TNT Embedded Technologies Guidebook

Please Note:

Due to limitations of Adobe Acrobat, you will need to zoom in on all screen captures for better viewing. If you would like a printed version of Phar Lap's *TNT Technologies Guidebook*, please email: info@pharlap.com.

Phar Lap Software, Inc.
60 Aberdeen Ave.
Cambridge, MA, 02138

tech-support@pharlap.com
phone: (617) 661-1510
fax: (617) 876-2972
www: http:/ / www.pharlap.com

ETS-TG-5-0598

Fifth Edition: May 1998

Printing History:
Fourth Edition: March 1997
Third Edition: September 1996
Second Edition: January 1996
First Edition: September 1995

This manual is printed on recycled paper.

Contents

Preface vii

- An Industrial Strength RTOS vii
- Support for Off-the-Shelf Win32 Tools vii
- Modular Subsystems Provide Networking, MicroWeb Server Technology and More viii
- TNT Embedded ToolSuite Provides Development Tools viii
- All About x86 viii
- System Requirements ix
- Ordering Information x
- About Phar Lap x
- Format of *TNT Technologies Guidebook* xi
- Organization of the Manual xi
- For Further Information xii

Chapter 1 Introduction 1

- TNT Embedded ToolSuite Features 2
- The Realtime ETS Kernel 3
- Networking 4
- Embedded StudioExpress 5
- The Visual System Builder 5
- Developing Your Application 6

Chapter 2 A Quick Tour of the Realtime ETS Kernel 7

- Key Concepts of the Realtime ETS Kernel 8
- A Sample Realtime Program 12
- The ETS Project Wizard 14
- Creating a New ETS Workspace 15
- Building the Program 17
- Running with the Realtime ETS Kernel 21

The Typical Development Cycle 22
RAM-Based Applications 23
ROM-Based Applications 23

Chapter 3 A Quick Tour of the Visual System Builder and CFIGKERN 27

Preconfigured Templates 27
The Property Sheets 28
CFIGKERN 30

Chapter 4 Debugging with the Realtime ETS Kernel 33

A Sample Debug Session 34
Embedded StudioExpress Extensions 38
Target Port Input/ Output 38
Target System Information 39
Debugging Multithreaded Programs 40
Sample Output from EtsDumpThreads() 43
The Event Logging System 44

Chapter 5 Options for the Realtime ETS Kernel 47

Booting from Disk 50
Booting from ROM 52
BIOS Extension ROM Boot Method 53
Boot Jump Method 54
DOS Boot Option 54
Kernel Build Options 56
Replaceable Code 56
The Kernel Initialization Process 56

Chapter 6 Realtime ETS Kernel Programming Environment 61

The Protected-Mode Environment 62
C Run-Time Library Support 63
Host/Local File System 64
TCP/IP and WinSock 1.1 Networking 65
Floating-Point Emulation 65
Communications with the Development Host 66

Accessing Memory Mapped Devices 66
Host Command Line and Environment 66
Replaceable Code 67
Building and Using Dynamic Link Libraries (DLLs) 67
Interrupts and Exceptions 69
ETS Kernel Interrupt Processing 69
Installing an Interrupt Handler in Your Application 70
Structured Exception Handling 72
Realtime ETS Kernel Device Drivers 73
ETS PC Card Support Package 74
Priority Inversion Avoidance 75
Realtime ETS Kernel Version 9.1 Memory Requirements 76
Minimal C++ Run-Time Libraries 77

Chapter 7 Network Programming with ETS TCP/IP 79

Network Protocols 80
The ETS MicroWeb Server 81
MicroWeb Server Components 82

Appendix A The Realtime ETS Kernel API 85

Memory Management Routines 87
Threads and Synchronization Routines 87
File Management Routines 90
DLL Management Routines 91
Time Routines 92
TCP/IP Device Driver Configuration APIs 93
Event Logging Routines 94
Console Routines 95
Interrupt Control Routines 96
Process-Related Routines 97
Miscellaneous Routines 98
Windows Sockets APIs 99
C Run-Time Library Alternate Functions 101
HTTP Server APIs 102

FTP Server APIs 103
PC Card APIs 104
PCI Bus APIs 105
Porting Routines 106

Appendix B Supported C Run-Time Library Routines 109

Appendix C Realtime ETS Kernel Performance Measurements 113

C.1 Measuring Interrupt Latency 113
C.2 Measuring Interthread Yield Times 116

Appendix D Other Supported Compilers 119

Index Error! Bookmark not defined.

Preface

This manual, *TNT Embedded Technologies Guidebook*, provides an overview of Phar Lap's TNT Embedded ToolSuite, an embedded development toolkit that features a Windows-friendly realtime operating system (RTOS). Phar Lap's RTOS, the Realtime ETS Kernel, combines the benefits of Windows development tools with hard realtime performance. While this manual discusses the entire ToolSuite, emphasis is on its core component, the Realtime ETS Kernel.

AN INDUSTRIAL STRENGTH RTOS

The Realtime ETS Kernel was built from the ground up to meet the requirements of industrial-strength embedded applications. Based on a Phar Lap-defined subset of the Win32 API, the Realtime ETS Kernel provides unequalled hard realtime functionality in a Windows-friendly package, including:

- A deterministic scheduler with 31 priority levels
- A threads-based multitasker
- Priority-based scheduling

SUPPORT FOR OFF-THE-SHELF WIN32 TOOLS

The Realtime ETS Kernel's support for Microsoft Visual C++ 5.0 and the Developer Studio Integrated Development Environment (IDE) allows you to use the industry's leading compiler/ IDE combination to create embedded and realtime systems. Phar Lap's Embedded StudioExpress is an add-in that seamlessly integrates the Realtime ETS Kernel with Developer Studio. It lets you take advantage of the Developer Studio IDE, so you can do all of your development in a single environment. It also enables you to debug your embedded application using the Developer Studio's integrated debugger.

MODULAR SUBSYSTEMS PROVIDE NETWORKING, MICROWEB SERVER TECHNOLOGY AND MORE

The Realtime ETS Kernel is scalable. You can select only the components you need and incorporate them into your embedded system. The kernel's subsystems include:

The ETS TCP/IP network stack — which includes support for standard network and Internet protocols, including SLIP, PPP, and FTP

- The ETS MicroWeb Server— which includes an embedded HTTP Server and HTML-On-The-Fly libraries
- Flash File System Support
- A DLL Loader

TNT EMBEDDED TOOLSUITE PROVIDES DEVELOPMENT TOOLS

TNT Embedded ToolSuite was designed to provide all the software development tools you need to build embedded and realtime systems. In addition to the Realtime ETS Kernel, the ToolSuite includes:

- LinkLoc — a 32-bit linker/locator

  Embedded StudioExpress
- ETS Visual System Builder — a point-and-click Windows utility for easily configuring the Realtime ETS Kernel for custom hardware systems
- 386|ASM — a 32-bit assembler

ALL ABOUT X86

As the company that pioneered 32-bit x86 software development tools, Phar Lap understands the requirements of x86 processors. The Realtime ETS Kernel supports all 32-bit x86 hardware, from the Intel 386EX to the NS486 and Intel Pentium family.

SYSTEM REQUIREMENTS

In order to use TNT Embedded ToolSuite for embedded development, you will need two 32-bit x86-compatible systems:

Host System:

486 or Pentium personal computer with:

- 32MB memory
- 20MB free disk space
- Serial or parallel port
- Windows NT or Windows 95

One of the following compilers:

- Microsoft Visual C++, Version 4.x or later
- Borland C++, 32-bit compiler, Version 4.5-5.x
- Aonix ObjectAda Real-Time for Intel/ETS

Optional:

In-circuit emulator

Target System:

Embedded system with:

- 386, 386EX, 486, NS486, Pentium, Pentium Pro, MMX, Pentium II or compatible processor
- Sufficient memory for the ETS Monitor and your application (RAM with ROM or Flash memory)
- 100K bytes minimum required memory
- Serial or parallel port for debugging between target and host.

Optional Target Hardware:

- Keyboard
- Video Monitor
- IDE Controller
- Floppy Disk Controller
- Floating-Point Coprocessor
- Network Interface Card
- PC Card Controller
- Flash Disk

ORDERING INFORMATION

The TNT Embedded ToolSuite, Realtime Edition Software Development Kit (SDK) sells on a per-seat basis. Each programmer developing code using any component of the ToolSuite will need a complete development system.

Run-time licenses are required for distribution of the Realtime ETS Kernel and each of its subsystems.

ABOUT PHAR LAP

Phar Lap Software provides software tools that make developing embedded and hard realtime systems as easy as developing Windows applications. Founded in 1986, Phar Lap pioneered 32-bit x86 development tools, and continues to lead in embedded development through advanced realtime capabilities, seamless integration with Windows-compatible development tools, and innovations such as Web-accessibility using standard Internet technology.

More than 50,000 programmers worldwide have used Phar Lap tools to build and deliver more than 3,000 applications and embedded system designs.

Phar Lap is also at the forefront of the Embedded Internet technology wave. Phar Lap's "World's Smallest Web Server" (http:/ / smallest.pharlap.com) demonstrates a realtime weather station application developed with the Realtime ETS Kernel. "Smallest"

displays live weather data from Phar Lap's headquarters in Cambridge, Massachusetts. The company's MicroWeb Server technology lets developers create Web-accessible embedded systems, such as security and office equipment systems, "smart home" applications, and manufacturing and process control devices.

FORMAT OF *TNT TECHNOLOGIES GUIDEBOOK*

TNT Embedded Technologies Guidebook includes examples using the Realtime ETS Kernel which have been excerpted from the product documentation. They are presented here in a format useful to readers who may be evaluating the product. *TNT Embedded Technologies Guidebook* is not a substitute for product documentation.

ORGANIZATION OF THE MANUAL

The following is a quick summary of the chapters and appendices in this book:

Chapter 1, Introduction
An overview of programming for the Realtime ETS Kernel

Chapter 2, A Quick Tour of the Realtime ETS Kernel
Building a simple realtime embedded program with each supported compiler and running it on your embedded system

Chapter 3, A Quick Tour of the Visual System Builder and CFIGKERN
How to use the Phar Lap Visual System Builder to manage your embedded application

Chapter 4, Debugging with the Realtime ETS Kernel
How to use CodeView or Turbo Debugger on the host, along with the new event logging system, to debug an embedded program on the target

Chapter 5, Options for the Realtime ETS Kernel
A description of the configurable features of the Realtime ETS Kernel

Chapter 6, Realtime ETS Kernel Programming Environment
Features of the Realtime ETS Kernel used in application programs

Chapter 7, Network Programming with ETS TCP/ IP
Overview of networking capabilities provided with ETS TCP/ IP

Appendix A, The Realtime ETS Kernel API
Functions in the Realtime ETS Kernel Libraries

Appendix B, Supported C Run-Time Library Routines
Table identifying C run-time routines implemented by the Realtime ETS Kernel

Appendix C, Realtime ETS Kernel Performance Measurements
An illustrated discussion of interrupt latency and other factors influencing performance of realtime systems

Appendix D, Other Supported Compilers and Languages
Descriptions of compilers and languages other than Microsoft Visual C++/ Developer Studio that can be used to build programs targeted to the Realtime ETS Kernel.

FOR FURTHER INFORMATION

If you have questions about the product functionality and capabilities of TNT Embedded ToolSuite and the Realtime ETS Kernel, please contact your Phar Lap sales representative for further information. You may reach Phar Lap sales between the hours of 8:30 a.m. and 5:30 p.m. Eastern Standard Time, Monday-Friday at:

Telephone: (617) 661-1510 Fax: (617) 876-2972

Email: sales@pharlap.com www.pharlap.com

Chapter 1

Introduction

Welcome to Phar Lap's TNT Embedded ToolSuite (ETS), a total solution toolkit for embedded systems development on the 32-bit x86 family of processors! The ToolSuite includes the ETS Kernel, Phar Lap's realtime operating system (RTOS).

The native PC marketplace has long since standardized on 32-bit x86-compatible architecture. This has resulted in several factors that make this platform an obvious and attractive solution for embedded systems developers. For instance:

- The price/performance advantage of the 32-bit x86 family
- The availability of inexpensive, industry-standard compilers and other mature third-party development tools that run on Windows
- Standard APIs such as Win32, which include functions for multithreading, multitasking, and deterministic priorities, and WinSock, which provides an interface to a standard TCP/IP protocol stack
- The proliferation of low-cost 32-bit x86 boards designed specifically for embedded systems

With expertise in 32-bit protected-mode and embedded development tools, Phar Lap Software created TNT Embedded ToolSuite to help embedded systems programmers capitalize on the many advantages of the 32-bit x86-compatible architecture.

In the past, embedded systems developers have had to obtain development tools from many vendors, selecting a linker from one company, a debugger from another. The result was usually a hit-and-miss solution with a high price tag. Now with TNT Embedded ToolSuite, developers have a full-featured single-source solution at a reasonable price. TNT Embedded ToolSuite is unique in providing all the components you need to build multithreaded realtime embedded applications for 32-bit x86 processors. The rich feature set includes robust networking capabilities, priority inversion avoidance, a powerful event logging system, and booting from a disk (floppy, IDE hard, PC Card ATA, or M-Systems flash), from ROM, or from DOS.

Ultimately, building your embedded application with the ToolSuite simply costs you less than building the same application with tools from multiple vendors. The ToolSuite itself is priced lower than its competition, and the Windows compilers supported by the ToolSuite are much less expensive than specialty compilers for embedded systems development.

The ability to use standard Windows tools for developing embedded software is one of the most significant benefits of using the ToolSuite. For example, you can use the familiar 32-bit C and C++ compilers to build embedded applications. And for the first time, you can debug your embedded applications using Microsoft's Developer Studio debugger. In fact, the ETS Project Wizard lets you use the full capabilities of the Developer Studio IDE, including automatic compilation of modified source files, to build ETS applications. When you can buy a high-quality 32-bit compiler from Microsoft for $300, you have an excellent reason to reevaluate the $3000 you might spend on a specialty embedded systems compiler to produce the same program.

TNT EMBEDDED TOOLSUITE FEATURES

TNT Embedded ToolSuite includes the following major components:

- The Realtime ETS Kernel, an RTOS that implements the C/C++ runtime libraries and a subset of the Win32 APIs on your embedded target
- Embedded StudioExpress, CVEMB, TDEMB, and SBEMB shells for embedded cross-debugging
- The ETS Project Wizard, a tool for creating Developer Studio workspaces targeted to the Realtime ETS Kernel
- LinkLoc, a 32-bit linker/locator
- The Visual System Builder, a Windows program for configuring the ETS Kernel and your application
- Full support for C/C++ run-time libraries
- The ability to boot from a disk (floppy, IDE hard, PC Card ATA, or M-System flash), from ROM, or from DOS
- Comprehensive user documentation

Phar Lap's most powerful product is TNT Embedded ToolSuite, Realtime Edition, which includes all the functionality of the Standard Edition plus the following:

- Support for deterministic multithreaded embedded applications, including reliable priority inversion avoidance
- Support for priority scheduling
- Support for round-robin scheduling with variable timeslices
- Robust networking capabilities including a built-in TCP/IP stack and support for Ethernet and WinSock 1.1, including multi-homed Ethernet and serial connections
- Support for Dynamic Link Libraries (DLLs)
- An MS-DOS compatible file system with support for FAT12, FAT16, and FAT32 formats as well as a wide variety of disk types including IDE (both CHS and LBA formats), floppy, PC Card ATA (both flash and rotating media), M-Systems Flash, and RAM disk.
- A floating-point emulation library
- Support for PC Card ATA disks, Ethernet adapters, serial ports, and modems

For more information about the features of the Realtime and Standard Editions, please contact your Phar Lap sales representative.

THE REALTIME ETS KERNEL

The Realtime ETS Kernel is physically divided into two separate pieces: the ETS Monitor and the ETS Libraries, which are linked with your application.

The ETS Libraries provide the Win32 APIs that support the C/ C++ run-time library, and also contain optional components of the Realtime ETS Kernel.

The ETS Monitor handles communications with the host debugger or program launcher, program downloading or loading off disk, and system hardware initialization and switching the processor to protected mode. The ETS Monitor can be booted from disk on a PC/ AT-compatible target, or it can be linked into a ìcontainerî file along with your application for programming a ROM or downloading to a ROM emulator.

The Realtime ETS Kernel is a simple, compact RTOS (as small as 75 Kb) for running embedded programs. The two main functions of the kernel are to initialize 32-bit protected mode and to provide the foundation for a C/ C++ run-time library, making it almost as easy to develop programs for embedded systems as it is to develop console applications for Windows. For a complete discussion, see Chapter 4.

The kernel also includes an optional host communications module that allows your embedded system to communicate with a host PC running Windows. You can use familiar Windows tools to develop your embedded programs, then download the compiled and linked program via a cable to the target system. The Realtime ETS Kernel running on the target loads your program and, while you're debugging, can synchronize with the Developer Studio debugger running on the host.

Before your program is loaded, the kernel performs the low-level details of setting up a protected-mode environment. Once your program is running, the ETS Libraries (which are linked with your application) provide the functionality of most of the C run-time library. The only unsupported functions are those that assume the existence of resources that may not be available on the embedded system.

Optional kernel components include the structured exception handling library, timer, keyboard, and screen drivers, host communications software and a disk loader (loads an embedded application from the same boot disk as the kernel).

The system requirements for using TNT Embedded ToolSuite are listed in the Preface. The default hardware configuration for the embedded system is PC/ AT. Many configurations are supported, as listed in Chapter 2. For a complete list of currently supported configurations, please call your Phar Lap Software sales representative.

To accommodate the wide range of target systems available in today's market, all the hardware-specific code in the Realtime ETS Kernel is contained in "replaceable" modules for which source code is provided.

NETWORKING

The Realtime ETS Kernel provides robust networking capabilities through the popular TCP/ IP and WinSock 1.1 protocol specifications. A properly equipped embedded system built with TNT Embedded ToolSuite can thus run network applications over the Internet, or even act as a network server. (Visit http:/ / smallest.pharlap.com to see an

interesting implementation of Phar Lap's realtime embedded HTTP server.) All the standard application-level protocols are supported, including FTP, SMTP, and HTTP. Network connections are supported for Ethernet, SLIP, and PPP. See Chapter 7, "Network Programming with ETS TCP/ IP," for a complete discussion.

EMBEDDED STUDIOEXPRESS

Embedded StudioExpress is a Developer Studio add-in that lets you debug your ETS applications using the Developer Studio's integrated development environment. Additionally, Embedded StudioExpress extends Developer Studio by providing special integrated functions for manipulating the remote target system. There are three parts to Embedded StudioExpress:

- The host communications package that works behind the scenes to turn Developer Studio into a powerful debugging environment so you can debug your embedded program using this same source-level debugger that you use for native Windows programs.
- The ETS Project Wizard that lets you create Developer Studio workspaces targeted to the Realtime ETS Kernel so you can use the full capabilities of the Developer Studio IDE for building ETS applications. C and C++ compilation options can be set from within the IDE, and Phar Lap's LinkLoc linker is run directly from the Build menu, using options set from the Linker Settings dialog. Chapter 2 shows the use of the ETS Project Wizard.
- The StudioExpress Toolbar features that allow you to display a comprehensive view of the embedded target. These features are illustrated in Chapter 4.

THE VISUAL SYSTEM BUILDER

Optional kernel components are easily selected with the Visual System Builder, a Windows program for managing and configuring your embedded application environment. Besides selecting the components to be included in your custom kernel, the Visual System Builder also makes it easy to replace target-specific kernel modules with your own code.

You do not have to use the Visual System Builder. If you prefer, you can create and edit your own linker command files. However, we think

that if you try the Visual System Builder, you'll want to continue using it. Please refer to Chapter 3 for more details.

DEVELOPING YOUR APPLICATION

In a perfect world, you would be able to do all the development of your embedded program using the final, stable hardware that will be used for production runs of your application. This, however, is rarely the case. In fact, we recommend that your initial ETS development environment consist of two PC-compatible computer systems (one is the host, the other is the target), with the COM1 serial ports of each machine connected using the standard LapLink serial cable shipped with TNT Embedded ToolSuite. This way, you're using the default kernel, libraries, and drivers, and can focus on developing your program.

Of course, eventually you will replace the PC/ AT target with your custom hardware, but at that point the hardware-independent portion of your application will be working and you can concentrate your efforts on debugging your custom drivers and replacement modules.

Clearly, TNT Embedded ToolSuite provides a powerful, versatile environment for developing your embedded application. The remaining chapters in this book provide a more detailed description of additional features of the TNT Embedded ToolSuite and its RTOS, the Realtime ETS Kernel.

Chapter 2

A Quick Tour of the Realtime ETS Kernel

This chapter uses a simple embedded program to illustrate the basics of embedded system development with TNT Embedded ToolSuite. Embedded software development attains a new level of ease and sophistication with the Realtime ETS Kernel. In fact, it's almost as simple to develop programs for your embedded system as it is to develop Windows console applications.

Designed to work with popular 32-bit C/ C++ compilers and most of the standard C run-time library, the Realtime ETS Kernel is an RTOS that creates a protected-mode environment for your program on the embedded system.

Using Visual C++ along with the Realtime ETS Kernel brings 32-bit power and the ease of C-language programming to embedded systems. Appendix D describes support for compilers and languages other than Microsoft Visual C++/ Developer Studio.

The Realtime ETS Kernel bootstraps the embedded system and establishes a protected-mode environment for your program. When your program gets control at the beginning of main(), the Realtime ETS Kernel has already performed the low-level details of setting up a protected-mode environment for your embedded program:

- The embedded processor is in protected mode.
- Selectors are set up for the code and data segments.
- Registers and flags are initialized.
- Global variables are automatically initialized from compressed data in ROM.
- The C run-time library is initialized.

Thus, the programming environment for your embedded system is similar to that for other 32-bit protected-mode systems. The Realtime ETS Kernel performs all system-level initializations as part of the

bootstrap process, leaving only application-specific initializations for your program. As detailed in Appendix B, most of the C run-time library is available to your program.

The Realtime ETS Kernel can be run in either WaitHost or NoWaitHost mode. The difference between these modes is that in WaitHost mode, the kernel boots the target system, initializes itself, and then waits for the host computer to instruct it to start executing the embedded program. In NoWaitHost mode, the kernel boots the target system, initializes itself, and then begins executing the embedded program (loaded either in ROM or from disk). In general, WaitHost mode is used during program development and NoWaitHost mode is used for production runs.

The examples in this chapter use the Realtime ETS Kernel in WaitHost mode.

KEY CONCEPTS OF THE REALTIME ETS KERNEL

The paragraphs below provide a quick summary of the main points of programming with the Realtime ETS Kernel.

- **Supported Win32 APIs**
  The Realtime ETS Kernel supports a Phar Lap-defined subset of the Win32 API with functions for threads, events, mutexes, semaphores, critical sections, and thread-local storage. In addition, we have added some functions to the Realtime ETS Kernel API to allow you to specify the granularity of the timer tick and time slices. Please see Appendix A for a complete list of supported Win32 and ETS API functions.

- **Single Process, Multiple Threads**
  The Win32 API supports multi-process as well as multithreaded applications. The Realtime ETS Kernel, however, supports only multithreaded programs. There is a single process within an ETS application, and all ETS Realtime Kernel threads share the same 32-bit flat address space.

