest->Next();
}
```

System Status Flags

All RTPEG-32 objects have system status flags associated with them. The system status flags are important for the correct operation of the library, but are generally not often needed by the application software. PegThing provides public functions to examine and/or modify system status flags for an object. The system status flags have names that start with PSF_, which stands for PEG System Flag.

The following code segment can be used to enquire which child of the current object has input focus:

```
PegThing * pTest = First();
while (pTest)
{
    if (pTest->StatusIs(PSF_CURRENT))
        break; // this object has input focus
    pTest = pTest->Next();
}
```

The following status flags are available:

PSF_VISIBLE	The object is visible on the screen. This flag should not be modified by the application level software. Clearing or setting this flag will not have the effect of removing or displaying the object. The PegThing member functions Add and Remove are used for that purpose.
PSF_CURRENT	the object is in the current branch of the display tree. If the object is a leaf object (i.e. it has no children) and it is current, then it is the object which will receive keyboard input messages.
PSF_SELECTABLE	This flag is tested by PegPresentationManager to determine if an object is enabled and allowed to receive input messages. The application level software can modify this flag.
PSF_SIZEABLE	determines whether or not an object can be resized. The application can modify this flag.
PSF_MOVEABLE	determines whether or not an object can be moved. The application can modify this flag.
PSF_NONCLIENT	allows a child object to draw outside the client area of its parent. The application can modify this flag after the object is constructed but before the object is displayed.
PSF_ACCEPTS_FOCUS	indicates that the object will become the receiver of input events when selected. The application can modify this flag, but normally this is not advised. If this flag is modified for a particular object, it is important for correct operation that 'breaks' in the tree of objects accepting focus are avoided. In other words, if a parent window cannot accept focus, then neither should any of the window's child objects be allowed to accept focus.
PSF_ALWAYS_ON_TOP	This flag insures that the object is always on top of its siblings. The application level software can modify this flag.
PSF_VIEWPORT	instructs PegPresentationManager that the object should be given a private screen viewport. Objects that have a viewport are drawn differently than objects that do not have a viewport. In general, large objects or objects which have a very complex drawing routine should be given viewports, while

small or simple objects should not. By default all PegWindow derived objects receive viewports, and all other objects do not. This flag should not be changed except immediately after the object is constructed.

Style Flags

All RTPEG-32 objects also have a set of *style* flags associated with it, which allow control of many things related to how an object appears and functions. The style flags are interpreted in different ways by different object types, and some style flags apply only to certain types of objects. PegThing provides functions for reading or modifying an object's style flags.

The following code segment can be used to set the AF_ENABLED style flag for a button. Note: this is an example only. The PegButton class provides member functions for accomplishing this task.

```
PegButton * pChild = (PegButton *) First();
pChild->Style(pChild->Style() | AF_ENABLED);
```

Determining the Position of an Object

One of the most basic properties of all RTPEG-32 objects is the object's position on the screen. PegThing maintains this information, along with clipping and Z-ordering information to insure that objects are only allowed to draw to the areas of the screen that are owned by the object. An object's position is held in the PegThing public member variable mReal, which is a value of type PegRect. PegThing also maintains a separate but related member called mClient, which is an additional PegRect member that indicates the client rectangle of the object. For many objects, the client area and the real area are equal.

For example, by using the mReal variables and the PegRect::Overlap function, we can easily determine if two objects overlap using the following code segment:

```
PegThing * pThing1 = First();
PegThing * pThing2 = pThing1->Next();

if (pThing1->mReal.Overlap(pThing2->mReal))
{
    // objects overlap
}
else
{
    // objects do not overlap
}
```

Using Object Types

PegThing maintains a type value for each object derived from it in the public member variable muType. This value can be used to safely upcast a PegThing * to a pointer to a specific RTPEG-32 control or window type. RTPEG-32 does not internally use the object type value, with the exception of checking the range, which can be above or below TYPE_WINDOW. You can safely create and use your own object types and assign new type values to these classes. If your new object is derived at some point from PegWindow, it should have an object muType >= TYPE_WINDOW. The object type ranges are specified in file \peg\include\pegtypes.hpp. This is sometimes useful as a debugging aid in addition to the occasional need to upcast a PegThing pointer to a specific derived class pointer. Additional information about object types is provided in the reference of the muType accessor functions.

Derived object types inherit the object type value of their parent. The type value can be overridden if desired by re-assigning the object type after calling the base class constructor.

Object type values are divided into two groups. One group is for classes derived from PegWindow, and the other group is for all other object types. When assigning custom object types, one should use the value FIRST_USER_WINDOW_TYPE or FIRST_USER_CONTROL_TYPE as the base. This insures that private type values will be unique and will not overlap on the RTPEG-32 class types.

Object type values can be useful when searching child object lists for objects of a certain type, for example PegString objects. This value is also useful when debugging since at times you may have a pointer to a PegThing and wish to know exactly what type of PegThing it points to. After checking the muType member of a PegThing, one can safely upcast a PegThing pointer to a pointer to a specific RTPEG-32 object type. The possible return values of the Type() function are defined in header file pegtypes.hpp.

The following code fragment illustrates one possible method of locating the status bar attached to a window:

```
PegThing * pTest = First(); // get pointer to first child object
while (pTest)              // search to the end of list if necessary
{
    if (pTest->Type() == TYPE_STATUS_BAR)
    {
        PegStatusBar *p StatBar = (PegStatusBar *) pTest;
        // use pStatBar to call member functions or change attributes
        break; // found the status bar, exit the loop
    }
    pTest = pTest->Next(); // continue down the list of children
}
```

Of course, it is simpler to call the PegWindow member function StatusBar(), which does exactly what is shown above and returns a pointer to the PegStatusBar object if one is found.

Using Object IDs

A few object ID values starting at 1000 are reserved for proper operation of dialog boxes and message windows. Therefore, custom IDs should always begin enumerations with a value of 1, so as not to overlap the reserved ID values, which are at the very top of the valid ID range. The reserved object IDs are:

```
enum PegButtonIds {
    IDB_CLOSE = 1000,
    IDB_SYSTEM,
    IDB_OK,
    IDB_CANCEL,
    IDB_APPLY,
    IDB_ABORT,
    IDB_YES,
    IDB_NO,
    IDB_RETRY,
};
```

Buttons with the IDs listed above are given special treatment by dialog and message window classes. For further information, see PegDialog and PegMessageWindow.

Object ID values can be used to identify an object. When an object sends a notification signal to a parent window, the object ID is contained in the iData member of the notification message.

A child object can be located using the object's ID with the Find() function. Find will search the child list of the current object for an object with an ID value matching the passed in value. Object IDs are also useful for identifying top-level windows. It is often the case that one window needs to locate another window without knowing whether the other window is actually displayed. The following code segments illustrate using Window ID values to locate a top-level window:

```
Window1::Window1(...) : PegDecoratedWindow(...)
{
    Id(ID_WINDOW1);
}

PegDecoratedWindow * Window2::FindWindow1(void)
{
    return Presentation()->Find(ID_WINDOW1);
}
```

Messages

Messages are identified by the 16-bit member field `wType` of structure `PegMessage`. The first 4000h message `wType` values are reserved for internal messages, which leaves message values 4000h through FFFFh available for user definition. The first message value which is available for user defined message is `FIRST_USER_MESSAGE`, declared in `Peg.hpp`.

Below are the system messages which would potentially be of interest in the application level software:

<code>PM_ADD</code>	This message can be sent to add an object to another object. The message <code>pTarget</code> field should contain a pointer to the parent object, and the message <code>pSource</code> field should contain a pointer to the child object.
<code>PM_CLOSE</code>	Recognized by <code>PegWindow</code> derived objects, and causes the recipient to remove itself from its parent and delete itself from memory.
<code>PM_CURRENT</code>	Sent when an object becomes a member of the branch of the presentation tree which has input focus.
<code>PM_DESTROY</code>	This message can be sent to <code>PegPresentationManager</code> to destroy an object. The <code>pSource</code> member of the message should point to the object to be destroyed.
<code>PM_DIALOG_NOTIFY</code>	Sent to the owner of a <code>PegDialog</code> when the dialog window is closed if the dialog window is executed non-modally. The message <code>iData</code> member will contain the ID of the button used to close the dialog window.
<code>PM_DRAW</code>	Sent to an object to force that object to redraw itself.
<code>PM_EXIT</code>	Sent to <code>PegPresentationManager</code> causes termination of the program.
<code>PM_HIDE</code>	Sent whenever an object is removed from a visible parent.
<code>PM_KEY</code>	Sent to the current input object when keyboard input is received. The message <code>iData</code> member contains the corresponding ASCII character code, if any, and the <code>lData</code> member of the message contains the keyboard scan code, if available.
<code>PM_LBUTTONDOWN</code>	Sent when the user generates mouse click input. The position of the mouse click is included in the message <code>Point</code> field.
<code>PM_LBUTTONUP</code>	Sent when the user releases the left mouse button.
<code>PM_MAXIMIZE</code>	This message can be sent to any <code>PegWindow</code> derived object. If the target window is sizable (as determined by the <code>PSF_SIZEABLE</code> status flag), it will resize itself to fill the client rectangle of its parent.
<code>PM_MINIMIZE</code>	Similar to <code>PM_MAXIMIZE</code> , this message can be sent to any <code>PegWindow</code> derived object. If the window is sizable, it will create a proxy <code>PegIcon</code> , add the icon to the parent window, and remove itself from its parent.
<code>PM_NONCURRENT</code>	This message is sent to an object when it loses membership in the branch of the presentation tree which has input focus.
<code>PM_PARENTHIZED</code>	Sent to all children of a <code>PegWindow</code> derived object if the window is resized. This makes it very easy for child windows that want to maintain a certain proportional spacing or position within their parent to catch this message and resize themselves whenever the parent window is sized.
<code>PM_POINTER_ENTER</code>	Sent to an object when the mouse pointer passes over it.
<code>PM_POINTER_EXIT</code>	Sent to an object when the mouse pointer leaves it.
<code>PM_POINTER_MOVE</code>	Sent to an object whenever the mouse pointer moves over it.
<code>PM_SHOW</code>	Sent to an object when it is added to a visible parent, before the object is first drawn. This allows an object to perform any necessary initialization prior to drawing itself on the screen.

PM_SIZE	Cause an object to resize or move. This is equivalent to calling the <code>Resize()</code> function. Note that there is no difference between moving and resizing an object. The new size and position is included in the message <code>Rect</code> field.
PM_RESTORE	Can be sent to any sizeable <code>PegWindow</code> derived object to cause that window to restore its size and position after it has been maximized or minimized.
PM_RBUTTONDOWN	The right mouse button is pressed. RTPEG-32 objects do not process right mouse button messages.
PM_RBUTTONUP	The right mouse button is released. RTPEG-32 objects do not process right mouse button messages.
PM_TIMER	This message is sent to an object that has started a timer via the <code>PegMessageQueue TimerSet</code> function when that timer expires. The ID of the timer is included in the <code>iData</code> member of the message.

Overriding the `Message()` Method

Overriding the message handler method should in most cases return a result of 0. A non-zero return value is used to terminate modal window execution. `PegWindow` derived classes such as `PegDialog` and `PegMessageWindow` return non-zero results when a signal from a child control is received that causes the window to close.

A `Message()` method should make sure that it passes the unhandled messages down to the base class to insure that normal default operation occurs, unless of course the message should be intercepted. For system messages, the base class's `Message` handler should be called first before any custom processing is done.

A typical `Message()` function for a derived class would appear as follows (assuming in this example that the class is derived from `PegWindow`):

```
SIGNED MyClass::Message(const PegMessage &Mesg)
{
    switch (Mesg.wType)
    {
    case UIM_SHOW:
        PegWindow::Message(Mesg);
        // add your own code here:
        break;
    case USER_DEFINED_MSG1:
        // code for your user message
        break;
    case USER_DEFINED_MSG2:
        // code for another user defined message:
        break;
    case SIGNAL(IDB_OK, PSF_CLICKED):
        // code for OK button clicked:
        break;
    default:
        // pass all other messages down to the base class:
        return PegWindow::Message(Mesg);
    }
    return 0;
}
```

Drawing to the Screen

Drawing on the screen is performed through methods of class `PegScreen`. This is most often done from within an overridden `Draw()` function, as was described in the previous chapter. Drawing always starts with a call to `BeginDraw()` and completes by calling `EndDraw()`. When RTPEG-32 recognizes that an object needs to be re-drawn (i.e. the object was just `Add()`-ed, or the object has been moved), it re-draws the object by calling the object's `Draw()` function.

Drawing can also be performed by functions other than Draw(). Such functions must be members of a PegThing derived class, or at least have access to a PegThing object, since all of the PegScreen drawing functions require as a parameter a pointer to the PegThing object calling the drawing function. PegScreen requires this pointer to insure that an object is not allowed to draw outside of the area it 'owns' on the screen.

PegScreen only allows drawing to occur to areas of the screen which have been invalidated. Areas of the screen are invalidated by calling the Invalidate() function, which is a member of PegScreen but also provided in inline form as a member of PegThing. Under most circumstances the screen invalidation is handled automatically by RTPEG-32 as the user moves things around on the screen, or as the program adds and removes visible objects. If all drawing is done from with an overridden Draw() function, there is no need to invalidate the screen, since the Draw() function is called specifically because an area of the screen has been invalidated.

If you need to draw on the screen outside at random times, or for example based on a periodic timer, you must invalidate the area you are going to draw to before you start drawing. If you want to be allowed to draw anywhere within the client area of your object, you can simply call the Invalidate() function with no parameters, which invalidates the area of the screen corresponding to an object's client area. You can also calculate and specify a more limiting rectangle to clip your drawing, and pass that rectangle to the Invalidate() function. No matter how large the invalidated rectangle on the screen, you are never allowed to draw outside of an object's borders.

The following function is an example function that could be used to draw a series of lines to the screen at any time. This example will paint the entire client area of the object black, and then fill the client area of the object with RED horizontal lines, 1 pixel wide, spaced 4 pixels apart.

```
void MyObject::DrawLines(void)
{
    PegColor LineColor(RED, BLACK, CF_FILL);
    SIGNED yPos = mClient.wTop;

    Invalidate();           // invalidate my client area
    BeginDraw();           // prepare for drawing
    Rectangle(mClient, Color, 0); // fill with black
    while (yPos <= mClient.wBottom)
    {
        // draw red lines:
        Line(mClient.wLeft, yPos, mClient.wRight, yPos, Color);
        yPos += 4;
    }
    EndDraw();             // Done
}
```

Overriding the Draw() Method

You can create a custom interface appearance by deriving your own control types from the RTPEG-32 control types. For example, you may want to define a button type that has an appearance different from the standard RTPEG-32 button types provided.

The following code listings illustrate creating a new PegButton derived class, and overriding the Draw() function to generate a custom button appearance. In this case we are going to draw a wide button border, and use custom colors for the button client area and text. This example code can also be found in the example program file Robot\Robobutn.cpp. The following is the class definition for the RoboButton class:

```
class RoboButton : public PegButton
{
public:
    RoboButton(PegRect &R, WORD wId, PEGCHAR *Text);
    void Draw(void);
    void DataSet(PEGCHAR * Text)
    {
        mpText = Text;
        Screen()->Invalidate(mClient);
    }
}
```

```
    }  
    private:  
        PEGCHAR * mpText;  
};
```

The above class definition tells the compiler that we are defining a new class based on PegButton. This definition also informs the compiler that we are going to override the PegButton Draw() function, since we have defined a function named Draw() with the same return type and parameters as are defined in the virtual PegButton Draw() function. Note: If your parameter list does not match the base class parameter list, the compiler will assume that you are overloading, not overriding, a base class function.

The following listing is the Draw() function implementation for the RoboButton class:

```
void RoboButton::Draw(void)  
{  
    PegColor Color;  
    BeginDraw(); // Note19  
    if (Style() & BF_SELECTED) // Note20  
        Color.uForeground = BLACK;  
    else  
        Color.uForeground = LIGHTGRAY;  
    // draw the top: // Note21  
    Line(mReal.wLeft, mReal.wTop, mReal.wRight, mReal.wTop, Color, 3);  
    // draw the left:  
    Line(mReal.wLeft, mReal.wTop, mReal.wLeft, mReal.wBottom, Color, 3);  
    if (Style() & BF_SELECTED) // Note22  
        Color.uForeground = LIGHTGRAY;  
    else  
        Color.uForeground = BLACK;  
    // draw the right shadow:  
    Line(mReal.wRight, mReal.wTop, mReal.wRight, mReal.wBottom-2, Color, 1);  
    Line(mReal.wRight-1, mReal.wTop+1, mReal.wRight-1, mReal.wBottom-2, Color, 1);  
    Line(mReal.wRight-2, mReal.wTop+2, mReal.wRight-2, mReal.wBottom-2, Color, 1);  
    // draw the bottom shadow:  
    Line(mReal.wLeft, mReal.wBottom, mReal.wRight, mReal.wBottom, Color, 1);  
    Line(mReal.wLeft+1, mReal.wBottom-1, mReal.wRight, mReal.wBottom-1, Color, 1);  
    Line(mReal.wLeft+2, mReal.wBottom-2, mReal.wRight, mReal.wBottom-1, Color, 1);  
    // fill in the button client area:  
    Color.Set(LIGHTRED, DARKGRAY, CF_FILL);  
    Rectangle(mClient, Color); // Note23  
    // draw the text centered:  
    PegPoint Put;  
    Put.x = (mClient.wLeft+mClient.wRight) >> 1;  
    Put.x -= TextWidth(mpText, &SysFont) >> 1;  
    Put.y = mClient.wTop+1;
```

19 Always start a custom drawing function with BeginDraw(). This call informs the screen driver that drawing is about to begin. If this button is being drawn as part of a larger window drawing operation, the screen driver will recognize that this is a nested call to BeginDraw().

20 This if statement is testing the button style flag BF_SELECTED to determine if the button is depressed. If the button is depressed, we want to draw the button shadow on the top and left, instead of on the right and bottom. This provides the 3D action desired for this button class.

21 This line is using the "Line" wrapper function of PegThing to call the PegScreen::Line() function. The line endpoints, color and width are passed to the Line() function. In this case we are drawing a 3-pixel wide line, to create a wide button border.

22 We are again testing the button style flag BF_SELECTED to toggle the border colors.

23 On this line we are using the wrapper function Rectangle() to draw a rectangle on the screen. The rectangle will have a RED border, and will be filled with the color DARKGRAY. This will fill the client area of the button.

```

    if (Style() & BF_SELECTED)
    {
        Put.x++;
        Put.y++;
    }

    Color.Set(WHITE, BLACK, CF_NONE);
    DrawText(Put, mpText, Color, &SysFont); // Note24
    EndDraw();                             // Note25
}

```

Drawing to Memory

An alternative to drawing directly to the screen is to draw to an offscreen bitmap. Such a bitmap can be displayed at any location by calling the `PegScreen Bitmap()` member function. This is the preferred method of displaying flicker-free animation, and can be used for many other purposes as well.

Drawing to memory works almost exactly like on-screen drawing. `PegScreen` member functions are used to draw to an offscreen bitmap. Offscreen drawing is more of a drawing mode in which all screen output is redirected to a bitmap.

Offscreen bitmaps must be created first using `PegScreen` member function `CreateBitmap()`. Then, the alternate form of the `BeginDraw()` function that accepts a pointer to the bitmap is called to start drawing. When drawing is done, the alternate form of the `EndDraw()` function must be called to put the `PegScreen` driver back to normal drawing mode. While the `PegScreen` driver is in the offscreen drawing mode, all of the normal `PegScreen` drawing functions draw into the bitmap.

The following example is a code fragment demonstrating offscreen drawing. This example is also provided in the Gauge demo program. It draws a blank gauge bitmap offscreen, drawing the 'needle' or position indicator on top of the background bitmap, and then copying the offscreen bitmap to the visible screen. The resulting effect is that the gauge updates smoothly without any noticeable flicker.

```

void Gauge::Draw(void)
{
    if (!mpBitmap) // first time?
    {
        // Create the bitmap, and draw into it:
        mpBitmap = Screen()->CreateBitmap(gbDialBitmap.wWidth,
                                          gbDialBitmap.wHeight);

        DrawToBitmap();
    }

    // now just copy the bitmap to the screen:
    BeginDraw();
    PegPoint Put;
    Put.x = mReal.wLeft;
    Put.y = mReal.wTop;
    Bitmap(Put, mpBitmap);
    EndDraw();
}

void Gauge::DrawToBitmap(void)
{
    PegPoint Put;
    SIGNED x1, y1, x2, y2;
    SIGNED iRadius = (gbDialBitmap.wWidth / 2) - 8;
}

```

²⁴ After calculating where to draw the button text, this call writes the text for the button on the button face using the RTPEG-32 font `SysFont`. Any other custom font could be used just as well.

²⁵ The `Draw()` function must end with a call to `EndDraw()` to inform the screen driver that drawing is complete. A common mistake is to forget to call the `EndDraw()` function, which can cause unpredictable results in terms of screen appearance.

```
// Open the bitmap for drawing:
Screen()->BeginDraw(this, mpBitmap);    // Note26
Put.x = Put.y = 0;

// copy the background bitmap into memory bitmap:
Bitmap(Put, &gbDialBitmap);            // Note27

// find the center:
x1 = mReal.Width() / 2;
y1 = mReal.Height() / 2;

// find the end:
double angle = ((4.0 * PI) / 5.0) + (((double) miCurrent / 100.0) * PI);
x2 = x1 + cos(angle) * iRadius;
y2 = y1 + sin(angle) * iRadius;

// draw the indicator line:
PegColor LineColor(RED, RED, CF_NONE);  // Note28
Line(x1, y1, x2, y2, LineColor, 2);

// Close the bitmap for drawing:
Screen()->EndDraw(mpBitmap);            // Note29
}
```

The real work of drawing the gauge is done in the class member function `DrawToBitmap`. `DrawToBitmap` draws the background of the gauge and the gauge indicator line into an offscreen bitmap, which can then be copied to the screen at any time.

Using PegTimer

For this example we will create a derived `PegDecoratedWindow` class. In this derived window class we will override the `Message()` function to provide custom functionality. The custom operation is to start a periodic `PegTimer` and wait for timer expiration messages to arrive. The window will change colors each time the timer expires.