® **The Realtime ETS Kernel Scheduler**
There is one significant difference between the implementation of threads in the Realtime ETS Kernel and in Windows NT/ 95. In the Realtime ETS Kernel, the scheduling of tasks is deterministic, whereas in Windows NT/ 95 it is not. To put this another way, under ETS,

The highest priority runnable thread will run.

This thread will run until it blocks, a higher priority thread becomes runnable, or, if time slicing is enabled, the time slice expires and there are other runnable threads of equal priority.

A thread is runnable so long as it is not waiting on a synchronization object (event, semaphore, mutex, or critical section) or waiting in the Sleep() system call. Each thread has a priority which can range from -15 (lowest) to 15 (highest) with a default of 0. The function SetThreadPriority() is used to change the priority.

The ETS multitasking library includes a scheduler that determines which thread will run. From the scheduler's point of view, each thread has a state which is either runnable or waiting. When anything happens to change the state of *any* thread, the scheduler looks at *all* the runnable threads to determine which one has the highest priority so that the highest priority runnable thread will run.

Sometimes a thread needs an object owned by a lower-priority, unrunnable thread. This scheduling anomaly, known as **priority inversion**, can lead to unexpected, non-deterministic behavior in your program. **The multitasking scheduler in the Realtime ETS Kernel prevents the ill effects of priority inversion.** It automatically watches for priority inversions and carefully manipulates thread priorities to allow the higher-priority thread to run. See Chapter 6 for more information.

® **Time Slicing**
The frequency with which the Realtime ETS Kernel switches among same-priority threads is controlled by the length of the time slice. The default time slice is 10 milliseconds. Time slicing can be disabled by setting a time slice length of zero. The ETS Libraries include functions for adjusting the length of the time slice from your embedded application.

Although not required, the time slice length should be an integer multiple of the timer tick period to obtain accurate time slices. The timer tick period is the rate of the hardware clock. The default timer tick period is 10 milliseconds. The programmer has the option of setting the timer tick period to the granularity required by the application. The smaller the period, the more precisely you can schedule time-outs in the Sleep() and Wait() system calls.

® **Choosing the Timer Tick Period**

The limits to the timer tick period are hardware-dependent. For PC/ AT machines, the period is usually between 1 and 55 milliseconds. There are functions in the ETS API for specifying sub-millisecond timer periods. There is, however, a cost associated with very short periods. On some lower performing systems, you could end up spending a significant amount of processor time servicing the timer tick interrupt.

We chose the default value of 10 milliseconds as one that gave good performance on lower-end systems. On a 20 MHz 386SX machine with a timer tick period of 10 milliseconds, about 5% of the CPU time is spent servicing the timer tick interrupt. The percentage is correspondingly less on machines with faster CPUs. On fast enough machines, you can significantly improve the resolution of the timer by making the period of the timer tick smaller without affecting the performance of your application.

® **An Example of Thread Scheduling**

In the simplest case, there is no time slicing. The highest priority runnable thread runs until:

- A higher priority thread becomes runnable.
- The thread calls Sleep().
- The thread finishes running (calls _endthread()).
- The thread has to wait for a synchronization object.
- The thread makes a system call, many of which are implemented using synchronization objects. For example, critical sections are used in the C run-time library print routines so that two threads are not simultaneously accessing the same file.

The situation gets a bit more complicated when time slicing is enabled. The thread scheduler works the same, but with some additional considerations:

- The highest priority runnable thread is still runnable when the time slice ends. If there are other runnable threads with the same priority, the one that has been waiting to run for the longest time will be scheduled. If this is the only runnable thread with this priority, it will continue to run.
- At the beginning of the time slice, Thread1 is the thread running. During the time slice, Thread2 of higher priority becomes runnable, so the scheduler starts Thread2. If Thread2 completes before the end of the time slice, the scheduler runs Thread1 for the remainder of the slice.

® **Scheduler Callback Functions**
Callback functions are called whenever the Realtime ETS Kernel creates a thread, terminates a thread, or switches context from one thread to another. There are default functions within the Realtime ETS Kernel that perform the necessary actions, but you can use the ETS Library function EtsRegisterCallback() to cause your own function(s) to be called as well.

® **Using Threads to Service Hardware Interrupts**
You may find it convenient to create threads to handle the interrupts. The ISR for the interrupt simply signals an event and returns. The actual interrupt processing is done in a thread which is waiting for the event, mutex, or semaphore. The following Win32 APIs may be called from an ISR:

- GetCurrentThreadID()
- GetTickCount()
- PulseEvent()
- QueryPerformanceCounter()
- ReleaseSemaphore()
- ResetEvent()
- ResumeThread()
- SetEvent()

The significance of this approach is that the priority of the thread is **not** the same as the priority of the hardware interrupt. This allows you to quickly dispatch hardware interrupts while high-priority threads are running and then service them at a priority appropriate to your application.

A SAMPLE REALTIME PROGRAM

Let's look at an example. Many manuals begin with the familiar "hello world" sample program, but we need to modify the program for realtime. This version of HELLO.C isn't a particularly realistic example of a realtime program, but neither is "hello world" a realistic example of C programs. It is, however, a a good way to illustrate building and running realtime programs with TNT Embedded ToolSuite.

HELLO.C has a simple main() program that starts up *num_threads* (which is initialized to 5) different threads, each one executing the OneHello() procedure. OneHello() is essentially the traditional "hello world" program in a loop. There are, however, some additions for realtime. Let's look at the code (there isn't much of it) and then explain the realtime features.

```
#include <windows.h>
#include <stdio.h>
#include <process.h>

#define STACK_SIZE (16 * 1024)

void OneHello(void *id);

volatile DWORD alive;
int num_threads = 5;

void main()
{
    int i;
    int id = 0;
    // start threads
    for (i = 0 ; i < num_threads ; i++)
    {
        if (_beginthread(OneHello, STACK_SIZE, (void *) id)
                                                    != (ULONG)-1)
        {
            ++id;
            InterlockedIncrement((LPLONG)&alive);
        }
    }
```

```
    while (alive)
        Sleep(50);
    printf("main thread terminates\n");
    exit(0);
}

void OneHello(void *id)
{
    int i;

    for(i = 0; i < 5; ++i)
        printf("Hello From Thread %c\n", 'A' + (int)id);
    InterlockedDecrement((LPLONG)&alive);
    _endthread();
}
```

Looking at main(), we see that it simply calls _beginthread() in a loop to start the multiple threads of OneHello(). Although the Win32 API supported by the Realtime ETS Kernel provides a complete set of thread functions (for example, CreateThread() and ExitThread()), you should always use the _beginthread() and _endthread() functions provided by the C run-time library. This is required if the thread calls any other C run-time library function, and is good practice in all cases.

After starting up the threads, main() waits for all the other threads to finish executing before calling exit() to terminate itself. This is very important: the last thread to finish executing **must** call exit() so the C run-time library performs an orderly, normal shutdown. Calling _endthread() or ExitThread() will not cause this orderly shutdown and information could be lost; for example, open files are not flushed to disk correctly unless exit() is called.

Let's look at the definition of _beginthread():

```
unsigned long _beginthread(
    void( *start_address )( void * ),
    unsigned stack_size,
    void * arglist );
```

We see that the first argument is the address of the function that starts the thread, which is OneHello() for this example. Although in this example, OneHello() simply executes a few instructions and then returns, in a more realistic example this function would be more like the typical main() function in a traditional C program.

The next argument is the size, in bytes, of the stack to be allocated by _beginthread(). For this example, we're using a 16KB stack. You

should specify the appropriate stack size for the thread, remembering that a stack of this size is allocated for each thread created by this call. If the stack is too big, you can be wasting memory. If it is too small, the stack may overflow, which can be very hard to debug. The thread stack is allocated from the heap, making it difficult to determine in advance what will be overwritten if the stack overflows.

The final parameter is a pointer to the arguments to be passed to the newly created thread. In this example, the argument is the variable *alive*, which is a count of the number of threads. Note that *alive* has been declared with the volatile attribute. This indicates to the compiler that *alive* is going to be accessed by multiple threads simultaneously, and thus should not be stored in a CPU register. Just before terminating, each thread decrements the value of *alive* and main() monitors this variable. When it returns to 0, main() knows that all the threads have finished executing and it safe for main() to call exit().

It is also important to note that the value of *alive* is changed by calling the Win32 functions InterlockedIncrement() and InterlockedDecrement(). Use of these functions prevents more than one thread from simultaneously trying to change the value of *alive*.

The code for OneHello() looks just like any other C code. There is a loop in which printf() is called to display a message. The formal parameter *id* is used to identify the thread. Rather than returning when it is finished processing, OneHello() decrements the value of *alive* and calls _endthread().

Now that we've seen what a realtime program looks like, we need to know how to build.

THE ETS PROJECT WIZARD

The ETS Project Wizard creates a new Developer Studio workspace targeted to the ETS Kernel. This means that you can use the full capabilities of the Developer Studio IDE, including automatic compilation of modified source files, for building ETS applications. C and C++ compilation options can be set from within the IDE, and Phar Lap's LinkLoc linker is run directly from the Build menu, using options set from the Linker Settings dialog.

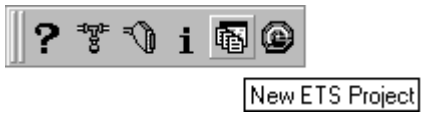
An ETS workspace under Developer Studio is essentially a Win32 Console Application project that has a / ETS switch added to its Link command line. When Developer Studio attempts to link an application,

Embedded StudioExpress intercepts the operation and looks for the / ETS:FILENAME switch on the Link command line. If it does not find this switch, it just passes the command line on to Microsoft Link to allow the application to be linked as normal. If the / ETS switch is on the command line, then Embedded StudioExpress translates the Developer Studio linker switches to LinkLoc format and invokes LinkLoc using the filename included in the / ETS switch as a linker command file. In addition, the complete list of object file names is passed to LinkLoc with no changes.

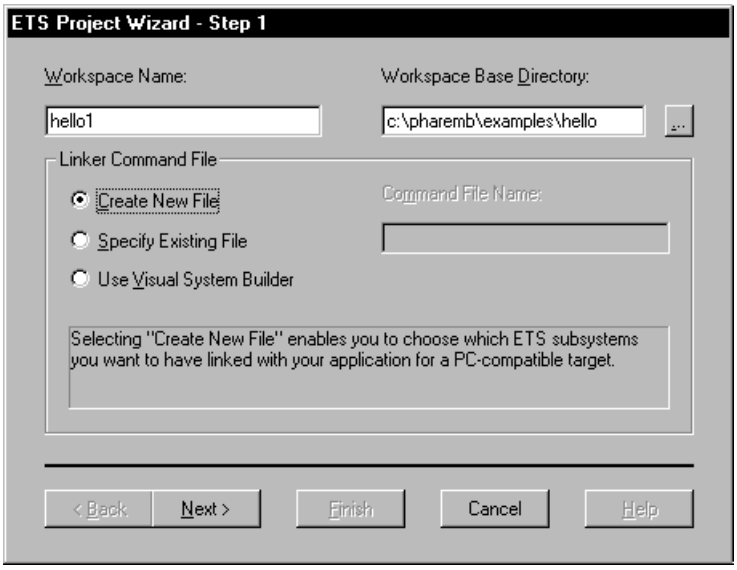
A linker command file is simply a text file that contains the switches and options that would be specified on the LinkLoc command line. The ETS Project Wizard uses a linker command file to specify the LinkLoc switches required for your program. One of the ETS Project Wizard dialogs lets you choose which ETS subsystems you want to have linked with your application.

CREATING A NEW ETS WORKSPACE

After starting Developer Studio, select New ETS Project from the Embedded StudioExpress toolbar:



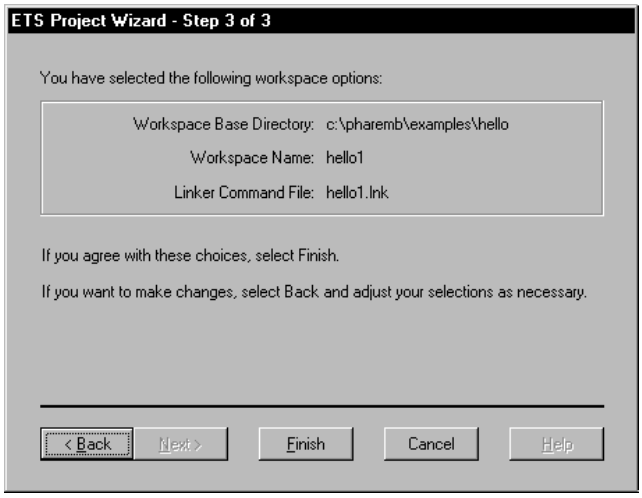
There are two general areas on the first ETS Project Wizard screen: one for specifying the name and location of the workspace and one for specifying the linker command file.



If you select "Create New File," the Subsystem Selection dialog lets you select the ETS components that will be linked with your application. Let's take a look at this dialog to see the ETS Kernel components that can be included with your application:

For this sample program, we need to select the multithreaded library as well as the PC-Compatible Screen Driver. Selecting this driver allows us to display console output on the target screen.

Before the ETS Project Wizard actually creates the workspace and linker command file, it shows you the options you have selected and gives you a chance to make changes, if necessary.



When the ETS Project Wizard creates the HELLO1.LNK linker command file, the file is included in your project and you can later edit it to make changes if you find that you want to include a different set of subsystems.

When you select Finish, the ETS Project Wizard closes the currently open Developer Studio workspace (if there is one). You may be prompted by Developer Studio to save your work. The ETS Project Wizard then creates the newly specified workspace and tells Developer Studio to open it.

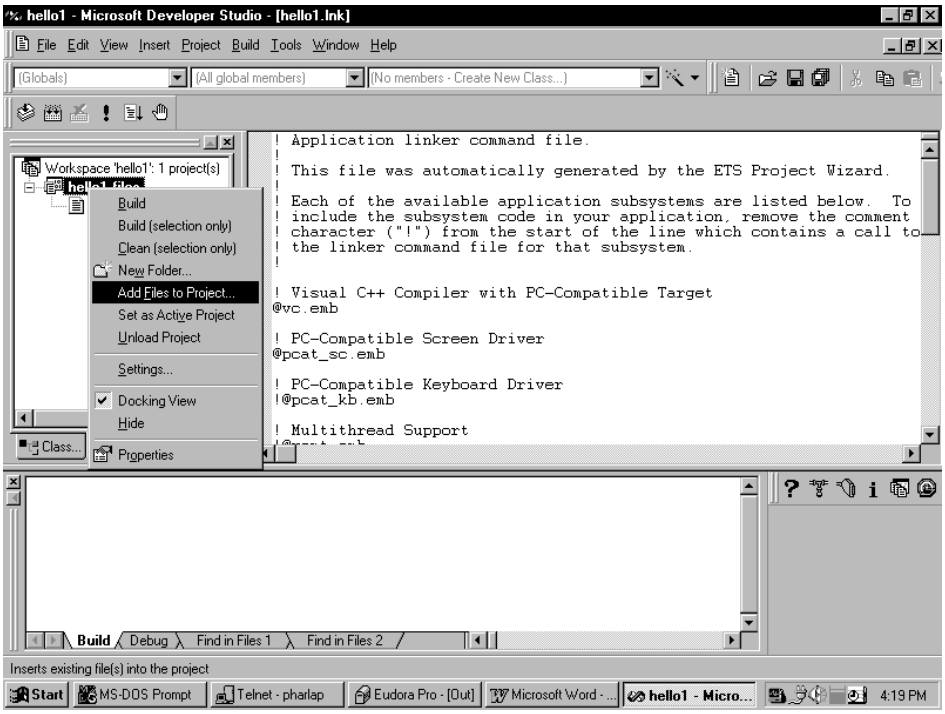
After using the ETS Project Wizard to create the workspace and project, you're ready to continue with building your program.

BUILDING THE PROGRAM

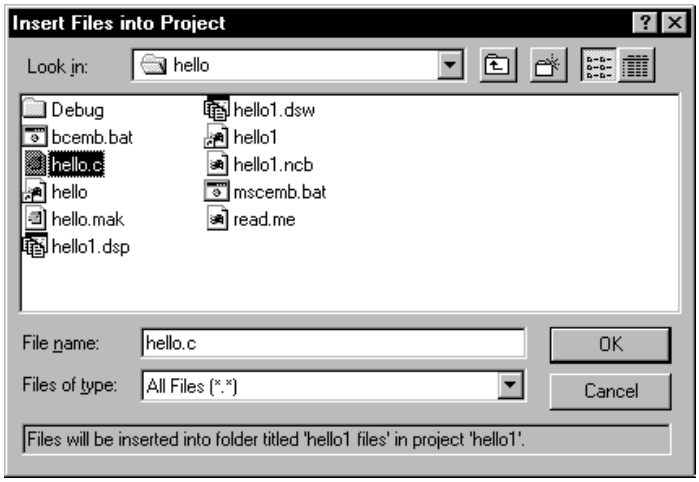
Using Developer Studio to build programs targeted to the ETS Kernel is the same as using Developer Studio to build programs targeted to Windows. You add your files to the project, creating them if they don't already exist. You use the standard Developer Studio features to specify whether you're building a debug or release version of your program. Then you use the tools on the Developer Studio Build menu to build and run the program.

The first thing we're going to do is add the HELLO.C file to the project so Developer Studio knows to include it in the compilation and linking commands. To do this, select Workspace from the View menu and then click on the File View tab in the Workspace window.

Right-click on "hello1" and then select Add Files to Project:

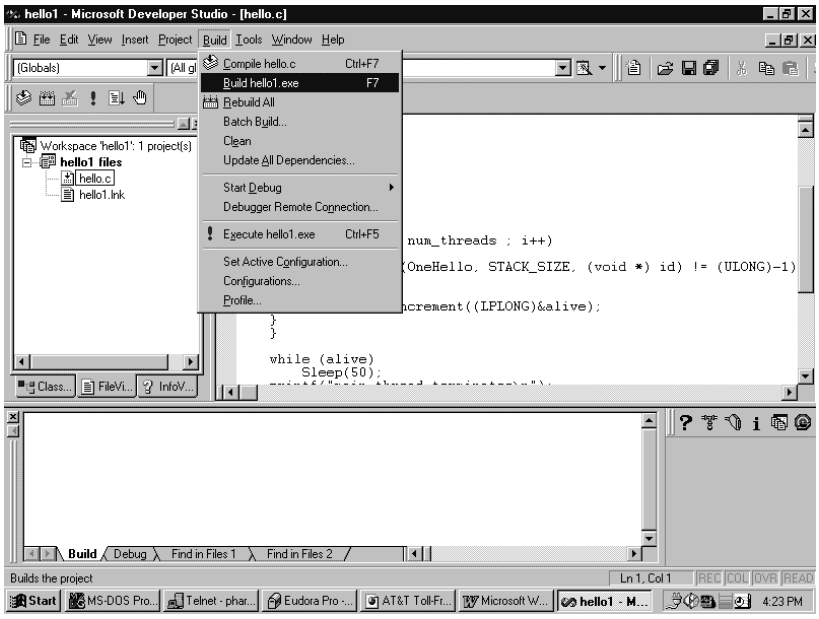


Select the files you want to add to the project. For this project, the only file we need to add is HELLO.C. Choose All Files (*.*) in the Files of type box to see more than just HELLO.C:



To edit a project file, double click on its name in the Workspace window.

To build a debuggable version of HELLO1.EXE, choose Build hello1.exe from the Developer Studio Build menu:



The output from the build process appears in the Build window at the bottom of the screen:

```
-------------Configuration: hello1 - Win32 Debug-------------
Compiling...
hello.c
Linking...
LinkLoc: 9.1 -- Copyright (C) 1986-98 Phar Lap Software, Inc.
Microsoft (R) Debugging Information Compactor  Version
5.00.7022
Copyright (C) Microsoft Corp 1987-1997. All rights reserved.

hello1.exe - 0 error(s), 0 warning(s)
```

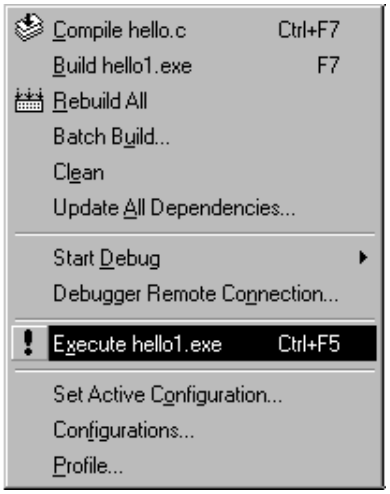
The resulting HELLO1.EXE is ready to run or debug.

RUNNING WITH THE REALTIME ETS KERNEL

When an ETS application is linked, LinkLoc marks its file header as an ETS application. Embedded StudioExpress automatically notices these specially marked applications and downloads them to the embedded target. It ignores standard Win32 applications which Developer Studio loads normally on the development host.

Because Embedded StudioExpress ignores standard Win32 applications, there is no need to remove it when you aren't building ETS applications. It is there to use when you need it, but it doesn't prevent you from using Developer Studio for non-ETS programs.

Once you've built your program, you can run it by choosing Execute hello1.exe from the Build menu.



StudioExpress will download your program to the embedded target. A console window opens on the host screen before the download starts. This console window will be used to perform keyboard input for the application running on the embedded target.

After the download is complete, the following text appears on the target screen:

```
Hello From Thread A
Hello From Thread A
Hello From Thread A
Hello From Thread A
Hello From Thread A
Hello From Thread B
Hello From Thread B
```

```
Hello From Thread B
Hello From Thread B
Hello From Thread C
Hello From Thread C
Hello From Thread C
Hello From Thread C
Hello From Thread C
Hello From Thread D
Hello From Thread E
Hello From Thread D
Hello From Thread E
Hello From Thread D
Hello From Thread E
Hello From Thread D
Hello From Thread E
Hello From Thread D
Hello From Thread E
main thread terminates
```

The host console window will display the text "Press a key to exit." As directed, press any key on the host keyboard to exit the program.

THE TYPICAL DEVELOPMENT CYCLE

The HELLO example gives you the flavor of building and running programs for the Realtime ETS Kernel. Running from Developer Studio is easy and simplifies the example, but does not utilize the full power of TNT Embedded ToolSuite. While the Realtime ETS Kernel may be the heart of ETS's support for programs written in C or C++, Embedded StudioExpress is the soul of its convenience for programmers. When run in WaitHost mode from Developer Studio, the Realtime ETS Kernel is part of a powerful debugging environment that facilitates debugging embedded programs using the same source-level debuggers used for non-embedded programs. Chapter 4, "Debugging with the Realtime ETS Kernel," contains detailed information about debugging ETS programs.

One of the first questions you have to answer when building an embedded system is whether the application is going to be RAM-based or ROM-based. If the answer is RAM-based, your development environment will be similar to the one described in this book. If you're building a ROM-based application, there will be some difference in how you boot ETS on the embedded target, but you'll use the same procedures for building and debugging your program.

RAM-Based Applications

Most embedded targets that run RAM-based applications are PC/ AT-compatible or PC/ 104 systems. The production version of the system loads the application from some kind of disk: floppy, fixed IDE, PC Card, or M-System flash.

A typical development cycle for a RAM-based application might contain the following steps:

1. **Boot ETS Monitor in WaitHost mode from disk, download embedded program from development host:** In this stage, your target system is connected to a host PC with a cable between the communications ports. You take full advantage of the host tools to develop your program. During this stage of development, you're making frequent changes to your program and usually running with a debugger to track down problem areas.
2. **Boot ETS Monitor in WaitHost mode from disk, run embedded program from disk:** Your target system is still connected to a host PC with a cable between the communications ports. During this stage of development, you're running a lot of tests and using the debugger only when a problem is encountered. This scenario might be suited for Alpha and Beta releases of your product. If your program is particularly large, you may find it faster to load the program from disk instead of downloading it from the development host.
3. **Boot ETS Monitor in NoWaitHost mode from disk, run embedded program from disk:** This scenario describes the production version of your product. The embedded system contains everything needed to run the program.