```
class MyWindow : public PegDecoratedWindow
{
public:
    MyWindow(const PegRect &Rect);
    SIGNED Message(const PegMessage &Msg);
private:
    SIGNED miColor;
};

void PegAppInitialize(PegPresentationManager * pPresent)
{
    PegRect WinRect;
    WinRect.Set(0, 0, 100, 100);
    MyWindow *pWin = new MyWindow(WinRect);
    pPresent->Center(pWin);
    pPresent->Add(pWin);
}
```

26 This line of code is calling the alternate form of `BeginDraw()`, passing not only a pointer to the Gauge class itself, but also a pointer to the bitmap that the class wants to draw to. This places the `PegScreen` driver into offscreen drawing mode.

27 This line of code is drawing a pre-generated bitmap (the bitmap was generated prior to compile using `PegImageConvert`) into the memory bitmap. This bitmap forms the background of the gauge.

28 This line of code draws the gauge indicator line. The line is drawn from the center of the gauge to the outside edge. This line is also drawn to the offscreen bitmap, not to the visible screen.

29 The offscreen drawing mode is terminated by calling the alternative form of `EndDraw()`, which accepts as a parameter a pointer to the bitmap that was drawn to. The offscreen bitmap is now ready to be displayed.

```

MyWindow::MyWindow(const PegRect &Rect):
    PegDecoratedWindow(Rect, FF_THIN)
{
    Add(new PegTitle("A colorful Window!"));
    RemoveStatus(PSF_SIZEABLE);
    miColor = 0;
}

SIGNED MyWindow::Message(const PegMessage &Mesg)
{
    switch(Mesg.wType)
    {
        case PM_SHOW:
            PegDecoratedWindow::Message(Mesg);
            SetTimer(1, ONE_SECOND * 2, ONE_SECOND / 2);
            break;
        case PM_TIMER:
            SetColor(PCI_NORMAL, miColor);
            Invalidate();
            Draw();
            miColor++;
            miColor &= 0x0f;
            break;
        case PM_HIDE:
            KillTimer(1);
            PegDecoratedWindow::Message(Mesg);
            break;
        default:
            return PegDecoratedWindow::Message(Mesg);
    }
    return 0;
}

```

In the above Message() function, the derived window class catches the system messages PM_SHOW and PM_HIDE, and the PM_TIMER messages generated by the timer. The PM_SHOW message is received when the window is first displayed. This is a convenient place to start the timer. In this case we set the timer to wait 2 seconds before the first timeout, and to expire every 500 ms (ONE_SECOND / 2) thereafter.

This timer ID value is simply set to '1'. If you are using many timers, you will probably want to enumerate the timer ID values, but in this case we are only using one timer and so we simply hard-coded the timer ID value. Note that the PM_SHOW message handler also passes the message on down to the base PegDecoratedWindow class. It is important to pass system messages on down to the base class in case the base class is also catching the message.

The PM_HIDE message is received when the window is removed. This is a convenient place to stop the timer. You must remember to kill timers that you have started before your windows are deleted, or PM_TIMER messages will be sent to invalid destinations and your software will most likely crash. Since a window is always removed before it is deleted, the PM_HIDE system message handler is an excellent place to kill any active timers.

The PM_TIMER message handler is where we change the window color and re-display the window. A member variable has been defined named miColor, and we will use this variable to keep track of which color to display next. This example assumes we are running in 16-color mode, and therefore prevents the color index from passing 15 (0x0f).

Note that after changing the window color by using the SetColor() function, we have to tell the window to re-draw. The window does not automatically redraw since you may make several changes to the window and you do not want each change to cause a window re-draw operation.

Viewports

Viewports are used to improve drawing efficiency and to allow background drawing operations to occur without overwriting foreground graphics.

Viewports are rectangular areas of the screen owned by the object visible in that rectangle. Each viewport has only one owner, while one object may own several viewports.

RTPEG-32 maintains the screen viewports, and you do not ordinarily have to concern yourself with how they work. There is one exception, however, that you may need to be aware of. Normally, only PegWindow derived objects have viewport status. That means that other smaller objects like PegButton and PegIcon do not own viewports, and simply inherit the viewport(s) of their parent window.

The viewport management algorithm employed does not allow breaks in the viewport tree. That is, an object that owns viewports (i.e. a PegWindow derived object) should only be added to another object that owns viewports. This does not mean that you cannot add PegWindow derived objects to objects that are not derived from PegWindow, because you can. However, when you do this you should set the PSF_VIEWPORT status flag of the parent object, to make it a viewport owner.

An example should clarify this concept. Suppose you want to create a simple object container class. This container class will simply serve as a parent for a group of lists, windows, and other controls. This is a common thing to do, as it allows you to add and remove the entire group of objects at any time simply by adding or removing the container. Since the container class does not need to actually draw anything, you decide to derive it from PegThing, the most basic class. Since at least some of the children of the PegThing container are PegWindow derived objects, you will need to make the PegThing container class a viewport owner. If you don't do this, the PegWindow derived children of the container class won't show up on the screen. You can make the PegThing container class a viewport owner simply by adding the PSF_VIEWPORT system status in the container class constructor:

```
AddStatus(PSF_VIEWPORT);
```

Now your container class will work correctly, and both PegWindow derived children and simple children will be displayed when the parent container class is displayed.

Fonts

The PegFont type contains information about each font used in your application. The PegScreen text output and text information routines require a pointer to a PegFont structure as a parameter. You do not have to create these structures manually, since the PegFontCapture utility program will automatically generate this data structure for you, along with the associated offset table and data table.

By default, only two fonts called SystemFont and MenuFont are used. SystemFont is slightly larger than MenuFont, and is used for the title text in windows, and the strings in text boxes and string objects. MenuFont is used by menu bar and menu buttons, and by the PegPrompt and PegTextButton objects. You can easily change the font used by any text-related objects at run time by calling the SetFont() member function for that object after the object has been created.

Please note that fonts used by RTPEG-32 use the standard 1252 ANSI code page (also called ISO 8859-1 or Latin 1) or Unicode encoding, while RTTarget-32's text mode display routines use the OEM 437 code page. RTPEG-32 switches RTTarget-32's input code page to 1252 at initialization. The Unicode version of RTPEG-32 does not use code pages.

Default Fonts

Custom fonts can always be used through method SetFont(). To make a font a default font, use static member method SetDefaultFont or class PegTextThing:

```
static void SetDefaultFont(const UCHAR uIndex, PegFont * pFont);
```

The function sets a default font for an application and not just for one particular object. Parameter uIndex specifies for which purpose the given font is the new default:

PEG_TITLE_FONT	Titles
PEG_MENU_FONT	Menus
PEG_TBUTTON_FONT	Text Buttons
PEG_RBUTTON_FONT	Radio Buttons
PEG_CHECKBOX_FONT	Ceck Boxes
PEG_PROMPT_FONT	Prompts
PEG_STRING_FONT	String
PEG_TEXTBOX_FONT	Text Boxes
PEG_GROUP_FONT	Groups
PEG_ICON_FONT	Icons
PEG_CELL_FONT	Spreadsheet Cells
PEG_HEADER_FONT	Spreadsheet Headers
PEG_TAB_FONT	Tab Control Labels
PEG_MESGWIN_FONT	Message Windows
PEG_TREEVIEW_FONT	Tree View

The Vector Font

Fonts created with PegFontCapture are variable width bitmapped fonts. The disadvantage of bitmapped fonts is that a new font is needed for each point size and style used by the application. RTPEG-32's built-in vector font can be used to create any number and size of bitmapped fonts at run time. They can then be used just like any other fonts.

The PegScreen member functions MakeFont() and DeleteFont() can be used to create and delete a bitmapped font from the vector font. The following is an example of using the vector font in an application program.

```
void MyWindow::Draw(void)
{
    BeginDraw();

    // create 20 point font, not bold, italic:
    PegFont * MyFont = Screen()->MakeFont(20, FALSE, TRUE);

    // use MyFont here to draw text on the screen
    DrawText(..., MyFont);

    ...
    // after the program is done with the font, is should be deleted:
    Screen()->DeleteFont(MyFont);
    EndDraw();
}
```

A bitmapped font created at run time can be assigned to any text related object such as a PegText-Button or PegString. However the user must insure that the font has been created before it is assigned to the object, and it must also be deleted after the object has been destroyed.

Scrolling

PegWindow provides the capability of adding scroll bars and using those scroll bars to pan or move the client area of the window. Scroll bars are added by calling the SetScrollMode() PegWindow member function.

The scroll bars added to the window make use of two virtual PegWindow functions: GetHScrollInfo and GetVScrollInfo. When a scroll bar needs to update itself, it calls these parent window member functions to learn the scroll bar limit, current setting, and percentage visible data. GetHScrollInfo() and GetVScrollInfo() receive a pointer to a PegScrollInfo structure. It is the job of these functions to fill in the PegScrollInfo wMin, wMax, wCurrent, wStep, and wVisible values so that the scroll bar is correctly positioned.

The PegWindow class provides default implementations of GetHScrollInfo and GetVScrollInfo. These implementations examine all client-area children of the window to determine the outer limits that the scroll bars should allow scrolling to. This default implementation also uses the window client area width and height as the scroll bar 'visible' value.

The default implementation works well in most cases, and makes it very easy to create scrolling client areas. All you need to do is add a child window to a scrolling parent that is much larger than the parent client area. The default implementation will adjust the scroll bars such that the entire child window can be viewed by moving the horizontal and/or vertical scroll bars.

In some cases the default operation does not provide the required functionality. In these cases you can override the `GetHScrollInfo` and `GetVScrollInfo` functions to return custom scrolling information. For example, suppose you want to create a continuous-time plot of data values, and use a horizontal scroll bar to move back and forth in the time period displayed. In this case you would create a derived `PegWindow` class in order to draw the chart data in the window client area. You would also provide an overridden version of the `GetHScrollInfo` function to make the horizontal scroll bar reflect the accumulated time values. In this case, the `ScrollInfo` minimum value might be the starting time of data recording, the maximum value would be the current time, and the visible amount would be the time period visible in the window client area.

Chapter 3

Screen Drivers

RTPEG-32 is shipped with six different screen drivers, all of which are implemented as classes derived from PegScreen. However, the internal class declarations are not accessible to the application. Only methods defined by the abstract base class PegScreen should be used by an application.

Each driver implements a unique function to initialize it. These functions return a pointer to the PegScreen derived calls of the respective driver or NULL, if initializing the driver fails.

The DOS utility program VESATEST.COM shipped with On Time RTOS-32 can be used to determine which drivers are suitable for a particular target computer.

The source code of all drivers is available in directory Driver\Peg. If you need additional screen drivers, please contact On Time for further information.

Driver VESA_8

This driver is initialized with function:

```
PegScreen * CreatePegScreen_VESA_8(int Width = 800, int Height = 600);
```

VESA_8 supports any VESA 8-bit color modes if the complete video memory is accessible in a linear address range. VESA BIOS version 2.0 or higher is required. This driver does not set the video mode at run time; rather, it expects that the desired mode has been set by RTTarget-32's boot code using the *GMode* command.

Parameters Width and Height are only evaluated by the Windows emulation drivers. On the target, the screen resolution is determined by the current graphics mode set by the boot code through a GMode command.

If the current video mode is not a VESA 8-bit color mode, or the video mode does not use a linear frame buffer, NULL is returned.

Driver VESA_16

This driver is initialized with function:

```
PegScreen * CreatePegScreen_VESA_16(int Width = 800, int Height = 600);
```

VESA_16 supports any VESA 15- or 16-bit color mode if the complete video memory is accessible in a linear address range. VESA BIOS version 2.0 or higher is required. This driver does not set the video mode at run time; rather, it expects that the desired mode has been set by RTTarget-32's boot code using the *GMode* command.

Parameters Width and Height are only evaluated by the Windows emulation drivers. On the target, the screen resolution is determined by the current graphics mode set by the boot code through a GMode command.

If the current video mode is not a VESA 15- or 16-bit color mode, or the video mode does not use a linear frame buffer, NULL is returned.

Driver VESA_24

This driver is initialized with function:

```
PegScreen * CreatePegScreen_VESA_24(int Width = 800, int Height = 600);
```

VESA_24 supports any VESA 24-bit color mode if the complete video memory is accessible in a linear address range. VESA BIOS version 2.0 or higher is required. This driver does not set the video mode at run time; rather, it expects that the desired mode has been set by RTTarget-32's boot code using the *GMode* command.

Parameters Width and Height are only evaluated by the Windows emulation drivers. On the target, the screen resolution is determined by the current graphics mode set by the boot code through a *GMode* command.

If the current video mode is not a VESA 24-bit color mode, or the video mode does not use a linear frame buffer, NULL is returned.

This driver is somewhat slower than VESA_8, VESA_16, and VESA_32 due to poor pixel alignment in the video memory.

Driver VESA_32

This driver is initialized with function:

```
PegScreen * CreatePegScreen_VESA_32(int Width = 800, int Height = 600);
```

VESA_32 supports any VESA 32-bit color modes if the complete video memory is accessible in a linear address range. VESA BIOS version 2.0 or higher is required. This driver does not set the video mode at run time; rather, it expects that the desired mode has been set by RTTarget-32's boot code using the *GMode* command.

Parameters Width and Height are only evaluated by the Windows emulation drivers. On the target, the screen resolution is determined by the current graphics mode set by the boot code through a *GMode* command.

If the current video mode is not a VESA 32-bit color mode, or the video mode does not use a linear frame buffer, NULL is returned.

Driver VGA_4

This driver is initialized with function:

```
PegScreen * CreatePegScreen_VGA_4(int Width = 640, int Height = 480);
```

VGA_4 supports standard VGA BIOS mode 12h and VESA mode 102h, which is 16 color and 640x480/800x600 pixels resolution. This driver does not set the video mode at run time; rather, it expects that the desired mode has been set by RTTarget-32's boot code using the *GMode 12h* or *GMode 102h* command.

Parameters Width and Height are only evaluated by the Windows emulation drivers. On the target, the screen resolution is determined by the current graphics mode set by the boot code through a *GMode* command.

If the current video mode is not 12h or 102h, NULL is returned. Note that this driver is significantly slower than the VESA drivers due to the pixel plane organisation of the video memory.

Driver VGASCRN

This driver is initialized with function:

```
PegScreen * CreatePegScreen(void);
```

Driver VGASCRN does not require any BIOS support, but it needs 100% VGA compatible video hardware. This driver will set the video controller to 16 color and 640x480 pixels resolution.

Note that this driver is significantly slower than the VESA drivers due to the pixel plane organisation of the video memory.

Chapter 4

Demo Programs

This chapter describes the various graphics demo programs shipped with RTPEG-32. If you intend to run these programs under the debugger, be sure to use the Graphics Monitor (GraphMon) to boot the target. All demos except PegDemo expect to run with a 256 color VESA mode.

Demo Cross Reference

The table below lists all RTPEG-32 demo programs with the classes used:

PegDemo	Robot	Table	TreeView
PegDialog	PegButton	PegTitle	PegTitle
PegWindow	PegPrompt	PegMenuBar	PegTreeView
PegDecoratedWindow	PegThing	PegTable	PegTreeNode
PegTextBox	PegPrompt	PegMessageWindow	
PegTextButton	PegSlider	PegIcon	
PegTitle	PegTextButton	PegTextButton	
PegMenuBar	PegTimer	PegPrompt	
PegStatusBar	PegAnimatedWindow	PegRadioButton	
PegMessageWindow		PegGroup	
PegString			
PegEditBox			
PegPrompt			
PegHorzList			
PegBitmapButton			
PegComboBox			
PegSlider			
PegProgressbar			
PegSpinButton			
PegRadioButton			
PegCheckBox			
PegGroup			
PegMLTextTextButton			
PegTimer			
PegVertList			
PegHorzList			

Notebook	Spread	Guage	Dialog	Terminal
PegTitle	PegTitle	PegWindow	PegCheckBox	PegTerminalWin
PegMenuBar	PegMenuBar		PegPrompt	
PegNotebook	PegStatusBar		PegString	
PegTextBox	PegSpreadSheet			
PegGroup	PegMessageWindow			
PegPrompt				
PegCheckBox				
PegRadioButton				

All RTPEG-32 command line demo makefiles contain an additional target with letter "W" appended to the demo's name to build a Windows version. For example, the command:

```
make PegDemoW
```

would build program PegDemoW.exe which can execute under Windows.

Program PegDemo

PegDemo is the largest of all RTPEG-32 demos and serves as an introduction to what RTPEG-32 can do. Many different classes are demonstrated. Unlike all other demos which use the VGA_8 driver, this demo also shows how a program can dynamically detect at run-time which driver should be used. By defining preprocessor symbol RTK32 and/or PEG_UNICODE, PegDemo shows how to use RTPEG-32 with RTKernel-32 and multiple threads and/or with Unicode support enabled.

Program Robot

This demo shows how several bitmaps are displayed in rapid sequence, giving the impression of a video film.

Program Table

This demo shows how to use class PegTable and how a table can be populated with objects of different types.

Program TreeView

This demo displays the RTPEG-32 class hierarchy using class PegTreeView.

Program Notebook

Notebook demonstrates several features of the PegNotebook class.

Program Spread

This program shows a spreadsheet using the PegSpreadSheet class.

Program Gauge

This demo shows how off-screen drawing can be used to display animated controls.

Program Dialog

Program Dialog is a small and simple program showing how information is passed to and retrieved from controls which make up a dialog box.

Program Terminal

This demo shows a PegTerminalWin and how to add a simple command processor to the window.

Program Unicode

Demo program Unicode is only included with the RTPEG-32 Source Code product for Microsoft Visual C++ and Borland C++. It allows switching the display of all strings between English and Japanese by clicking on a button at run-time. All string references make use of the String Table, which is a multi-dimensional array of Unicode strings produced by WindowBuilder. (Note that the program's windows themselves are not included in the WindowBuilder project of this demo.)

The demo program switches between the normal fonts (SysFont and MenuFont) and the Unicode font when switching between English and Japanese. This isn't really necessary but the appearance of the normal fonts is slightly better than the Unicode font. The Unicode font used for this demo is MS Song.

The process for creating the Unicode font is this: Using Fontcapture, generate individual fonts for each code page required for the Unicode font. For this demo, the following code pages were used to produce a few font files:

- Latin-1 code page (unilatin.cpp)
- Hiragana code page (hiragan.cpp)
- Katakana code page (katakan.cpp)

- CJK code page. There is a limitation in the PegFont that prevents any one font from being more than 65k bits wide. Since CJK contains about 28,000 characters, this code page is broken into sections. The natural place to divide this is by Unicode hex ranges, i.e. 0x4000 to 0x4fff, 0x5000 to 0x5fff and so on. The font files generated, Kanji4 (CJK characters 0x4000 to 0x4fff) and Kanji5 - Kanji9 each contains 1000 hex CJK characters.

These are the code pages needed to support English, Japanese, Chinese, and Korean.

To maintain a multi-language string table in WindowBuilder, do the following: In WindowBuilder, create a project. Under Configure|Languages, define how many languages are required and what they are named. Next create the Unicode font(s). Click on the Fonts tab and use Project|Add Font. Select the *Composite* option to create one large font from several smaller fonts. Select the *Reduce* option to have WindowBuilder create a reduced PegFont which contains only those characters actually used in the string table, instead of all 30,000 characters contained in the total composite font. Enter a name for the composite font. When *Composite Font* is checked, the button changes to say *component fonts*. Clicking on this button will display a table that allows defining which sub-fonts will be included in the composite font. In this demo, the Add button was used to add all of the individual fonts defined above.

Under Project|String Table, all strings of the application can then be entered and edited. When done, Project|Update|Strings will generate the strings source files (by default, files wbstring.hpp and wbstring.cpp). These files enumerate all string ids and the encoded Unicode strings and must be compiled with the application.

Chapter 5

Utility Programs

RTPEG-32 is delivered with three host tools to aid developing embedded GUIs. These programs are described in this section.

Window Builder

Window Builder is a rapid prototyping and design tool used to quickly create windows and dialogs. Window Builder is a Win32 application program and will run under Windows 95/98/NT/2000. Window Builder is actually a PEG application program to guarantee that all GUI objects are displayed in Window Builder by the same code as on your target.

Project Files

Window Builder project files have the extension ".wbp". The project file maintains information about the source files, target system, images, strings and fonts, etc. used by the application. You can save your work at any time, and later re-open the project file and modify your target screens.

All project file path information, such as the location of image files referenced by your project, is maintained in a relative path format. This means that Window Builder project files can easily be copied from one computer to another as long as all related font and image files are maintained in the same sub-directory structure. If Window Builder does not find a required image or font file using the relative path information, it always attempts to find the file in the directory containing the Window Builder project.

Output Files

The goal of Window Builder is to produce C++ source files, ready to compile and run on the target system. These source files may be one of three types: C++ class definitions, image file data structures, or string table data. For most .cpp source files, Window Builder also creates a corresponding header file. These header files contain class prototypes, message definitions, control IDs, string IDs, and other definitions required for the application software to compile and run.

Each top-level module in a Window Builder project will generate one C++ source module and one corresponding header file. These files contain the PegWindow derived classes that constitute the application screens. The source code created by Window Builder contains both the object initialization code required to create the window or dialog under construction, and the message handling functions required to process any signals enabled for individual controls. While it is your job to insert the actual signal handling code, Window Builder creates a framework with the individual signal case statements.

If you use BMP, GIF, or JPG images in your project, Window Builder will also produce a single image source file containing all images added to the project.

Project Window

The Window Builder environment contains three main windows: the Project window (upper left), the Preview window (lower left), and the Target window on the right. The Project window is used to modify global configuration settings, maintain the source files included in the open project, and instruct Window Builder to perform various operations affecting the entire application program. The Preview window displays images and fonts that you have added to your project and allows you to drag those images and fonts to various target objects. The Target window provides true WYSIWYG emulation of the target system display screen.

All global project information is maintained in Window Builder Project window. A project consists of any number of source files and associated classes, along with fonts, images, and strings used. While it is possible to use multiple project files for a single system, this is discouraged since in this case there is no way for Window Builder to prevent the duplication of source code or data.

Internally, the Project window directly corresponds to and continuously updates the Window Builder project file with extension .wbp (Window Builder Project). The Project window contains a command menu and a PegNotebook control described in the following sections.

Project | New/Open/Save/Close

These commands create, open, save, or close a project file.

Project | Add Module

This command adds a new source module to the current project. One or more source modules must be added to a new project before you can edit it with Window Builder. The Source page of the notebook must be selected for this command to be active.

Project | Add Image

This command adds a new image to the current project. After an image has been added, it can be used to fill the client area of various controls. The source for the image must be a .BMP (Windows Bitmap) or .GIF (Compuserve .GIF) file.

Window Builder uses RTPEG-32's run-time image conversion classes that are included with the library to read, decompress and process these images. When you update your output image file, these images will be saved in C++ data array format.

The Image page of the notebook must be selected for this command to be active.

Project | Add Font

This command is used to add a custom font. Once a font is added, it can be applied to any text-related object simply by dragging the font from the Preview window to the object that should be assigned the custom font. The input for adding a font should be a C++ font file created with PegFontCapture.

The Fonts page of the notebook must be selected for this command to be active.

Project | Update | Source

This command instructs Window Builder to update the currently selected source and include files to reflect the changes in the current project. Please refer to section *Source Code Generation* later in this chapter for details.

Project | Update | Images

This command instructs Window Builder to re-process all input image files, and save the resulting PegBitmap information to the C++ image file in the source directory. Note that the output image file is completely regenerated each time an image update is performed.

The process of generating the output image file can be quite complex and take up to a few minutes, depending on the target screen color resolution.

If the target system supports 256 or more colors, Window Builder will scan each image in the project, create an optimal palette for displaying those images, remap the image colors back to the optimal palette, RLE compresses each image, save the custom palette, and save each newly-encoded image file in C++ style source data structures.

For 16 color targets, the Update Images command is slightly less complex. In this mode, Window Builder dithers each image to a fixed orthogonal 16-color palette, RLE compresses the images for which this saves memory, and saves the resulting bitmaps in C++ style source data structures. For 2 and 4 color targets, the Update Images command simply saves each image in the selected format.

Project | Update | Strings

This command instructs Window Builder to re-create the string data table and associated string ID header file. Note that this file is completely regenerated each time an Update Strings command is issued.

For Unicode enabled systems, this command does far more than simply write out all strings as C++ string arrays. Rather, Window Builder performs the following operations:

- Scans string tables for all languages, creating the global table of required glyphs.
- Re-scans all fonts used by the application, saving only the required glyphs.

- Re-encode string information using new encoding of glyph indexes.
- Create C++ wide-string arrays using newly encoded character strings.

Project | String Table

This command brings up the string table edit window. This window allows you to define the literal strings and string IDs used for each language in the system. The string table is further described in section *String Table*.

Configure | Directories

This dialog window allows you to specify the complete path for your source files, include files, and image files. The Source Files directory indicates where the source files generated by Window Builder will be saved. The Header Files directory indicates where the header files generated by Window Builder will be saved. This directory can be the same directory as the source files directory, if desired.

The Image directory indicates where the source image files are to be found. Note that all of your source images must be saved in a common directory. This is where Window Builder will look for your image files in order to convert them into the PegBitmap source code format. All converted images are saved in a single output file located in the source files directory. The name of this single output file is specified in the Image File Name field.

The String Filename field allows you to specify the name of the Window Builder output file that will contain your project string data. There are actually two output files for string information; one contains the literal string table, and the other is a header file containing the string IDs for each string. Both of these files will have the filename specified in the String Filename field; however, the literal string table will have the extension .cpp and will be saved in the Source Files directory, while the associated header file will have extension .hpp and will be saved in the Header Files directory.

The Backup directory indicates where Window Builder will save backup copies of files before updating them. Backups are created for all source, image, string, font, and project files. To disable file backups, set the Backup directory to a NULL string. It is **not** recommended that you disable file backups.

Configure | Target

The target's screen resolution and color depth can be set here. For a target with more than 256 colors, specify 256 colors.

Configure | Default Fonts

This command allows you to inform Window Builder of the default font settings you would like to use. These settings default to the standard settings provided in the RTPEG-32 library. If you modify the default setting, Window Builder must be informed of this to prevent the generation of unnecessary 'SetFont()' function calls. If you are using custom fonts for some or all of the default fonts, you must first install these fonts before you can configure them using this command. Installing new fonts into your project is described in a section *Fonts Page*.

Configure | Languages

This command invokes a dialog window that allows you to specify the number of languages supported by the system software and the character encoding used for string storage. This in effect determines the layout of the String Table that will be generated by Window Builder, and also determines how strings will be entered when new objects are created that display text or string data.

You will be prompted to enter a name for each supported language. These language names will be saved as an enumeration in the string data file. For example, you might configure your system to support 3 languages, *English*, *German*, and *Dutch*. The first language name is the default language at application startup.

If your application does not require multiple language support, or if you for any reason do not wish to maintain your string data in the Window Builder string table, you can de-select the *Enable String Table* checkbox on the language configuration page.

Configure | Style

This command invokes a dialog window that allows you to enter several lines of ASCII text that will be included in the source file headers generated by Window Builder. You can also modify various style settings which control the format of the source code generated by Window Builder.

Configure | Remote

With this option, Window Builder can be instructed to display the complete target screen in a separate window on the Windows desktop in addition to the smaller window in the target preview pane. If you select *Test Mode Only*, the remote target screen will be updated only when you select *View / Test* in the target window menu. *Contiguous* causes the remote screen to be updated automatically while you edit.

Source Page

Window Builder organizes the application program possibly containing hundreds of unique application windows into unique modules. Each module corresponds to one window or dialog and produces one source file and one include file.

The Source page of the Project window notebook control contains a PegTreeView depicting each of the modules included in the current project. Each top-level node of the PegTreeView control represents a top level class constructed with Window Builder. If you expand a top-level node, you will see the corresponding source and header file associated with that top level class. If you select a source module by left clicking with the mouse, the Target window will display the objects defined within that source module.

The Target Window always operates on the selected module. If no module is selected, none of the Target Window editing commands are operational. You can at any time create a new module by selecting the *Project / Add Module* command.

The *Filename* field allows you to specify the output filename for the source file Window Builder will generate for this new module. Any valid filename may be entered into this field. You do not need to specify an extension, as Window Builder will automatically write both a .cpp and a .hpp file for this module.

The *ClassName* field allows you to specify a name for the PegWindow derived class you are creating, i.e. the name of the class which will define the window or screen you are developing.

The *Parameters* field allows you to specify any user-defined parameters you would like to pass to the class constructor (in addition to the parameters Window Builder will always pass to the constructor). If you wish to pass extra parameters, you should type them on this line exactly as they should appear in the constructor prototype, i.e. Type-Name, Type Name, etc. for each parameter.

The *Startup Window* checkbox specifies that this window will be the first displayed when your application executes. When this checkbox is selected, Window Builder automatically writes the PegAppInitialize function in this module such that this window is created and added to PegPresentationManager during program startup.

The *Base Class* field allows you to specify which RTPEG-32 library class will be the base class for your new Window. This will usually be PegDialog, but could also be PegWindow, for example.

The *Overrides* group is used to tell Window Builder which function of the base class will be overridden by the class you are defining. Only two options are supported by Window Builder (although you can, of course, add your own function overrides to the completed class). These are the Message function and the Draw function. The default setting of this field indicates that you will override the Message() function (to catch signals from child controls) but will not override the Draw() function. This is the most common situation.

The *Absolute Position* checkbox allows you to use an alternate form for the class definition. Normally, Window Builder produces a class that accepts a left-top corner position as the first two incoming parameters. The window and all child controls are positioned relative to this left-top position. If desired, you can produce a class that is absolutely positioned, i.e. there is no left-top incoming parameters and the window and child controls use absolute pixel positioning.

When you have completed entering in the required information, a new object of the selected type is created and displayed in the Target window, and the new source file is added to the source page tree view control.

To remove a source module and its associated objects from your project, select the source module in the tree control and press the *Delete* key on your keyboard. Following confirmation, the source module is removed from the current project. Note that the actual source files corresponding to the selected node are not deleted. Window Builder simply removes all information about the source file from the current project.

To modify the parameters associated with a source module after the module has been created, you can right-click with the mouse on the module in the Source notebook page. This will bring up a dialog window allowing you to change the module name, file name, and other module parameters.

Note that you are not allowed to delete sub-nodes from the PegTreeView displayed within the source page of the notebook. This is accomplished by individually deleting objects from within the Target window as described below.

Images Page

The Images Notebook page lists the BMP, GIF, and JPG image files included in the current project. Images are imported into a project by using the *Project | Add Image* command on the menu bar while viewing the Images notebook page.

Images are deleted by selecting the image in the Image notebook page and pressing the *Delete* key. Any objects that had been using a deleted image are re-configured to use a default bitmap.

When an image is selected with the mouse or keyboard on this notebook page, a preview of the image is shown in the preview window. The image can be applied to a bitmap-based control by dragging the image from the preview window to the target control.

When you assign an image to certain types of objects by clicking on the image and dragging it onto the object, Window builder will ask you if you would like to re-size the object to fit the image. If you select yes, the target object is re-sized such that the image fits neatly within the object. If you select no, the image is centered within the client area of the target object and the target object size is not modified. For other object types, resizing is done without prompt since this is the only mode of operation supported by that object type.

Fonts Page

The Fonts page is very similar to the Images page. This page lists the fonts that have been added to the project and allows you to drag-and-drop fonts onto specific objects.

In a new project, two fonts are listed on the Fonts page. These are the System font and the Menu font. These fonts are part of the RTPEG-32 library and are always available for use. Note that you cannot delete these fonts from a project.

You can add any number of additional fonts to a project and apply them to any text-display control. You must pre-generate the fonts you will add to your project by using the FontCapture utility program. Window Builder is able to read the C++ source files produced by FontCapture and create PegFont data structures in memory using these source input files.

The operation of adding a new font to a project varies depending on the language configuration settings. If the current project is configured to use only ASCII characters without the String Table, adding a new font is simply a matter of choosing a font file produced by the FontCapture program.

Composite Fonts and Reduced Fonts

If the project is configured to support multiple languages and Unicode, the operation of adding and defining a font becomes more complex. This is due to several factors, primarily the large number of characters (> 30,000) which may be required for the support of multiple languages.

The Font Name field assumes the name of the source font for standard (i.e. non-Composite, non-Reduced) fonts. For Composite or Reduced fonts, Window Builder will produce a completely new PegFont from the input font data when the String Table file is generated. In this case, you can assign any name to the new font by typing into the Font Name field.

Composite Fonts are fonts collections, produced and managed by Window Builder, containing multiple sub-fonts produced by the FontCapture program. Composite fonts are needed to overcome limitations in the character set or alphabet included in most TrueType or BDF font files. In many cases you will find it is impossible to obtain one single TrueType or BDF font that contains all of the characters required by your application program. For example, one TrueType font may contain the Latin and Cyrillic characters, while another contains Kanji and Hangul. It is very rare to find a single font that contains characters for many different alphabets.

In order to avoid re-assigning the font associated with every PEG object when a language change is made, it is desirable to have a single font (or multiple fonts of different sizes, each containing the same character set) that contains all of the characters used by your application program. This is the reason for Composite fonts. You can combine any number of sub-fonts into one SuperFont potentially containing all the characters from every sub-font.

The true power of Composite Fonts is realized when combined with the Reduce Font option. This option instructs Window Builder to produce a new PegFont wherein only those characters actually used by your application strings are included in the new PegFont. This information about which characters to include is obtained by examining all strings found in the project String Table (described below). By using the Reduce Font option, you can save a tremendous amount of ROM storage for your fonts for languages with very large alphabets, such as Asian languages. If you select the Composite Font option, you can then select the *Component Fonts* button to edit a table which defines each sub-font that will be included in the composite font.

For each sub-font that you add to your composite font, the range of characters used from the sub-font will default to the full range of characters contained in the sub-font. Since it is possible that several sub-fonts may contain overlapping characters, you may need to edit the First Char/Last Char ranges displayed in this table so that each sub-font provides a non-overlapping range of characters to the final composite font.

When you have completed defining the sub-fonts that will make up your composite font, you simply close this table by pressing the *Done* button, and after naming your composite font click the *OK* button on the Font properties dialog.

You can return and re-edit your Composite font settings at any time by right-clicking on the font name in the Font tree display. Note, however, that while you can change the component font list for a Composite font, you cannot change a previously added non-Composite font into a Composite font, nor can you change a Composite font into a normal font. Instead, you must delete the font from your project and newly add the font using the desired settings.

Target Window

The Window Builder Target window displays as accurately as possible a representation of the target system display screen.

When you create objects within the target window, you are actually defining new instances of RTPEG-32 objects. These objects are dynamically constructed and added to the Target Window, and operate just as any normal RTPEG-32 objects. This is important to remember. As you create your windows and dialogs within Window Builder, you are creating a working RTPEG-32 program. You can at any time interact with the objects you have created, just as the end user of your system software will interact with the final system.

This also causes some confusion sometimes when you first begin working with Window Builder. For example, consider the case where you have created a new modal dialog window, and you set the dialog system status to be non-movable. Your intention, of course, is that the end-user will not be able to move the dialog window. However, you will not be able to reposition the dialog window from within Window Builder either! In order to move the dialog window, you will have to temporarily set the system status to be movable, move the dialog to the desired position, and then reset the system status to the desired value.

As you work within the Target window, the internal copy of the Window Builder project file is updated to reflect all of your changes. The source code files for your project are **not** updated until the *Project / Update / Source* command is invoked.

The target window becomes active when a source module is selected in the project window. If no source files are included in your project, you must first create a new source module before you will be able to do editing in the target window. When you create a new source module, a default object of the type defined in the new source module is generated and is the initial object displayed in the target window.

Target Window Status Line

The Target window status bar helps you in positioning and sizing the objects you create and place on the Target screen. The first status bar field indicates the type of object that has been selected. The next field indicates the top-left pixel position of the selected object(s). The next field indicates the width and height of the selected object(s). The last field on the status line indicates the current pointer position when the pointer is over the Target window. This pointer position is relative to the top-left corner of the target screen, which is position 0,0.

Selecting Objects in the Target Window

Almost all selection and editing of objects in the Target window is done using the mouse. When you click on an object in the Target window, a dark border is drawn around the object to indicate that the object has been selected. You can re-size any object by dragging the dark border with the left mouse button held down until the desired size is obtained.

You can move an object by either dragging a selected object with the mouse, or by using the keyboard arrow keys.

Multiple objects can be selected by holding the *Ctrl* key down while right-clicking on additional objects. When multiple objects are selected, the selection box expands to contain all selected objects.

The target window menu commands always operate on the currently selected object. You should select an object or group of objects before selecting one of the menu commands described below.

Add Menu

The Add Menu brings up sub-menus of various control and window objects. Selecting an object adds it to the selected current object. In this case, the currently selected object would usually be the top-level window or dialog, or possibly a PegGroup.

Note that adding a Vertical Scroll or Horizontal Scroll adds a client area scroll bar. This is a user-defined scroll bar rather than a scroll bar which acts to scroll the window client area. Normal non-client-area scroll bars are added by adjusting the window properties.

Edit | Copy

Copies the selected object or objects, including all status and style flags. Only one object can be selected when the *Edit / Copy* command is issued; however, that object can have any number of children. When an object such as a PegGroup is copied, and the PegGroup has children, the Group and all of the group's children are copied.

When this command is selected, Window Builder automatically changes the selection box to contain the parent of the current object. This allows you to quickly copy and paste an object into the object's parent, which is the most common operation. You can select an object, copy it, and then select an entirely different object to paste the copy into.

Edit | Paste

This command pastes an exact copy of the copied objects into the center of the selected object. Window Builder automatically selects the parent of the copied object as the target for the paste command. You can override this operation by selecting any other parent before selecting the paste command.

Edit | Properties

This command invokes a properties dialog for the selected object. One and only one object must be selected in order for this menu command to be active. You can also invoke the edit properties dialog window by right-clicking with the mouse on the selected object.

The properties dialog is context sensitive depending on the type of object which has been selected. In general, you can adjust the border style, system status flags, and style flags for a given object by selecting each page of the properties dialog notebook control. Many object types have additional settings which can be controlled using the properties dialog.

The properties dialog is also where you specify the text string associated with many object types such as PegPrompt or PegString. While you can directly type text into a PegString object, this does not set the initial text value displayed by the string. The initial text displayed is determined by the String ID associated with the object in the object properties dialog. The same is true for PegTextBox objects.

For text-based control types, the properties dialog includes a notebook page with a table labeled *Extended* that allows you to select the string ID associated with an object. This string ID is a member of the string table maintained by Window Builder. You can view the string table by selecting the drop-down list of the *ID* field.

If you have disabled the use of the Window Builder string table in the *Project | Configure | Language* dialog, the String page of the properties notebook allows you to directly enter the ASCII string used to initialize a control.

Layout Menu

This group of commands is used to evenly align any number of child controls. Before activating this command, one or more child controls should first be selected using the method described above. The above group of commands can then be used to align the group of objects as desired.

View | Test Mode

This command places the target window in test mode. In test mode, all of the Window Builder windows are hidden, leaving only your newly created window or dialog on the screen. While in this mode, your new window or dialog will operate exactly as on the final target system, although any message processing code you have added to the window or dialog will not be operational from within Window Builder.

While in test mode, you will not be able to select and edit objects. You can exit edit mode by closing the window or dialog under test, or by pressing the *Stop* button placed in the lower right hand corner of the screen.

View | Maximize

The maximized view is similar to Test Mode, but it only hides the project and preview windows, enlarging the available view area for the target screen. The menu is still available, and the image can be edited.

Note that you can also maximize Window Builder itself to get more space for the Target window.

String Table

If you have enabled the use of string tables in the Window Builder *Project | Configure | Language* dialog, Window Builder will maintain a table containing all strings for all languages used by the application. In this environment, all PegTextThing derived classes are constructed using StringID information, rather than literal strings. This allows your system software to easily convert between different supported languages.

The String Table is composed of an array of literal strings, a two-dimensional array of string pointers, and an enumeration of string ID values. String ID values are just indexes selecting the correct row from the two-dimensional table. Each table column is associated with one of the supported languages. If the application supports only one language, the String Table is simply a single list of literal strings, an array of string pointers, and an associated String ID enumeration.

The correct string table column is selected by the current language, which is maintained in a static member variable of the PegTextThing class. This variable should be loaded with one of the enumerated language names when the active language is selected by calling the PegTextThing::SetLanguage() function. The default language is the first language configured in the *Configure | Languages* dialog box.

The string table is saved to the filename specified in the *Configure | Directories* dialog. The enumeration of the language names and the string table IDs is saved in the corresponding String Table header file. Each source file that uses the String Table must include the String Table header file in order to resolve the string IDs and language names. This include is added to each source file generated by Window Builder.

The String Table is edited by selecting the *Project | String Table* command on the Project window menu bar.

The left-hand side of the String Table Editor window displays a PegSpreadSheet object containing each of the strings used in your system. Each row of the table corresponds to a StringID, and each column of the table corresponds to a supported language. The enumerated language names are displayed as the table column headers.

The String Table can be displayed in a two-column or three-column format. You can change the format by right-clicking over the spreadsheet and selecting the desired format in the pop-up menu. You can also sort the string table entries by using the right-click pop-up menu. This menu provides commands to shift the selected entry up or down in the string table.

The first language listed on your language configuration page is your project's *Reference Language*. This language will usually be English, but may be any language desired. The reference language is important because this is the language you are working in when you work in the target window. This is also the language which is always displayed when you view the table in three column mode.

String Table Edit Fields

The right-hand side of the String Table Editor window displays a series of fields for editing the selected string. The first field, the ID field, is where you can modify the string ID name, which is the name associated with each string ID. This name will be included in your string file as an enumerated list, and you will use this name in your application software when you want to refer to a particular string. You can edit this name simply by typing on the keyboard.

You can select the font to use while working in the string table using the drop-down list box labeled Font. The selected font is displayed in the grid in the lower-right portion of the screen. This grid allows you to select characters while editing the current string.

The second field on the right side of the String Table Editor window is the string literal edit window. This field displays the string literal value using any of the fonts which are part of your project.

There are three methods for editing strings displayed in the string literal edit window: First, if the current language alphabet is supported by your keyboard, you can simply type the string value. Second, you can simply click on characters displayed in the font viewer window. As you click on the characters, they are inserted into the current string at the current insertion point. Finally, you can type the JIS (*Japanese Industrial Standard*) or Unicode encoding value in hexadecimal for the character you wish to insert. For people who know the encodings for common characters, this is faster than finding the characters in the font display window.

As you edit the selected string, the width of the string (in pixels) is displayed in the Width field.

The Notes button brings up a small note editor window. Notes are useful for including additional information about each string, usually for the benefit of translators who will translate your English or reference language strings into strings for the other languages.

The reference language is also important in the event that a translation is not required or available for certain strings in your application. Empty (NULL) strings in any other language are replaced with the corresponding (untranslated) string of the reference language.

Merging String Tables

The *Merge* button on the String Table Editor window invokes a series of dialogs that walk through the merge process. In order to understand the reason for the merge operation, we need to examine the life-cycle of a typical multi-language project development.

1. The system developers define the initial string table. The total number of languages and the language names are defined using the *Configure / Languages* dialog.
2. The String ID names and the Reference Language (English) strings are initialized for all strings in the application using the String Table Editor.
3. The Window Builder Project file, along with the WindowBuilder executable program, are distributed to translators who will each fill in one column of the string table. These translators may reside at the same location, but could also be located all around the globe.
4. The translators return WindowBuilder project files to you, and the returned project files each have one or more additional columns of the string table filled in with translated strings.

The problem should now be obvious: The Merge operation will merge strings for selected languages from a second project file into the current project file. The process is actually very simple as you are guided step-by-step through the merge process. When WindowBuilder performs the merge, it looks for matching string ID names in the secondary project. For each matching string ID name, if the selected language in the secondary project has a non-NULL string value, that string value is copied into the current project for that specific string ID and language.

Source Code Generation

The final goal of running Window Builder is to produce the C++ source code you will use to display your application screens. You will need to edit and add your own program logic to the source files produced by Window Builder. Most significantly you will need to add program logic to catch signals generated by your child controls. You may also need to make any number of other additions and changes to the source files produced by WindowBuilder.

At the same time, you will want to be able to run Window Builder again and again to modify your screens and update the source files without losing any of your hand-coded changes. This is not difficult to do as long as you understand how Window Builder updates your source files and follow a few simple rules.

When you instruct Window Builder to produce/update the source files using the *Project / Update / Source* command, Window Builder first looks to see if the source file already exists. If it does, Window Builder enters *Merge Mode*. In Merge Mode, Window Builder is very careful not to lose any of your custom modifications. The rules are this: Window Builder will find and re-write the section of the source file delimited by the start of the constructor and the comment line which reads:

```
/* WB End Construction */
```

To avoid losing your changes, never make any manual edits between the start of the class constructor and this comment delimiter.

Window Builder also searches for the `Message()` member function, if present, and updates this function to contain any new SIGNAL cases not already present. Window Builder will NOT remove case statements from your Message function, even if the control which generated a specific SIGNAL is no longer a child of the window. In short, deleting obsolete sections from your source files is your responsibility; in the interest of safety Window Builder will not delete source lines from your Message function.

Any and all code outside of the class constructor and Message function is maintained without modification during the source code merge process. That is, any other editing that you have done will be preserved entirely during the source file update process.

Child Object Pointer Control

You can control the type and name of the pointer (if any) used when each child object of the top-level window is created. Controlling how pointers are used is done by adjusting the basic properties, using the properties dialog, for each child control. There are four types of pointers used by Window Builder during

code generation: Member pointers, Automatic Named pointers, Automatic Temporary pointers, and Implicit pointers. We will describe each type below and describe how you can control the use of pointers in the generated source code.

Implicit Pointers

An implicit pointer is used by Window Builder when no references to an object are made after the object has been created and you have not chosen to create a member or automatic pointer. In this case, Window Builder does not need to keep the address of the newly created child in any variable, and therefore uses an in-line, implicit pointer to pass the child's address to the `Add()` function. The following is an example of source code produced by Window Builder that uses an implicit pointer:

```
Add(new PegPrompt(ChildRect, "Text"));
```

Note that the return value from the new operator is not saved, but is passed directly to the `Add()` function. When no other pointer type is needed, this is the default pointer style used by Window Builder.

Temporary Pointers

This type of pointer is used by Window Builder when reference to an object is required after it has been created, but you have not requested an automatic or member pointer to be created. In this case, Window Builder will create a temporary automatic pointer to hold the address of the child object instance. The temporary pointer is called *Automatic* because it is created on the execution stack, i.e. space for the pointer is allocated automatically by the compiler on the stack, and the space is destroyed when the function (in this case the class constructor) returns.

A common example of this might be a `PegGroup` container added to the top level window. During code generation, Window Builder needs to maintain the address of the `PegGroup` instance while creating and adding child controls to the group. Window Builder will default to using a temporary pointer for this purpose, which produces the following source code:

```
PegThing * pChild1;

pChild1 = new PegGroup(...); // keep temp pointer to object
pChild1->Add(..);             // add second-generation children to object
pChild1->Add(..);             // ditto
Add(pChild1);                // add object to top-level window
```

Window Builder will always use the generic names `pChildx` for temporary automatic pointers. Window Builder will reuse the temporary pointers for new objects if needed and available during code generation. In some cases, multiple temporary pointers are required simultaneously, in which case Window Builder will create and use as many temporary object pointers as needed.

Automatic Named Pointers

Similar to automatic temporary pointers, Automatic Named pointers are created on the execution stack and only exist during the class constructor. Named pointers are created by typing a name into the *Pointer Name* field in the object properties dialog basic properties page and unchecking the *Member Pointer* box.

Member Pointers

A member pointer is a pointer to a child object which is maintained as a member variable of the parent window class. This pointer is initialized in the class constructor and used at all times to reference the child object. You can instruct Window Builder to create a member pointer for a child object by checking on the *Member Pointer* checkbox in the properties dialog and typing a name in the *Pointer Name* field.

Image Convert

Image Convert (program Pimagcon) is a utility program that can be used to convert .BMP, .GIF, and .JPG files into binary or source code formats supported by RTPEG-32.

Input files can be 1, 2, 4, 8, 16, or 24 bit-per-pixel formats. Likewise, Image Convert can generate 1, 2, 4, 8, 16, or 24 bit-per-pixel image files. During the image conversion process, the palette used to encode the output image can be saved in source format, suitable for use in calling the PegScreen SetupPalette() function.

The actual operation of Image Convert strongly depends on the selected output format. Image Convert may perform color reduction, dithering, RLE compression, and transparency encoding for all input file types. In addition, for 8-bpp output formats, Image Convert adds optimal palette generation.

While Image Convert can output PegBitmap structures in many formats, this is of little use if the PegScreen driver being used on the target is not capable of properly displaying the bitmaps in the format in which they are saved. For example, while Image Convert can save PegBitmap structures using 2-bpp encoding, this format should only be used if your derived PegScreen driver class understands and can properly display images saved in 2-bpp format. The PegScreen derived interface classes provided with On Time RTOS-32 support both 8-bpp bitmap encoding and the native encoding corresponding to the color depth of the target display.

All of the options that can be selected on the Image Convert dialog window control the output of Image Convert. The format and data content of the input file(s) is determined by Image Convert by reading and parsing the input file header information.

If the target supports 256 or more colors, Image Convert can also perform advanced palette reduction and optimization, allowing the creation and use of any number of color palettes, each of which is optimized for the images displayed in the application. This option is best utilized along with batch image processing (described below), which allows a custom palette to be created for optimal display of multiple images. The input files for list processing can be any combination of the supported file types, and can even have different internal formats in terms of the color resolution associated with each input file. Image Convert utilizes an improved form of Heckbert's *Median Cut* algorithm for color reduction.

Input File

The input file string allows you to select the source image file. You can either type in the name of the file, or you can use the *Browse* button to select a file from your computer.

Only one file path\name can be entered in the Input File string field. If you want to process multiple input images at one time, you should enter the image names in an ASCII command file and select the command file as the input file. This is described in more detail in the section Batch Conversion.

The basic input file type is determined by Image Convert based on the input filename extension. For MS or OS/2 Bitmap files, the filename extension should be .BMP. For GIF files, the filename extension should be .GIF, and for JPEG files the filename extension should be .JPG. To process multiple input files, the filename extension should be .CMD, which is interpreted as a command file.

Image Convert verifies that the file is truly of the type indicated by the file extension by attempting to read the file header information and verifying that the file header makes sense for the indicated file type. An error is reported if the file header information and filename extension do not correspond.

Output File

The output file string field allows you to specify where and to what file name Image Convert will save the generated output file. Image Convert also uses this filename as the name of the final PegBitmap object. For this reason, you should enter the filename in exactly the form you want the resulting bitmap to be named, including upper and lower case characters. Image Convert pre-fixes the letters *gb* to the bitmap name, and post-fixes the letters *Bitmap*. For example, the following output name:

```
C:\mybitmaps\House.cpp
```

will result in the final bitmap being named *gbHouseBitmap*. This is the name you will use in your source code when referring to the bitmap. To use this bitmap on a PegBitmapButton button, for example, you would then do something similar to the following in your source code:

```
extern PegBitmap gbHouseBitmap;

void MyWindow::MyWindow(...)
{
    Add(new PegBitmapButton(20, 20, &gbHouseBitmap));
    ...
}
```

The naming convention for the resulting PegBitmap structures is slightly different when batch processing, which is described later.

Compression

Image Convert can optionally apply a simple RLE compression technique to the output data. The effectiveness of this compression depends on many factors. If the input image is a computer generated image with few colors, RLE compression can be very effective. If the image was produced with a RAY-tracing package or from an actual photograph, RLE compression is less successful.

When RLE compression is enabled, Image Convert is required to save the output data in 8-bpp format, regardless of what format was selected in the Output Colors field. This means that for 1-bpp, 2-bpp, and 4-bpp input images, turning on RLE compression forces Image Convert to first expand the image to 8-bpp format, and then apply RLE compression. Depending on the exact image file, this can actually cause the final output file to be larger than if compression is not used.

For this reason, selecting RLE compression is actually only a suggestion to Image Convert. If RLE compression is effective at reducing image size, the compression is performed. If compression does not reduce the output image size, RLE compression is omitted. This decision is made automatically by Image Convert during the conversion process. Therefore, the only reason to disable RLE compression is if the PegScreen derived screen driver does not support RLE encoded PegBitmap formats. The PegScreen drivers provided with On Time RTOS-32 do support RLE encoding.

RLE compression is almost always beneficial if you are using 8-bpp bitmap encoding, especially for very large images. Compression ratios typically vary from 10:1 to 3:2, depending again on the source of the image being processed.

The use of dithering on the output bitmap has a negative impact on RLE compression effectiveness. Therefore, you should disable the Dither option if data size is the most important consideration in your application. This forces Image Convert to do a best match color mapping of input to output colors.

Palette Options

Image Convert can apply various color optimization and dithering methods when converting the input images to PegBitmap encoded data structures. The input images can be any combination of 2, 4, 16, 256, or true-color (i.e. 24-bpp) images. Image Convert will convert the images to the best possible representations on the target system.

If the images contain a higher number of colors than are available on the target display, Image Convert will reduce the number of colors in the source image. This reduction will either perform a best-match remapping or a dithering algorithm, depending on whether or not the dithering option is selected.

Fixed Orthogonal

The Fixed Orthogonal palette option instructs Image Convert to use a pre-defined palette covering the rainbow of colors available for 16 or 256 color targets. This is the only palette option when running with fewer than 256 colors. When targeting 256-color operation, you must choose between the fixed pre-defined system palette (the Fixed Orthogonal palette) or an optimal system palette created for the images.

Generate Optimal

The Generate Optimal palette option is only available if the target supports 256 color (i.e. 8-bpp) output. This is the opposite of using a Fixed Orthogonal palette. When this option is selected, Image Convert will create a custom palette for use with the input images. The custom palette will be saved at the top of the output file, and will be named PegCustomPalette. The custom palette is simply an array of 256*3 unsigned characters, which is passed to the PegScreen::SetupPalette function when you want to use the custom palette.

Using a custom palette when running in 256-color or higher modes provides the best possible image display. Since the resulting palette is generally modified extensively as compared to the palette that was included in the input images, the input images are automatically re-encoded by Image Convert to use the newly created palette. This all happens transparently when the *Generate Optimal* option is selected in the Image Convert dialog.

The custom palette created by Image Convert is always named `PegCustomPalette`, and is found at the top of the C++ output file generated by Image Convert. This palette always starts with the 16 standard colors, and is followed by up to 240 colors selected to produce the best possible image display for your input images. The custom palette is simply an array of unsigned characters containing the Red, Green, and Blue components of each color. This array of RGB values should be programmed into the video controller palette registers prior to displaying the associated bitmap(s). This palette can be directly passed to the `PegScreen::SetupPalette()` function, as shown below:

```
PegScreen()->SetupPalette(PegCustomPalette, 256);
```

It is also possible to use multiple custom palettes. When multiple custom palettes are used, it is the responsibility of the application level software to install the correct custom palette before the corresponding images are displayed. For systems that display 'one window at a time', it is a simple matter to install the correct palette when each window is displayed. For other systems, it can be complex to use multiple palettes, and one optimal palette is generally preferred.

Floyd-Steinburg Dither

The Floyd-Steinburg Dither option instructs Image Convert to dither the images when re-encoding them to the target palette. Dithering can be used in any of the output color depths, with or without a custom palette. When an optimal palette is created for multiple images, the actual colors contained in the final palette may not exactly match the original image colors. Likewise, when Image Convert is outputting bitmaps for 16-color targets using input images that contain 256 or more colors, Image Convert must translate those original colors into the best possible representation using only the 16-color palette.

The dithering option tells Image Convert how to convert the original image colors to the new system palette colors. If dithering is selected, Image Convert will pick colors such that the average value in each multi-pixel area is equal to the average value of the original input colors for the same multi-pixel area. If dithering is disabled, Image Convert will simply translate each pixel into its nearest color in the target palette.

Save As

The Output Format option specifies whether Image Convert should generate C++ source code or binary data. If the target system has means for file I/O, you can greatly reduce the RAM or ROM storage requirements for your bitmaps by saving them as binary files, and retrieving them from files or Win32 resources at run-time.

When C++ source data structures are generated, Image Convert writes a normal ASCII file that can be opened and modified with any editor. This ASCII file contains the bitmap data array, along with the corresponding `PegBitmap` structure definition. If an optimal palette is generated, the C++ file will also contain the custom palette.

When binary format is selected, Image Convert generates a binary data file containing one or more `PegBitmap` data structures and bitmap data definitions. The binary file starts with an 8 byte header as shown below:

```
// Header, one occurrence per binary image file:
char  cVersion[4]           // four byte version string, "1.00"
UCHAR uReservedA            // 1 byte reserved
UCHAR uHavePalette          // 1 byte palette flag
UCHAR uReservedB[2]         // 2 unused bytes
```

The `uHavePalette` byte of the header signals the presence or absence of a color palette in the binary file. If the binary file contains a custom palette, `uHavePalette` will be non-zero and the custom palette immediately follows the file header, and can be declared as shown below:

```
// Palette, max one occurrence per binary image file
UCHAR Palette[256*3]          // only present when custom palette is generated.
```

Following the short header and optional palette data are the bitmap header and bitmap data fields. The bitmap header and data fields are repeated for each image contained in the file. Each bitmap is contained in the binary file as shown below:

```
// repeated for each bitmap in file:
Char  cName[28]                // bitmap name left justified
UCHAR uType                    // 1 byte, PegBitmap.uType
UCHAR uBitsPerPix              // 1 byte, PegBitmap.uBitsPerPix
WORD  wWidth                   // 2 bytes, PegBitmap.wWidth
WORD  wHeight                  // 2 bytes, PegBitmap.wHeight
WORD  wReserved                // 2 bytes, unused
LONG  lSize                    // 4 bytes, size of data array in bytes
UCHAR uData[lSize]             // bitmap data values
```

For each bitmap, lSize bytes of bitmap data immediately follow the bitmap header information. When multiple PegBitmaps are generated using batch conversion, each successive bitmap header immediately follows the previous bitmap data. There are no padding or alignment bytes inserted between bitmaps. For multi-byte fields such as wWidth and wHeight, byte swapping may be required when reading the bitmap header data depending on the endian type of the CPU and the method used to read the bitmap data value. Image Convert always writes multibyte values MSB (most significant byte) first. Byte swapping is not required for the actual bitmap data, as this section always contains only single-byte values.

Reading a bitmap or series of bitmaps from a binary file can be accomplished with the pseudo-code shown below. Several variations of this example could also be used:

```
UCHAR * ReadBitmap(FILE * pSrc, PegBitmap &Bitmap)
{
    UCHAR uTemp[30];
    UCHAR *pPalette;
    LONG  lDataSize;

    fread(uTemp, 1, 8, pSrc);    // read in the header
    // ** check here for correct version string **
    if (uTemp[5])                // does file contain a palette?
    {
        pPalette = new UCHAR[256 * 3];
        fread(pPalette, 1, 256*3, pSrc);    // read the palette
    }
    else
        pPalette = NULL;

    fread(uTemp, 1, 28, pSrc);    // read image name
    // ** verify image name **

    fread(&Bitmap.uType, 1, 1, pSrc);
    fread(&Bitmap.uBitsPerPix, 1, 1, pSrc);
    fread(&Bitmap.wWidth, 1, 2, pSrc);
    fread(&Bitmap.wHeight, 1, 2, pSrc);
    fread(uTemp, 1, 2, pSrc);    // skip unused bytes
    fread(&lDataSize, 1, 4, pSrc);    // get data size
    Bitmap.pStart = new UCHAR[lDataSize];    // get RAM for bitmap data
    fread(Bitmap.pStart, 1, lDataSize, pSrc);
    return pPalette;
}
```

Note that if the binary file contains multiple bitmaps, the application software could also pass to ReadBitmap() the name of the image file to load. In that case, ReadBitmap would loop through the binary images until the correct bitmap name is found.

Transparency

The Transparency field can be used to specify a transparent color in the input image. This field only applies to .BMP files, as GIF and JPEG files encode transparency information internal to the input file. In all cases, the transparent color will be saved as index 255 in the output PegBitmap, since index 255 is always interpreted as transparent by the PegScreen bitmap functions. Transparency forces the output PegBitmap structure to use 8-bpp encoding, since the default transparent color value is 255. This is true even if the source image contains only 2, 4, or 16 colors. If the source image is encoded using less than 8 bits-per-pixel, Image Convert will expand the image to 8 bits-per-pixel format when transparency is enabled.

.BMP files do not inherently support transparency. To use image transparency, the source image(s) must be created such that all areas that should be displayed transparently are painted with an otherwise unused color. You must then inform Image Convert which color should be interpreted as transparent. There are two methods of specifying the transparent color:

Specify RGB Value:

If the source images are 24-bpp (true-color) images, you must specify an actual RGB value to use as the transparent color. If you select the 'RGB' button, you should enter the Red, Green, and Blue values in the string fields which are displayed. You can determine the correct values through examination with a paint program.

Use Upper-Left Color:

When this method is selected, Image Convert will assume that the upper left corner pixel is in the transparent color. This method can only be used with 16 and 256 color input images.

Output Colors

The output colors field allows you to specify how the generated PegBitmap structures will be encoded, and tells Image Convert how many colors are available on the target system. PegBitmap structures can be encoded using 1, 2, 4, 8, 16, or 24 bits-per-pixel.

If transparency or RLE compression are enabled, the resulting PegBitmap structures are saved using 8-bpp encoding, regardless of the selection in the Output Colors field.

The PegScreen driver classes provided with On Time RTOS-32 all support 8-bpp encoding (enabling the use of transparency), RLE encoding, and the native encoding format corresponding to the output color depth.

Batch Conversion

Image Convert can be used to perform list or batch conversion of multiple images. This is helpful in many cases and absolutely essential when the goal is to create an optimal palette for multiple images. Batch conversion is selected by specifying an input file with a filename extension of .CMD, which indicates that the source file is actually a command file, rather than an individual image file.

Command files are simply lists of input images, along with optional comments. They do not instruct Image Convert in terms of the type of conversion to perform. This is still done by selecting the appropriate options in the Image Convert dialog.

The command file should be an ASCII command file, with one command per line. Each command line should contain a single input image path and filename, and the output image name. The source file and output image names can be separated by any combination of space, comma, and tab characters. The output image name is the name to be used by the application when passing the image address to one of the PegScreen bitmap display functions.

When performing batch conversion, all of the resulting output images are saved in one output file, along with the custom palette if a custom palette has been generated.

Very large command files are often easier to maintain by entering comments within the file to indicate where groups of bitmap files are used. Comment lines are indicated by a single '#' character in the first column of a line.

The following is an example of a typical command file:

```
#
# This is a comment line
# Each command line specifies one input file,
# and the name of the resulting PegBitmap structure.
#
\graphics\bitmaps\stop.bmp StopSign
\graphics\bitmaps\go.bmp GoSign
#
# note that .bmp and .gif files can be processed
# within the same .cmd file
#
\graphics\targa\yield.gif YieldSign
```

In the above example, the goal is to produce three PegBitmaps and a single optimal palette for use with these three images. The source images are stop.bmp, go.bmp, and yield.gif.

The resulting PegBitmaps will be named gbStopSignBitmap, gbGoSignBitmap, and gbYieldSignBitmap, respectively.

Image Convert will process each of the input files using the options selected in the Image Convert dialog, and will save all output to the single file specified in the 'Output File' field of the conversion dialog.

Font Capture

Font Capture (program Pfontcap.exe) is a Windows utility program to generate additional fonts for use by an application. Font Capture generates C++ files containing font information, character widths, and character data. This information is stored in a PegFont data structure, allowing easy use of the new fonts in applications.

After starting Pfontcap, the *Source Font* group allows selecting the source font type. Font Capture supports the conversion of MS Windows TrueType fonts or Adobe Postscript .BDF fonts. With TrueType, the *Font...* button can be used to invoke a Windows dialog to select from the installed Windows fonts. If BDF is selected, the *Font...* button changes to *File...*, allowing to select the .BDF file. Please note that some fonts installed on your computer may carry licensing terms limiting their use outside the Windows operating system.

The *Range* group allows you to specify the range of character glyphs that will be encoded in the output font. When the ASCII option is selected, the range of characters is fixed to ASCII-0 through ASCII-127, which is the normal range for single language applications.

Font Capture also allows you to specify a custom range of characters to be encoded. When you select the Custom option, the Configure option becomes active, allowing you to fully define the range of glyphs that will be recorded in the output file.

In the simplest case, a particular font is only used to display a certain range of characters. For example, you may define one font that will be used only for displaying numbers. In this case, you do not need or want to encode the entire ASCII character range in the output file. Instead, you can enter a limited character range by selecting the *Custom* button, and entering the range of characters in the Range Configuration dialog in hexadecimal. The First Char and Last Char fields allow you to define the start and ending characters to be encoded. For non-Unicode systems, range 0..FFh (256) is recommended.

The *Output Format* group allows selecting between normal bitmapped font output and outlined font format.

The *Outline* checkbox can be used to generate a font with an added single pixel wide outline of each glyph. Font Capture encodes *Outline* fonts in a 2-bpp format, where bitmap value 0 indicates the pixel should be the foreground color, bitmap value 1 indicates that the pixel should be in the outline color, and bitmap value 2 indicates that the pixel should be either the background color or transparent, depending on the PegColor.uFill value passed to the text drawing function.

The *Solid* and *Add Space* checkboxes are modifiers for the outline font generation mode. The *Solid* checkbox causes the font outline to appear somewhat heavier than the default outline. The *Solid* choice is beneficial when working with large fonts. The *Add Space* option adds a single pixel of spacing between each generated character when generating an outline font. This is beneficial when working with very small outlined fonts.

The field *Current Font* displays the currently selected font name and size. Note that the height information will normally be larger than the actual point size selected, since this field includes both the ascent and descent values of the selected font, which is a more accurate indication of true font height.

Output File is where you tell Font Capture the name and location of the file created during the capture process. You would usually set this to the same location as your other application source files, and use a name that will help you remember which font is stored in the file.

Font Name is the name of the newly created PegFont. This field is case sensitive, so the font names *MyFont* and *myfont* are not identical, exactly like any other variable names in a C++ program.

The *Sample Text* field is displayed in the currently selected font, allowing you to review your selection before capturing the font to a file.

The *Capture!* button causes Font Capture to display each character of the selected font, and scan that character to generate the .cpp output file.

Using Custom Fonts

Using fonts generated with FontCapture is very simple. Every object that supports text output has a member function called `SetFont(PegFont *)`. Therefore, after constructing an object that should use the new font, you simply call that object's `SetFont` function, passing a pointer to your new font. For example, assume that you told Font Capture to generate a new font called *MyNewFont*, by typing this in the font name field of Font Capture. In this case, the following code fragment illustrates the process of altering the font used:

```
extern PegFont MyNewFont;

PegTextButton * pButton = new PegTextButton(10, 10, 100, MESG1, "NewFont");
PButton->SetFont(&MyNewFont);
```

Likewise, if you have overloaded a `Draw()` function for a window or other object, and you are drawing text on the screen, you can simply pass the pointer to your new font to any of the `PegScreen` text information or output functions.

Index

- #define, 23
- #defineN, 23
- #else, 23
- #endif, 23
- #error, 23
- #if, 23
- #ifsection, 23
- #include, 23

- \$, 42

- .DEF files, 124

- _beginthread, 198, 229
- _endthread, 199, 229
- _thread variables
 - see *TLS data*

- 16550, 209

- 386, 9, 14
 - protected mode, 14
 - real mode, 14
 - virtual 8086 mode, 14
- 387, 37, 112, 131, 168, 221

- 64-bit integers, 202

- 80387
 - see 387

- A20, 37, 48
- abort, 167
- access
 - to pages, 17
 - values, 24
- active protocol, 207
- addresses, 17
- addressing
 - sectors, 251
- Align command, 26
- aligned access, 250
- allocation unit
 - see *cluster*
- AMD Élan SC400, 110
 - demo program, 114
- AMD Élan SC520, 110
 - demo program, 114
- AMD SC520
 - configure for, 227
- API
 - C, 282
 - C++, 282
 - mixing, 282
 - RTFiles-32, 254
 - RTTarget-32 native, 67
 - Win32, 281
- APIs
 - alternate, 196
 - RTKernel-C, 196
 - Win32, 197
- application
 - header, 28
- Application Image Report, 43
- argc, 34
- argv, 34
- Assign, 24
- assignment
 - of drive letters, 254
- attribute, 263
- auto init, 163
- automatic library protection
 - see *library protection*

- baud rate
 - debugger, 49
 - Monitor, 39
- bin files, 20
- BinFile
 - command, 40
- BIOS
 - booting, 46
 - CMOS RAM, 287, 289, 290
 - data area, 118
 - extension, 46
 - graphics mode, 38
 - parameter block, 252, 289
 - PCI, 88
 - PnP, 90
- BIOSBOOT.EXE, 29
- BIOSDemo, 113
- BIOSVector
 - Locate command, 30
- bitmaps, 353
- Blocked
 - task state, 160
- BlockedGet, 169
- BlockedPut, 169
- BlockedReceive, 169
- BlockedSend, 169
- BlockedWait, 169
- BOOT.EXE, 29
- boot code, 45
 - options, 37
 - Report in LOC file, 43
- boot record, 252, 275, 276
- BootCode
 - Locate command, 29
- BootData
 - Locate command, 29
- bootdisk
 - creating, 83
- BootDisk
 - program, 45
- BOOTFLAGS
 - command, 37
- booting, 45
 - BIOS, 46
 - from MS-DOS, 47
 - reboot, 71
 - reset, 46
- Bootprog, 113

- BootVector
 - Locate command, 29
- Borland C/C++
 - compiling, 136
 - debug symbols, 137
 - IDE, 137
- Borland Delphi
 - compiling, 141
- breakpoint
 - hardware, 55
- buffer
 - cache, 278
 - configuration, 283
- buffers, 255
 - for serial ports, 210

- C++, 237
- C++ streams, 282
- C/C++ API, 282
- C/C++ run-time system, 119
- CORTT.OBJ, 136, 138, 139
- cache, 255, 278
 - read-ahead, 256
- calibrate
 - high-res timer, 202
- calling
 - ring 0, 76
- Centronics port, 84
- chaining programs, 74
- character encoding, 334
- character I/O, 67
- CharNextA, 102
- CharToOemA, 102
- CharUpperA, 102
- CharUpperBuffA, 102
- child, 311
- CHS, 250, 251
- ClassDemo, 115
- clear screen, 215
- CLI, 187
- CLKHRTPC, 219, 220
- CLKMicroSecsToTicks, 204
- CLKMilliSecsToTicks, 204
- CLKPC, 219
- CLKSecondsToTicks, 204
- CLKSetResolution, 204
- CLKSetTimerIntVal, 204
- CLKTicksToMicroSecs, 204
- CLKTicksToMilliSecs, 204
- CLKTicksToSeconds, 204
- clock, 203
 - overflow, 172
 - reading the, 173
 - resolution, 172
 - setting the, 172
- CloseHandle, 99, 102, 197, 25
- closing files, 272
- cluster, 250 - 252
- CMOS RAM
 - access, 79
 - of BIOS, 287, 289, 290

- code page, 81, 334
- code position, 171, 172
- code sharing, 160
- codes
 - returned by RTFiles, 307
- color, 312
- color display, 38
- COM port
 - configuration on host, 49
 - I/O functions, 85
 - settings on target, 39
- COM ports, 205
- COMAllocateBuffers, 210
- COMDemo, 226
- COMDisableInterrupt, 210
- COMEnableFIFO, 209
- COMEnableInterrupt, 210
- COMError, 212
- COMHasFIFO, 209
- COMLineStatus, 211
- Commandline command, 34
- commit, 259, 261, 272
- COMModemControl, 212
- COMModemStatus, 212
- communication
 - between tasks, 160
- compact
 - flash disk, 280
- CompareStringA, 102
- CompareStringW, 102
- compiling
 - Borland C/C++, 136
 - Delphi, 141
 - Microsoft C/C++, 137
 - Watcom C/C++, 139
- COMPort
 - command, 39
- COMPortInit, 209
- compression, 19, 32
- Compression Report, 43
- COMReceivePolled, 211
- COMSendBlock, 210
- COMSendBlockTimed, 210
- COMSendChar, 210
- COMSendCharPolled, 211
- COMSendCharTimed, 210
- COMSetBoardType, 208
- COMSetIOBase, 208
- COMSetIRQ, 208
- COMSetModemStatusHook, 211
- COMSetProtocol, 209
- COMWaitSendBufferEmpty, 211
- configuration
 - of RTFiles-32, 283
 - of RTKernel-32, 162
 - of RTTarget-32, 66
 - RTTARGET.INI, 49
- configuration files, 21
- Configuration Report, 42
- configure
 - PCI cards, 88
 - PnP cards, 90
- Console
 - file system, 128
- console I/O, 100
- contiguous file, 261
- control
 - GUI, 311
- conversion
 - clock device, 204
 - finetime, 202, 203
 - ticks, 205
- cooperative
 - scheduling, 158, 159, 233
- coordinates
 - screen, 312
- Copy
 - Locate command, 32
- core file system, 254
- CPL, 17, 37, 117
- CPL 0
 - calling, 76
- CPU
 - see 386
- CPU load, 217
- CPU time, 163, 171, 172
- CPU386, 223
- CPU386F, 223
- CPUMoni, 217
- CPUMonitorStart, 217
- CPURelativeLoad, 217
- CreateDirectoryA, 102
- CreateEvent, 200
- CreateEventA, 102
- CreateFile, 100
- CreateFileA, 102
- CreateFileW, 102
- CreateMutex, 200
- CreateMutexA, 102
- CreatePegScreen, 338
- CreatePegScreen_VESA_16, 337
- CreatePegScreen_VESA_24, 337
- CreatePegScreen_VESA_32, 338
- CreatePegScreen_VESA_8, 337
- CreatePegScreen_VGA_4, 338
- CreateProcessA, 102
- CreateSemaphore, 200
- CreateThread, 102, 198
- critical error, 273
- Critical Section, 236
 - Win32, 164, 200
- cross debugging, 8, 49, 62
- cross development, 8
- Current, 169
 - task state, 160
- cursor, 100
- custom drivers
 - for RTFiles-32, 302, 304
- cyclic task, 236
- data files, 256
- data security, 299
- data tables
 - RTFiles-32, 254, 283
- date, 101
- DBGShell, 62
- deadlock, 169, 239
- Debug Monitor, 51
 - see Monitor
- debug symbols, 19
 - Borland C/C++, 137
 - Microsoft Visual C++, 139
 - Watcom C/C++, 140
- Debug Version, 191, 228
 - example using, 226
- DebugBreak, 102
- debugger
 - configuration, 49
 - running the, 51
- debugging, 49, 62, 278
- DecompCode
 - Locate command, 32
- DecompData
 - Locate command, 33
- DEF files, 124
- Delay, 41
- Delaying, 169
 - task state, 160
- DeleteCriticalSection, 102, 200
- DeleteFileA, 102
- deleting files, 266
- Delphi
 - compiling, 141
- demo programs, 109, 226, 297, 83
- descriptors, 14, 15
- DestroyWindow, 102
- device, 250, 251
 - disk drivers, 287
 - files, 256
 - list, 250, 254, 284, 30
- Device
 - memory type, 24
- device driver
 - raw I/O, 257
- DeviceIOControl, 102
- Dialog, 340
- DigiBoard
 - serial ports, 207
- directory, 253, 264
 - files, 256
 - root, 253
- DisableThreadLibraryCalls, 102
- discardable entities, 9, 19, 33
- DiscardSectors, 304
- disk
 - booting, 45
 - drivers, 287
 - formatting, 270
 - info, 268
 - label, 270
 - mounting, 254
 - PCMCIA, 96
- DiskBuffer
 - Locate command, 30
- diskette
 - see floppy disk
- DiskOnChip
 - driver, 285, 291
- display, 38, 68
- distributables, 142
- DLL, 101, 122, 125

- and threads, 102
- DEF files, 124
- resolving references, 35
- with threads, 75
- DLL command, 26
- DLLDemo, 112
- DLLDemo2, 112
- DLLDemo3, 112
- DLM, 125
 - demo program, 112
- DMA
 - for floppy driver, 287
- DOS
 - booting from, 47
 - emulation, 98
- DOSDateTimeToFileTime, 102
- downloading, 46
 - avoiding repeated, 119
 - RTRun, 46
- DPMI
 - emulation, 98
- Draw
 - method, 328, 329
- drive, 250
 - mounting, 254
- drive letter
 - assignment, 254
 - skipping, 286
- driver
 - example, 304, 306
 - flags, 163
 - flash, 294, 305
 - floppy disk, 287
 - IDE, 290
 - keyboard, 80
 - M-Systems DOC, 291
 - NULL device, 296
 - RAM disk, 293
 - RTFiles-32 custom, 302, 304
 - RTFiles-32 demo, 297
 - screen, 312, 337
 - SRAM, 292
 - system, 254, 283
- drivers, 243
 - disk, 287
 - of RTKernel-32, 218
- drives, 252
- DrvDemo, 297
- DRVRT32, 225
- DTR/DSR, 206
- DuplicateHandle, 99, 102, 197, 25
- Duration, 172
- Dynamic Link Report, 43
- email, 3
- embedded system, 8
- EmuDemo, 112
- emulation
 - DOS, 98
 - DPMI, 98
 - floating point, 131
 - Win32, 98
- EndM, 23
- endthread, 199
- EnterCriticalSection, 103, 200
- entity, 9
- EnumCalendarInfoA, 103
- EnumSystemLocales, 103
- EnumThreadWindows, 103
- environment, 34
- EPROM
 - programming, 40
- error codes, 258, 272, 307
- error handler, 273
- error handling
 - Win32, 197
- error messages, 43, 143, 245
- errors
 - of serial ports, 212
- Event
 - Win32, 164, 200
- events, 158
- examples, 4, 12
 - demo programs
 - MBDemo, 178
 - MSGDemo, 182
 - SemaDemo, 175
- exception handling, 102
- exceptions, 18, 78
 - 13, 15
 - 14, 17
- EXE File Report, 43
- executable file, 8
- exit, 167
 - function, 165
- ExitProcess, 103
- ExitThread, 103
- ExitThread (Win32), 199
- EXLED, 114
- extended memory, 73
- extended partition, 252
- extending files, 261
- FAPIDemo, 297
- FAT, 250, 251
 - 12, 252
 - 16, 252
 - 32, 252
 - copies, 253
- FatalAppExit, 103
- file
 - attribute, 263
 - closing, 272
 - committing, 272
 - contiguous, 261
 - deleting, 266
 - extending, 261
 - finding, 265, 281
 - flushing, 272
 - handles, 258
 - info, 262
 - instance, 250
 - name, 267
 - opening, 273
 - renaming, 266
 - table, 279
 - temporary, 266
 - time, 263
 - unique, 266
- File
 - Locate command, 33
- file allocation table
 - see FAT
- file I/O, 33
 - Win32, 100
- file system, 127
 - console, 128
 - parallel port, 128
 - RAM files, 128
- file system core, 254
- file table, 283
- filebuf, 282
- files
 - data, 256
 - directory, 256
 - logical drive, 256
 - physical device, 256
 - special, 256
- FileTimeToDosDateTime, 103
- FileTimeToLocalFileTime, 103
- FileTimeToSystemTime, 103
- FillConsoleOutputAttribute, 103
- FillConsoleOutputCharacterA, 103
- FillRAM command, 25, 116
- FindClose, 103
- FindFirstFileA, 100, 103
- finding files, 265, 281
- FindNextFileA, 100, 103
- FindResourceA, 103
- FindResourceExA, 103
- FineTime, 202
 - arithmetic, 202
- fixed memory manager, 106
- fixup, 8
- Fixup Table Report, 43
- flags
 - driver, 163
 - kernel, 163
 - task, 166
- flash disk, 279, 280
 - driver, 285, 291, 294, 295, 305
- FlashDemo, 297
- flat memory model, 15
- floating point, 163, 166, 221
 - demo program, 112
 - emulation, 131
- floppy disk, 45
 - DMA buffer, 287
 - driver, 285, 287
 - motor timeout, 288
 - read-ahead buffer, 288
- FLT387, 221
- FLTEMUMT, 221
- FLTNUL, 221
- FLTPII, 221
- FlushConsoleInputBuffer, 103
- FlushFileBuffers, 103
- flushing files, 272
- focus, 317
- Font Capture, 358
- fonts, 334, 358
- format, 270, 275, 276

- FormatMessageA, 103
- FPU, 37
- FreeEnvironmentStringsA, 103
- FreeEnvironmentStringsW, 103
- FreeLibrary, 101, 103
- FreeResource, 103
- FTAdd, 202
- FTCalibrate, 202
- FTDivide, 202
- FTElapsedMicroSecs, 203
- FTElapsedMilliSecs, 203
- FTElapsedSeconds, 203
- FTIntMultDiv, 202
- FTMicroSecsToTime, 203
- FTMilliSecsToTime, 203
- FTMultiply, 202
- FTReadTime, 203
- FTSecondsToTime, 203
- FTSetResolution, 202
- FTSubtract, 202
- FTTimeToMicroSecs, 203
- FTTimeToMilliSecs, 203
- FTTimeToSeconds, 203
- function keys, 215
- garbage collection
 - flash disk, 280
- Gauge, 340
- GDT, 14, 15
- general protection fault, 15
- GetACP, 103
- GetActiveWindow, 103
- getch, 273
- GetCommandLineA, 103
- GetCommandLineW, 103
- GetConsoleCursorInfo, 103
- GetConsoleMode, 103
- GetConsoleScreenBufferInfo, 103
- GetCPInfo, 103
- GetCurrentDirectoryA, 103
- GetCurrentProcess, 103
- GetCurrentProcessId, 103
- GetCurrentThread, 103
- GetCurrentThread (Win32), 199
- GetCurrentThreadId, 103, 198
- GetDateFormatA, 103
- GetDiskFreeSpaceA, 103
- GetDiskGeometry, 304
- GetDriveTypeA, 103
- GetEnvironmentStrings, 103
- GetEnvironmentStringsW, 103
- GetEnvironmentVariableA, 103
- GetExitCodeProcess, 103
- GetExitCodeThread (Win32), 199
- GetFileAttributesA, 103
- GetFileSize, 103
- GetFileTime, 103
- GetFileNameA, 103
- GetFileType, 103
- GetFullPathNameA, 103
- GetKeyboardType, 103
- GetLargestConsoleWindowSize, 103
- GetLastError, 103
- GetLocaleInfoA, 103
- GetLocaleInfoW, 103
- GetLocalTime, 101, 103
- GetLogicalDrives, 103
- GetModuleFileNameA, 103
- GetModuleFileNameW, 103
- GetModuleHandleA, 101, 103
- GetNumberOfConsoleInputEvents, 103
- GetNumberOfConsoleMouseButtons, 103
- GetOEMCP, 103
- GetProcAddress, 101, 103
- GetProcessHeap, 103
- GetShortPathNameA, 103
- GetStartupInfoA, 103
- GetStdHandle, 103
- GetStringTypeA, 103
- GetStringTypeExA, 103
- GetStringTypeW, 103
- GetSystemDefaultLangID, 103
- GetSystemDefaultLCID, 103
- GetSystemInfo, 103
- GetSystemMetrics, 103
- GetSystemTime, 101, 103
- GetTempFileNameA, 103
- GetThreadContext, 103
- GetThreadLocale, 103
- GetThreadPriority, 200
- GetTickCount, 103, 118
- GetTickCount (Win32), 199
- GetTimeFormatA, 103
- GetTimeZoneInformation, 103
- GetUserDefaultLCID, 103
- GetVersion, 103
- GetVersionExA, 103
- GetVolumeInformationA, 103
- global data, 160
- global descriptor table, 14
- GlobalAlloc, 103
- GlobalFree, 103
- GlobalHandle, 103
- GlobalLock, 103
- GlobalMemoryStatus, 103
- GlobalReAlloc, 103
- GlobalUnlock, 103
- glossary, 8, 250
- GMode command, 38
- GPF, 15
- graphics, 38, 73, 129, 54
- GUI, 310
 - run under Windows, 320
- handle
 - see task
- handles, 159, 258
 - Win32, 99, 197, 281
- hard disk
 - booting, 45
- hardware
 - breakpoint, 55
- hardware configuration
 - of serial ports, 207
- hardware interrupt
 - see interrupt
- headings
 - task list, 171
- heap
 - for RTKernel-32, 221
 - real-time, 188
- heap manager
 - RTTHeap, 107
- hex file, 20
- hierarchy
 - RTPEG-32, 319
- high resolution time, 202
- host, 8
 - COM port configuration, 49
- hotline, 2
- http, 3
- Halt
 - instruction, 37, 71
- Header
 - Locate command, 28
- Heap
 - Locate command, 31
- HeapAlloc, 103
- HeapCompact, 103
- HeapCreate, 103
- HeapDestroy, 103
- HeapFree, 103
- HeapReAlloc, 103
- HeapSize, 103
- HeapValidate, 103
- Hello, 111
- Hello2, 111
- HelloFiles, 297
- HelloGUI, 115
- HelloSc400, 114
- HelloSc520, 114
- HexFile
 - command, 40
- High Resolution Timer, 220
- Hostess card
 - serial ports, 207
- HRTNULL, 220
- HRTPC, 220
- HRTPEnt, 220
- HRTSC520, 220
- I/O address
 - of serial ports, 208
- I/O sensitive, 17
- i386
 - see 386
- IDE
 - Borland C/C++, 137
 - driver, 285, 290
- IDE disk
 - PCMCIA, 96
- Idle Task, 160
- ids
 - of RTPEG-32 objects, 326
- IDT, 15, 18
- IgnoreMsg command, 36
- ILINK32, 136
- Illegal, 170
- image, 9
- image conversion, 353
- Image Convert, 353

- Image Report, 43
- information
 - about disks, 268
- information messages, 143
 - ignoring, 36
- INI file, 49
- Init command, 34
- InitCode, 41
- initialize
 - target hardware, 41
- InitializeCriticalSection, 103, 200
- inport, 70
- input focus, 317
- installation, 3
- instance
 - file, 250
- Intel
 - hex files, 40
- inter-task communication, 160
- InterlockedDecrement, 103
- InterlockedExchange, 103
- InterlockedIncrement, 103
- Internet, 3
- interrupt, 18
 - 21h, 98
 - 31h, 98
 - CPU time, 163
 - demo program, 111
 - disabling of, 187
 - driver, 219
 - enabling of, 187, 188
 - end of, 187
 - handler, 231
 - handling, 183, 219
 - high-level handler, 184
 - hooked, 164
 - IRQ, 184
 - latency, 157, 241
 - list, 186
 - low-level handler, 184
 - priorities, 186
 - reset target, 117
 - RTDisableInterrupts, 70
 - RTDisableIRQ, 70
 - RTEnableInterrupts, 70
 - RTEnableIRQ, 69
 - RTInstallISR, 69
 - RTIRQEnd, 70
 - RTRestoreInterrupts, 70
 - RTRestoreVector, 69
 - RTSaveAndDisableInterrupts, 70
 - RTSaveVector, 68
 - RTSetIntVector, 69
 - RTSetTrapVector, 69
 - stack, 164, 184, 185
 - timer, 118
 - vectors, 184
 - wait for, 71
- interrupt controller, 37
- interrupt handling
 - RTKernel-32 example, 226
 - RTTarget-32 example, 111
- interrupts, 117
- introduction
 - mailboxes, 160
 - message passing, 160
 - semaphores, 160
- Invalidate, 329
- IOPL, 17
- iostream, 282
- IRQ
 - definition, 184
 - of serial ports, 208
 - of target COM port, 39
- IRQRT32, 219
- IsBadCodePtr, 104
- IsBadReadPtr, 104
- IsBadWritePtr, 104
- IsValidCodePage, 104
- IsValidLocale, 104
- KBGetCh, 213
- kbhit, 213
- KBInit, 213
- KBKeyAvailable, 213
- KBKeyPressed, 213
- KBPutCh, 213
- kernel drivers, 218, 243
- kernel tracer, 163, 188
 - buffer size, 189