ROM-Based Applications

Depending on the availability of the target hardware, you may want to start development using a PC/ AT-compatible target. If so, you would follow Steps 1 and 2 in the previous section.

However, you probably want to start running your program on hardware that more closely resembles the intended embedded system as soon as possible. The ToolSuite documentation describes the

procedures to be used for getting the Realtime ETS Kernel up and running on a variety of different target boards.

A typical development cycle for a ROM-based application might contain these steps:

1. **Boot ETS Monitor in WaitHost mode from ROM, download embedded program from development host:** In this stage, your target system is connected to a host PC with a cable between the communications ports. You take full advantage of the host tools to develop your program. During this stage of development, you're making frequent changes to your program and usually running with a debugger to track down problem areas.
2. **Boot ETS Monitor in WaitHost mode from ROM, run embedded program from ROM:** Your target system is still connected to a host PC with a cable between the communications ports, but the ROM for your embedded system now contains your program as well as the ETS Monitor. During this stage of development, you're running a lot of tests and using the debugger only when a problem is encountered. This scenario might be suited for Alpha and Beta releases of your product. If your program is particularly large, you may find it faster to load the program from ROM instead of downloading it from the development host.
3. **Boot ETS Monitor in NoWaitHost mode from ROM, run embedded program from ROM:** This scenario describes the production version of your product. The embedded system contains everything needed to run the program.

If your target is not one of those directly supported by the Realtime ETS Kernel, you will also have to port the Realtime ETS Kernel to the target system. Version 9.1 supports the following hardware targets:

PC/104 Systems

Adastra Systems Corporation	http://www.adastra.com
Ampro Computers Inc.	http://www.ampro.com
Real Time Devices, Inc.	http://www.rtdusa.com
VersaLogic Corporation	http://www.versalogic.com
WinSystems Inc.	http://www.winsystems.com

Single Board Computers

Adastra Systems Corporation	http://www.adastra.com
Ampro Computers Inc.	http://www.ampro.com
Cell Computing, Inc.	http://www.cellcomputing.com
Megatel Computer Corporation	http://www.megatel.ca
RadiSys Corporation	http://www.radisys.com
S-MOS Systems, Inc.	http://www.smos.com
Teknor Industrial Computer Inc.	http://www.teknor.com
VersaLogic Corporation	http://www.versalogic.com

Industrial PCs

or Industrial Computers	http://www.or-computers.com
Texas Micro Inc.	http://www.texasmicro.com

STD-32 Systems

Octagon Systems	http://www.octa.com
VersaLogic Corporation	http://www.versalogic.com

Compact PCI Systems

Texas Micro Inc.	http://www.texasmicro.com
Ziatech Corporation	http://www.ziatech.com

CPU Modules

Forth-Systeme Modul 386EX

Forth-Systeme Modul SC400

ZF MicroSystems http://www.zfmicro.com

x86 CPU Evaluation Boards

AMD SC300/310
http://www.amd.com/products/lpd/elan300/elan300-310.html

AMD SC400/410
http://www.amd.com/products/lpd/elan400/21027a.html

Intel/Radisys 386EX EXPLR2
http://developer.intel.com/design/intarch/PRODBREF/27291601.htm

NS486
http://www.national.com/appinfo/ns486/evboard.html

Intel Pentium Processor Modules

Intel
http://www.intel.com/design/intarch/prodbref/971978.htm

Chapter 3

A Quick Tour of the Visual System Builder and CFIGKERN

You have a choice of methods for configuring your project. The Visual System Builder (VSB) presents an efficient graphical interface for specifying linker options and configuring the Realtime ETS Kernel. If you wish, you may instead use the CFIGKERN utility to specify kernel options.

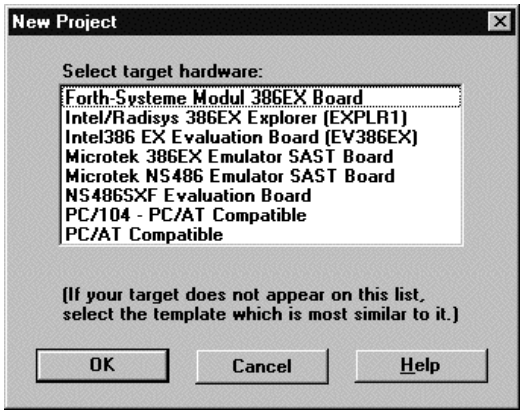
The Visual System Builder (VSB) is a Windows program that acts as a task-oriented interpreter between you and LinkLoc, providing a simple and reliable way to select the options you need for your particular compiler and target. This saves you the trouble of learning and remembering dozens of switches. You simply click on choices that describe the kind of system you want to build. The VSB then automatically selects the linker switches and kernel options appropriate to a system with the properties you've specified.

When you save your project, the VSB writes the switches and options into linker command files that are referenced by Developer Studio when building your application. Given the great number of options required for even a small system, the VSB can greatly simplify application development. And because your options are generated automatically, you're safe from typos and other syntax errors.

PRECONFIGURED TEMPLATES

Many projects will inevitably share the same or similar target hardware. The Visual System Builder supplies preconfigured templates for several hardware configurations.

A template is provided for you automatically when you begin your project. You choose a template at the beginning of the project from the New Project dialog box:

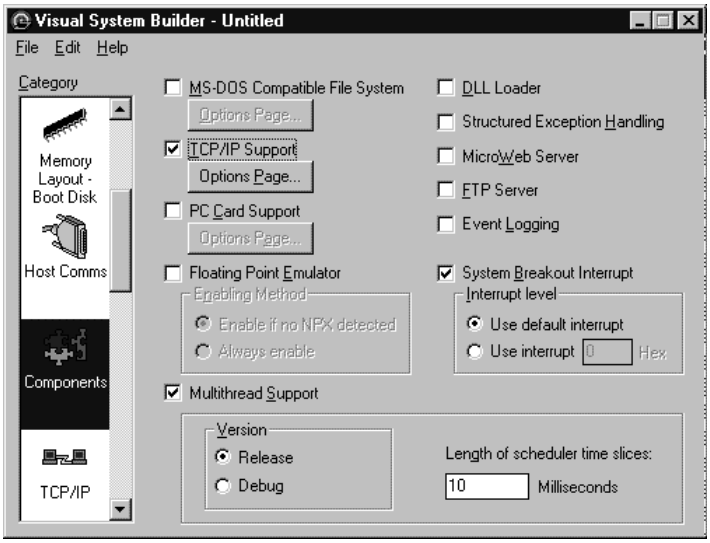


Each template is customized for a particular target hardware configuration and already includes appropriate settings for most of the options in the VSB. Using a template essentially frees you from having to make all but a few decisions for your project. If no template is provide for your particular target, you can begin with the template for the hardware most like yours.

Phar Lap frequently adds templates for new target hardware. Check with your Phar Lap sales representative for the most up-to-date information.

THE PROPERTY SHEETS

The Visual System Builder presents options on property sheets. Labeled icons in a scrollable window at the left side of the VSB dialog let you select the property sheet you want to work on. Each VSB property sheet presents you with options for different types of targets and applications.



The Components Property Sheet

Several property sheets are standard for every project:

- Compiler/Linker
- Monitor
- Memory Layout
- Host Communications
- Components
- Monitor Drivers
- Kernel Drivers
- Extra Linker Switches

These additional sheets present options for the Realtime Edition:

- File System
- TCP/IP
- IP Addresses
- Network Driver
- PC Card Support

Certain property sheets, such as TCP/ IP, are present only when an option has been selected on one of the standard pages.

Most of the property sheets will not allow you to choose options that are incompatible with options that have already been specified. In order to provide maximum flexibility, however, other options do not include this insurance. If you have trouble building your application, make sure that the specifications you have entered do not conflict.

CFIGKERN

CFIGKERN is a command-line utility that can be used to display and change certain options in a previously-linked kernel. It thus provides an easy way to change many settings in your project without having to relink. CFIGKERN does not allow you to change settings that require relinking.

If no switches are specified when CFIGKERN is run on a kernel, CFIGKERN prints out a summary of the options that can be configured in the named kernel file. The following is an excerpt from such a report:

```
CFIGKERN: 9.1 -- Copyright (C) 1986-98 Phar Lap Software,
Inc.

Dump of ETS Kernel settings in monitor file 'A:\DISKKERN.BIN'

    Boot Type: Disk (WaitHost)
    Debug Flag: 00000000h
    Monitor Code Base Address:  00007E00h
    Monitor Data Base Address:  00001000h
    Floating Point Coprocessor Detect: Auto
    Null Pointer Protection: Enabled, Range 0 to FFFh
    Bad Jump Protection: Disabled
```

CFIGKERN allows you to change settings in the following areas:

- Host Communication
- Application Loader
- Error Detection
- Specifying Target Memory
- Local File System
- Floating-Point Emulation
- Multitasking
- General Networking

- Specifying IP Addresses
- Driver Configuration

Chapter 4

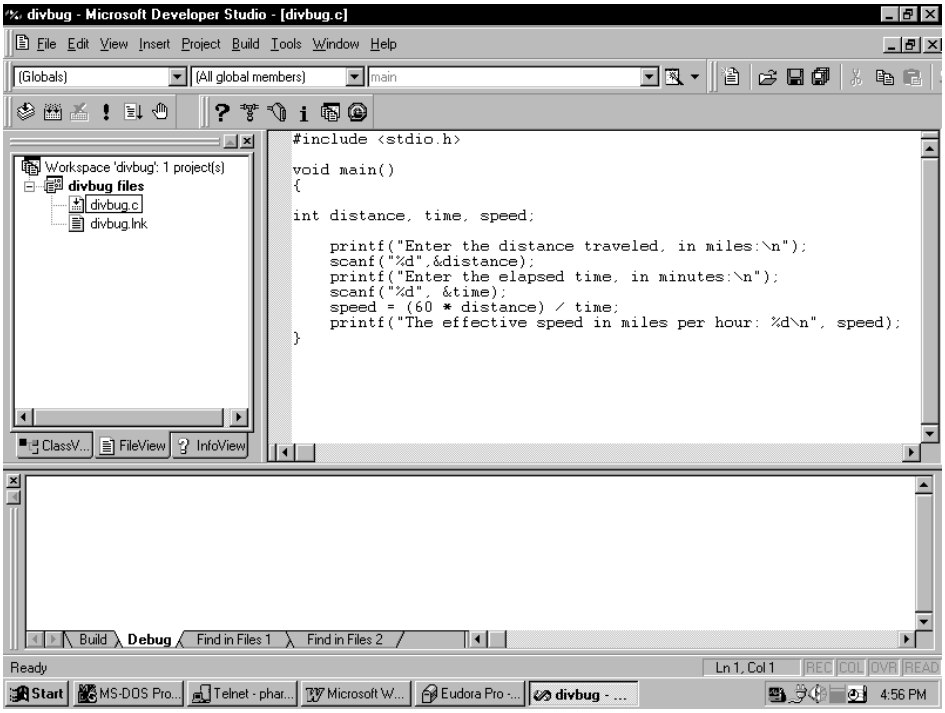
Debugging with the Realtime ETS Kernel

The Microsoft Developer Studio debugger is included with Visual C++ and Embedded StudioExpress is included with TNT Embedded ToolSuite. These two components work together to let you debug your embedded application with the Developer Studio Integrated Development Environment (IDE).

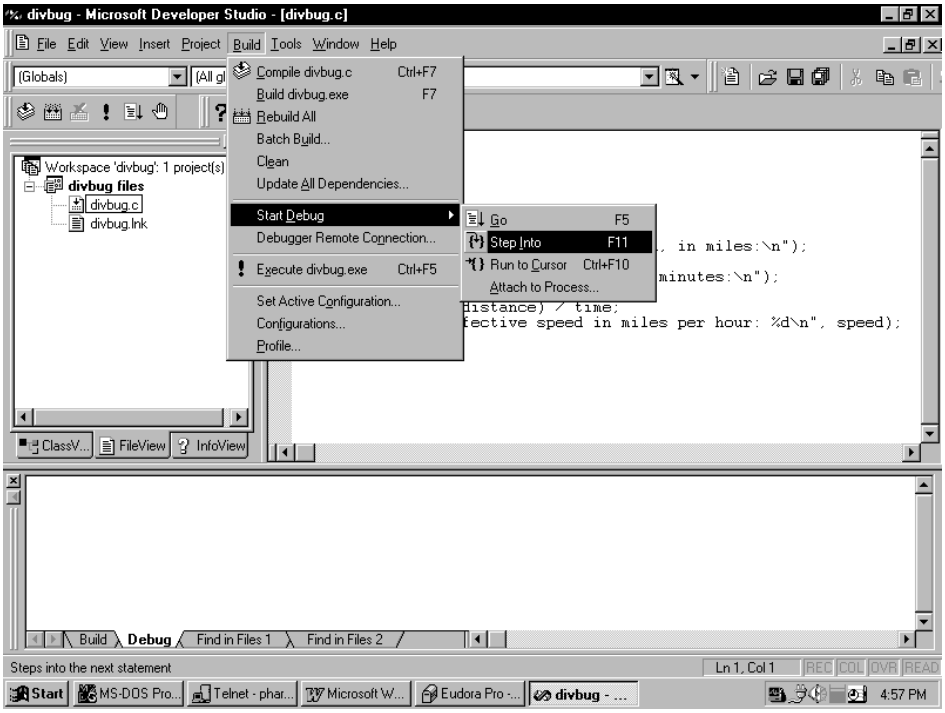
The beginning of this chapter discusses general issues related to debugging embedded programs and show you how to use the basic features of Developer Studio to debug a simple embedded program. We then describe the additional debugging features available with Embedded StudioExpress. These features extend the Developer Studio debugger, adding capabilities specific to debugging embedded applications under the Realtime ETS Kernel.

A SAMPLE DEBUG SESSION

To demonstrate the debug capabilities of the Developer Studio debugger and Embedded StudioExpress, we'll use the DIVBUG sample program. This program inputs two numbers and divides the first by the second without first checking to make sure that the denominator is not zero. If the second number is zero, this results in an integer divide by zero exception (exception 0). DIVBUG uses the C run-time library functions printf() to prompt the user for input on the host machine and scanf() to collect the data.

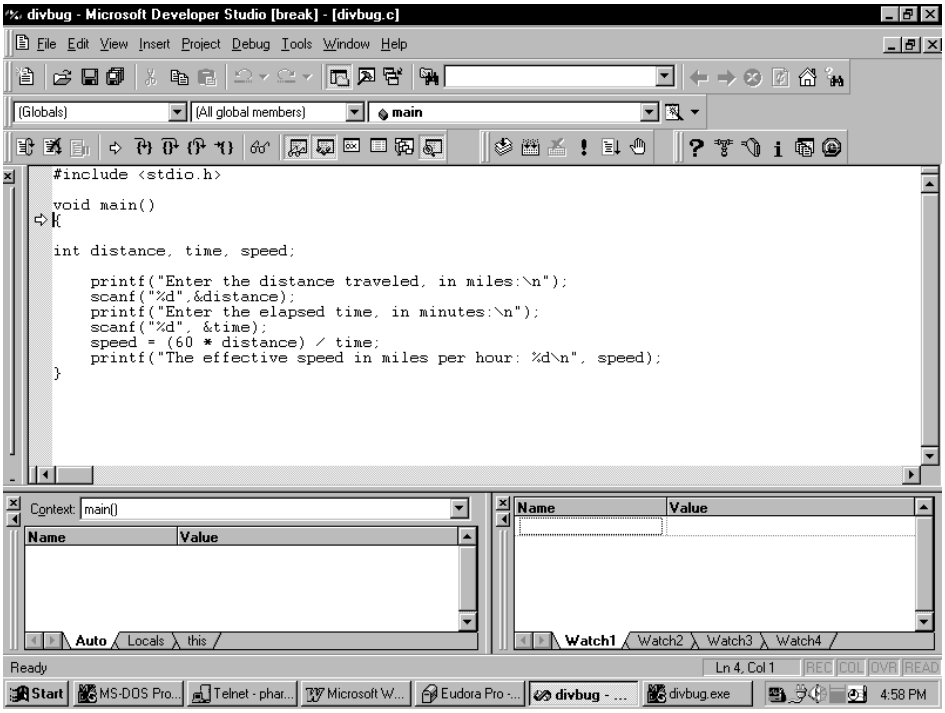


To start a debug session, select Start Debug from the Build menu. For this sample, we're going to choose Step Into:



StudioExpress will download your program to the embedded target. If you carefully watch the host screen, you'll see a console window open after the download is complete. Notice the new button labeled divbug.exe on the Windows taskbar. This console window will be used to perform keyboard input and screen output for the DIVBUG application running on the embedded target.

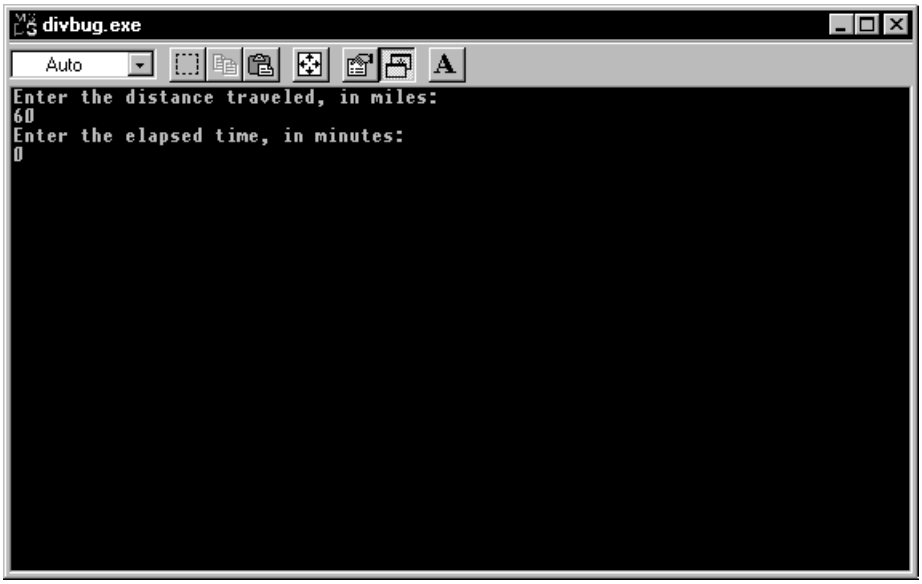
Because we started the debugger by choosing Step Into, a breakpoint is automatically set at main() as indicated by the arrow next to the beginning of this function:



At this point, you can use any of the normal Developer Studio debugging commands.

Recall that the DIVBUG sample program inputs two numbers and divides the first by the second without first checking to make sure that the denominator is not zero. If the second number is zero, this results in an integer divide by zero exception (exception 0). DIVBUG uses the C run-time library functions printf() to prompt the user for input on the host machine and scanf() to collect the data.

We'll let the program run (by selecting Go from the Debug menu) and see what happens. When prompted for input in the console window, specify 60 for the distance and 0 for the time.

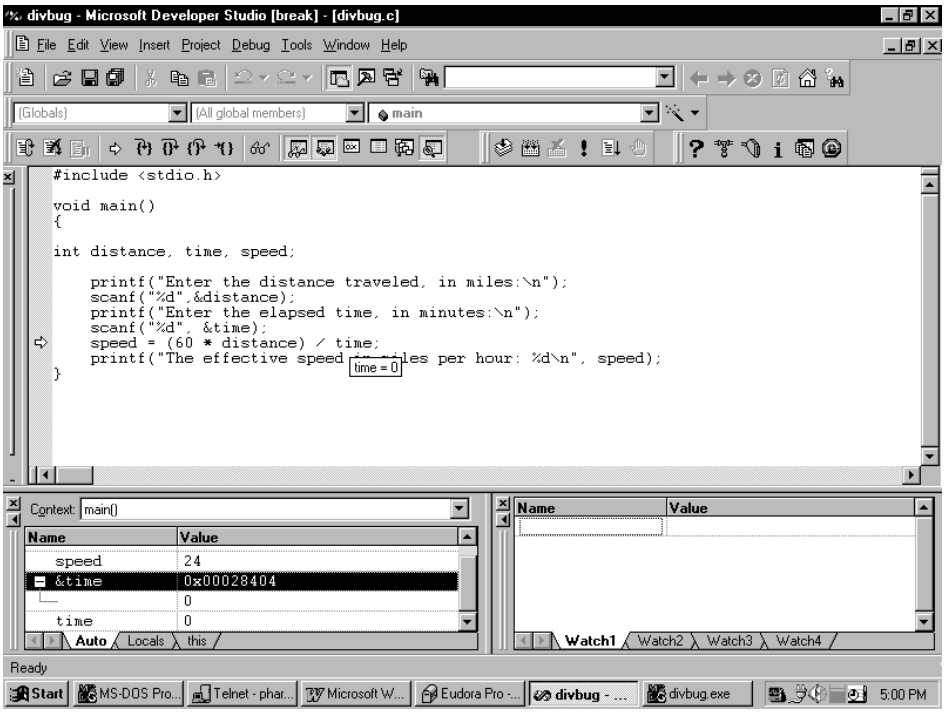


As expected we get an integer divide by zero error:

Click OK to return to the debugger. The breakpoint is at the line

```
speed = (60 * distance) / time;
```

Notice that if you place the cursor over the variable *time*, the current value is displayed:



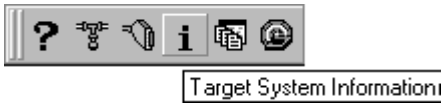
To correct this problem, we should add some checking code to the program to verify that the value of time is non-zero.

EMBEDDED STUDIOEXPRESS EXTENSIONS

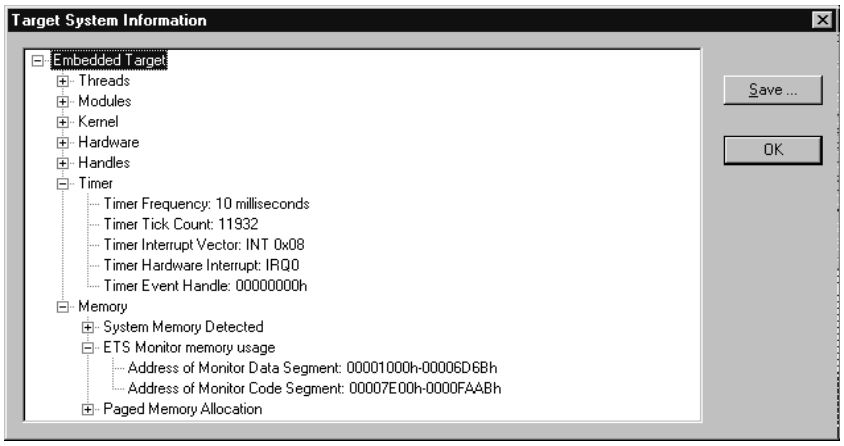
Target Port Input/Output

The Target Port Input/ Output dialog allows you to do direct port I/ O on the embedded target from the debugger. Direct port I/ O can be very useful when debugging device drivers, analyzing I/ O states, configuring hardware, etc. To access this component, click on the ETS Target Input/ Output icon on the ETS Toolbar.

Target System Information



The Target System Information component of Embedded StudioExpress displays just about everything there is to know about what's happening on your embedded target. To access this component, click on the Target System Information icon on the StudioExpress Toolbar. If we were to do this when the exception occurred in DIVBUG, we'd see a list of subsystems for which information is available. Clicking on the name of any target subsystem will cause StudioExpress to display information about the components of that subsystem. At any level of the display, a plus sign (+) preceding an object name indicates more information is available. Click on the plus sign to see the next level of detail. For example, the following screen shows detailed information about the timer and memory subsystems.



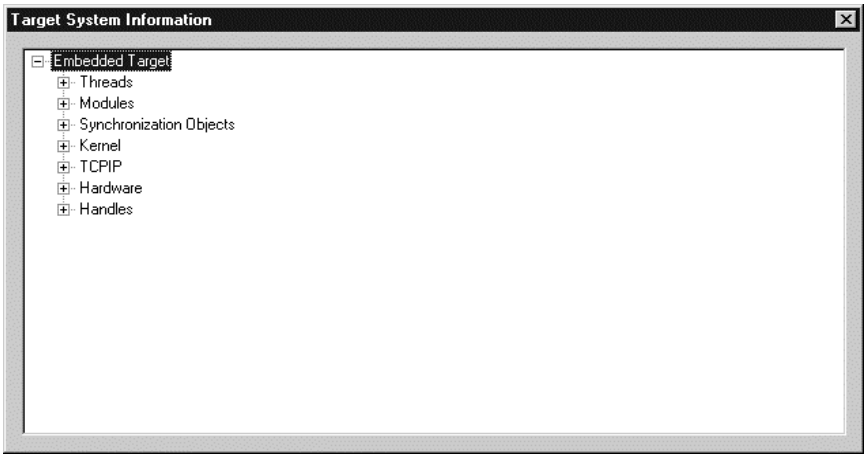
Clicking on the Save button causes the entire contents of the Target System Information window to be copied to a user-specified text file, which is then opened in the main window of the debugger:

We have tried to include just about anything you could possibly want to know about the state of the embedded system in the Target System Information. In most cases, you should be able to find everything you need to fix your problem without going to the lowest levels of detail. But, know that the information is there if you need it.

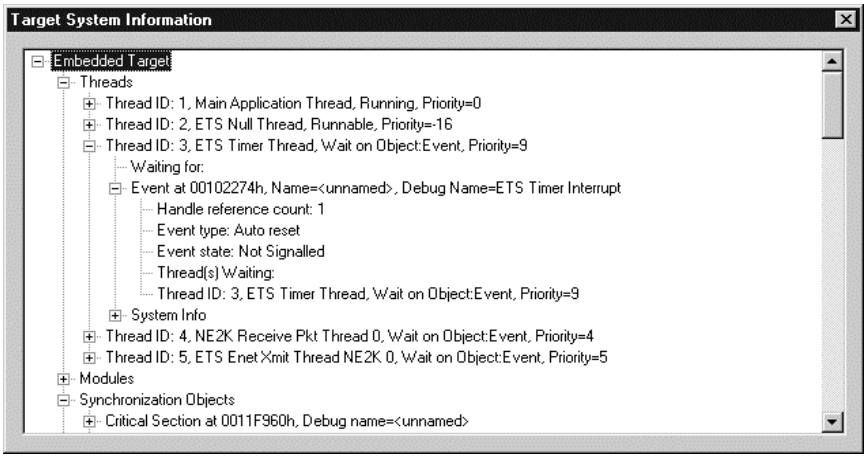
DEBUGGING MULTITHREADED PROGRAMS

The techniques that you use for debugging single threaded programs aren't always going to be adequate for multithreaded programs. In general, your realtime program will have several threads running and it is quite helpful to know the state of those threads when you're debugging. You may also have questions that just don't come up when debugging single threaded programs. For example, why is this particular thread blocked? The embedded system can also become deadlocked. This happens when no threads are runnable and each is waiting on a resource owned by another thread.

The Target System Information displayed by Embedded StudioExpress is a comprehensive view of the current state of the embedded target. The following window appears when you select Target System Information from the StudioExpress Toolbar:



Clicking on the name of any target subsystem will cause StudioExpress to display information about the components of that subsystem. If, for example, you click on "Threads," you'll see the state of all the active threads. Clicking on any thread will display more information about that thread. For example, the ETS Timer Thread is usually waiting for an event to be signaled by the ETS Timer Interrupt Handler:



We see that the object being waited for is at address 00102274h. We can also see that the ETS Timer Thread is the only one waiting for this object. If other threads were also waiting, they would be listed.

In addition to the information displayed by Embedded StudioExpress, you may find it helpful to take advantage of the following procedures and Realtime ETS Kernel features when debugging multithreaded programs.

- **Use the Realtime ETS Kernel API function EtsSetDebugName() to name your threads and synchronization objects.** This information is then automatically displayed by StudioExpress when examining the state of the target system. It will also be displayed if you call EtsDumpThreads().
- **Use the ETS event logging system to keep track of significant events in your program, including system APIs and events and user events.** The basic event logging system is described later in this chapter.
- **If your embedded system appears to be deadlocked, and if there's an NMI button on the embedded target, you can signal an NMI to stop the target, giving control to the ETS Monitor.** This will restore you to the debugger prompt. This technique will work even if interrupts are disabled on your target. If interrupts are enabled, you can effect the same thing by using Ctrl-Break on the development **host**. This will stop the program on the target, giving you control at the debugger prompt.

- **The Realtime ETS Kernel creates a very low priority (-16) system thread named "ETS Null Thread."** Because the priority is so low, this thread will run only if no other threads can run. This can happen if the embedded target gets deadlocked. If you're using the StudioExpress Toolbar or if you call EtsDumpThreads(), see if the current thread is the "ETS Null Thread." If so, you should look for signs of deadlock in your program.
- **When you enter the "go" command to a debugger, your program starts to run and you have no control over which threads are running.** While this behavior is desirable in a running program, it greatly complicates the debugging process. In order to simplify your debugging, you should freeze all threads except the one you're debugging. Please note that only application threads — not system threads — can be frozen. In particular you cannot freeze the timer, keyboard or null system threads. In practice, the inability to freeze the timer thread is the only one that might be troublesome. The timer thread will run according to its priority, calling any registered timer callback functions.

  Once you've corrected any problems in the thread you're debugging, unfreeze the other threads and see what happens. Repeat this process, each time freezing all threads but one until you've debugged the whole program.

When you're debugging a multithreaded program, you might also want to specify the debug version of the ETS Multithread Library. This version of the library contains some additional functionality which facilitates debugging, but has a performance cost unnecessary in production programs:

- A stack frame is created on every function call. This means that some compiler optimizations have been disabled, but it allows you to more easily look at threads other than the current one while debugging. If you look at a stack backtrace, you will eventually find the call to your user code in the other threads.
- There is additional error checking within the library itself.
- When switching threads, the ETS Multithread Library code reads the Thread ID out of memory at a specific time so hardware-assisted debuggers can trap on this read. The ETSCHECKPOINTS structure defined in EMBKERN.H resides at the start of the monitor code segment and contains "checkpoint" locations.

```
typedef struct tagETSCHECKPOINTS
{
    char    ChkReturnToApp;
    char    ChkEnterMonitor;
    char    ChkThreadSwitch;
    char    reserved[13];
} ETSCHECKPOINTS;
```

When switching threads, the ETS Multithread Library reads location `ChkThreadSwitch`. This is a signal to a hardware-assisted debugger that the next data read will be the 32-bit Thread ID.

Sample Output from EtsDumpThreads()

The following is a portion of the active threads list from the software running Phar Lap's "World's Smallest Web Server." This software can be demonstrated by pointing your Web browser at http:/ / smallest.pharlap.com.

```
Current Thread List:

THREADBLK at 00101A48, System Thread, ID=2, BasePri=-16,
CurrPri=-16
 Thread Name = "ETS Null Thread"
 Flags = 0801, In Run Queue

THREADBLK at 001045BC, Application Thread, ID=4,
BasePri=0, CurrPri=0
 Thread Name = "ETS TCP/IP Receive Thread"
 Flags = 0C06, Waiting on IPC Object(s), In Timeout
Queue
 Waiting on EVENTBLK at 001047C0 (name="/event/ETS TCPIP
Incoming Packet")

THREADBLK at 001043A0, Application Thread, ID=5,
BasePri=0, CurrPri=0
 Thread Name = "ETS TCP/IP Transmit Thread"
 Flags = 0C01, In Run Queue
 IPC Objects owned by this thread:
  MUTEXBLK at 00104838, No waiters

*** Current Thread: ***
THREADBLK at 00143E2C, Application Thread, ID=107,
BasePri=0, CurrPri=0
 Thread Name = "HTTP Connection to 192.107.36.201"
 Flags = 0C01, In Run Queue
```

The information displayed for each thread includes the following:

- Whether the thread was created by the system (Realtime ETS Kernel) or the application
- The number of the thread
- The base and current priority of the thread
- The thread name
- The status of the thread
- Any synchronization objects the thread either owns or is waiting on

There were eight threads running when the above list was generated. The four threads shown illustrate interesting and helpful information that can appear in the thread list:

- **ETS Null Thread**
  The very low priority system thread that should run only when no other threads are runnable. If "ETS Null Thread" is the current thread, it is a good indication that your program is deadlocked. When this snapshot was taken, "ETS Null Thread" is sitting in the run queue while higher priority threads are running.
- **ETS TCP/IP Receive Thread**
  A typical application thread. It is waiting for an incoming TCP/ IP packet.
- **ETS TCP/IP Transmit Thread**
  Another application thread. It owns a mutex object on which no other threads are waiting.
- **HTTP Connection to 192.107.36.201**
  The currently active thread. This thread manages the connection to the Web browser.

THE EVENT LOGGING SYSTEM

The ETS event logging system provides a convenient, automatic way of keeping track of significant events in the life of your program. This can be particularly helpful in locating and fixing troublesome bugs in your

code. The ETS event logging system is particularly useful for debugging multithreaded programs.

Here's a simple abridged example of an event log:

```
t=90 tid=5 API Enter: GetLastError
  Params= none
t=90 tid=5 API Exit: Sleep
  Params= none
t=80 tid=6 API Enter: ExitThread
  Params= 0x00000000h
t=80 tid=6 API Exit: TlsSetValue
  Params= 0x00000001h
t=80 tid=6 API Enter: TlsSetValue
  Params= 0x0000001Dh 0x00000000h
```

The buffer is LIFO (last-in, first-out). Starting at the bottom, then, we see that the time (t) is 80 milliseconds, and the thread ID (tid) is 6. The thread has just entered the Thread Local Storage Set Value function (TlsSetValue()) with 01Dh passed as an argument.

Moving up, we see that the next entry is still at timer tick 80. Thread 6 is exiting TlsSetValue() at this point. No argument is being passed on exit.

Moving up to the third and middle entry, and still at the same timer tick, we see that thread 6 is executing ExitThread().

At the next entry we see that the timer tick has been incremented by the Realtime ETS Kernel to 90 milliseconds. Thread 5 is being awakened by the Realtime ETS Kernel, as you can see by the Exit call to Sleep().

In the last entry in this example, we see thread 5 executing a call to GetLastError() with no argument.

As you can see even from this brief example, the event log makes it very easy to track the sequence of events in program execution.

Now that you've seen some typical output from an event log, let's look at the more technical aspects.

Events are logged to a circular buffer of log entries which can then be displayed. Under the Realtime ETS Kernel, an event is a variable-length structure defined in the file EVENTLOG.H:

```
typedef struct
{
  DWORD TimeStamp;  // Time event was logged.
  DWORD ThreadId;   // Thread id that logged event.
  DWORD EventId;    // Id identifying event.
  DWORD ParamCnt;   // Number of DWORD parameters in event.
  DWORD Params[];   // Array of DWORD parameters.
} EVENT_LOG_ENTRY;
```

As you saw in the example above, the TimeStamp and ThreadId fields are filled in automatically for all events. Every logged event is identified by a DWORD EventId. Event IDs below 0x8000000 are used for system events; Event IDs of 0x8000000 and higher are available for user-defined events. To log a user event, you specify the EventId, ParamCnt, and Params fields.

You specify which events are to be logged by setting bits in the log flags, corresponding to the events of interest. ETS events are grouped into four subsystems:

- **SUBSYS_API**
  Creates a log entry whenever infrequently called ETS or Win32 APIs are entered or exited.
- **SUBSYS_VAPI**
  Enables logging of frequently called ETS and Win32 APIs.
- **SUBSYS_LOADER**
  Enables logging of events whenever any function in the DLL Loader Library is called.
- **SUBSYS_EMBMT**
  Enables logging when key events happen inside the Realtime ETS Kernel multitasking library.

The pseudo-subsystem SUBSYS_ALL can be used to enable logging of all ETS system events.

Chapter 5

Options for the Realtime ETS Kernel

The Realtime ETS Kernel includes a number of configuration options, providing a great degree of flexibility in building your system. Some options must be specified when the kernel is built, while others can be changed later with the CFIGKERN utility. This chapter identifies these options.

The most basic configuration option is the boot method. There are two basic choices. Disk kernels can load embedded programs from disk or over the communications cable from a host PC. ROM kernels can load programs over the communications cable, from ROM on the embedded system, and from disk (BIOS Extension ROM kernels only).

The Realtime ETS Kernel has two operational modes, independent of boot method: WaitHost and NoWaitHost. In WaitHost mode, the Realtime ETS Kernel boots the target system, initializes itself, optionally loads your program, and waits for a signal from the host PC before executing your program. Therefore, to run in WaitHost mode, you must have a host PC connected to your embedded system with a communications cable. In NoWaitHost mode, the Realtime ETS Kernel boots the target system, initializes itself, loads your program, and begins executing it. There are three ways to set the run mode:

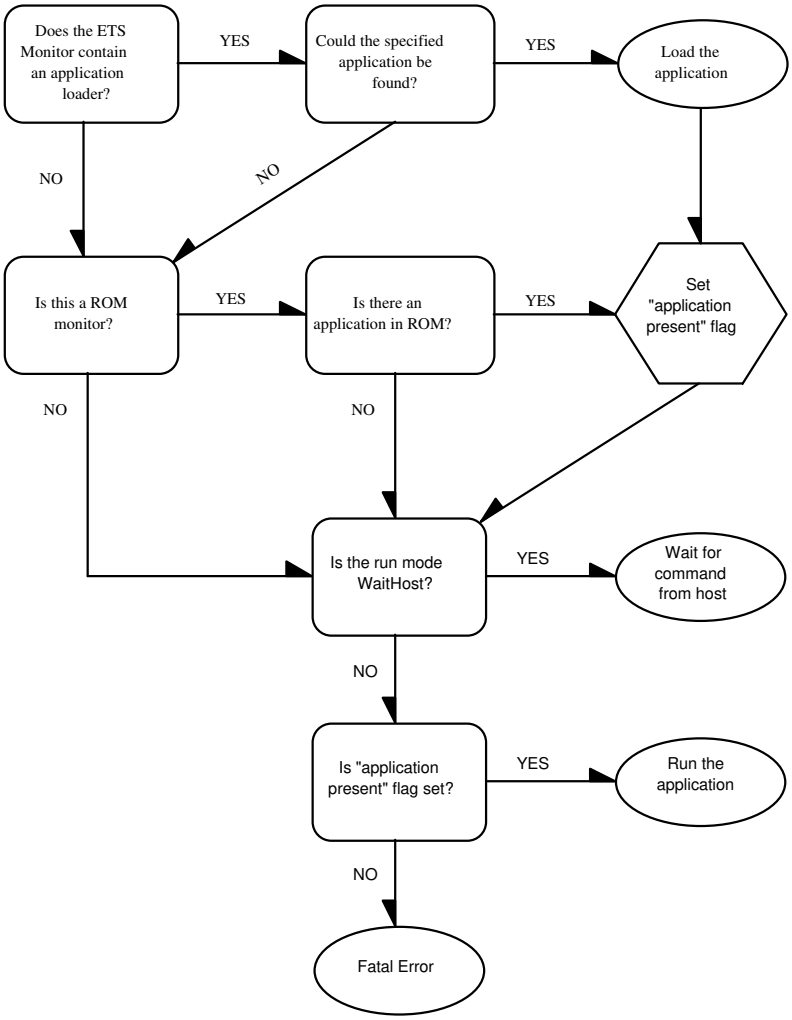
- When the kernel is built.
- By running the CFIGKERN utility on the kernel.
- By writing a function called EkCustomRunMode(), which is linked into the kernel. This function could, for example, read a hardware switch at boot time to select WaitHost or NoWaitHost mode dynamically.

You must choose whether to include host communications code. This code must be included for any kernel that must run in WaitHost mode, for instance during debugging.

If a kernel includes an application loader, one of the last steps in the kernel initialization is to load an application. There are two ways to specify the application file:

1. The default is to load the second file on the boot disk.
2. You can use the CFIGKERN switches -APPNAME and -APPDRIVE to specify the name and drive location of the application. If no drive is specified, the default is to use the boot drive for disk kernels and drive A: for ROM kernels. The file name is a standard DOS file name with no provisions for directory specification, so the file must be located in the root directory on the specified drive.

The figure opposite illustrates the process of loading an application and how the different kernel options can affect this process.



Realtime ETS Kernel Application Loading Process

To summarize, if an application loader is present, the Realtime ETS Kernel will try to load the specified application. If no loader is present or if the specified application could not be found, ROM kernels will use the application in ROM, if one is present.

The combination of NoWaitHost run mode and no application (either loaded or in ROM) causes the Realtime ETS Kernel to generate a fatal error and exit.

If the kernel run mode is WaitHost, you can choose either to run the loaded application (if any), or download a different application and run it instead.

Although it is easy to customize a kernel with the Visual System Builder, for your convenience several pre-made kernels are included with the TNT Embedded ToolSuite to suit a wide variety of platforms and boot methods.

The "BIOS Extension" and "Boot Jump" methods for booting the Realtime ETS Kernel are discussed later in this chapter.

BOOTING FROM DISK

When the Realtime ETS Kernel boots from a disk, the process is the same for all disks: floppy, IDE hard, PC Card ATA, or M-Systems flash. Because the boot sector loader calls PC BIOS functions to load the ETS Monitor, the target system must have a PC/ AT-compatible BIOS if you want to boot from disk.

When the machine is turned on, the BIOS tries to boot by first checking the A drive and, if there is no disk, then checking the C drive. By convention, bootable disks have a boot sector loader in the boot record.

The BIOS loads the boot sector loader into memory and begins to execute this code. The boot sector loader is a simple program (it has to fit in one 512-byte disk sector) that loads the first file on the boot disk and transfers control to that code. The only constraint imposed by the boot sector loader is that the file to be loaded must be contiguous on disk. It can be as large as necessary, it just cannot be fragmented.

When you boot DOS, the first file on the disk is MSDOS.SYS. When you boot the Realtime ETS Kernel, however, this file is usually DISKKERN.BIN, a disk kernel. The ETS boot sector loader loads the Realtime ETS Kernel into memory and begins to execute it.

Because of the different physical characteristics of flash, floppy, and hard disks, there are some procedure differences in the way you create the boot disk. Once the disk contains DISKKERN.BIN, however, there are no differences. In particular, the procedures for loading your program are the same for all disk kernels, regardless of type of disk.

It is also possible to put the Realtime ETS Kernel in a BIOS Extension ROM which will load your application from disk each time the target is booted. This strategy eliminates the step of loading the Realtime ETS Kernel from the disk. Once the Realtime ETS Kernel starts executing, the boot sequence continues unchanged.

Once running, the Realtime ETS Kernel starts the booting process by determining the disk drive from which the application should be loaded. The drive letter can be configured into the kernel or you can use the default drive, the one from which the kernel booted (the "boot

Next, the Realtime ETS Kernel checks to see if an application name has been configured into the kernel. If it has, the Realtime ETS Kernel tries to load it from the root directory of the designated disk. If no application name has been configured, the Realtime ETS Kernel tries to load the second file in the root directory of the designated disk.

When you boot the embedded system, the Realtime ETS Kernel will initialize itself and then load your program from the disk. What happens next depends on whether the kernel is running in WaitHost mode or NoWaitHost mode.

Running in WaitHost mode requires that the kernel was built with the host communications option. Assuming that your kernel has been appropriately configured for WaitHost mode, you must choose Execute or Start Debug from the Developer Studio Build menu on the host to synchronize with the kernel and start execution. Use the Nodownload option to prevent downloading the file (since it was already loaded from disk).

If you don't use the Nodownload option, the application from the host will be downloaded, overwriting the one loaded from disk, and the Realtime ETS Kernel will execute the downloaded application instead of the one loaded from disk.

If the kernel is running in NoWaitHost mode, your program will begin executing as soon as the kernel has finished loading it. No additional action is required, or even possible.

Loading your embedded program from disk is useful during Beta testing or when there is no host PC, or as a way to try new versions of your application in the field if your embedded system includes a disk drive.

BOOTING FROM ROM

When your program is fairly stable, and you're ready to prepare for the final production, you need to program a ROM with the Realtime ETS Kernel and your program.

Before you can begin programming the ROM, you must decide on the method by which the Realtime ETS Kernel will be booted. There are two possible methods: BIOS Extension ROM and Boot ROM. Samples of both kernels (BIOSKERN.EXE and BOOTKERN.EXE) for PC/ AT machines are included in TNT Embedded ToolSuite.

The ToolSuite also contains sample kernels for several commercially available target boards.

There are two ways to run an embedded program loaded from ROM, depending on whether the kernel is running in WaitHost mode or NoWaitHost mode.

When a ROMable program is built, the resulting ROM image contains both the Realtime ETS Kernel and a protected-mode embedded program. When the kernel boots, it initializes itself and then loads this embedded program.

If the kernel is running in NoWaitHost mode, this program will begin executing as soon as the kernel has finished loading it. No additional action is required, or even possible.

If it is running in WaitHost mode, the kernel does not begin executing the code but waits for a signal from the host. This gives you a chance to load a different program from the host. Normally, when you are planning to download your application, the application program you place in ROM with the Realtime ETS Kernel will be the tiny MINAPP sample program distributed with TNT Embedded ToolSuite.

BIOS Extension ROM Boot Method

In this book, the term "BIOS extension" kernel refers to a kernel that runs after other software has initialized the system hardware.

IBM PC/AT-Compatibles

For an IBM PC/ AT-compatible, the BIOS extension method uses the BIOS capabilities that allow additional components to be initialized as part of the BIOS power-on self test (POST) and initializations. During POST processing, the BIOS code scans specific address ranges, looking for a three-byte signature:

Byte 0	0x55
Byte 1	0xAA
Byte 2	Number of 512-byte blocks in the ROM

The address ranges searched are:

C0000h – DFFFFh	search the address range in increments of 2K
E0000h	search the address, the ROM must occupy the entire 64K

When the signature is found, the BIOS issues a far call to the address in the first byte following the signature. When the call returns, the BIOS continues processing. When all BIOS processing has been completed, an INT 19h is issued to boot and start.

When the BIOS extension method is used to boot the Realtime ETS Kernel from ROM, the ROM must be located at one of the specified addresses. A valid signature will be found while the BIOS is searching the address ranges. The code called will install an INT 19h handler and return. After the BIOS has completed all self-testing and loaded all the other extensions, the BIOS issues an INT 19h as usual. Now, however, the Realtime ETS Kernel INT 19h handler will intercept the interrupt and boot the Realtime ETS Kernel instead of DOS. Since this kernel is already in ROM, control is passed to the beginning of the code.