- KERNEL32.DLL, 98, 101
- keyboard, 37, 66, 100, 101, 212
 - change driver, 80
 - code page, 81
 - reset target, 117
 - RTPEG-32, 318
 - RTSetKeyboard, 80
- label
 - set disk, 270
- language
 - of keyboard, 80
- LBA, 250, 251
- LCMapStringA, 104
- LCMapStringW, 104
- LDT, 14, 15
- LeaveCriticalSection, 104, 200
- LIBPROT, 230
- libraries
 - order of, 134
 - reentrance, 228
- library protection, 175, 229
- licensing terms, 5
- line status register, 88, 211
- linear addresses, 17
- linear flash disk
 - driver, 294
- LINK
 - Microsoft's, 137
- Link command, 35, 43
- Link Report, 43
- linker
 - DEF files, 124
- linking, 134
 - with LINK, 138
 - with TLINK32, 136
 - with wlink, 139
- list flags, 170
- list of files, 279
- listing file, 42
- Loader, 113
- loading
 - program, 74
- LoadLibrary (Win32), 126
- LoadLibraryA, 101, 104
- LoadLibraryExA, 104
- LoadResource, 104
- LoadStringA, 104
- LOC file, 42
- local descriptor table, 14
- LocalAlloc, 104
- LocalFileTimeToFileTime, 104
- LocalFree, 104
- LocalReAlloc, 104
- locate, 8, 43
 - RTLoc, 19
- Locate command, 27
 - BIOSVector, 30
 - BootCode, 29
 - BootData, 29
 - BootVector, 29
 - Copy, 32
 - DecompCode, 32
 - DecompData, 33
 - DiskBuffer, 30
 - File, 33
 - Header, 28
 - Heap, 31
 - Nothing, 33
 - NTSection, 28, 116
 - PageTable, 31
 - Section, 27, 116
 - Stack, 30
- LockResource, 104
- logical drive, 250, 252
 - files, 256
- low-level handler, 184
- LowLevelFormat, 304
- IstrcmpA, 104
- IstrcmpiA, 104
- IstrcpyA, 104
- IstrcpynA, 104
- IstrlenA, 104
- M-Systems
 - DOC driver, 285
- Macro, 23
- mailbox
 - clearing, 179
 - deleting, 179
 - example, 178, 226
 - introduction, 178
 - large data types, 235
 - name, 179
- main
 - arguments, 34
- Main Task, 160
 - priority, 165
- MakeDef, 124
 - demo program, 112
- MakeDLM, 125
- MAP file, 42

- MAPDemo, 112
- mass storage device, 251
- master boot record, 252, 275
- MediaChanged, 304
- MEMCHEAP, 222
- memory, 17
 - access, 24
 - defining, 24, 25
 - device, 24
 - installed, 73
 - management, 67, 75, 106, 107
 - mapping, 76
 - RAM, 24
 - real-time allocation, 188
 - remapping, 25, 116
 - ROM, 24
 - types, 24
 - uncommitted, 9, 106
 - Win32, 99
- memory driver
 - for RTKernel-32, 221
- memory management, 17
- memory mapping
 - demo program, 112
- memory model
 - flat, 15
- memory pool, 188
- memory requirements
 - of tasks, 166
- memory technology driver
 - see MTD
- MEMSTCH, 222
- MEMSTH, 222
- MEMW32, 222
- Message
 - RTPEG-32 Handler, 328
- message passing
 - example, 182, 226
 - function, 181
 - large data types, 235
- Message Reports, 43
- MessageBoxA, 104
- messages, 143, 245, 247, 272, 327
 - ignoring, 36
 - informational, 247
 - RTPEG-32, 313
- MetaWINDOW, 129
 - demo program, 115
- methods
 - as tasks, 237
- MetWorld, 115
- MFC, 115
- Microsoft C/C++
 - compiling with, 137
- Microsoft Visual C++
 - debug symbols, 139
- Microsoft Visual Studio, 62
- milliseconds, 101
- MMU, 17
- MMX, 221
- modal, 312
- modem
 - PCMCIA, 96
- modem control register, 88
- modem status register, 88, 211, 212
- Module, 9
- Monitor, 51
 - COM port settings, 39
 - reserving memory for, 26
 - resetting target, 117
- monochrome display, 38
- MountDevice, 303
- mounting
 - disks, 254, 267
- mouse, 100, 101
- MoveFileA, 104
- MS-DOS
 - booting from, 47
- MTD, 294, 305
 - example, 306
- MTD driver, 295
- MultiByteToWideChar, 104
- multitasking, 102, 128
 - cooperative, 158, 233
 - introduction, 157
 - non-preemptive, 158, 233
 - preemptive, 158, 233
 - real-time, 157
 - time sharing, 157
- Mutex
 - Win32, 164, 200
- mutual exclusion, 236
- name
 - mailbox, 179
 - semaphore, 175
 - task, 166
- NoAccess, 24
- non-preemptive
 - see cooperative
- Notebook, 340
- Nothing
 - Locate command, 33
- NS16550, 209
- NS486SXF, 111
 - demo program, 114
- NSHello, 114
- NTSection
 - Locate command, 28, 116
- NULL device
 - driver, 296
- NULL pointer, 118
- numeric syntax, 21
- object-oriented, 237
- occupied resources, 177
- OemToCharA, 104
- off-screen drawing, 331
- online, 3
- OpenEvent, 201
- opening files, 273
- OpenMutex, 201
- OpenSemaphore, 201
- optimization
 - file I/O, 298
- options
 - boot code, 37
 - RTLLoc, 19, 21
- OUTB, 41
- OUTD, 41
- output, 70
- Output
 - command, 40
- OutputDebugStringA, 104
- OUTW, 41
- overflow
 - RTKernel-32 clock, 172
 - stack, 118
- page fault, 17
- Page Table Detailed Report, 43
- Page Table Summary Report, 43
- PageTable
 - compression, 33
 - Locate command, 31, 116
- paging, 17, 116
 - page table report, 43
 - page table size, 31
- palette, 312
- panic stack, 184, 186
- parallel port, 84
 - file system, 128
- parameter block, 252
 - of BIOS, 289
- parent, 311
- partition, 250
 - extended, 252
 - info, 269
 - table, 250, 252, 275
- partition table, 275
- passive protocol, 207
- PC Card, 91
- PCCard, 113
- PCCardMT, 114
- PCI BIOS, 88
 - example, 113
- PCMCIA, 91
 - disk, 96
 - example, 113, 114
 - SRAM card driver, 292
 - UART, 96
- PE file, 8
- PeekConsoleInputA, 104
- PEG, 310
- PegDemo, 340
- PegExecute, 322
- PegFont, 334
- PegInitialize, 321
- PegMessage, 313
- PegMessageQueue, 313
- PegPresentationManager, 317
- PegScreen, 312, 328, 337
- PegThing, 318
- Pegw32.lib, 320
- Pentium
 - configure for, 226
- Pentium III, 221
- performance, 226, 241, 278
- physical addresses, 17
- physical device, 250
 - files, 256
- PIC, 37

- plug-and-play, 90, 91
- PM_ADD, 327
- PM_CLOSE, 327
- PM_CURRENT, 327
- PM_DESTROY, 327
- PM_DIALOG_NOTIFY, 327
- PM_DRAW, 327
- PM_EXIT, 327
- PM_HIDE, 327
- PM_KEY, 318, 327
- PM_LBUTTONDOWN, 327
- PM_LBUTTONUP, 327
- PM_MAXIMIZE, 327
- PM_MINIMIZE, 327
- PM_NONCURRENT, 327
- PM_PARENTSIZED, 327
- PM_POINTER_ENTER, 327
- PM_POINTER_EXIT, 327
- PM_POINTER_MOVE, 327
- PM_RBUTTONDOWN, 328
- PM_RBUTTONUP, 328
- PM_RESTORE, 328
- PM_SHOW, 327
- PM_SIZE, 328
- PM_TIMER, 328
- PMBOOT.EXE, 29
- PnP BIOS, 90
 - example, 113
- polling, 231, 233, 234
- ports
 - I/O, 195
 - parallel, 84
 - reading, 70
 - serial, 205
 - writing, 70
- preemptions, 163, 191
- preemptive
 - scheduling, 158, 159, 233
- preprocessor, 23
- printer port, 84
- priority, 159, 164, 237
 - base, 166
 - default, 165
 - enquiring, 170
 - execution, 166
 - inheritance, 174
 - interrupt, 186
 - resource, 166
 - Win32, 197, 199
- privilege level, 17, 37, 76
- processor
 - see 386
- program
 - command line, 34
 - entity, 9
 - environment, 34
 - header, 28
 - termination, 165, 272
- protected mode, 14
- protection, 17, 116 - 118
- protector
 - library protection, 229
- protocols
 - serial ports, 206
- PSF_ACCEPTS_FOCUS, 324
- PSF_ALWAYS_ON_TOP, 324
- PSF_CURRENT, 324
- PSF_MOVEABLE, 324
- PSF_NONCLIENT, 324
- PSF_SELECTABLE, 324
- PSF_SIZEABLE, 324
- PSF_VIEWPORT, 324
- PSF_VISIBLE, 324
- PulseEvent, 201
- Pwindbl, 342
- RaiseException, 104
- RAM, 24
 - file system, 128, 301
 - remapping, 25
- RAM disk
 - driver, 285, 293
- RAMFile, 33
- raw I/O, 257, 276
- read-ahead
 - cache, 256
 - floppy disk driver, 288
- ReadConsoleInputA, 104
- ReadConsoleInputW, 104
- ReadFile, 100, 104
- Readme, 2
- ReadOnly, 24
- ReadProcessMemory, 104
- ReadSectors, 303
- ReadWrite, 24
- Ready, 169
 - task state, 160
- real-time, 128
 - definition, 158
 - file I/O, 299
 - introduction, 157
 - memory management, 188
 - systems, 157
- real-time clock, 79, 80
- real mode, 14
- reboot, 71, 117
- redistributables, 142
- reentrance, 160
 - definition, 161
 - libraries, 229
 - run-time system, 228
- RegCloseKey, 104
- Region command, 24
- regions
 - virtual, 116
- RegOpenKeyExA, 104
- RegQueryValueExA, 104
- ReleaseMutex, 104, 201
- ReleaseSemaphore, 201
- Relocation Report, 43
- remapping RAM, 25
- remote debugging, 49
- RemoveDirectoryA, 104
- renaming files, 266
- Reports
 - in LOC file, 42
- ReqOpenKeyA, 104
- Reserve command, 26
- Reset
 - Ctrl-Alt-Del, 117
- reset vector
 - booting, 46
- ResetEvent, 201
- resources, 174
 - display of, 171
 - management, 239
 - occupying, 177
 - priority, 174
 - rules, 174
 - task suspension, 174
 - task termination, 174
- ResumeThread, 104
- ResumeThread (Win32), 199
- return codes, 258, 272, 307
- ring
 - see privilege level
- ring 0, 76
- Robot, 340
- ROM, 24
- ROMable, 19, 32
 - demo program, 114
- root directory, 253
- Round-Robin, 173, 236
- RS232, 85
- RT_CLOSE_FIND_HANDLES, 281
- RT_KEY_BY_INTERRUPT, 67, 101
- RT_MOUSE_BY_INTERRUPT, 67, 101
- RTB files, 73
- RTBench, 226
- RTBenchA, 227
- RTBenchP, 226
- RTBootPM, 74
- RTBootRM, 74
- RTCallRing0, 76
- RTCharOutHandler, 67, 68
- RTCcloseCOMPort, 86
- RTCMOSExtendHeap, 80
- RTCMOSRead, 79
- RTCMOSReadTime, 79
- RTCMOSSetSystemTime, 80
- RTCMOSSetTime, 101
- RTCMOSWrite, 79
- RTCMOSWriteTime, 79
- RTCom, 205
- RTCOMError, 88
- RTD32, 49
- RTDisableInterrupts, 70
- RTDisableIRQ, 70
- RTDisplayChar, 68
- RTDisplayHex, 68
- RTDisplayHexW, 68
- RTDisplayInt, 68
- RTDisplayString, 68
- RTDLLThreadEvent, 75
- RTEMU, 131
- RTEnableInterrupts, 70
- RTEnableIRQ, 69
- RTExtendHeap, 78
- RTF_ATTR_..., 259
- RTF_CACHE_DATA, 259
- RTF_COMMITTED, 259
- RTF_CREATE, 259

- RTF_CREATE_ALWAYS, 259
- RTF_DEVICE_LAZY_WRITE, 285
- RTF_DEVICE_MOUNT_CONT., 254, 285
- RTF_DEVICE_NEW_LOCK, 285
- RTF_DEVICE_NO_MEDIA, 285
- RTF_DEVICE_REMOVABLE, 285
- RTF_DEVICE_RESOURCE_ERROR, 288, 291
- RTF_DEVICE_SINGLE_FAT, 285
- RTF_LAZY_DATA, 259
- RTF_OPEN_DIR, 259
- RTF_OPEN_NO_DIR, 259
- RTF_OPEN_SHARED, 259
- RTF_READ_ONLY, 259
- RTF_READ_WRITE, 259
- RTFBufferInfo, 278
- RTFBufferStatistic, 278
- RTFClose, 260
- RTFCloseAll, 272
- RTFCmd, 297
- RTFCmdMT, 297
- RTFCommit, 261
- RTFCommitAll, 272
- RTFCreateBootSector, 276
- RTFCreateDir, 265
- RTFCreateMasterBootRecord, 275
- RTFCriticalErrorHandler, 273
- RTFDATA.C, 283
- RTFDefaultCriticalErrorHandler, 275
- RTFDelete, 266
- RTFDevice, 284, 287
- RTFDeviceList, 284
- RTFDiskInfo, 268
- RTFDOSDirEntry, 262
- RTFDrvDOC, 285
- RTFDrvDOCData, 292
- RTFDrvFlash, 285
- RTFDrvFlashCompact, 280
- RTFDrvFlashData, 295
- RTFDrvFlashInfo, 279
- RTFDrvFloppy, 285
- RTFDrvFLPYData, 289
- RTFDrvIDE, 285
- RTFDrvNULL, 286
- RTFDrvRAM, 285
- RTFDrvSRAM, 285
- RTFDrvSRAMData, 293
- RTFDumpFileTable, 279
- RTFErrorAction, 273
- RTFErrorMessage, 272
- RTFExpandName, 267
- RTFExtend, 261
- RTFFFileInfo, 262
- RTFFFindClose, 266
- RTFFFindFirst, 265
- RTFFFindNext, 266
- RTFFLPYTurnMotorOFF, 279
- RTFFormat, 270
- RTFGetAttributes, 263
- RTFGetCurrentDir, 264
- RTFGetDiskInfoEx, 268
- RTFGetFileInfo, 262
- RTFGetFileSize, 263
- RTFGetPartitionInfo, 269
- RTFHANDLE, 258
- RTFiles-32, 83, 127
 - configuration, 283
 - with RTTarget-32, 301
- RTFindPhysMem, 76
- RTFMakeFileName, 267
- RTFMakeTempFileName, 266
- RTFOpen, 258, 273
- RTFPartitionInfo, 269
- RTFRawDiscardSectors, 277
- RTFRawGetDiskGeometry, 277
- RTFRawLowLevelFormat, 277
- RTFRawMediaChanged, 277
- RTFRawMount, 276
- RTFRawRead, 276
- RTFRawSetMedia, 276
- RTFRawShutDown, 276
- RTFRawWrite, 277
- RTFRead, 260
- RTFRemoveDir, 265
- RTFRename, 266
- RTFResetDisk, 267
- RTFSeek, 260
- RTFSetAttributes, 264
- RTFSetCriticalErrorHandler, 273
- RTFSetCurrentDir, 264
- RTFSetDefaultOpenFlags, 273
- RTFSetFileTime, 263
- RTFSetVolumeLabel, 270
- RTFShutDown, 272
- RTFSK32.LIB, 284
- RTFSplitPartition, 275
- RTFSRTT.LIB, 283
- RTFTruncate, 262
- RTFWrite, 260
- RTGetExtMem, 73
- RTGetMetaWEvents, 131
- RTGetMode, 73
- RTGetMouseEvents, 101
- RTGetVideoRAMAddr, 73
- RTHalt, 71
- RTHaltCPL3, 71
- RTHandleInfo, 99
- RTIn, 70, 195
- RTInD, 70
- RTInitCOMPort, 86
- RTInstallISR, 69
- RTInW, 70, 195
- RTIP-32, 1
- RTIRQEnd, 70
- RTK_MAX_PRIO, 165
- RTK_MIN_PRIO, 165
- RTK_NO_TASK, 167
- RTK32_VER, 197
- Rtk32api.txt, 124, 125
- RTKAlloc, 194
- RTKAllocMemPool, 188
- RTKAllocUserData, 168
- RTKCallIRQHandlerFar, 185
- RTKClearMailbox, 179
- RTKClearStatistic, 172
- RTKClearTraceBuffer, 189
- RTKCreateMailbox, 179
- RTKCreateSemaphore, 175
- RTKCreateTask, 238
- RTKCreateThread, 165
- RTKCurrentTaskHandle, 169
- RTKDeallocTerminatedTasks, 194
- RTKDebugVersion, 191
- RTKDelay, 173, 236
- RTKDelayUntil, 173
- RTKDeleteMailbox, 179
- RTKDeleteSemaphore, 176
- RTKDemo, 226
- RTKDisableInterrupts, 187
- RTKDisableIRQ, 187
- RTKDisableTrace, 189
- RTKDOS.H, 196
- RTKDumpTrace, 190
- RTKDuration, 172
- RTKEnableInterrupts, 188
- RTKEnableIRQ, 187
- RTKEnableTrace, 189
- RTKernel-32, 284, 288, 302
 - alternate APIs, 196
 - clock, 172
 - Debug Version, 191, 228
 - initialization, 165
 - module, 162
 - scheduler, 158
 - task, 159
 - termination, 165
- RTKernel-C for DOS, 196
- RTKERNEL.H, 196
- RTKernelInit, 165
- RTKeybrd, 212
- RTKeyLanguage, 80
- RTKeyTable, 80
- RTKFatalError, 194
- RTKFree8087, 168
- RTKFreeBuffer, 188
- RTKGet, 180
- RTKGetBuffer, 188
- RTKGetCond, 180
- RTKGetIRQHandler, 184
- RTKGetLocalData, 169
- RTKGetMinStack, 170, 171
- RTKGetTaskPrio, 170, 171
- RTKGetTaskStack, 170, 171
- RTKGetTaskState, 169, 171
- RTKGetTime, 173
- RTKGetTimed, 181
- RTKGetUserData, 169
- RTKInt, 226
- RTKIRQEnd, 187
- RTKIRQInfo, 186
- RTKIRQTopPriority, 186
- RTKListTitles, 171
- RTKLoadSymbols, 172
- RTKLPSemas, 230
- RTKMessages, 179
- RTKMtdCFI2_x, 295
- RTKNextCond, 181
- RTKOpenSemaphore, 176
- RTKPreemptionsOFF, 192
- RTKPreemptionsON, 191, 192
- RTKProtect8087, 168

- RTKProtectLibrary, 230
- RTKPulse, 174, 177
- RTKPut, 180
- RTKPutCond, 180
- RTKPutFront, 180
- RTKPutFrontCond, 180
- RTKPutFrontTimed, 181
- RTKPutTimed, 181
- RTKReceive, 182
- RTKReceiveCond, 183
- RTKReceiveTimed, 183
- RTKResetEvent, 178
- RTKResourceOwner, 176
- RTKRestoreIRQHandlerFar, 185
- RTKResume, 167
- RTKRTLCreateThread, 166
- RTKSavelRQHandlerFar, 185
- RTKScheduler, 192
- RTKSemaInfo, 176
- RTKSemaType, 175
- RTKSemaValue, 176
- RTKSend, 182
- RTKSendCond, 183
- RTKSendTimed, 183
- RTKSetEvent, 174
- RTKSetIRQHandler, 184
- RTKSetIRQStack, 185
- RTKSetMessageHandler, 192
- RTKSetPriority, 168
- RTKSetTaskStartStopHook, 194
- RTKSetTaskSwitchHook, 192
- RTKSetTime, 172
- RTKSetTraceBufferSize, 189
- RTKSetUserData, 168
- RTKSignal, 174, 177
- RTKStackCheck, 191
- RTKStopTracing, 189
- RTKSuspend, 167
- RTKTaskInfo, 170
 - headings, 171
- RTKTaskState, 169
- RTKTerminateTask, 167, 238
- RTKTime, 172
- RTKTimeSlice, 173
- RTKToWin32Handle, 198
- RTKTraceAll, 189
- RTKTraceBuffer, 189
- RTKTraceHeader, 190
- RTKTraceNames, 190
- RTKUserTrace, 190
- RTKWait, 174, 177
- RTKWaitCond, 177
- RTKWaitTimed, 178
- RTKWin32ToRTKHandle, 198
- RTLineStatus, 88
- RTLLoadRTBFile, 73
- RTLLoc, 19
 - Align command, 26
 - BinFile command, 40
 - command line, 19
 - COMPort command, 39
 - config. files, 21
 - DLL command, 26
 - FillRAM command, 25
 - GMode command, 38
 - HexFile command, 40
 - LOC file, 42
 - Locate command, 27
 - locate process, 43
 - options, 19, 21
 - Output command, 40
 - Region command, 24
 - Reserve command, 26
 - VideoRAM command, 38
 - Virtual command, 25
- RTLLocateSection, 71
- RTLockHeap, 75
- RtlUnwind, 104
- RTMakeBootDisk, 83
- RTMapMem, 77
- RTMetaWInit, 131
- RTModemControl, 88
- RTModemStatus, 88
- RTNewEvents, 101
- RTOut, 70, 195
- RTOutD, 70
- RTOutW, 70, 195
- RTPCCardPresent, 93
- RTPCEnableIRQ, 96
- RTPCGetFirstTuple, 94
- RTPCGetFunctionID, 93
- RTPCGetNextTuple, 94
- RTPCGetTupleData, 94
- RTPCInit, 92
- RTPCIsATA, 96
- RTPCIsUART, 96
- RTPCMapATA, 97
- RTPCMapIOWindow, 95
- RTPCMapMemoryWindow, 95
- RTPCMapUART, 96
- RTPCPowerUp, 93
- RTPCSetConfigRegister, 95
- RTPCShutDown, 93
- RTPCUnmapCIS, 95
- RTPCUnmapSocket, 96
- RTPEG-32, 310
- RTPrintByte, 85
- RTPrinterInit, 84
- RTPrinterSetIOBase, 84
- RTPrinterStatus, 85
- RTProcessEvents, 101
- RTRaiseCPUException, 78
- RTReboot, 71
- RTReceiveBufferCount, 87
- RTReceiveChar, 88
- RTReceiveCharTimed, 88
- RTReleaseVirtualAddress, 77
- RTReserveVirtualAddress, 77
- RTRestoreBootSector, 84
- RTRestoreInterrupts, 70
- RTRestoreVector, 69
- RTRun, 46
- RTRunProgram, 74
 - example, 113
- RTS/CTS, 206
- RTSaveAndDisableInterrupts, 70
- RTSaveVector, 68
- RTSectionName, 73
- RTSendBlock, 87
- RTSendBlockTimed, 87
- RTSendBufferCount, 87
- RTSendChar, 87
- RTSendCharTimed, 87
- RTSetCodepageTranslation, 81
- RTSetDisplayHandler, 67
- RTSetFlags, 67
- RTSetIntVector, 69
- RTSetKeyboard, 80
- RTSetKeyboardTables, 80
- RTSetTrapVector, 69
- RTSignalEvent, 101
- RTT_BIOS_FindClassCode, 89
- RTT_BIOS_FindDevice, 89
- RTT_BIOS_GenSpecialCycle, 90
- RTT_BIOS_GetInterruptRouting, 89
- RTT_BIOS_Installed, 89
- RTT_BIOS_ReadConfigData, 90
- RTT_BIOS_SetPCInt, 90