National Semiconductor NS486SXF Evaluation Board

In the case of the National Semiconductor NS486SXF Evaluation Board, initialization is performed by the NSC boot program in the flash memory. The BIOS extension ROM program is downloaded to the

flash memory using the NSC utility program, FLASHLDR.EXE. When you reboot the target, the NSC boot program runs first and then jumps to the application program you downloaded with FLASHLDR.EXE.

The NSC boot program initializes the chip select registers to make the 128K flash memory answer at FFFC0000h–FFFFFFFFh, and makes the 1 MB of RAM memory appear from 0–100000h. The on-chip I/ O peripherals are programmed to appear at the PC/ AT-compatible I/ O addresses.

Boot Jump Method

Not all embedded systems will have an IBM PC-compatible BIOS. For those systems, the "Boot Jump" method is used to boot the Realtime ETS Kernel. When a 32-bit x86-compatible CPU is booted, control goes to address F000:FFF0. If the ROM containing the Realtime ETS Kernel is placed at this address, the machine will begin executing the code in the ROM when it is powered on.

A kernel that is booted using the Boot Jump method must initialize the chip set on the motherboard. Because there are so many different chip sets, the BOOTKERN.EXE distributed with TNT Embedded ToolSuite does not do chip set initialization, and therefore can only run on a motherboard with a chip set that doesn't require software initialization.

In spite of this limitation, BOOTKERN.EXE is useful because together with the BOOTJMP sample program provided with TNT Embedded ToolSuite, it illustrates how to build an embedded product with a Boot Jump kernel. The sample kernels for the Forth-Systeme Modul 386EX Board and the Intel386 EX Evaluation Board also employ the Boot Jump method.

DOS Boot Option

If your target system is a PC/ AT that can run DOS, you can also boot the Realtime ETS Kernel from the DOS prompt. This feature can be particularly convenient during program development. For example:

- You can use the standard, familiar DOS utilities for file management on your target.

- While you're developing your device drivers, you can use the DOS drivers to initialize your hardware devices.

- Many single-board computers are supplied with DOS. Booting the Realtime ETS Kernel directly from DOS is a quick and easy way to get up and running.
- If you're modifying the real-mode start-up code in the Realtime ETS Kernel, you can use real-mode DOS debuggers like SYMDEB or DEBUG to debug your changes to this code.
- A query can be added to the AUTOEXEC.BAT file, asking the user whether to run DOS or the Realtime ETS Kernel, to provide dual boot capabilities on the target. If the user wishes to run the Realtime ETS Kernel, invoke the DOSBOOT program described below from within AUTOEXEC.BAT.

If you're going to boot from DOS, there are some constraints on the environment:

Your embedded application must be loaded above 1 MB. You specify this load address with the -OFFSET switch to LinkLoc. For example,

```
C:\>linkloc @vc.emb test –offset 100000h
```

When you start the target with DOS, do not load DOS high.

The target cannot be running in Virtual 8086 mode. EMM386 and other memory managers put the machine into this mode. You should not load one of these programs if you're planning to boot the Realtime ETS Kernel.

Once booted from DOS, the Realtime ETS Kernel behaves as if it had been loaded directly by the disk boot loader. The only difference is where the kernel code and data are loaded in memory. When you boot the Realtime ETS Kernel directly, the kernel code and data are loaded at the locations specified when the kernel was linked. When you boot from DOS, they are loaded where DOS chooses to load them.

KERNEL BUILD OPTIONS

The Realtime ETS Kernel consists of three components, two of which are optional:

1. Basic operations (required)
2. Application loader (optional)
3. Host communications (optional)

In addition, the kernel can be loaded from a bootable diskette, using the BIOS Extension method from ROM, or using the Boot Jump method from ROM.

These options are all selectable from the Kernel Options property sheet in the Visual System Builder.

In addition, there are batch files shipped with the ToolSuite distribution for building these kernels. To build a kernel with different build options, you would edit one of these files, choosing the batch file that corresponds with the method used to boot the Realtime ETS Kernel on your system.

REPLACEABLE CODE

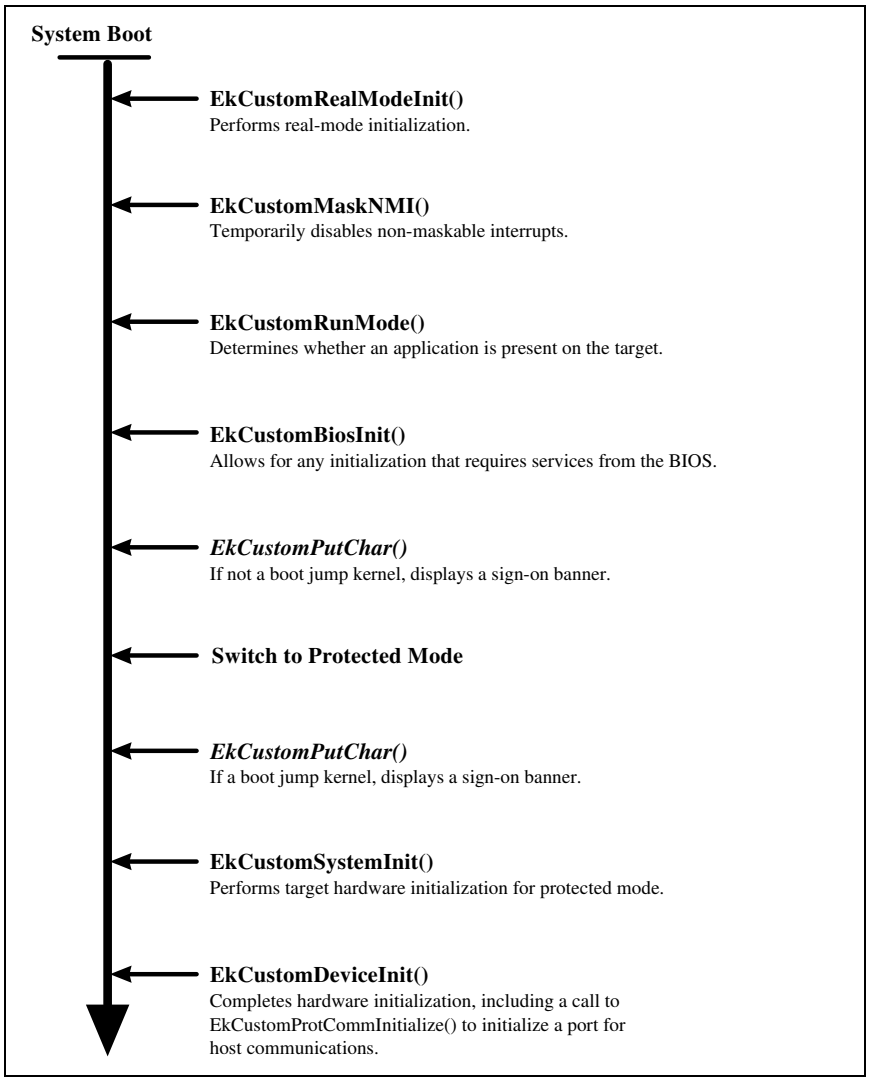
As part of its initialization, the Realtime ETS Kernel calls a number of functions which perform specified actions on the embedded system hardware. As stated previously, the default hardware configuration is a PC/ AT-compatible computer. Other platforms are also supported. Please contact your Phar Lap sales representative for the most up-to-date list of supported hardware. If you have different hardware, you can replace the target-specific files with ones that are applicable to your system. All the replaceable kernel functions begin with the prefix EkCustom. The EkCustom function are listed in Appendix A.

The Kernel Initialization Process

This section describes the initialization sequence for the Realtime ETS Kernel from the time that the embedded system is booted in real mode until user code begins to execute. During initialization, the Realtime ETS Kernel initializes the machine, communicates with the host if necessary, and then transfers control to the application. The description is from the point of view of the EkCustom functions. Details of kernel

initialization which are not target specific (like creation of descriptor tables and switching into protected mode) are described mostly in terms of the kernel doing additional processing.

The following diagram presents a timeline for the kernel initialization process.



Realtime ETS Kernel Initialization Sequence

The kernel starts up in real mode. This part of the kernel runs out of ROM only and does not touch any memory outside the ROM. It sets up

a register set which it wants to use (CS, DS, SS:SP, etc.) and then jumps to EkCustomRealModeInit(). The kernel data segment is not addressable at this time.

EkCustomRealModeInit() completes real-mode initialization so that the kernel can transfer control to protected mode.

EkCustomRealModeInit() is responsible for the following:

- Initializing all motherboard memory hardware (including DMA refresh) so that the kernel can touch RAM memory.
- Calling the kernel function EkCustomInitKernelData() to initialize the data segment to all zeroes.
- Detecting hardware devices and storing their addresses in the global KernelInfo data structure. For the PC/AT hardware, the following devices are detected:
  - Master 8259
  - Slave 8259
  - Monochrome Display Adapter
  - CGA or better Display Adapter

  For non-PC/AT hardware, EkCustomRealModeInit() should initialize the corresponding devices.

  If the machine is PC/AT-compatible and has an installed BIOS, then the kernel was either loaded from disk or run as a BIOS extension. In this case, the BIOS has already run its power-on self test and done hardware detection, so rather than detecting the hardware devices, EkCustomRealModeInit() calls the BIOS to detect the following hardware devices and store their addresses in the global KernelInfo data structure:
  - CPU Type and Coprocessor Type
  - Master 8259
  - Slave 8259
  - Monochrome Display Adapter
  - CGA or better Display Adapter
  - BIOS Memory values

® Regardless of whether a BIOS is present, enabling A20 to allow full 32-bit addressing.

When EkCustomRealModeInit() completes, it returns to the kernel by jumping to the address passed to it in the DX register. At this point the kernel data segment and stack are now usable.

The kernel collects some basic information about the system. First, it determines the type of CPU and coprocessor present. Then the kernel calls EkCustomMaskNMI() to temporarily disable non-maskable interrupts (NMIs).

The kernel calls EkCustomRunMode() which is responsible for making a decision as to whether or not the kernel should wait and synchronize with the host before continuing to execute the application. If the kernel is to wait for the host, EkCustomRunMode() should return FALSE. If the kernel is to continue and start execution of the application (assuming there is one in memory already), EkCustomRunMode() should return TRUE. This routine is replaceable in case the user wants to check some status information (like a DIP switch on the target) to decide whether or not to wait for the host debugger.

At this point, the kernel calls EkCustomCommInitialize(). This call is made for reasons of backwards compatibility only. The actual initialization of the communications hardware now takes place in protected mode.

The kernel then prepares to enter 16-bit protected mode. After it enters protected mode but before interrupts have been enabled, the kernel calls the function EkCustomSystemInit().

EkCustomSystemInit() is responsible for initializing any target hardware required for the kernel to run in protected mode. EkCustomSystemInit() must **not** enable interrupts while it does its work.

When EkCustomSystemInit() completes, it returns to its caller. At this time, the kernel continues its initialization. After it enables interrupts, it calls EkCustomDeviceInit(), which is responsible for final initialization of any hardware devices required to run the kernel. In particular, EkCustomDeviceInit() calls EkCustomProtCommInitalize() to initialize the communications hardware. The only difference between EkCustomSystemInit() and EkCustomDeviceInit() is that the machine is running with interrupts enabled so unmasked hardware interrupts will

be occurring while EkCustomDeviceInit() — and all subsequent code — is executing.

The kernel then completes its initialization. If EkCustomRunMode() returned TRUE (NoWaitHost), the kernel then transfers control to the application code. If EkCustomRunMode() returned FALSE (WaitHost), the kernel then uses the EkCustomComm functions to initiate a conversation. At this point, the kernel is under the control of the host computer, which may or may not instruct it to start running (and/ or debugging) application code.

If EkCustomRunMode() returned TRUE and no application is present, the kernel will display a message on the screen (if there is one) and then assume that EkCustomRunMode() should have returned FALSE. It will then attempt to synchronize with the host.

After the application loaded by the kernel terminates, the kernel re-initializes itself to get the machine into the same state as it was when the application was first loaded. This is so that another application can be re-loaded into the machine.

The kernel calls all the protected-mode EkCustom functions that were called at cold-boot time, this time with a parameter indicating that this is a warm reboot.

Chapter 6

Realtime ETS Kernel Programming Environment

In previous chapters of this book, we've talked about the Realtime ETS Kernel, told you how to build programs that run with it on your embedded target, discussed various ways to run those programs, and showed you how to use the host debuggers to find problems in your program. Now, it is time to look more closely at the Realtime ETS Kernel itself.

Using Visual C++ and Developer Studio along with the Realtime ETS Kernel brings the power of 32-bits and the ease of C-language programming to embedded systems. The simple HELLO and DIVBUG sample programs mentioned in previous chapters show that developing an embedded program can be quite straightforward, and not all that different from developing programs for general-purpose operating systems like MS-DOS, Windows, or UNIX.

Like a general-purpose operating system, the Realtime ETS Kernel provides the following features and capabilities through Win32- and WinSock 1.1-compatible APIs:

- Memory allocation
- Console I/O
- Host file I/O
- System timer services
- Structured exception handling
- Multi-threading and synchronization
- TCP/IP networking
- Local file I/O
- Loading and using DLLs
- Other miscellaneous APIs

While the Realtime ETS Kernel does provide many features of a general purpose operating system, it is by design an **embedded** operating system. As an embedded operating system, the Realtime ETS Kernel has additional characteristics specific to embedded computer applications:

- The Realtime ETS Kernel is scaleable. You can choose which operating system features you want to include in your embedded application. If your application does not need a particular feature or subsystem, you can exclude it to reduce the memory footprint of the Realtime ETS Kernel.
- The Realtime ETS Kernel guarantees your application the capability of responding to real-world events in real time.
- The Realtime ETS Kernel provides many additional APIs for taking over interrupt vectors, logging events, configuring the system, retrieving status information and debugging realtime applications.
- The Realtime ETS Kernel operates on physical memory only. In order to guarantee realtime responsiveness, it does **not** provide demand-paged virtual memory. Embedded applications must be aware of this fact. In particular, an embedded application cannot allocate unlimited memory and must use fixed-size stacks.
- The Realtime ETS Kernel can be customized for non-standard hardware. All hardware-dependent modules in the Realtime ETS Kernel are shipped in source code form and can be modified and replaced as necessary.

THE PROTECTED-MODE ENVIRONMENT

Regardless of which configuration options have been selected, the Realtime ETS Kernel always sets up a protected-mode environment for your embedded program. When your program gets control at the beginning of main(), the kernel has already performed the low-level details of setting up this environment:

- The embedded processor is switched to protected mode.
- The code and data segment selectors are set to a flat address space starting at address 0 and extending to 4 GB.
- Registers and flags are initialized.
- Interrupts are enabled.

- Global variables have been initialized from compressed data in ROM.
- If it has been linked with your program, the C run-time library is initialized.

Having the Realtime ETS Kernel automatically perform this initialization greatly simplifies development of programs for embedded systems. For example, your program no longer has to specify and manage system resources such as Descriptor Tables.

C RUN-TIME LIBRARY SUPPORT

As listed in Appendix B, "Supported C Run-Time Library Routines," most of the C run-time library compiler is supported by the Realtime ETS Kernel. In general, unsupported routines are those needing a resource not likely to be present on an embedded system.

In general, behavior of C run-time library routines under the Realtime ETS Kernel is the same as under Windows. There are, however, some differences:

- By default, the application gets all available RAM. Also, EtsCustomGetMemPool() can be used to size and locate the heap dynamically at run time.
- The environment and command line are hardwired at build time. Please see "Host Command Line and Environment" below for more information.
- File I/O is supported over the communications cable to the host when running in WaitHost mode, and locally on the embedded system if the local file system library has been linked with the application. Please see "Host/Local File System" for details.
- Screen/keyboard I/O are supported over the communications cable to the host when running in WaitHost mode, and locally on the embedded system if the embedded processor has a screen and keyboard. The local console is supported in both WaitHost and NoWaitHost mode.
- Timer services are supported on the target via the timer driver.

The EtsCustom functions and other ETS functions are documented in Appendix A.

Using the C run-time library can greatly simplify program development. However, as with most time-savers, there is an associated cost. In this case, this cost is the increased size of your embedded program. If you don't need the functionality provided by

the C run-time library, you can write a program with an assembly language entry point that calls C code, and then not use any C run-time library calls in your C code.

HOST/LOCAL FILE SYSTEM

The Realtime ETS Kernel supports file operations on both the host and target systems. File operations are performed using the regular C run-time library functions — for example, fprintf() is used for formatted output to a file and fread() is used for unformatted input from a file.

The host file system is available only when running in WaitHost mode and is automatically included when you link your program. This is particularly useful during development, when it is often more convenient to manipulate files on the host system. You can also use a host file to simulate data received from an input device. If your stand-alone embedded system will eventually have a file system, you can use the host file system for initial development of your application, switching over to the local file system at the appropriate time.

The local (target) file system is available in both WaitHost and NoWaitHost modes. This file system is an MS-DOS compatible FAT file system, with the following features:

- Support for FAT12, FAT16, and FAT32 formats
- Support for a variety of disk types, including IDE (both CHS and LBA formats), floppy, PC Card ATA (both flash and rotating media), M-Systems Flash, and RAM Disk

The function EtsSelectFileSystem() selects whether file operations that take a file name (e.g., opening, creating, or deleting) are directed to the local file system or to the host file system. There is no limit to the number of times EtsSelectFileSystem() can be called, so your program can toggle file I/ O operations between the development host and embedded target as desired. Once a file is open, its file handle identifies whether the file resides locally or on the development host.

The default file system selection (before the first call to EtsSelectFileSystem()) is local if the local file system is linked in, otherwise host. If neither file system is available, then file operations return an error.

When a new thread is created, the new thread uses the system default file system as described above. If file operations are to be directed to a specific file system, you should call EtsSelectFileSystem() before making any calls that use the file system.

TCP/IP AND WINSOCK 1.1 NETWORKING

The ETS Realtime Kernel includes a built-in TCP/ IP stack with support for a subset of the WinSock 1.1 API. ETS TCP/ IP supports Ethernet, SLIP, and PPP network connections. The following protocols are supported:

Application Protocols	Network Protocols
File Transfer Protocol (FTP)	ARP
Finger	BOOTP
Hypertext Transfer Protocol (HTTP)	DNS
Simple Mail Transfer Protocol (SMTP)	Ethernet
	ICMP
	IP
	PPP
	RARP
	SLIP
	TCP
	UDP

Chapter 7, "Network Programming with ETS TCP/ IP," describes ETS TCP/ IP and WinSock support in more detail.

FLOATING-POINT EMULATION

Most compilers do not provide floating-point emulation libraries. Rather, they depend on the operating system, in this case the Realtime ETS Kernel, to provide this service.

By default, the Realtime ETS Kernel does not include floating-point emulation. If your embedded program uses floating-point operations and the target computer does not have a coprocessor, you must link the emulation library with your program. If the embedded system has a

coprocessor, it will be used for all floating point operations even if the emulation library is present.

COMMUNICATIONS WITH THE DEVELOPMENT HOST

The communications option (serial or parallel) of the Realtime ETS Kernel supplies much of the functionality of the WaitHost run mode. This option allows you to debug your program with Developer Studio and provides access to the host keyboard, screen, and file system.

You must connect the ports of the development host and embedded target systems with a LapLink cable. You may connect a serial port on the development host to a serial port on the embedded target or you may connect a parallel port on the development host to a parallel port on the embedded target. You may **not** connect a serial port on one system to a parallel port on the other.

ACCESSING MEMORY MAPPED DEVICES

The Realtime ETS Kernel sets up the descriptor for the 32-bit data segment with a base of 0, and a size of 4 GB. With the paging unit shut off, all addresses are physical addresses. Thus your program can directly access all physical memory in the embedded system, including memory mapped devices, with a 32-bit pointer.

If, for example, the target system has a video display screen, your program can write directly to the physical address of the screen.

HOST COMMAND LINE AND ENVIRONMENT

Embedded programs can access their command line and environment variables using standard C programming techniques. Command line arguments are accessed through the traditional *argc* and *argv* parameters to main(). Environment variables can be accessed through the getenv() and _putenv() functions or through the *envp* parameter to main(). The effects of _putenv() are analogous to those in MS-DOS: calling _putenv() only changes the environment for the duration of the embedded program. No changes are made to the host environment.

When the Realtime ETS Kernel is running in WaitHost mode, the command line contains the name of the program and any arguments

following it. For Developer Studio, these arguments are specified in the Project Settings dialog and *argv[0]* is the name of your embedded program.

The environment contains all the environment variables defined on the host when Developer Studio launched your program.

When running in NoWaitHost mode, the Realtime ETS Kernel calls the EtsCustomGetCommandLine() and EtsCustomGetEnvStrings() functions to specify default values for the command line and environment strings. The default functions just initialize the command line to "ETS dummy command line" and the environment to "ETS=dummy". If these default values are not adequate for your program, you can write replacement functions that supply appropriate values.

REPLACEABLE CODE

As mentioned earlier, the Realtime ETS Kernel and the ETS libraries linked with your application can be customized to meet the needs of your embedded hardware. In general, there are two kinds of customizations: those implemented via link-time switches and options and those implemented via replaceable code modules. If the Realtime ETS Kernel includes the necessary code (a floating-point emulation library, for example), you can specify the customization at link time, most easily by using the Visual System Builder. If, however, the code appropriate to your hardware is not available, you can replace the applicable module in the Realtime ETS Kernel or linkable libraries with code that targets your system.

Appendix A identifies all the replaceable modules in the Realtime ETS Kernel and libraries.

BUILDING AND USING DYNAMIC LINK LIBRARIES (DLLS)

The Realtime ETS Kernel supports dynamic link libraries (DLLs). Normal subroutine libraries are included in the .EXE file that uses the library. In contrast, DLLs are libraries of code and data that are not included in the .EXE file; instead, the .EXE links to the DLL while it is running.

A DLL is a Microsoft-defined executable file containing functions that are available to other programs. DLLs are used extensively in Windows 95 and Windows NT, as well as OS/ 2. The Realtime ETS Kernel brings DLLs to embedded systems!

DLLs provide an ideal mechanism for giving your application an add-in (user-extension) facility. Third-party developers would package their add-ins as DLLs. Because the functions in a DLL are also accessible via their ASCII string names, accessing new services in a DLL is easy: no more messy glue routines. And again because functions in a DLL are accessible via their ASCII string names, this facility is perfect for any type of program that is user-extensible.

The benefits of DLLs are:

- More modular software development
- Not tightly linked to compiler or language
- Overlays for protected mode
- ASCII named functions for linking at run time

In the Realtime ETS Kernel, DLLs are loaded into RAM at run time by calling LoadLibrary().

INTERRUPTS AND EXCEPTIONS

There are three types of interrupts This chapter describes how these interrupts are handled by programs using the Realtime ETS Kernel:

- Hardware interrupts (caused by external hardware)
- Software interrupts (caused by executing an INT instruction)
- Processor exceptions (generated by the processor when it detects certain programming errors)

Two "special" hardware interrupts are the timer (IRQ0 for PC/ AT-compatible machines) and keyboard (IRQ1). The Realtime ETS Kernel includes replaceable drivers for these hardware devices in source form. Before writing your own driver for one of these devices, consider whether the ETS driver could meet or be modified to meet your needs.

If your ETS application needs to handle processor exceptions, you may want to consider the structured exception handling mechanism provided by Visual C++.