- RTT_BIOS_WriteConfigData, 90
- RTT_PNP_CallPnPBIOs, 91
- RTT_PNP_Installed, 91
- RTT32.LIB, 66
- Rtt32api.txt, 124
- RTT32DLL.DLL, 123
- RTTarget-32, 283, 288
 - library, 66
 - native API, 67
 - version, 105
 - with RTFiles-32, 301
- RTTARGET.H, 67
- RTTARGET.INI, 49
- RTTarget32Flags, 66
- RTTBOOT, 47, 48
- RTTCOM, 85
 - demo program, 112
- RTCcloseCOMPort, 86
- RTCOMError, 88
- RTInitCOMPort, 86
- RTLineStatus, 88
- RTModemControl, 88
- RTModemStatus, 88
- RTReceiveBufferCount, 87
- RTReceiveChar, 88
- RTReceiveCharTimed, 88
- RTSendBlock, 87
- RTSendBlockTimed, 87
- RTSendBufferCount, 87
- RTSendChar, 87
- RTSendCharTimed, 87
- RTTextIO, 213
- RTTHeap, 107, 222, 229
- RTTickFactor, 101
- RTUnlockHeap, 76
- RTWait, 71
- RTWaitEvent, 101
- run-time system, 119
 - reentrance, 228
 - running without, 111, 119, 136, 138, 139
- running
 - program on target, 45

- the debugger, 51
- scheduler, 158
 - calling the, 192
 - rules, 158
- scheduling
 - preemptive, 192
- screen, 213
 - driver, 337
- screen driver, 312
- screen I/O, 67, 68
- screen memory, 38
- scrolling, 335
- searching files, 265, 281
- secondary FAT, 253
- Section
 - Locate command, 27, 116
- sector, 250, 251
 - aligned access, 250
- security, 299
- seek, 260
- segment
 - registers, 14
- selectors, 14
- semaphore
 - binary, 174
 - counting, 174
 - demo program, 226
 - events, 174
 - example, 175
 - function, 173
 - name, 175
 - resource, 174
 - Win32, 200
- SerDemo, 112
- serial I/O
 - demo program, 112, 226
- serial port
 - PCMCIA, 96
- serial ports, 85, 205
- SerInt, 111
- Set command, 34
- SetConsoleCtrlHandler, 104
- SetConsoleCursorInfo, 104
- SetConsoleCursorPosition, 104
- SetConsoleMode, 104
- SetConsoleScreenBufferSize, 104
- SetConsoleWindowInfo, 104
- SetCurrentDirectoryA, 104
- SetEndOfFile, 104
- SetEnvironmentVariableA, 104
- SetEnvironmentVariableW, 104
- SetEvent, 104, 201
- SetFileAttributesA, 104
- SetFilePointer, 104
- SetFileTime, 104
- SetHandleCount, 104
- SetLastError, 104
- SetLocalTime, 101, 104
- SetSidHandle, 104
- SetSystemTime, 101, 104
- SetThreadLocale, 100, 104
- SetThreadPriority (Win32), 199
- SetUnhandledExceptionFilter, 104
- SetVolumeLabelA, 104
- shut down, 272
- ShutDown, 303
- shutdown mode, 71
- SHGetFileInfoA, 104
- sibling, 311
- signals
 - RTPEG-32, 315
- size of file, 263
- SizeofResource, 104
- Sleep, 104
- Sleep (Win32), 199
- software interrupt, 18
- source level debugging, 49
- source position, 171, 172
- special files, 256
- speed
 - file I/O, 298
- Spread, 340
- SRAM
 - driver, 285, 292
- SRCNULL, 223
- SRCTDS, 223
- SSE, 221
- stack, 164
 - checking, 166, 191, 239
 - enquiring, 170
 - interrupt, 164, 185
 - overflow, 118
 - panic, 184
 - task, 160, 164, 166
- Stack
 - Locate command, 30
- status flags
 - RTPEG-32, 324
- STI, 188
- storage device, 251
- streams, 282
- style flags
 - RTPEG-32, 325
- subdirectory, 253
- support, 2
- Suspended, 166, 167, 169
 - task state, 160
- SuspendThread (Win32), 199
- symbol table, 172
- SysAllocStringLen, 104
- SysDemo, 112
- SysFreeString, 104
- SysRead, 24
- SysReAllocStringLen, 104
- SYSRT32, 218
- SYSSTD, 218
- SysStringLen, 104
- System, 24
- system driver, 254, 283
- system tick, 101
- system time, 80, 101, 283, 28
- SystemTimeToFileTime, 104
- Table, 340
- table of files, 279
- target, 8
- task
 - communication, 160
 - CPU time, 171, 172
 - creation, 165, 166, 238
 - cyclic, 236
 - definition, 157, 159
 - function, 165
 - handle, 159, 166
 - Idle, 160
 - list, 170
 - Main, 160
 - memory needs, 166
 - name, 166
 - parameter, 166
 - priority, 159, 164, 170
 - stack, 160, 164, 166, 170
 - state, 160, 169
 - suspension, 174
 - synchronization, 157
 - TaskHandle, 166
 - termination, 167, 174, 238
 - timer, 236
- task switch
 - activating, 159
 - blocking, 159
 - coprocessor, 168
 - hook, 192
 - number of, 171
 - time slice, 159
 - types, 159
- TaskState, 169
- TCP/IP
 - see RTIP-32
- technical support, 2
- temporary files, 266
- Terminal, 340
- Terminated, 170
- TerminateProcess, 104
- TerminateThread (Win32), 199
- terminating tasks, 229
- termination, 165, 272
 - of tasks, 167
- terms
 - RTPEG-32, 311
- terms used, 8, 250
- TF_MATH_CONTEXT, 166
- TF_NO_MATH_CONTEXT, 166
- TF_SUSPENDED, 166
- thread
 - see task
 - definition, 157
- thread id
 - Win32, 197
- Thread Local Storage
 - see TLS data
- threads, 76
- Threads, 226
- throughput
 - file I/O, 298
- TIElapsedAndMark, 205
- TIElapsedTime, 205
- TIFineTimeToSeconds, 205
- time, 101, 172, 202
 - measuring, 203
 - system time, 80

- time of day, 283, 284
- time of files, 263
- time sharing, 157, 236
- time slicing, 159, 165, 236
 - cooperative, 173
 - RTKTimeSlice, 173
- Timed
 - task state, 160
- TimedGet, 169
- TimedPut, 169
- TimedReceive, 169
- TimedSend, 169
- TimedWait, 169
- timer, 37, 204
 - chaining, 219
 - interrupt, 118
 - task, 236
 - tick, 101, 172, 203
 - tick interval, 204, 205
- Timer
 - RTPEG-32, 332
- timer units
 - of clock device, 203
- TimerInit, 205
- TimeSlice, 159, 173
- TISecondsToTicks, 205
- TISetTimerInterval, 205
- TITicksToSeconds, 205
- TLINK32, 136
- TLS, 102
- TLS data, 9
 - Borland C/C++, 136
 - Microsoft Visual C/C++, 138
- TlsAlloc, 104
- TlsFree, 104
- TlsGetValue, 104
- TlsSetValue, 104
- tracer
 - see *kernel tracer*
- traps, 18
- TreeView, 340
- truncate, 262
- TS_BLOCKED_GET, 169
- TS_BLOCKED_PUT, 169
- TS_BLOCKED_RECEIVE, 169
- TS_BLOCKED_SEND, 169
- TS_BLOCKED_WAIT, 169
- TS_CURRENT, 169
- TS_DEADLOCKED, 169
- TS_DELAYING, 169
- TS_ILLEGAL, 170
- TS_READY, 169
- TS_SUSPENDED, 169
- TS_TERMINATED, 170
- TS_TIMED_GET, 169
- TS_TIMED_PUT, 169
- TS_TIMED_RECEIVE, 169
- TS_TIMED_SEND, 169
- TS_TIMED_WAIT, 169
- Turbo Vision
 - demo program, 114
- TVDemo, 114
- types
 - of RTPEG-32 objects, 325
- UART, 205
 - PCMCIA, 96
- uncommitted memory, 9, 106, 107
- UnhandledExceptionFilter, 104
- UnhookWindowsHookEx, 104
- Unicode, 80, 99, 197, 320, 340
- unique files, 266
- UnlockFile, 104
- user events
 - kernel tracer, 190
- USER32.DLL, 101
- V24, 85
- V86 mode, 14
- VariantClear, 104
- VariantCopy, 104
- VariantCopyInd, 104
- VC, 62
- VCL, 115
- vector font, 335
- version
 - of RTTarget-32, 105
- VESA, 38
- VESA_16, 337
- VESA_24, 337
- VESA_32, 338
- VESA_8, 337
- VESATEST.COM, 39
- VGA, 38
- VGA_4, 338
- VGASCRN, 338
- VideoRAM command, 38
- viewports, 334
- virtual 8086 mode, 14
- virtual addresses, 17
- Virtual command, 25
- virtual heap, 107
- virtual regions, 116
 - example, 114
- VirtualAlloc, 104
- VirtualFree, 104
- VirtualProtect, 104
- VirtualQuery, 104
- Visual C/C++
 - see *Microsoft C/C++*
- Visual Studio, 62
- volume, 250
- W32apimt.txt, 125
- W32Bench, 227
- wait
 - for interrupt, 71
- WaitForSingleObject, 104, 201
- warning messages, 43, 143
 - ignoring, 36
- Watcom C/C++
 - compiling with, 139
 - debug symbols, 140
- WClearScreen, 215
- WCloseWindow, 215
- WCursorOFF, 216
- WCursorON, 216
- WCursorXY, 216
- WDefineFunctionKey, 215
- WFrame, 215
- WGetS, 216
- WGotoXY, 216
- WideCharToMultiByte, 104
- Win32
 - adding functions, 105
 - API, 98, 197, 281
 - Critical Section, 200
 - emulation, 98, 301
 - error handling, 197
 - Event, 200
 - handles, 99, 197, 281
 - Mutex, 200
 - priorities, 197
 - priority, 199
 - semaphore, 200
 - thread, 198
 - thread id, 197
- Win32api.txt, 124
- window
 - GUI, 311
- Window Builder, 342
- windows
 - on screen, 213
- Windows
 - emulation of RTPEG-32, 320
- WNewWindow, 215
- WOpenWindow, 215
- World Wide Web, 3
- Wprintf, 217
- WPutC, 216
- WPutS, 216
- WriteConsoleA, 104
- WriteConsoleInput, 101
- WriteConsoleInputA, 104
- WriteConsoleOutputA, 104
- WriteConsoleW, 104
- WriteFile, 100, 104
- WriteSectors, 303
- WSetColor, 216
- WSetCursor, 216
- WSetScreenSize, 214
- WSetUserInput, 214
- WSetVideoRAMAddress, 214
- wsprintfA, 104
- XON/XOFF, 206

Table of Contents

Welcome to On Time RTOS-32	1
Hardware and Software Requirements	2
This Manual	2
Technical Support	2
Support Web Page and Mailing Lists	3
Installation	3
Licensing Terms and Liability	5
Part I RTTarget-32	6
Features of RTTarget-32	7
Terms and Definitions	8
Chapter 1 Running Win32 Programs without Win32	10
Benefits of Running without Windows	10
Benefits of Running with Windows	11
Preparing a Program for RTTarget-32	11
Locating a Program	11
Cross Debugging a Program	12
A Complete Example	12
Chapter 2 The i386 Microprocessor	14
Real-Address Mode	14
Virtual 8086 Mode	14
Protected Mode	14
16-Bit Protected Mode	15
32-Bit Protected Mode	15
Descriptors and Descriptor Tables	15
Privilege Levels	17
Paging	17
Virtual, Linear, and Physical Addresses	17
Exceptions and Interrupts	18
Chapter 3 RTLoc: Locating a Program	19
Invoking RTLoc	19
RTLoc Options	19
Options Command	21
Configuration Files	21
Specifying Numeric Values	21
Preprocessor Directives	23
Macros	23
Defining the Target Hardware	23
Region Command	24
Virtual Command	25
FillRAM Command	25
Defining Program Location	26
DLL Command	26
Align Command	26
Reserve Command	26
Locate Command	27
Section	27
NTSection	28
Header	28

BootCode	29
BootData	29
BootVector	29
BIOSVector	30
DiskBuffer	30
Stack	30
Heap	31
PageTable	31
Copy	32
DecompCode	32
DecompData	33
File	33
Nothing	33
Defining Program Options	34
Set Command	34
Commandline Command	34
Init Command	34
Link Command	35
IgnoreMsg Command	36
Defining Boot Code Options	37
BOOTFLAGS Command	37
VideoRAM Command	38
GMode Command	38
COMPort Command	39
Creating Output Files	39
Output Command	40
HexFile Command	40
BinFile Command	40
Initializing Target Hardware	41
OUT Commands	41
Delay Command	41
InitCode Commands	41
The LOC File	42
The Locate Process in Detail	43
Chapter 4 Running a Program on the Target	45
Booting from Disk	45
Program BootDisk	45
Booting from a BIOS Extension	46
Booting from the CPU Reset Vector	46
Downloading	46
Program RTRun	46
Booting from MS-DOS	47
Program RTTBOOT	48
Chapter 5 Cross Debugger RTD32	49
File RTTARGET.INI	49
Prerequisites for Cross Debugging	51
The Debug Monitor	51
Differences from Borland's TD32	51
A Quick Example	52
Debugger Reference	52
RTD32 Command Line	52
Navigating in RTD32	53
Expressions	53

Menu Commands	54
File	54
Edit	54
View	54
Run	54
Breakpoint	55
Data	56
Options	56
Window	57
Help	57
Debugger Windows	58
Source Module	58
Inspect	58
Watch	58
Breakpoints	58
Stack	58
Log	58
Variables	58
File	58
CPU	58
Code Pane	59
Register Pane	59
Stack Pane	60
Register	60
Numeric Processor	60
Dump	60
Execution History	60
Class Hierarchy	60
Global Descriptor Table	60
Interrupt Descriptor Table	60
Clipboard	60
Keyboard Shortcuts	61
Chapter 6 Using Microsoft Visual Studio	62
Program DBGShell	62
Setting up a Project	62
Cross Debugging	64
Chapter 7 RTTarget-32 Library	66
RTTarget-32 Flags	66
RTTarget-32's Native API	67
Function RTSetFlags	67
Function RTSetDisplayHandler	67
Function RTDisplayChar	68
Function RTDisplayString	68
Function RTDisplayInt	68
Function RTDisplayHex	68
Function RTDisplayHexW	68
Function RTSaveVector	68
Function RTRestoreVector	69
Function RTSetIntVector	69
Function RTSetTrapVector	69
Function RTInstallISR	69
Function RTEnableIRQ	69
Function RTDisableIRQ	70
Function RTIRQEnd	70

Function RTDisableInterrupts	70
Function RTEnableInterrupts	70
Function RTSaveAndDisableInterrupts	70
Function RTRestoreInterrupts	70
Functions RTIn, RTInW, RTInD	70
Functions RTOut, RTOutW, RTOutD	70
Function RTReboot	71
Function RTHalt	71
Function RTHaltCPL3	71
Function RTWait	71
Function RTLocateSection	71
Function RTSectionName	73
Function RTGetExtMem	73
Function RTGetGMode	73
Function RTGetVideoRAMAddr	73
Function RTLoadRTBFile	73
Function RTRunProgram	74
Function RTBootRM and RTBootPM	74
Function RTDLLThreadEvent	75
Function RTLockHeap	75
Function RTUnlockHeap	76
Function RTCallRing0	76
Function RTFindPhysMem	76
Function RTReserveVirtualAddress	77
Function RTReleaseVirtualAddress	77
Function RTMapMem	77
Function RTExtendHeap	78
Function RTRaiseCPUException	78
Function RTCMOSRead	79
Function RTCMOSWrite	79
Function RTCMOSReadTime	79
Function RTCMOSWriteTime	79
Function RTCMOSSetSystemTime	80
Function RTCMOSExtendHeap	80
Function RTSetKeyboard	80
Function RTSetKeyboardTables	80
Function RTSetCodepageTranslation	81
Function RTInitMouse	81
Function RTInitTextMouse	82
Function RTSetMousePos	82
Function RTMouseDone	82
Function RTTextMouseDone	82
Function RTMakeBootDisk	83
Function RTRestoreBootSector	84
Function RTPrinterSetIOBase	84
Function RTPrinterInit	84
Function RTPrinterStatus	85
Function RTPrintByte	85
Serial I/O Functions	85
Function RTInitCOMPort	86
Function RTCloseCOMPort	86
Function RTSendChar	87
Function RTSendCharTimed	87
Function RTSendBlock	87
Function RTSendBlockTimed	87

Function RTSendBufferCount	87
Function RTReceiveBufferCount	87
Function RTReceiveChar	88
Function RTReceiveCharTimed	88
Function RTCOMError	88
Function RTLineStatus	88
Function RTModemStatus	88
Function RTModemControl	88
PCI BIOS Functions	88
Function RTT_BIOS_Installed	89
Function RTT_BIOS_FindDevice	89
Function RTT_BIOS_FindClassCode	89
Function RTT_BIOS_GetInterruptRouting	89
Function RTT_BIOS_SetPCInt	90
Function RTT_BIOS_GenSpecialCycle	90
Function RTT_BIOS_ReadConfigData	90
Function RTT_BIOS_WriteConfigData	90
Plug-and-Play BIOS Functions	90
Function RTT_PNP_Installed	91
Function RTT_PNP_CallPnPBIOS	91
PC Cards (PCMCIA)	91
Function RTPCInit	92
Function RTPCShutDown	93
Function RTPCCardPresent	93
Function RTPCPowerUp	93
Function RTPCGetFunctionID	93
Function RTPCGetFirstTuple	94
Function RTPCGetNextTuple	94
Function RTPCGetTupleData	94
Function RTPCSetConfigRegister	95
Function RTPCUnmapCIS	95
Function RTPCMapMemoryWindow	95
Function RTPCMapIOWindow	95
Function RTPCEnableIRQ	96
Function RTPCUnmapSocket	96
Function RTPCIsATA	96
Function RTPCIsUART	96
Function RTPCMapUART	96
Function RTPCMapATA	97
DOS Emulation	98
DPMI Emulation	98
Win32 Emulation	98
Win32 Handles	99
Function RTHandleInfo	99
Win32 Memory Management	99
Win32 File I/O	100
Win32 Console I/O	100
Console Input Event Management	100
Win32 Time Management	101
Win32 DLLs	101
Win32 Exception Handling	102
Win32 Thread Local Storage (TLS)	102
Win32 API Function Cross Reference	102
Adding other Win32 Functions	105

RTTarget-32's Memory Managers	106
Fixed Memory Manager	106
Virtual or Uncommitted Memory Manager	107
Alternate Heap Manager RTTHeap	107
Chapter 8 Demo Programs	109
Running Demos with Command Line Tools	109
Running Demos in Visual Studio 6.0	109
Preparing a Standard PC to Act as a Target	110
Preparing a Standard PC to Act as a Target for GUI Demos	110
Preparing the AMD Élan SC400 Evaluation Board	110
Preparing the AMD Élan SC520 Evaluation Board	110
Preparing the NS486 Evaluation Board	111
Program Hello	111
Program Hello2	111
Program SerInt	111
Program SerDemo	112
Program MAPDemo	112
Program EmuDemo	112
Program DLLDemo	112
Program DLLDemo2	112
Program DLLDemo3	112
Program SysDemo	112
Program Loader	113
Program BootProg	113
Program BIOSDemo	113
Program PCCard	113
Program PCCardMT	114
Program EXLED	114
Program HelloSc400	114
Program HelloSc520	114
Program NSHello	114
Program TVDemo	114
Program ClassDemo	115
Program MetWorld	115
Program HelloGUI	115
Chapter 9 Advanced Topics	116
Choosing a Locate Method	116
Locate Section or NTSection	116
Physical or Virtual Regions	116
Running with or without Paging	116
Running at CPL 0 or 3	117
Installing Hardware Interrupt Handlers	117
Catching NULL Pointer Assignments	118
Catching Stack Overflows	118
Running with or without Run-Time System	119
Avoid Repeated Downloads	119
Switching between Configurations with and without Debug Monitor	120
Using Data Compression	121
Downloading and Cross Debugging	121
Applications Booted from Disk	121
Applications copied from ROM to RAM	122
Applications Running in ROM	122
Using DLLs through RTLoc	122
Using RTT32DLL.DLL	123

Linking RTT32.LIB into the EXE	123
Using a Custom RTTarget-32 System DLL	124
Utility MakeDef	124
Differences from Win32	125
Loading DLLs through a File System	125
Advantages of DLMs	126
Disadvantages of DLMs	127
Installable File System	127
Multithread Applications	128
Using the MetaWINDOW Graphics Library	129
Prerequisites	129
Initialization	130
Limitations	130
Function RTMetaWInit	131
Function RTGetMetaWEvents	131
Using the 387 Emulator	131
Linking the Emulator in C/C++ Programs	132
Linking the Emulator in Delphi Programs	132
Emulator Licensing Terms	133
Appendix A Compiling and Linking with On Time RTOS-32	134