ETS Kernel Interrupt Processing

The Realtime ETS Kernel creates Interrupt Descriptor Table (IDT) entries (or vectors) for all 256 interrupts. Because each table entry is an interrupt gate, interrupts are disabled when the handler starts to execute. On most PC/ AT systems, hardware interrupts IRQ0 through IRQ7 are mapped to INT 8 through INT 0Fh; hardware interrupts IRQ8 through IRQ15 are mapped to INT 070h through INT 077h. Replaceable modules in the ETS Monitor control this mapping.

The Realtime ETS Kernel uses the following interrupts:

- Processor exceptions are used for debugging.
- One hardware interrupt (IRQ0 for PC/ATs) is used by the optional timer driver.
- One hardware interrupt (IRQ1) is used by the optional keyboard driver.
- Software interrupt INT FEh is used to provide kernel services.
- Software interrupt INT FFh is used to provide host services.

Additionally, some other hardware interrupts are used by the Realtime ETS Kernel for components such as the local file system, the ETS PC Card Support Package and the ETS TCP/ IP stack.

INSTALLING AN INTERRUPT HANDLER IN YOUR APPLICATION

An interrupt handler is a section of code that is automatically executed by the CPU when a particular interrupt occurs. This code which fields and processes an interrupt is often called an interrupt service routine or "ISR." The terms "interrupt handler", "interrupt service routine" and "ISR" will be used interchangeably throughout this section.

Before installing an interrupt handler in your ETS application, you should first spend a little bit of time thinking about the software architecture you are going to use. You will need to choose a strategy for hooking the interrupt, and a strategy for managing preemption, reentrancy and interrupt latency. The strategy you choose will be affected by the type of interrupt you are hooking: a software interrupt, processor exception, or a hardware interrupt.

For a software interrupt or processor exception, the architecture may be quite simple because the options and trade-offs are few. Software interrupts occur only when issued by your application. Processor exceptions usually result from an error in the application. Both of these events are synchronous — they only occur when generated by application code. Thus reentrancy is only a problem if your application can generate multiple simultaneous software interrupt requests, or if your processor exception handler itself generates another processor exception.

The recommended way for an application to handle a software interrupt is by installing a C-language interrupt handler.

There are two ways for an application to handle processor exceptions:

- You can use the facilities for structured exception handling, provided by Visual C++.
- You can handle them directly by installing a C-language exception handler.

Hardware interrupts are, by nature, more complicated. They occur asynchronously and are usually arbitrated by a hardware device called a programmable interrupt controller (or PIC). Because hardware interrupts usually represent some sort of "real-world" happening, they often arrive with realtime service and response requirements. If you are including the ETS Realtime Multi-Tasker in your application, you will also have to choose whether you want to service your device directly in the hardware interrupt handler, or signal a driver thread that will service the device when it wakes up.

A few minutes spent thinking about the overall strategy for your hardware interrupt handler will likely save you a lot of time down the road when bringing up your finished driver.

Before choosing a software architecture for your hardware ISR, we recommend that you do the following:

- Understand the realtime requirements of the hardware device you will be driving and how they relate to the requirements of the other hardware devices in your target system. (For example: a software architecture that spends 10 milliseconds with interrupts disabled in a driver interrupt service routine would not be suitable if there were another device in the system that required its interrupt be serviced every 5 milliseconds.)
- Read all the ToolSuite documentation about interrupts to understand your options for hooking the interrupt and managing reentrancy, latency, and the programmable interrupt controller (or PIC). This documentation includes annotated sample programs and references to ETS drivers that illustrate different architectures for different types of interrupts.

Once you've digested all this information, you'll be ready to design the handler for your application. Again, some recommendations:

- If possible, use a C function as your interrupt handler. This is the simplest way to hook an interrupt and should suffice for all but the highest frequency interrupts that must avoid the overhead of the Realtime ETS Kernel umbrella handler.

- If possible, start with an ETS sample program or driver. These programs and drivers have been extensively tested by Phar Lap and are known to work.
- If you have any questions about the architectural choices for your interrupt handler(s), feel free to contact Phar Lap's technical support department for advice. Phar Lap's support engineers have the benefit of years of experience with the Realtime ETS Kernel and can recommend a design that will meet your needs as simply as possible.

Structured Exception Handling

Visual C++ includes a powerful language construct called "structured exception handling." The construct includes four keywords:

- __try
- __except
- __finally
- __leave

and two intrinsic Win32 functions:

- GetExceptionCode()
- GetExceptionInformation()

The syntax of a structured exception handler is:

```
__try
{
     body, or guarded code
}
__except (exception-filter)
{
     exception handler
}
```

These try-except statements can be nested to any depth. If any exception occurs while executing in the __try body, the exception-filter expression is evaluated.

The exception filter can use the intrinsic functions GetExceptionCode() to obtain a code identifying the exception that occurred, and

GetExceptionInformation() to obtain detailed information about the program's registers when the exception occurred. The exception-filter expression can invoke a function, so the filter can be quite elaborate, including changing register values for the program before continuing from the exception.

REALTIME ETS KERNEL DEVICE DRIVERS

The Realtime ETS Kernel contains drivers for many PC/ AT-compatible peripherals on the embedded target. Target systems that include one of these supported peripherals can use the driver shipped with the Realtime ETS Kernel — in most cases, without making any changes to the driver. Your application can then access the device through the high-level Win32 or C run-time library functions. The Realtime ETS Kernel handles all the details of seamlessly integrating the driver into the Win32 API layer presented to the application.

The Realtime ETS Kernel includes built-in support for the following types of device drivers:

Driver Type	**Description**
Timer	The timer device driver provides time-of-day and timer interrupt services. The timer driver is required by the Realtime ETS Kernel.
Screen	The screen device driver provides services for writing ASCII text to the display.
Keyboard	The keyboard device driver provides services for receiving and processing keystrokes, and processing keyboard interrupts.
Disk	The local file system contains block device drivers for several different types of disk devices.
Network	The ETS TCP/ IP stack contains Ethernet drivers for several different types of Ethernet interfaces. It also contains a 16550-compatible serial driver for SLIP/ PPP connections.

Driver Type	Description
PC Card	The ETS PC Card Support Package contains a device driver for Intel 82365-compatible PC Card host controllers. It also contains enablers for several popular PC Card devices.
Serial	Although the Realtime ETS Kernel does not include direct support for serial devices through the Win32 communications APIs, it does include several serial device drivers in source form.

Because the timer driver is required, it is installed by default. The other drivers are all optional and are installed only if specifically mentioned.

The drivers distributed with the Realtime ETS Kernel are for the PC/ AT hardware architecture. Like all hardware-dependent code in the Realtime ETS Kernel, they are distributed as both source and object code. If you want to use these drivers, but the device in your target system is not PC/ AT-compatible, you must modify the source code to these drivers as appropriate.

ETS PC CARD SUPPORT PACKAGE

Because they're small, portable and interchangeable, PC Cards (formerly called PCMCIA Cards) are a convenient way to add functionality to your embedded system using standard connections and components. The ETS PC Card Support Package provides the software needed to use these devices.

The ETS PC Card Support Package is a set of libraries and source code needed to use PC Card ATA Disks, Ethernet Adapters, Serial Ports, and Modems with your embedded target. The libraries implement key functions from the PC Card Standard, as well as some functions specific to the Realtime ETS Kernel. The source code includes enablers that recognize specific PC Cards and some sample programs. We have included enablers for most PC Cards. However, new cards are coming on the market all the time and you may have a card for which we have yet to write an enabler. Source code is provided for the enablers, to be used as a starting point for writing your own.

Note that while the PC Card Standard supports linear flash memory devices (which are not ATA compatible) , this type of card is not supported by the ETS PC Card Support Package.

PRIORITY INVERSION AVOIDANCE

Priority inversion is a scheduling anomaly that can occur when a thread is waiting on an object held or owned by a lower-priority, unrunnable thread. This can lead to unexpected, non-deterministic behavior in your embedded program. The following scenario illustrates the problem.

- Thread H is a high priority thread that is waiting on resource X.
- Thread L is a low priority thread that currently owns resource X.
- Thread M is a medium priority thread that is runnable.
- Thread M will run forever, effectively becoming higher priority than Thread H. Thread L will never get a chance to run, thus never relinquishing ownership of resource X, and never enabling Thread H to run.

Priority inversion will never happen under the Realtime ETS Kernel. The Realtime ETS Kernel includes code that watches for such situations and takes actions to avoid the priority inversion. If a thread is holding a mutex or critical section object (resource X in the scenario above) and a higher priority thread is waiting for that object, the priority of the owning thread is immediately increased to that of the waiting thread. As soon as the owning thread relinquishes ownership of the object, its priority is returned to the original level. The Realtime ETS Kernel does not perform this temporary priority adjustment for events and semaphores, which cannot be "owned" by a particular thread.

The view presented above is a bit simplistic, but it is important that you understand the basic concepts first. Suppose the lower priority thread owns several objects, each of which is being waited on by a different thread and all of the waiting threads have different priorities. In this situation, the simple case is generalized as follows: a thread's temporarily adjusted priority is the maximum of it's original priority and that of the highest priority thread waiting on an object it owns.

Let's look at another example to illustrate what happens:

1. Thread L owns objects X1, X2, and X3. The priority of Thread L is 5.
   - Thread H1 is waiting on object X1. The priority of Thread H1 is 3.
   - Thread H2 is waiting on object X2. The priority of Thread H2 is 7.
   - Thread H3 is waiting on object X3. The priority of Thread H3 is 9.
2. The priority of Thread L is temporarily set to 9.
   - Thread L releases ownership of X1. Thread H1 is waiting on X1, but its priority is lower than the original priority of Thread L so the Realtime ETS Kernel makes no adjustments.
   - Thread L releases ownership of X3.
3. The priority of Thread L is adjusted to 7. This is the same as Thread H2, the highest priority thread waiting on an object owned by Thread L.
4. Thread H3 takes ownership of X3 and runs at priority 9 until it is no longer runnable.
5. Thread L runs at priority 7.
   - Thread L releases ownership of X2.
6. The priority of Thread L is returned to its original level of 5. At this point, all threads have returned to their original priority levels.

The Realtime ETS Kernel automatically prevents priority inversion from occurring, with no action on your part.

REALTIME ETS KERNEL VERSION 9.1 MEMORY REQUIREMENTS

The Realtime ETS Kernel contains a number of customization and configuration options. One reason for this architecture is to minimize the memory required. Your embedded program loads only the kernel features that are actually used, reducing the overhead.

The following table summarizes the memory requirements of the various Realtime ETS Kernel components.

NOTE: The Realtime ETS Kernel supports embedded systems that do not use a ROM. For memory requirements on these systems, just add the ROM and RAM figures together.

Component	ROM (CODE)	RAM (DATA)

Base ETS Kernel (single task)	**min**	**max**	**min**	**max**
ETS Monitor	16K	32 K	12 K	24 K
ETS Kernel Libraries	32 K	50 K	5 K	10 K
Total for Base Kernel	48 K	82 K	17 K	34 K
Multithread support*	28 K		16 K	
Structured exception support	3 K		< 1K	
Floating-point emulator	23 K		<1 K	
Local file system	42 K		28 K	
DLL Loader	10 K		4 K	
PC Card support	32 K		16 K	
M-Systems Flash support	20 K		4 K	
TCP/IP minimal stack (2 sockets)**	100 K		68 K	
TCP/IP typical stack (10 sockets)**	100 K		100 K	
TCP/IP server stack (50 sockets)**	100 K		260 K	
SLIP/PPP (with serial driver) driver	20 K		6.5 K	
Ethernet driver (approx.)	4 K		1 K	

* Minimum sizes to create a multithreaded Realtime ETS Kernel (103 K). Minimum kernel starts at 75 K.

** The default number of sockets is 10.

Minimal C++ Run-Time Libraries

	ROM Size	**RAM Size**	**RAM only System**
Microsoft Visual C++ ver. 5.0	14 k	4 k	19 k
Borland C++ ver. 5.0	20 k	4 k	24 k

Chapter 7

Network Programming with ETS TCP/IP

The Realtime ETS Kernel provides robust networking capabilities through the popular TCP/ IP and WinSock 1.1 protocol specifications. A properly equipped embedded system built with TNT Embedded ToolSuite can thus run network applications over the Inter/ intranet, or even act as a network server.

Standard application-level protocols are supported, including HTTP and FTP. Network connections are supported for Ethernet, SLIP, CSLIP, and PPP.

The Realtime ETS Kernel supports the WinSock 1.1 API. WinSock 1.1 provides an application interface to the TCP/ IP stack. The WinSock API was designed for Microsoft Windows, and is closely based on the popular UNIX networking API known as Berkeley Sockets.

The ETS TCP/ IP stack supports the following popular Ethernet controller chips:

SMC	8003, 8216, 8416, 91C92, and 91C94
3COM	3C509
Novell	NE2000 and compatibles
Digital	2104x and 2114x

The source code for these drivers is included with ToolSuite, making it easier for you to write your own driver if your card is not one of those supported.

The ETS TCP/ IP stack also supports the SLIP, CSLIP, and PPP serial IP protocols over an 8250, 16450, or 16550 UART. The source code for the serial drivers used by the PPP/ CSLIP driver is also included with ToolSuite, making it easier for you to write your own driver if you use a different serial chip.

In addition, TNT Embedded ToolSuite includes the Phar Lap MicroWeb Server, a collection of libraries and plug-ins linked with your

application to create an embedded Web server than runs under the Realtime ETS Kernel on your target system. The MicroWeb Server is described later in this chapter.

NETWORK PROTOCOLS

The ETS TCP/ IP stack supports several network protocols, used for a variety of applications:

Protocol	Typical Application
ARP	translate Internet address to Ethernet address
BOOTP	used to auto configure network devices
DNS	domain name lookups
Ethernet	local area network
Finger	get information from remote computer
FTP	transfer files between computers
HTTP	transfer Web pages
ICMP	network management
IP	basic network protocol used by TCP and UDP
PPP	point to point protocol, used to transfer data over serial lines
RARP	used to auto configure network devices
SLIP	serial line IP, used to transfer data over serial lines
SMTP	send and receive electronic mail
TCP	stream protocol used by most other protocols
UDP	datagram protocol used by several other protocols

In addition, TNT Embedded ToolSuite includes several network client and server programs:

Network Clients	Network Servers
Finger Client	Finger Server
Email Client	FTP Server
Time Client	HTTP Server

The ToolSuite includes source code for all these programs, except the FTP server.

THE ETS MICROWEB SERVER

The ETS MicroWeb Server is a collection of libraries and plug-ins linked with your application to create an embedded Web server that runs under the Realtime ETS Kernel on your target system. The World Wide Web has enjoyed unprecedented popularity, with Web browsers becoming a de facto standard for user interfaces. The MicroWeb Server components work with the ETS TCP/ IP Stack to implement a Web interface for your embedded application.

Any product that communicates across a network must be compatible with the other products running on that network. For example, the information from a medical instrument that resides in a patient's hospital room may have to be accessed by one or more nurses' stations, the attending physician(s) in their offices, and various hospital departments. Everyone accessing this information could have computers from different manufacturers running different operating systems.

In general, there are two major compatibility problems:

- The machines must all be able to communicate with each other.
- It would be nice if there could be a single user interface that can be used for each hardware/software platform.

The widespread acceptance of TCP/ IP as the network protocol of choice has effectively solved the first problem. This solution has been in wide use since the early 1980s.

It is only recently that a solution to the second problem has become available, based on Web technology. Over the next year nearly every operating system will either ship with a Web browser or have browser technology built right into the system. The Web browser has become the most common graphical user interface (GUI) in use today.

Having a Web browser on every computer is only half of the solution. In order to have bi-directional communications between your embedded product and a user's computer, the embedded product must contain a Web server that can transmit and receive data in the form of HTML pages. (HTML stands for

Hyper Text Markup Language and is the language through which Web servers and browser clients communicate.)

The MicroWeb Server gives you the tools you need to implement the server functionality in your embedded product. The availability of Web browsers on users' computers frees you from the responsibility of implementing a user interface for those computers.

MicroWeb Server Components

The MicroWeb Server is a collection of libraries, plug-ins, and sample programs. The libraries and plug-ins are linked with your application to create a Web server. Each sample program is a functioning Web server application built from the MicroWeb components and can serve as the basis for your embedded Web server.

The MicroWeb Server contains four libraries, each of which is distributed as source code in addition to the linkable library.

The following components comprise the ETS MicroWeb Server:

- **HTTP Server Library**
  This library implements an HTTP/ 1.0 server that complies with RFC 1945.
- **HTML Page Library**
  The functions in this library are the interface used to build HTML pages in memory (finding URIs in the data structures, handling common errors, etc.).
- **HTML On-The-Fly Library**
  HTML stands for Hyper Text Markup Language, the text formatting convention of the World Wide Web. Phar Lap provides a library of functions that let your network programs perform on-the-fly HTML formatting of realtime data for display by a Web server. Using Phar Lap's HTML On-the-Fly library, your embedded system can publish realtime data to a local area network or to the entire Internet community. This library creates the HTML entirely in memory, and does not create any disk files. If you want to save the HTML to disk, your application must open and write the file.

- **HTML On-The-Fly Forms Library**
  This library is analogous to the HTML On-The-Fly Library, but the functions are for creating forms and decoding the returned form data.

There is one MicroWeb Server plug-in, which is distributed only as a linkable library. Unlike traditional libraries, the interface to the plug-in does not contain callable functions. A plug-in is a series of HTML pages that can be accessed from your Web server application using the defined interface.

On-line Debugger Plug-In	This plug-in provides a convenient way to help debug your Web server application.

ToolSuite includes three sample programs that illustrate the use of the MicroWeb Server components. Each program is a functioning stand-alone Web server that can be used as the basis for your application.

- **WEBDISK**
  A sample Web server that serves files off disk. HTTPSERV.LIB is the only MicroWeb Server component used.
- **WEBMEM**
  A sample Web server that constructs HTML pages in memory. All the MicroWeb Server components are used and linked into one executable.
- **WEBSERVE**
  A sample Web server that first attempts to construct HTML pages in memory, then (optionally, depending on a compile-time option) attempts to serve files off disk before returning a page not found error. All the MicroWeb Server components are used and the MicroWeb DLL plug-in technology is used to create a modular server that can accept additional DLL "plug-ins" to construct HTML pages in memory.

Appendix A

The Realtime ETS Kernel API

The Realtime ETS Kernel API contains functions that augment the capabilities provided by the C run-time library. There are three types of services provided:

- Access to functionality in the ETS Kernel
- Access to functionality in the Win32 API
- Alternate functions for the C run-time library

The Realtime ETS Kernel additionally provides APIs that support realtime multitasking and networking, including:

- WinSock APIs
- HTTP Server APIs
- FTP Server APIs
- HTML Page Library APIs
- HTML On-The-Fly APIs

Finally APIs provide a convenient mechanism for customizing the Realtime ETS Kernel to your target hardware:

- Replaceable Driver Functions
- Replaceable Modules in the ETS Libraries
- Replaceable Modules in the ETS Monitor

This appendix presents tables of all the APIs included with or supported by the Realtime ETS Kernel that can be called by an embedded program. Each table presents APIs grouped by the type of function they perform.

Functions from the Win32 API are identified by a ^{W32} following the name.

The ETS Kernel APIs are listed in the following functional groups:

Group
Memory Management Routines
Threads and Synchronization Routines
File Management Routines
DLL Management Routines
Time Management Routines
TCP/IP Device Driver Configuration APIs
Event Logging Routines
Console Routines
Interrupt Control Routines
Process-Related Routines
Miscellaneous Routines
Windows Sockets APIs
C Run-Time Library Alternate Functions
HTTP Server APIs
FTP Server APIs
PC Card APIs
PCI Bus APIs
Porting Routines

MEMORY MANAGEMENT ROUTINES

C Routine	Description
GetProcessHeap[W32]	Get handle to process heap
HeapAlloc[W32]	Allocate memory for specified heap
HeapCreate[W32]	Create a heap
HeapDestroy[W32]	Destroy specified heap
HeapFree[W32]	Free memory from specified heap
HeapReAlloc[W32]	Reallocate heap memory
HeapSize[W32]	Return size of specified heap memory block
HeapValidate[W32]	Validate specified heap
HeapWalk[W32]	Walk memory blocks in specified heap
LocalAlloc[W32]	Allocate memory from local heap
LocalFree[W32]	Free local memory block
LocalReAlloc[W32]	Modify size of local memory block
LocalSize[W32]	Get current size of local memory block
VirtualAlloc[W32]	Reserve/commit memory pages
VirtualFree[W32]	Release memory pages
VirtualQuery[W32]	Get info about memory pages

THREADS AND SYNCHRONIZATION ROUTINES

C Routine	Function Description
CreateEvent[W32]	Create an event object
CreateMutex[W32]	Create a mutex object
CreatePipe[W32]	Create an anonymous pipe
CreateSemaphore[W32]	Create a semaphore object
CreateThread[W32]	Create a thread
DeleteCriticalSection[W32]	Release resources used by critical section object
EnterCriticalSection[W32]	Wait for ownership of specified critical section object
EtsCheckISRPriority	Compare Current Thread Priority to ISR Priority
EtsClearISRPriority	Unlock the ETS scheduler

THREADS AND SYNCHRONIZATION ROUTINES. *CONT.*

C Routine	Function Description
EtsDisableThreadStackOvfCheck	Disable Thread Stack Overflow Checking
EtsDisableThreadTimeCounting	Stop Per-Thread CPU Utilization Counting
EtsDumpThreads	Display information about active threads
EtsEnableThreadStackOvfCheck	Enable Thread Stack Overflow Checking
EtsEnableThreadTimeCounting	Start Per-Thread CPU Utilization Counting
EtsEnumerateThreads	Call Specified Function for Each Active Thread
EtsEnumerateThreadTimeCounts	Call Specified Function with Thread ID and Time Count
EtsExitProcess	Notify kernel of process termination
EtsForceThreadFuncCall	Force Function Call when Thread is Scheduled
EtsFreeUnusedThreadBlocks	Give Free Thread Block List Back to Memory Allocator
EtsGetThreadDebugName	Get ASCII name of thread
EtsGetTimeSlice	Get length of multitasking time slice
EtsMoveThreadToFront	Move a thread in front of same-priority peers
EtsSetISRPriority	Lock the ETS scheduler
EtsSetThreadDebugName	Set ASCII name of thread
EtsSetTimeSlice	Specify the length of the multitasking time slice
EtsTestCritSecOwner	Does Currrent Thread Own a Specified Critical Section?
ExitProcess[W32]	Terminate program
ExitThread[W32]	End a thread
GetCurrentProcess[W32]	Return handle for current process
GetCurrentProcessId[W32]	Return process identifier for current process
GetCurrentThread[W32]	Return handle for the current thread

THREADS AND SYNCHRONIZATION ROUTINES (CONT.)