General Rules	134
Order of Libraries	134
Borland C++	136
Microsoft Visual C++	137
Watcom C/C++	139
Borland Delphi	141
Appendix B Redistributable Components of RTTarget-32	142
Appendix C RTLoc Error, Warning, and Information Messages	143
Part II RTKernel-32	155
Chapter 1 Multitasking, Real-Time, and RTKernel-32	157
What is Multitasking?	157
Time Sharing	157
Real-Time Systems	157
Cooperative and Preemptive Multitasking	158
Real-Time	158
RTKernel-32's Scheduler	158
Task Switches	159
RTKernel-32 Tasks	159
Inter-Task Communications	160
Reentrance	161
Chapter 2 Module RTKernel-32	162
RTKernel-32 Configuration	162
StructureSize	162
DriverFlags	163
UserDriverFlags	163
Flags	163
DefaultTaskStackSize	164
DefaultIntStackSize	164
MainPriority	164
DefaultPriority	164
HookedIRQs	164
TaskStackOverhead	164

TimeSlice	165
RTKernel-32 Initialization	165
Function RTKernelInit	165
RTKernel-32 Exit Function	165
Task Management	165
Function RTKCreateThread	165
Function RTKRTLCreateThread	166
Function RTKTerminateTask	167
Function RTKSuspend	167
Function RTKResume	167
Function RTKSetPriority	168
Function RTKProtect8087	168
Function RTKFree8087	168
Function RTKAllocUserData	168
Function RTKSetUserData	168
Function RTKGetUserData	169
Function RTKGetLocalData	169
Enquiring Tasks	169
Function RTKCurrentTaskHandle	169
Function RTKGetTaskState	169
Function RTKGetTaskPrio	170
Function RTKGetTaskStack	170
Function RTKGetMinStack	170
Function RTKTaskInfo	170
Function RTKClearStatistic	172
Function RTKLoadSymbols	172
Time	172
Function RTKSetTime	172
Function RTKGetTime	173
Function RTKDelay	173
Function RTKDelayUntil	173
Function RTKTimeSlice	173
Semaphores	173
Function RTKCreateSemaphore	175
Function RTKOpenSemaphore	176
Function RTKDeleteSemaphore	176
Function RTKSemaInfo	176
Function RTKSemaValue	176
Function RTKResourceOwner	176
Function RTKSignal	177
Function RTKPulse	177
Function RTKWait	177
Function RTKWaitCond	177
Function RTKWaitTimed	178
Function RTKResetEvent	178
Mailboxes	178
Function RTKCreateMailbox	179
Function RTKDeleteMailbox	179
Function RTKClearMailbox	179
Function RTKMessages	179
Function RTKPut	180
Function RTKPutFront	180
Function RTKGet	180
Function RTKPutCond	180
Function RTKPutFrontCond	180

Function RTKGetCond	180
Function RTKPutTimed	181
Function RTKPutFrontTimed	181
Function RTKGetTimed	181
Function RTKNextCond	181
Message Passing	181
Function RTKSend	182
Function RTKReceive	182
Function RTKSendCond	183
Function RTKReceiveCond	183
Function RTKSendTimed	183
Function RTKReceiveTimed	183
Interrupt Handling	183
Function RTKSetIRQHandler	184
Function RTKGetIRQHandler	184
Function RTKSaveIRQHandlerFar	185
Function RTKRestoreIRQHandlerFar	185
Function RTKCallIRQHandlerFar	185
Function RTKSetIRQStack	185
Function RTKIRQInfo	186
Function RTKIRQTopPriority	186
Function RTKEnableIRQ	187
Function RTKDisableIRQ	187
Function RTKIRQEnd	187
Function RTKDisableInterrupts	187
Function RTKEnableInterrupts	188
Real-Time Memory Management	188
Function RTKAllocMemPool	188
Function RTKGetBuffer	188
Function RTKFreeBuffer	188
The Kernel Tracer	188
Function RTKSetTraceBufferSize	189
Function RTKEnableTrace	189
Function RTKTraceAll	189
Function RTKDisableTrace	189
Function RTKStopTracing	189
Function RTKClearTraceBuffer	189
Function RTKUserTrace	190
Function RTKTraceHeader	190
Function RTKDumpTrace	190
Miscellaneous RTKernel-32 Operations	191
Function RTKDebugVersion	191
Function RTKStackCheck	191
Function RTKCanPreempt	191
Function RTKPreemptionsON	192
Function RTKPreemptionsOFF	192
Function RTKScheduler	192
Function RTKSetMessageHandler	192
Function RTKSetTaskSwitchHook	192
Function RTKSetTaskStartStopHook	194
Function RTKFatalError	194
Function RTKAlloc	194
Function RTKDeallocTerminatedTasks	194
Functions RTIn, RTInW, RTInD, RTOut, RTOutW, RTOutD	195

Chapter 3	Alternate APIs for RTKernel-32	196
RTKernel-C 4.5 for DOS Compatible API		196
Win32 Thread Compatible API		197
Win32 Priorities		197
Win32 Handles		197
Win32 and RTKernel-32 Error Handling		197
Mixing RTKernel-32 and Win32 APIs		198
Function RTKWin32ToRTKHandle		198
Function RTKToWin32Handle		198
Function GetCurrentThreadId		198
Function CreateThread		198
Function ExitThread		199
Function TerminateThread		199
Function GetExitCodeThread		199
Function GetCurrentThread		199
Function Sleep		199
Function GetTickCount		199
Function SuspendThread		199
Function ResumeThread		199
Function SetThreadPriority		199
Function GetThreadPriority		200
Function InitializeCriticalSection		200
Function EnterCriticalSection		200
Function LeaveCriticalSection		200
Function DeleteCriticalSection		200
Function CreateEvent		200
Function CreateMutex		200
Function CreateSemaphore		200
Function OpenEvent		201
Function OpenMutex		201
Function OpenSemaphore		201
Function SetEvent		201
Function ResetEvent		201
Function PulseEvent		201
Function ReleaseMutex		201
Function ReleaseSemaphore		201
Function WaitForSingleObject		201
Chapter 4	Supplemental Modules	202
Module FineTime		202
Function FTSetResolution		202
Function FTCalibrate		202
Fine Time Arithmetic Functions		202
Function FTReadTime		203
Time Interval Measurements		203
Time Conversions		203
Module Clock		203
Function CLKSetResolution		204
Function CLKSetTimerIntVal		204
Time Conversions		204
Module Timer		204
Function TimerInit		205
Function TIElapsedTime		205
Function TIElapsedAndMark		205
Function TISetTimerInterval		205
Time Conversions		205

Module RTCom	205
Protocols	206
Hardware Configuration	207
DigiBoard Cards (PC/4, PC/8, PC/16)	207
Hostess Cards (4, 8, or 16 Ports)	207
Other Interrupt Sharing Cards (2 to 32 Ports)	208
Function COMSetBoardType	208
Function COMSetIOBase	208
Function COMSetIRQ	208
Function COMPortInit	209
Function COMHasFIFO	209
Function COMEnableFIFO	209
Function COMSetProtocol	209
Function COMAllocateBuffers	210
Function COMEnableInterrupt	210
Function COMDisableInterrupt	210
Function COMSendChar	210
Function COMSendCharTimed	210
Function COMSendBlock	210
Function COMSendBlockTimed	210
Function COMWaitSendBufferEmpty	211
Function COMSetModemStatusHook	211
Function COMReceiveCharPolled	211
Function COMSendCharPolled	211
Function COMLineStatus	211
Function COMModemStatus	212
Function COMModemControl	212
Function COMError	212
Module RTKeybrd	212
Function KBInit	213
Function KBKeyPressed	213
Function KBGetCh	213
Function KBPutCh	213
Module RTTextIO	213
Function WSetVideoRAMAddress	214
Function WSetScreenSize	214
Function WSetUserInput	214
Function WDefineFunctionKey	215
Function WClearScreen	215
Function WNewWindow	215
Function WOpenWindow	215
Function WCloseWindow	215
Function WFrame	215
Function WGotoXY	216
Function WCursorXY	216
Function WCursorOFF	216
Function WCursorON	216
Function WSetCursor	216
Function WSetColor	216
Function WPutC	216
Function WPutS	216
Function WGetS	216
Function Wprintf	217
Module CPUMoni	217
Function CPUMonitorStart	217

Function CPURelativeLoad	217
Chapter 5 RTKernel-32 Drivers	218
System Interface	218
Driver SYSSTD	218
Driver SYSRT32	218
Interrupt Handling	219
Driver IRQRT32	219
Kernel Clock	219
Driver CLKPC	219
Driver CLKHRTPC	219
High Resolution Timer	220
Driver HRTNULL	220
Driver HRTPC	220
Driver CLKHRTPC	220
Driver HRTPEM	220
Driver HRTSC520	220
Floating Point	221
Driver FLTNULL	221
Driver FLT387	221
Driver FLTPII	221
Driver FLTEMUMT	221
Memory Management	221
Driver MEMCHEAP	222
Driver MEMSTH	222
Driver MEMSTCH	222
Driver MEMW32	222
Source Code Positions	222
Driver SRCNULL	223
Driver SRCTDS	223
CPU	223
Driver CPU386F	223
Driver CPU386	223
Overview of all Drivers	224
Preconfigured Driver Library DRVRT32.LIB	225
Chapter 6 Demo Programs	226
Program Threads	226
Program RTKDemo	226
Program RTKInt	226
Program COMDemo	226
Program RTBench	226
Program RTBenchP	226
Program RTBenchA	227
Program W32Bench	227
Chapter 7 Advanced Topics	228
RTKernel-32's Debug Version	228
Reentrance of the C/C++ Run-Time Systems	228
Multithreaded Libraries	229
Replacements for Non-Reentrant Parts of the Run-Time System	229
Automatic Library Protection	229
How to Create Threads	231
Interrupt Handling	231
Structure of an Interrupt Handler	232
Avoid Polling	233

Preemptive or Cooperative Multitasking?	233
Advantages of Preemptive Scheduling	233
Advantages of Cooperative Scheduling	233
Waiting for Several Events	234
Avoid Large Data Types for Mailboxes and Message Passing	235
Mutual Exclusion	236
Avoid Time Slicing	236
Cyclic Tasks (Timer)	236
Priorities	237
Starting Objects' Methods as Tasks	237
Creating and Terminating Tasks	238
Chapter 8 Typical Error Sources	239
Program Termination	239
Stack Errors	239
Resource Management	239
Deadlock	239
Appendix A Performance and Interrupt Response Times	241
Appendix B Task Switches in Cooperative Scheduling	242
Appendix C Writing Custom Kernel Drivers	243
Appendix D Error and Information Messages	245
Error Messages	245
Informational Messages	247
Part III RTFiles-32	248
Terms and Definitions	250
Chapter 1 The FAT File System Structure	251
Sectors, Sector Addressing, and Clusters	251
Logical Drives and Partition Tables	252
The Boot Record	252
The File Allocation Table and Cluster Sizes	252
Directories and Files	253
Chapter 2 RTFiles-32 in Embedded Applications	254
Structure of an RTFiles-32 Program	254
RTFiles-32 APIs	254
Mounting Devices and Logical Drives	254
RTFiles-32 Buffers	255
File Types	256
Data Files	256
Directory Files	256
Logical Drive Files	256
Physical Device Files	256
Raw I/O	257
Chapter 3 RTFiles-32 Native API	258
Return Codes and File Handles	258
General File I/O	258
Function RTFOpen	258
Function RTFClose	260
Function RTFRead	260
Function RTFWrite	260
Function RTFSeek	260

Function RTFExtend	261
Function RTFCommit	261
Function RTFTruncate	262
Information about Files	262
Function RTFGetFileInfo	262
Function RTFGetFileSize	263
Function RTFSetFileTime	263
File Attributes	263
Function RTFGetAttributes	263
Function RTFSetAttributes	264
Directories	264
Function RTFGetCurrentDir	264
Function RTFSetCurrentDir	264
Function RTFCreateDir	265
Function RTFRemoveDir	265
Finding Files	265
Function RTFFindFirst	265
Function RTFFindNext	266
Function RTFFindClose	266
File Name Operations	266
Function RTFRename	266
Function RTFDelete	266
Function RTFMakeTempFileName	266
Function RTFMakeFileName	267
Function RTFExpandName	267
Disk and Volume Management	267
Function RTFResetDisk	267
Function RTFGetDiskInfoEx	268
Function RTFGetPartitionInfo	269
Function RTFSetVolumeLabel	270
Function RTFFormat	270
Miscellaneous File Functions	272
Function RTFCommitAll	272
Function RTFCloseAll	272
Function RTFShutDown	272
Function RTFErrorMessage	272
Function RTFSetDefaultOpenFlags	273
Function RTFSetCriticalErrorHandler	273
Function RTFDefaultCriticalErrorHandler	275
Function RTFCreateMasterBootRecord	275
Function RTFSplitPartition	275
Function RTFCreateBootSector	276
Raw I/O Functions	276
Function RTFRawMount	276
Function RTFRawSetMedia	276
Function RTFRawShutDown	276
Function RTFRawRead	276
Function RTFRawWrite	277
Function RTFRawMediaChanged	277
Function RTFRawDiscardSectors	277
Function RTFRawGetDiskGeometry	277
Function RTFRawLowLevelFormat	277
Functions for Debugging	278
Function RTFBufferInfo	278
Function RTFDumpFileTable	279

Device Dependent Functions	279
Function RTFFLPYTurnMotorOFF	279
Function RTFDrvFlashInfo	279
Function RTFDrvFlashCompact	280
Chapter 4 Alternate APIs for RTFiles-32	281
Win32 Emulation	281
RTTarget-32 Win32 Handles	281
RTTarget-32 Flag RT_CLOSE_FIND_HANDLES	281
ANSI C Run-Time System Functions	282
C++ iostreams	282
Mixing Different APIs	282
Chapter 5 Configuring RTFiles-32	283
RTFiles-32 Data Tables	283
The System Driver	283
Device List	284
Device Drivers	287
Floppy Disk Driver	287
DMA Buffer	287
RTFDevice.DeviceType	288
RTFDevice.DeviceNumber	288
RTFDevice.DeviceFlags	288
RTFDevice.Driver	289
RTFDevice.DriverData	289
IDE Hard Disk Driver	290
RTFDevice.DeviceType	290
RTFDevice.DeviceNumber	290
RTFDevice.DeviceFlags	290
RTFDevice.Driver	290
RTFDevice.DriverData	290
M-Systems DiskOnChip Driver	291
Memory Windows	291
RTFDevice.DeviceType	291
RTFDevice.DeviceNumber	292
RTFDevice.DeviceFlags	292
RTFDevice.Driver	292
RTFDevice.DriverData	292
PCMCIA SRAM Card Driver	292
RTFDevice.DeviceType	292
RTFDevice.DeviceNumber	292
RTFDevice.DeviceFlags	293
RTFDevice.Driver	293
RTFDevice.DriverData	293
RAM Disk Driver	293
RTFDevice.DeviceType	293
RTFDevice.DeviceNumber	293
RTFDevice.DeviceFlags	293
RTFDevice.Driver	293
RTFDevice.DriverData	293
Linear Flash Driver	294
RTFDevice.DeviceType	294
RTFDevice.DeviceNumber	294
RTFDevice.DeviceFlags	294
RTFDevice.Driver	295
RTFDevice.DriverData	295

MTD Drivers RTFMtdCFI2_8, RTFMtdCFI2_16, and RTFMtdCFI2_32	295
NULL Device Driver	296
RTFDevice.DeviceType	296
RTFDevice.DeviceNumber	296
RTFDevice.DeviceFlags	296
RTFDevice.Driver	296
RTFDevice.DriverData	296
Chapter 6 Demo Programs	297
Program HelloFiles	297
Program FAPIDemo	297
Program RTFCmd	297
Program RTFCmdMT	297
Program FlashDemo	297
Program DrvDemo	297
Chapter 7 Advanced Topics	298
Optimizing for Best Throughput	298
Optimizing for Best Data Security	299
Real-Time File I/O	299
Using RTFiles-32 with RTTarget-32	301
Win32 API Emulation with RTFiles-32 and RTTarget-32	301
Using RTFiles-32 with RTKernel-32	302
Implementing Custom Device Drivers	302
Structure RTFDriver	303
Driver Device Data Structure	304
Device-Specific Flags	304
Implementing Custom System Drivers	304
Flash Memory Technology Drivers (MTDs)	305
Appendix A RTFiles-32 Error Codes	307
Part IV RTPEG-32	310
Chapter 1 Overview	311
Windowing Interface Terminology	311
Window and Control	311
Parent, Child, Sibling	311
Base, Derived, Inherited	311
Modal Execution	312
Screen Coordinates	312
Palettes and Colors	312
Class PegScreen	312
Class PegMessageQueue	313
Messages	313
Message Flow and Routing	314
System Messages	314
User Defined Messages	314
Signals	315
Class PegPresentationManager	317
Event Driven Programming	317
Input Focus Tree	317
Keyboard Input Methods	318
Class PegThing	318
The Class Hierarchy	319
Unicode	320
Win32 Emulation Library	320

Chapter 2 Programming with RTPEG-32	321
Application Program Structure	321
Function PegInitialize	321
Function PegExecute	322
Rules Of Memory Ownership	322
Creating PegThings	322
Deleting/Removing PegThings	323
Obtaining a Pointer to PegPresentationManager	323
Finding an Object's Parent	323
Finding an Object's Children	323
System Status Flags	324
Style Flags	325
Determining the Position of an Object	325
Using Object Types	325
Using Object IDs	326
Messages	327
Overriding the Message() Method	328
Drawing to the Screen	328
Overriding the Draw() Method	329
Drawing to Memory	331
Using PegTimer	332
Viewports	334
Fonts	334
Default Fonts	334
The Vector Font	335
Scrolling	335
Chapter 3 Screen Drivers	337
Driver VESA_8	337
Driver VESA_16	337
Driver VESA_24	337
Driver VESA_32	338
Driver VGA_4	338
Driver VGASCRN	338
Chapter 4 Demo Programs	339
Demo Cross Reference	339
Program PegDemo	340
Program Robot	340
Program Table	340
Program TreeView	340
Program Notebook	340
Program Spread	340
Program Gauge	340
Program Dialog	340
Program Terminal	340
Program Unicode	340
Chapter 5 Utility Programs	342
Window Builder	342
Project Files	342
Output Files	342
Project Window	342
Project New/Open/Save/Close	343
Project Add Module	343
Project Add Image	343
Project Add Font	343

Project Update Source	343
Project Update Images	343
Project Update Strings	343
Project String Table	344
Configure Directories	344
Configure Target	344
Configure Default Fonts	344
Configure Languages	344
Configure Style	345
Configure Remote	345
Source Page	345
Images Page	346
Fonts Page	346
Composite Fonts and Reduced Fonts	346
Target Window	347
Target Window Status Line	348
Selecting Objects in the Target Window	348
Add Menu	348
Edit Copy	348
Edit Paste	348
Edit Properties	349
Layout Menu	349
View Test Mode	349
View Maximize	349
String Table	349
String Table Edit Fields	350
Merging String Tables	351
Source Code Generation	351
Child Object Pointer Control	351
Implicit Pointers	352
Temporary Pointers	352
Automatic Named Pointers	352
Member Pointers	352
Image Convert	353
Input File	353
Output File	353
Compression	354
Palette Options	354
Fixed Orthogonal	354
Generate Optimal	354
Floyd-Steinburg Dither	355
Save As	355
Transparency	357
Output Colors	357
Batch Conversion	357
Font Capture	358
Using Custom Fonts	359
Index	360