C Routine	Function Description
GetCurrentThreadId[W32]	Return thread identifier of current thread
GetExitCodeThread[W32]	Return termination status of specified thread
GetThreadPriority[W32]	Return priority value for specified thread
InitializeCriticalSection[W32]	Initialize a critical section object
LeaveCriticalSection[W32]	Release ownership of specified critical section object
OpenEvent[W32]	Return handle for existing event object
OpenMutex[W32]	Return handle for existing mutex object
OpenSemaphore[W32]	Return handle for existing semaphore object
PulseEvent[W32]	Release threads waiting on an event object
ReleaseMutex[W32]	Release ownership of the specified mutex object
ReleaseSemaphore[W32]	Release ownership of the specified semaphore object
ResetEvent[W32]	Set event object state to unsignaled
ResumeThread[W32]	Decrement the suspend count for a thread
SetEvent[W32]	Set event object state to signaled
SetThreadPriority[W32]	Set priority value for specified thread
Sleep[W32]	Suspend execution of current thread
SuspendThread[W32]	Suspend the specified thread
TerminateThread[W32]	Terminate a thread
TlsAlloc[W32]	Allocate a TLS (thread local storage) index
TlsFree[W32]	Release a TLS (thread local storage) index
TlsGetValue[W32]	Get value for TLS (thread local storage) index
TlsSetValue[W32]	Store value for TLS (thread local storage) index
WaitForMultipleObjects[W32]	Wait until one or all of several objects is signaled
WaitForSingleObject[W32]	Wait until object is signaled

FILE MANAGEMENT ROUTINES

C Routine	Description
CreateDirectory[W32]	Create a new directory
CreateFile[W32]	Create or open a file
DeleteFile[W32]	Delete file
DosDateTimeToFileTime[W32]	Convert DOS date and time to 64-bit file time
EtsQueryFileHandle	Query file handle
EtsSelectFileSystem	Select host or target file system
FileTimeToDosDateTime[W32]	Convert 64-bit file time to DOS date and time
FileTimeToLocalFileTime[W32]	Convert UTC file time to local file time
FileTimeToSystemTime[W32]	Convert 64-bit file time to system time format
FindClose[W32]	Close specified search handle
FindFirstFile[W32]	Search directory for specified filename
FindNextFile[W32]	Continue file search
FlushFileBuffers[W32]	Commit file buffers to disk
GetCurrentDirectory[W32]	Retrieve current directory
GetDiskFreeSpace[W32]	Get data about specifed disk
GetDriveType[W32]	Identify drive type
GetFileAttributes[W32]	Get attributes for specified file
GetFileInformationByHandle[W32]	Get info about specified file
GetFileSize[W32]	Return size of specified file
GetFileTime[W32]	Get times for creation, last access, and last modification for specified file
GetFileType[W32]	Return file type for specified file
GetFullPathName[W32]	Get full path and filename for specified file
GetLogicalDrives[W32]	Get data for currently available drives
GetLogicalDriveStrings[W32]	Return bitmask representing available disk drives
LocalFileTimeToFileTime[W32]	Convert local file time to UTC file time
MoveFile[W32]	Rename specified file or directory
ReadFile[W32]	Read data from file
RemoveDirectory	Delete empty directory
SetCurrentDirectory	Change directory
SetEndOfFile	Move end-of-file position for specified file
SetFileAttributes	Set attributes for specified file

FILE MANAGEMENT ROUTINES (CONT.)

C Routine	Description
SetFilePointer	Move file pointer for specified file
SetFileTime	Set times for creation, last access, and last modification for specified file
SystemTimeToFileTime	Convert a system time to a file time
WriteFile	Write data to specified file

DLL MANAGEMENT ROUTINES

DLL MANAGEMENT ROUTINES

C Routine	Function Description
FreeLibrary[W32]	Unload a previously loaded DLL
GetModuleFileName[W32]	Return path and file name for file containing module
GetModuleHandle[W32]	Return handle to specified module
GetProcAddress[W32]	Return address of specified DLL function
LoadLibrary[W32]	Load the specified DLL

TIME ROUTINES

C Routine	Description
DosDateTimeToFileTime[W32]	Convert DOS date and time to 64-bit file time
EtsGetRTCTime	Get Current Time from External Real-time Clock
EtsGetTimerPeriod	Get Frequency of Time-of-Day Clock
EtsMarkTimeSlice	Notify scheduler of timer tick
EtsResetSystemTimeFromRTC	Resynchronize In-Memory Time
EtsSetTimerPeriod	Specify frequency of time-of-day clock
FileTimeToDosDateTime[W32]	Convert 64-bit file time to DOS date and time
FileTimeToLocalFileTime[W32]	Convert UTC file time to local file time
FileTimeToSystemTime[W32]	Convert 64-bit file time to system time format
GetLocalTime[W32]	Get current local date and time
GetSystemTime[W32]	Get current system date and time
GetTickCount[W32]	Get elapsed time since Windows was started
GetTimeZoneInformation[W32]	Get current time-zone parameters
LocalFileTimeToFileTime[W32]	Convert local file time to UTC file time
QueryPerformanceCounter[W32]	Get current value of high-resolution performance counter
QueryPerformanceFrequency[W32]	Get resoulution of high-resolution performance counter
SetLocalTime[W32]	Set current local time and date
SetSystemTime[W32]	Set current system time and date
SystemTimeToFileTime[W32]	Convert a system time to a file time

TCP/IP DEVICE DRIVER CONFIGURATION APIS

The TCP/ IP Device Driver Configuration APIs are used to obtain configuration information on an Ethernet or PPP/ CSLIP driver, or to override configuration information specified with the Visual System Builder or the CFIGKERN utility.

For PPP/ CSLIP drivers, the configuration APIs are also used to provide needed information, such as a phone number to dial, that cannot be specified with the Visual System Builder or CFIGKERN.

C Routine	Description
EtsTCPBringDeviceDown	Bring Down Device Driver
EtsTCPBringDeviceUp	Initialize Device Driver
EtsTCPConfigureDevice	Set New Configuration for Device Driver
EtsTCPGetDeviceCfg	Get Base Configuration Info for Device Driver
EtsTCPGetDeviceExtendedInfo	Get Extended Configuration Info for Device Driver
EtsTCPGetDeviceHandle	Get Handle for Installed Device Driver
EtsTCPGetDeviceInstanceInfo	Get Device Ethernet Info
EtsTCPGetDeviceStatus	Get Status of Network Device
EtsTCPGetStackCfg	Get Stack Configuration Info
EtsTCPRegisterDevice	Register Device
EtsTCPSetDefaultGateway	Set Default Gateway for PPP/CSLIP Connection
EtsTCPSetDeviceEthernetInfo	Set Device Ethernet Info
EtsTCPSetDeviceInstanceInfo	Set Device Instance Info
EtsTCPSetStackCfg	Set Stack Configuration Info

EVENT LOGGING ROUTINES

C Routine	Description
EtsClearLogEnd	Enable overwriting existing entries in event log
EtsEnableLog	Enable or disable event logging
EtsEnumLogEntries	Return pointer to next logged event
EtsFormatLogEvent	Convert binary event log record to committed ASCII string
EtsGetRealtimeEventLogMask	Return mask for realtime event log
EtsGetSubsysLogFlags	Return events being logged
EtsInitializeLogBuffer	Initialize event logging
EtsLogEvent	Write event to log
EtsLogEventFromBuff	Write event from buffer to log
EtsSetLogEnd	Disable overwriting existing entries in event log
EtsSetRealtimeEventLogMask	Specify mask for realtime event log
EtsSetSubsysLogFlags	Specify events being logged

CONSOLE ROUTINES

C Routine	Description
EtsSelectConsole	Select host or local console
GetConsoleMode[W32]	Get input/output mode for a console
GetConsoleScreenBufferInfo[W32]	Get data about console screen buffer
GetLargestConsoleWindowSize[W32]	Get largest possible size for console window
GetNumberOfConsoleInputEvents[W32]	Get number of unread input records in console input buffer
PeekConsoleInput[W32]	Peek data from console input buffer
ReadConsole[W32]	Read character input from console input buffer
ReadConsoleInput[W32]	Read data from console input buffer
ScrollConsoleScreenBuffer[W32]	Move block of data in screen buffer
SetConsoleCursorPosition[W32]	Set cursor position in specified console screen buffer
SetConsoleMode[W32]	Set mode of console input buffer or screen buffer
WriteConsole[W32]	Write string to console screen buffer
WriteConsoleOutput[W32]	Write character and color attribute data to specified cell block in console screen buffer

INTERRUPT CONTROL ROUTINES

C Routine	Description
_dx_idt_rd	Read IDT descriptor
_dx_idt_wr	Write IDT descriptor
EtsCheckISRPriority	Compare Current Thread Priority to ISR Priority
EtsClearISRPriority	Unlock the ETS scheduler
EtsPicEnable	Enable or Disable an IRQ on the 8259 PIC
EtsPicEOI	Issue an EOI for the Specified Hardware IRQ
EtsPicGetIRQNumber	Get Interrupt Vector Corresponding to Hardware IRQ
EtsRestoreExceptionHandler	Restore Specified Exception Vector
EtsRestoreIDTHandler	Restore Contents of IDT Entry
EtsRestoreInterruptHandler	Restore Specified Interrupt Vector
EtsSaveExceptionHandler	Save Specified Exception Vector
EtsSaveIDTHandler	Save Contents of IDT Entry
EtsSaveInterruptHandler	Save Specified Interrupt Vector
EtsSetExceptionHandler	Install C Function as Exception Handler
EtsSetIDTHandler	Install Function as IDT Interrupt Handler
EtsSetInterruptHandler	Install C Function as Interrupt Handler
EtsSetISRPriority	Lock the ETS scheduler
RaiseException*W32*	Raise exception in calling thread
SetUnhandledExceptionFilter*W32*	Supersede top-level exception handler
UnhandledExceptionFilter*W32*	Pass unhandled exceptions to debugger, or display error message and execute exception handler

PROCESS-RELATED ROUTINES

C Routine	Description
EtsCallExitHandlers	Call registered exit functions
EtsHostGetCommandLine	Get Passed-in Command Line from Host
EtsHostGetCommandLineLen	Return Length of Passed-in Command Line from Host
EtsHostGetEnvSize	Return Size of Environment from Host
EtsHostGetEnvStrings	Return Environment from Host
GetCommandLine*W32*	Get command line for current process
GetEnvironmentStrings*W32*	Get address of environment block for current thread
GetEnvironmentVariable*W32*	Get value of variable from calling process
GetStartupInfo*W32*	Get contents of STARTUPINFO structure
GetStdHandle*W32*	Return handle for standard input, output, error device
SetEnvironmentVariable*W32*	Set value of environment variable
SetStdHandle*W32*	Set handle for standard input, output, error device

MISCELLANEOUS ROUTINES

C Routine	Description
EtsAddSubsysData	Add Record of Internal Data for Remote Debugging
CloseHandle[W32]	Close an open object handle
CompareString[W32]	Compare two strings at specified location
DuplicateHandle[W32]	Duplicate an object handle
EtsDisplayError	Display Error Messages on Host and Target Consoles
EtsGetKernelRunMode	Return run mode configured into kernel
EtsGetSystemInfo	Get configuration information
EtsGetVsbVarsPointer	Get pointer to kernel VSB_VARS structure
EtsRegisterCallback	Register device driver callback
GetACP[W32]	Get ANSI code page ID for system
GetCPInfo[W32]	Get data about specified code page
GetLastError[W32]	Return the last-error code
GetOEMCP[W32]	Get OEM code-page identifier
GetUserDefaultLCID[W32]	Get user default locale ID
GetVersion[W32]	Return operating system version number
InterlockedDecrement[W32]	Decrement specified LONG variable
InterlockedExchange[W32]	Atomically exchange two LONG variables
InterlockedIncrement[W32]	Increment specified LONG variable
IsBadCodePtr[W32]	Determine if current process has access to specified memory address
IsBadReadPtr[W32]	Verify that current process has read access to specified range of memory
IsBadWritePtr[W32]	Verify that current process has write access to specified range of memory
IsValidCodePage[W32]	Determine whether specified code page is valid
RtlUnwind[W32]	Unwind Stack Frames (Used by C runtime libraries to implement structured exception handling.)
SetLastError[W32]	Set last-error code
VkKeyScan[W32]	Translate a character to virtual-key code

WINDOWS SOCKETS APIS

C Routine	Description
accept()	Accept a connection on a socket
bind()	Associate a local address with a socket
closesocket()	Close a socket
connect()	Establish a connection to a peer
gethostbyaddr()	Get host information corresponding to an IP address
gethostbyname()	Get host information corresponding to a hostname
gethostname()	Return the standard host name for the local machine
getpeername()	Get the IP address of the peer to which a socket is connected
getprotobyname()	Get protocol information corresponding to a protocol name
getprotobynumber()	Get protocol information corresponding to a protocol number
getservbyname()	Get service information corresponding to a service name and protocol
getservbyport()	Get service information corresponding to a port and protocol
getsockname()	Get the local IP address for a socket
getsockopt()	Retrieve a socket option
htonl()	Convert a u_long from host to network byte order
htons()	Convert a u_short from host to network byte order
inet_addr()	Convert a string containing a dotted address into an in_addr
inet_ntoa()	Convert a network address into a string in dotted format
ioctlsocket()	Control the mode of a socket
listen()	Establish a socket to listen for incoming connection

WINDOWS SOCKETS APIS (CONT.)

C Routine	Description
ntohl()	Convert a u_long from network to host byte order
ntohs()	Convert a u_short from network to host byte order
recv()	Receive data from a socket
recvfrom()	Receive a datagram and store the source address
select()	Determine the status of one or more sockets, waiting if necessary
send()	Send data on a connected socket
sendto()	Send a datagram to a specific destination
setsockopt()	Set a socket option
shutdown()	Disable sends and/or receives on a socket
socket()	Create a socket
WSACleanup()	Terminate use of Windows Sockets
WSAGetLastError()	Get the error status for the last operation that failed
WSASetLastError()	Set the error code which can be retrieved by WSAGetLastError()
WSAStartup()	Initialize Windows Sockets, using highest compatible version

C RUN-TIME LIBRARY ALTERNATE FUNCTIONS

The following functions are used internally by the Realtime ETS Kernel. Each routine implements the functionality of the similarly-named procedure from the C run-time library. If space is at a premium, you can use these alternate functions to avoid loading the version from the C run-time library.

Additionally, some of these functions provide functionality that is not supported by Visual C++.

C RUN-TIME LIBRARY ALTERNATE FUNCTIONS

C Routine	Description
EtsAtoi	Convert ASCII to Integer
EtsBsearch	Binary Search of Sorted Array
EtsInp	Read Byte from Specified Input Port
EtsInpw	Read Word from Specified Input Port
EtsLtoa	Convert Long to ASCII
EtsMemcpy	Copy Bytes to Memory
EtsMemmove	Copy Bytes to Memory
EtsMemset	Fill Memory Locations
EtsOutp	Write Byte to Specified Output Port
EtsOutpw	Write Word to Specified Output Port
EtsStrcat	Concatenate Strings
EtsStrchr	Find Character in String
EtsStrcmp	Compare Strings
EtsStrcpy	Copy String
EtsStricmp	Compare Strings
EtsStrlen	Determine Length of String
EtsStrncmp	Compare Partial Strings
EtsStrncpy	Copy Partial String
EtsToupper	Convert Character to Uppercase

HTTP SERVER APIS

The HTTPSERV.LIB library implements an HTTP server that complies with RFC 1945. This library is part of the MicroWeb Server. Please note that the HTTP Server has one set of functions it exports and a separate set that it imports.

FUNCTIONS EXPORTED BY THE HTTP SERVER

C Routine	Description
StartWebServer	Initialize HTTP server
StopWebServer	Shut down HTTP server

FUNCTIONS IMPORTED BY THE HTTP SERVER

C Routine	Description
GetURI	Read entity-body for requested URI into memory for a GET or HEAD method
LogRequest	Record each processed HTTP request
PostURI	Process entity-body for a POST method
WebServerError	Report errors in HTTP server
WebServerInfo	Handle MicroWeb Server Messages
WebServerWarning	Handle MicroWeb Server Warnings

FTP SERVER APIS

The FTPSERV.LIB library implements an FTP server that complies with RFC 1945. Please note that the FTP Server has one set of functions it exports and a separate set that it imports.

FUNCTIONS EXPORTED BY THE FTP SERVER

C Routine	Description
FtpStartServer	Initialize FTP Server
FtpStopServer	Shut Down FTP Server

FUNCTIONS IMPORTED BY THE FTP SERVER

C Routine	Description
FtpAuthenticate	Authenticate FTP Client
FtpLogSession	Record Each FTP session
FtpServerError	Report FTP Server Errors

PC CARD APIS

C Routine	Description
EtsCSAccessConfigurationRegister	Access PC Card Socket Register Info
EtsCSAdjustResourceInfo	Identify Resources to the ETS PC Card Support Package
EtsCSGetCardServicesInfo	Return Information About the ETS PC Card Support Package
EtsCSGetConfigurationInfo	Get Description of PC Card Socket and Configuration
EtsCSGetFirstClient	Get Handle of Registered Client
EtsCSGetFirstConfigurationInfo	Get Description of PC Card Socket and Configuration
EtsCSGetFirstTuple	Get First Tuple from CIS
EtsCSGetNextClient	Get Handle of Registered Client
EtsCSGetNextConfigurationInfo	Get Description of PC Card Socket and Configuration
EtsCSGetNextTuple	Get Next Tuple from CIS
EtsCSGetTupleData	Get Data Associated with Returned Tuple
EtsCSGetStatus	Get Current Status of PC Card and Socket
EtsCsIsACard	Query Card Information Structure on PC Card
EtsCSMapMemPage	Map Memory Area of PC Card into Window
EtsCSModifyConfiguration	Modify Configuration of a PC Card and Socket
EtsCSModifyWindow	Change Attributes or Access Speed of a Window
EtsCSParseTuple	Interpret Tuple Data
EtsCSRegisterClient	Register Client with Client Services
EtsCSReleaseConfiguration	Remove Configuration Information for Card
EtsCSReleaseIO	Release I/O Addresses
EtsCSReleaseIRQ	Release Interrupt Request Line
EtsCSReleaseWindow	Release Block of System Address Space
EtsCSReportError	Interpret Tuple Data
EtsCSRequestConfiguration	Configure PC Card and Socket

PC CARD APIS (CONT.)

C Routine	Description
EtsCSRequestIO	Allocate I/O Addresses
EtsCSRequestIRQ	Reserve Interrupt Request Line
EtsCSRequestWindow	Assign Window to PC Card and Socket
EtsCSValidateCIS	Validate Card Information Structure (CIS)
EtsPcCardGetATACount	Return Number of PC Card ATA Disks in System
EtsPcCardGetATA	Get Configuration of PC Card ATA Disk
EtsPcCardGetSerialPort	Get Configuration of Serial Port PC Card
EtsPcCardGetSerialPortCount	Return Number of Serial Port PC Cards in System

PCI BUS APIS

PCI BUS APIS

C Routine	Description
EtsPCICall	Directly Access PCI BIOS
EtsPCIFindDevice	Query PCI BIOS About a Specific Device
EtsPCIFindDeviceByClass	Query PCI BIOS About a Specific Device Type
EtsPCIInit	Check for Presence of PCI BIOS and 32-bit Entry Point
EtsPCIReadCfgByte	Read Byte from PCI Configuration Register
EtsPCIReadCfgDWord	Read Double Word from PCI Configuration Register
EtsPCIReadCfgWord	Read Word from PCI Configuration Register
EtsPCIReadConfig	Read from PCI Configuration Register
EtsPCIWriteCfgByte	Write Byte to PCI Configuration Register
EtsPCIWriteCfgDWord	Write Double Word to PCI Configuration Register
EtsPCIWriteCfgWord	Write Word to PCI Configuration Register
EtsPCIWriteConfig	Write to PCI Configuration Register

PORTING ROUTINES

REPLACEABLE DRIVER FUNCTIONS

C Routine	Description
EtsCustomGetKeyboardDriver	Initialize Keyboard Driver
EtsCustomGetScreenDriver	Initialize Screen Driver
EtsCustomGetTimerDriver	Initialize Timer Driver

REPLACEABLE MODULES IN THE ETS LIBRARIES

C Routine	Description
EtsCustomAdjustResourceInfo	Specify Memory and IRQ Resources to ETS PC Card Support Package
EtsCustomAdjustResourceInfo2	Specify IO Resources to ETS PC Card Support Package
EtsCustomAlloc	Allocate heap memory at a specified location
EtsCustomCalloc	Allocate heap memory
EtsCustomClearCoProcesor	Clear coprocessor error
EtsCustomEmuInit	Initialize Floating-Point Emulator
EtsCustomExitProcess	Terminate the Process
EtsCustomFree	Release memory
EtsCustomFSExit	Shut Down Target File System
EtsCustomFSInit	Initialize Target File System
EtsCustomGetCommandLine	Get Host's Command Line
EtsCustomGetDriveType	Return Type of Specified DiskDrive
EtsCustomGetEnvStrings	Get Host's Environment Strings
EtsCustomGetFloppyType	Get Floppy Type
EtsCustomGetKeyboardDriver	Initialize Keyboard Driver
EtsCustomGetMemPool	Return Available Memory Ranges
EtsCustomGetScreenDriver	Initialize Screen Driver
EtsCustomGetTCPIPCfg	Get Internet Protocol Configuration
EtsCustomGetTimerDriver	Initialize Timer Driver
EtsCustomGetTimeZone	Get Time Zone
EtsCustomGetTimeZoneInformation	Get Current Time Zone Parameters
EtsCustomInstallFPUExceptionHandler	Install FPU exception handler
EtsCustomRemoveFPUExceptionHandler	Remove FPU exception handler
EtsCustomSetTCPIPCfg	Override Configured TCP/IP Settings
EtsCustomTCPInit	Install Network Device Drivers

REPLACEABLE MODULES IN THE ETS MONITOR
(16-BIT MONITOR FUNCTIONS)

C Routine	Description
EkCustomBiosInit	Perform initialization that uses the BIOS
EkCustomBreakDisable32	Disable communications break interrupt
EkCustomBreakEnable32	Enable communications break interrupt
EkCustomBreakInitialize	Initialize break interrupt handler
EkCustomClear32NMI	Clear source of non-maskable interrupt
EkCustomClearNMI	Clear source of non-maskable interrupt
EkCustomCommBreakOff	Turn off line break
EkCustomCommBreakOn	Turn on line break
EkCustomCommClearStatus	Clear latched status indicators
EkCustomCommGetStatus	Get communications device status
EkCustomCommInitialize	Initialize host communications driver
EkCustomCommReadCharacter	Read a message character
EkCustomCommSetSpeed	Set the communications speed
EkCustomCommStartReceive	Set up for start of message reception
EkCustomCommStartSend	Set up for start of message transmission
EkCustomCommWriteCharacter	Write a message character
EkCustomCoprocInit	Initialize floating-point coprocessor
EkCustomDeviceInit	Initialize custom target hardware
EkCustomGetScreenCursor	Get cursor position on target screen
EkCustomHaltMonitor	Stop Monitor Operations
EkCustomLinearFill	Fill memory at linear address
EkCustomLinearRead	Read memory at linear address
EkCustomLinearWrite	Write memory at linear address
EkCustomMaskNMI	Mask non-maskable interrupt
EkCustomMask32NMI	Mask non-maskable interrupt
EkCustomProtCommInitialize	Initialize host communications driver
EkCustomPutChar	Write character to target display adapter
EkCustomRealModeInit	Complete real-mode initialization
EkCustomRunMode	Set kernel run mode
EkCustomSetScreenCursor	Set cursor position on target screen
EkCustomSystemInit	Initialize system-critical hardware

Appendix B

Supported C Run-Time Library Routines

This appendix lists the functions in the C run-time library supported by the Realtime ETS Kernel:

abort
abs
_access
acos
_alloca
asctime
asin
assert
atan
atan2
atexit
atof
atoi
atol
_beginthread
_beginthreadex
bsearch
_cabs
calloc
ceil
_cexit
_c_exit
_cgets
_chdir
_chdrive
_chgsign
_chmod
_chsize
_clear87
clearerr
_clearfp
clock
_close
_commit
_control87
_controlfp
_copysign
cos
cosh
_cprintf
_cputs
_creat
_cscanf
ctime
difftime
_disable
div
_dup
_dup2
_ecvt
_enable
_endthread
_endthreadex
_eof
_exit
exp
_expand
fabs
fclose
_fcloseall
_fcvt
_fdopen
feof
ferror
fflush
fgetc
_fgetchar
fgetpos
fgets
_filelength
_fileno
_findclose
_findfirst
_findnext
_finite
floor
_flushall
fmod
fopen
_fpclass
_fpieee_flt
_fpreset
fprintf
fputc
_fputchar
fputs
fread
free
freopen
frexp
fscanf
fseek
fsetpos
_fsopen
_fstat
ftell
_ftime
_fullpath
_futime
fwrite
_gcvt
getc
_getch
getchar
_getche
_getcwd
_getdcwd*
_getdrive
_getdrives
getenv
_get_osfhandle
_getpid
gets
_getw
gmtime
_heapchk
_heapwalk
_hypot
_inp
_inpd
_inpw

* Supported for local file I/O, not for host

isalnum
isalpha
__isascii
_isatty
iscntrl
__iscsym
__iscsymf
isdigit
isgraph
islower
_isnan
isprint
ispunct
isspace
isupper
isxdigit
_itoa
_j0
_j1
_jn
_kbhit
labs
ldexp
ldiv
_lfind
localeconv
localtime
log
log10
_logb
longjmp
_lrotl
_lrotr
_lsearch
_lseek
_ltoa
_makepath
malloc
_matherr
__max
_memccpy
memchr
memcmp
memcpy
_memicmp
memmove
memset
__min
_mkdir
_mktemp
mktime
modf
_msize
_nextafter
_onexit
_open
_open_osfhandle
_outp
_outpd
_outpw
perror
_pipe
pow
printf
putc
_putch
putchar
_putenv
puts
_putw
qsort
rand
_read
realloc
remove
rename
rewind
_rmdir
_rmtmp
_rotl
_rotr
_scalb
scanf
_searchenv
setbuf
_setjmp
_setmode
setvbuf
sin
sinh
_snprintf
_sopen
_splitpath
sprintf
sqrt
srand
sscanf
_stat
_status87
_statusfp
strcat
strchr
strcmp
_strcmpi
strcpy
strcspn
_strdate
_strdup
_strerror
strerror
strftime
_stricmp
strlen
_strlwr
strncat
strncmp
strncpy
_strnicmp
_strnset
strpbrk
strrchr
_strrev
_strset
strspn
strstr
_strtime
strtod
strtok
strtol
strtoul
_strupr
strxfrm
_swab
tan
tanh
_tell
_tempnam
time
tmpfile
tmpnam
__toascii
_tolower
_toupper
_tzset
_ultoa
_umask
ungetc
_ungetch
_unlink
_utime
va_arg
va_end
va_start
vfprintf
vfwprintf
vprintf
_vsnprintf
_vsnwprintf
vsprintf
vswprintf
vwprintf
_write
_y0
_y1
_yn

Appendix C

Realtime ETS Kernel Performance Measurements

Phar Lap Software has written a number of performance measuring programs for the Realtime ETS Kernel. The programs primarily measure interrupt latency, or the time to interrupt a thread, and context switch times, or the time it takes one thread to yield to another. These measurement programs with their results are described below.

C.1 Measuring Interrupt Latency

The ISRTIME.C program measures the "time to interrupt thread" for the Realtime ETS Kernel on a particular embedded system. This number is often called the "interrupt latency" because it is the length of the delay between when a hardware interrupt is signaled and when the thread that processes the interrupt is awakened by the hardware ISR function.

ISRTIME.C gathers information using the instrumentation code in the default PC/ AT-compatible timer driver shipped with TNT Embedded ToolSuite (LIB\BUILD\PCTIMER.C and PCTIMERA.ASM). This instrumentation code measures the latency for each IRQ0 periodic timer interrupt. If you build a custom timer driver for your target hardware that does not contain the instrumentation to measure interrupt latency, ISRTIME will not display meaningful results. In fact, it will most likely fail to link.

The PC/ AT-compatible timer driver gathers its data as follows:

1. Each time the periodic timer counts down from its preset value and crosses zero, it requests an IRQ0 and automatically resumes counting down from its preset value.
2. The IRQ0 handler is installed directly in the IDT and will interrupt any task executing with interrupts enabled. The IRQ0 handler does a small amount of work (to mask further IRQ0 interrupts and issue an EOI so other interrupts can occur) and then wakes up the timer ISR task by calling

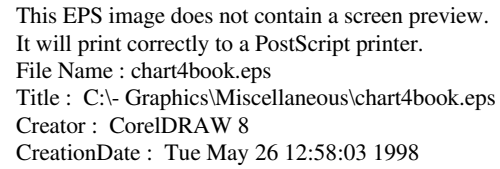
SetEvent() to set the event on which the timer ISR task is waiting. If the interrupted task is lower priority than the timer ISR task, (as will be the case most of the time) the call to SetEvent() will preempt the current thread and immediately schedule the timer ISR task.

3. The timer ISR task is a loop which calls WaitForSingleObject() on an event which gets signaled by the IRQ0 hardware interrupt service routine. Each time the IRQ0 handler signals the event, the timer ISR task wakes up, processes the interrupt, and goes back to sleep inside WaitForSingleObject().
4. Immediately after the timer ISR task returns from WaitForSingleObject(), it calls a function that reads the current value of the countdown timer. By subtracting this value from the timer's preset value, you get an extremely accurate measure of the elapsed time since the periodic timer had its last zero crossing and requested its interrupt. The timer ISR task samples each of these values and keeps enough data to track the maximum, minimum, and average values.

Thus ISRTIME can measure the time to interrupt by calling the private function inside the PC/ AT timer driver which retrieves the current set of interrupt latency statistics.

Because the samples gathered by the timer driver are real-world numbers gathered while performing its assigned duties, you should be able to look at these numbers with a reasonable expectation that you will see similar numbers if your application dispatches tasks to process hardware interrupts.

The following chart shows the average results from the ISRTIME program on four different machines:



The hardware timer ISR and timer ISR task have not been optimized for this test, so you can probably slightly reduce the interrupt latency to your hardware ISR task by minimizing the amount of work the hardware ISR does before calling SetEvent(). (The timer driver masks interrupts, issues an EOI, and calls a function in the Realtime ETS Kernel to update time-slicing data structures before calling SetEvent().)

The timing results can vary somewhat, even though each interrupt follows the exact same code path. If your target hardware has a memory cache (as most x86 boards do), you can often see a 20% to 30% difference between the measured minimum latency and the measured maximum latency. This performance difference is caused by memory wait states which get inserted when the ISR and task dispatch code path is not in the memory cache. This was empirically determined by disabling the memory cache in a 486 target machine. The variance in the data was eliminated when the memory cache was disabled, and the

minimum, maximum, and average latencies converged on the same (though dramatically higher) value. With the memory cache enabled, it is often possible to get a 5% difference in the measured results simply by moving bits of code around in memory to change instruction alignments.

When comparing the performance numbers for the Realtime ETS Kernel with those for other realtime kernels, it is important that you understand exactly what each test is measuring in order to make a fair comparison. Hardware memory caches and test programs that are optimized for a particular hardware configuration can make a big difference in performance between seemingly identical computers.

When designing your embedded application, you should use the measured maximum latency as a worst-case estimate for the time needed to schedule your ISR task. Your application can also call the timer driver (using the same calls as ISRTIME) to measure the interrupt latency for the IRQ0 interrupt under typical conditions for your application.

The measured latency can also be affected when interrupts are disabled by the application. This happens transparently when tasks make system calls to the multitasking scheduler. So, the best way to get a truly worst case number for your application is to make the measurement calls into the timer driver while your application is under heavy load.

C.2 Measuring Interthread Yield Times

YLDTIME.C measures the time cost of both directed and non-directed yields. This program only uses Win32 functions (no Phar Lap APIs), and can thus be used to compare the performance of your program under Windows NT or Windows 95 to that under the Realtime ETS Kernel or the Phar Lap TNT Realtime DOS-Extender systems.

Undirected Yield is the amount of time it takes to switch from one task to another when the current task calls Sleep(). YLDTIME measures this by having two tasks which are both calling Sleep(0) in a loop. Each task records the start time before calling Sleep(0) and records the end time when its Sleep(0) system call returns. So, the elapsed time between each start and end pair is the amount of time it takes one task to enter Sleep(0) and yield control to the other task which returns from Sleep(0).

Directed Yield Up is the amount of time it takes for a lower priority task to wake up a higher priority task which is currently blocked inside

WaitForSingleObject(). YLDTIME measures this by having a high priority task which loops calling WaitForSingleObject() to wait for an event, and a low priority task which loops calling SetEvent() on that same event. The low priority task records the start time before calling SetEvent(), and the high priority task records the end time when its call to WaitForSingleObject() returns. Thus the elapsed time between each start and end pair is the amount of time it takes a thread to enter SetEvent(), wake up and switch to the higher priority thread, and for the higher priority thread to return from WaitForSingleObject().

Directed Yield Down is the amount of time it takes for a higher priority task to enter WaitForSingleObject() and yield control to a lower priority task. YLDTIME measures this with the same two tasks as for the Directed Yield Up measurement, but with different start and end times. The high priority task records the start time before calling WaitForSingleObject() and the low priority task records the end time when its call to SetEvent() (where it was blocked because it woke up a higher priority task) returns. Thus the elapsed time between each start and end pair is the amount of time it takes a thread to enter WaitForSingleObject(), block, and switch control to another thread.

YLDTIME uses the Win32 function QueryPerformanceCounter()to record the starting and stopping times. The calling function saves the value of the high resolution performance counter at the time of the call. When calculating the results of the tests, YLDTIME calls QueryPerformanceFrequency() to determine the frequency of the performance counter. By dividing the number of counts (returned by QueryPerformanceCounter()) by the frequency of the counter, we can determine the number of elapsed seconds. This allows you to measure very small amounts of time with much more accuracy than would be possible by using traditional C run-time library timing functions.

Appendix D

Other Supported Compilers

In addition to Microsoft Visual C++ and Developer Studio, you can also use the following compilers to build ETS programs:

- Borland C++, Version 4.5 or 5.x
- Aonix ObjectAda Real-Time for Intel/ETS
- Microsoft MASM, Phar Lap 386|ASM, or Borland TASM

If you're using Borland C++, you use the command-line compiler to build your program. Once your program is built, you can use the TDEMB cross-developed shell that is part of TNT Embedded ToolSuite. TDEMB turns Borland Turbo Debugger into a debugger for embedded programs. You have access to all the debugging features of Turbo Debugger.

Aonix ObjectAda includes an integrated development environment for building and debugging your program. For more information on ObjectAda, visit the Aonix Web site (http://www.aonix.com/).

You may also write part or all of your program in assembly language. TNT Embedded ToolSuite includes 386|ASM. Assemblers from Microsoft and Borland are also supported.

Index

32-bit environments, vii-viii, x

A

Aonix ObjectAda compiler, 117
APIs, 83-105
  C run-time library alternate functions, 99
  console routines, 93
  DLL management routines, 89
  event logging routines, 92
  file management routines, 88-89
  FTP Server, 101
  HTTP Server, 100
  interrupt control routines, 94
  memory management routines, 85
  miscellaneous routines, 96
  PC Card, 102-103
  PCI Bus, 103
  porting routines, 104-105
  process-related routines, 95
  Realtime ETS Kernel, 60, 84-105
  TCP/ IP device driver configuration, 91
  threads and synchronization routines, 85-87
  time routines, 90
  Win32, 1, 8, 84-105
  Windows sockets, 97-98
application protocols, 63, 78
applications
  developing your own, 6, 21-24
  installing interrupt handler, 68-71
  loading and kernel options, 47
assemblers, 117

B

batch files, 54
_beginthread function, 13
Berkeley sockets, 77
BIOS
  booting from disk, 48-50
  extension, ROM boot method, 51-52
  kernel build options, 45-48, 51-52
  power-on self test (POST), 51
boot drive, 49
boot methods
  BIOS Extension, 51-52
  Boot Jump, 52
  Bootable ETS Kernel, 48-50
  booting from disk, 48-50
  booting from disk, 48-50
  booting from DOS, 52-53
  booting from ROM, 50-52
  bootstrap process, 7-8, 21-24, 45-48
  memory usage, 74-75
  options, 2, 45-53
  selecting via Visual System Builder (VSB), 25-28
  typical development cycle, 21-24
bootable diskettes
  application loading options, 45-48
  boot sector loader, 48
  choosing templates, 25-28
  typical development cycle, 21-24
Bootable ETS Kernel, 21-24, 45-48
BOOTJMP sample program, 52
Bootstrap Protocol (BOOTP), 63, 78
Borland compilers, 117
breakpoints, 34, 35
browsers, 79
building. *See also* compilers
  kernel, options, 45-48, 54
  using Developer Studio, 17-19
  Visual System Builder project, 25-28

C

C functions and interrupts, 68-71. *See also* interrupts
C/ C++ runtime libraries, 2. *See also* libraries

cable, communications, 6, 22-23, 45-48, 64
callback functions, 11
CFIGKERN utility, 28-29
chip set initializations, 52
clients, network, 78. *See also* network programming
COM ports and host communications, 64
command files
  creating your own, 14-17
  Visual System Builder output, 6, 25-28
command line, 14-15, 64-65
communications. *See also* host communications
  changing kernel options with CFIGKERN, 28-29
  custom routines, 54-58
  initializing, 55, 57
  parallel and serial, viii, 6, 64
Compact PCI Systems, 24
compilers
  _beginthread function, 13
  building projects with VSB, 25-28
  building realtime programs, 17-19
  command files, 14-17
  _endthread function, 10, 13
  memory usage, 75
  supported, 7, 117
  typical development cycle, 21-24
  using with Realtime ETS Kernel, 7
configuration
  default hardware, 3-4
  files, 25-28
  kernel, 5-6
console APIs, 93
CPU Modules, 24
critical sections, Realtime ETS Kernel support, 8
CSLIP connections, 77. *See also* network programming
Ctrl-Break, 39
custom kernel, creating bootable diskette, 48-50. *See also* bootable diskettes

D

deadlock, diagnosing, 38-40
DEBUG debugger, 53
debugging
  breakpoints, 34, 35
  building debug version of programs, 17, 19
  cross debuggers, 2, 117
  event logging, 39, 42-44
  in Developer Studio IDE (Integrated Development Environment), 31-36
  multithreaded programs, 38-42
  program resides in ROM on target, 50-52
  sample debug session, 32-36
  Target Port Input/ Output, 36
  Target System Information, 37
  typical development cycle, 6, 21-24
  using Embedded StudioExpress, vii, 5, 36-37
  using ETS Multithread Library, 40-41
defaults. *See also* initialization
  COM1 for serial communications, 64
  development environment, 6
  device drivers, 71-72
  hardware configuration, vii-viii, 3-4
  interrupt handling, 67-71
  floating point emulation software, 63
  thread priority, 8-9
  time slice, 9
  timer tick period, 10
Developer Studio IDE (Integrated Development Environment), vii, 5, 14
  building programs with, 17-19
  debugging with, 31-36
  running programs, 20-21
development cycle
  advantages of embedded systems, 1-2
  building programs, 17-19
  command files, 15, 25
  debugging, 31-36
  default environment, 6
  running programs, 20-21
  typical, 6, 21-24
  Visual System Builder in, 25-28
device drivers, 61, 67, 71-72
  Ethernet, 3, 5, 77
devices, memory mapped, 64
direct screen writes, 64

disk. *See also* bootable diskettes
  boot method and product development, 22-23
  booting from, 48-50
  drivers, 71-72
  flash, 49
  floppy, booting from, 49
  loader, 4, 45-48
DIVBUG sample program, 32-36
DLLs
  building and using, 3, 66
  management APIs, 89
  memory usage, 74-75
DOS boot option, 2, 52-53. *See also* boot methods
drivers. *See* device drivers
dynamic link libraries (DLLs). *See* DLLs

E

EkCustom functions, 54, 105
embedded programs
  advantages of developing, 1-2
  application loading, 45-48
  BIOS extension ROM boot method, 1-52
  booting . *See* boot methods
  deadlock, diagnosing, 38-40
  debugging. *See* debugging
  developing, 3-4, 7-12, 21-24
  distributing, x
  loading over communications cable, 49
  memory, 74-75
  networking, 4-5
  protected-mode environment, 3-4, 7-8, 54-58, 60-61
  Realtime ETS Kernel and, 3-4
  running, 20
  size and C run-time libraries, 62
  TNT Embedded ToolSuite features, viii, 2-3
  typical development cycle, 21-24
  Visual System Builder, 2, 5-6
Embedded StudioExpress, 2, 5
  creating new ETS Workspace, 15-17
  running programs, 20-21
  Toolbar, 5, 15, 36, 37
Embedded StudioExpress Extensions
  Target Port Input/ Output, 36
  Target System Information, 37
_endthread function, 10, 13
environment variables, 25-28, 64-65
error checking. *See also* debugging
  event logging system, 39, 42-44
Ethernet, 3, 5, 63,77. *See also* network programming
  controller chips, 77
  PC Card support, 72
ETS Kernel. *See* Realtime ETS Kernel APIs
ETS Kernel APIs, 83-105
ETS MicroWeb Server, 79-81
ETS Multithread Library, debug version, 40-41
ETS PC Card Support Package, 72
ETS Project Wizard, 2, 5, 14-15
ETS Workspace, 15-17
Ets*() functions, 83-105
event logging, 39, 42-44
event logging APIs, 92
events, Realtime ETS Kernel support, 11
exceptions, 4, 67-71
  Divide By Zero, 32, 35-36

F

file management APIs, 88-89
file system, host/ local, 59, 62-63, 74-75
File Transfer Protocol (FTP), 63, 77, 78
Finger protocol, 63, 78
flags, setting up, 7
flash disk, 49, 72
floating point emulation, 3, 63, 74-75
floppy disks, booting from, 49
Forth-Systeme Modul 386 EX Board, 24, 26
FTP Server APIs, 101

H

handlers, C interrupt, 68-71
hard disk, booting from, 49

hardware
  32-bit x86 embedded systems development, 1, 21-24
  custom, 60
  default configuration, 3-4
  Fourth-Systeme Modul 386 EX Board, 24, 26
  hardware-specific code, 5, 54-58, 104-105
  initializing, 54-58
  Intel 386 EX Evaluation Board, 24, 26
  Intel/ Radisys 386 EX Explorer, 24, 26
  interrupts, 11-12, 67-71
  kernels, 45-48
  measuring interrupt latency, 111-114
  measuring interthread yield times, 114-115
  Microtek Emulator Boards, 26
  NS486SXF Eval Board, 24, 26, 51-52
  PC/ AT compatible Boards, 22-23, 26, 51
  system information, 57
  system requirements, ix-x, 3-4
  targets for ROM-based applications, 22-24
  timer tick period and, 9-10
  VSB templates configured for, 25-26
host communications. *See also* target
  booting from disk, 48-50
  bootstrap process, 7-8, 21-24, 45-48
  I/ O with host, 61-62
  initializing, 55, 57
  kernel build options, 45-48, 54
  kernel initialization process, 54-58
  module, 4, 64
  program load method, 22-23
  services and interrupts, 67-71
  typical development cycle, 6, 22-23
  using Ctrl-Break, 39
  WaitHost mode, 8, 21-23, 45-48
host/ local file system, 59, 62-63, 74-75
HTML (Hyper Text Markup Language)
  On-The-Fly Libraries, 80-81
  pages, 79-81
  support, 79-81
HTTP (Hypertext Transfer Protocol), 42, 63, 77, 78
  HTTP Server Library, 80
  HTTP Server APIs, 100

I

I/ O, 36, 61-62, 71-72
ICMP (Internet Control Message Protocol), 78
Industrial PCs, 24
initialization. *See also* defaults
  chip set, 52
  communications code, 55, 57
  kernel, 54-58
  Realtime ETS Kernel set up, 3-4, 7-8, 21, 54-58
Intel 386 EX Evaluation Board, 24, 26
Intel Pentium Processor Modules, 24
Intel/ Radisys 386 EX Explorer, 24, 26
Internet applications, 4-5. *See also* network programming
interrupts, 67-71
  control APIs, 94
  hardware, using threads to service, 11-12
  Interrupt Descriptor Table (IDT), 67
  interrupt service routines (ISR), 11-12
  latency, measuring, 111-114
interthread yield times, measuring, 114-115

K

kernel. *See also* Realtime ETS Kernel
  application loading, 45-48
  build options, 45-48,
  building with batch files, 54
  changing options with CFIGKERN, 28-29
  collecting system information, 57
  command files, 25-28
  configuration, 5-6
  initialization, 54-58
keyboard driver and I/ O, 4, 61, 71-72

L

LapLink parallel and serial connection, viii, 6, 64
libraries, 7
  C run-time alternate support, 99
  C run-time support, 2. 7, 61-62, 99, 107-109
  C++ and memory, 75
  dynamic link (DLLs), 3, 66, 89
  floating-point emulation 3, 63, 74-75
  MicroWeb Server, 80-81
  replaceable modules, 54-58, 65, 104-105
  thread functions, 8-11
License, Realtime ETS Kernel Run-Time, x
linking
  linker command files, 5-6, 14-15, 25-28
  linker command line, 14-15
  RTHELLO program, 15-17
  using ETS Project Wizard, 14-17
  Visual System Builder (VSB), 25-28
  with LinkLoc, 2, 15
LinkLoc switches
  ETS Project Wizard, 14-15
  Visual System Builder (VSB), 25
loading process, 45-48
local file system, 59, 62-63, 74-75
logging events, 39, 42-44

M

memory
  HTML (Hyper Text Markup Language) On The Fly library, 80
  initializing, 54-58
  management APIs, 85
  mapped devices, 64
  Memory Layout property sheet (VSB), 27
  requirements, ix
  usage and embedded programs, 13-14, 74-75
Microsoft compilers, supported, vii, 1-2, 7, 117
Microtek Emulator Boards, 26
MS-DOS, 2, 3, 52-53
multithreaded applications
  debugging, 38-42
  ETS Multithread Library, 40-41
  Realtime ETS Kernel and, 2, 8-11
  TNT Embedded Tool Suite Realtime Edition, 3
mutex, Realtime ETS Kernel support, 8, 11

N-O

network
  clients, 78
  drivers, 71-72
  servers, 78
network programming
  Berkeley routines, 77
  Ethernet board support, 3, 5,77
  HTML (Hyper Text Markup Language) support, 79-81
  Internet Control Message Protocol (ICMP), 78
  introduction, 4-5, 77-78
  MicroWeb Server, 79-81
  TCP/ IP support, 3, 5, 63, 77-81
  WinSock support, 3, 4, 63, 77, 97-98
network protocols, 63, 78
NoWaitHost mode, 8, 22-23, 45-48
NS486SXF Evaluation Board, 24, 26, 51-52
options, Realtime ETS Kernel, overview, 45-48

P

parallel communications, 64. *See also* host communications
  kernel options, 45
  system requirements, ix-x
PC Card APIs, 102-103
PC Card ATA disks, 3, 49, 72
PC card device drivers, 71-72
PC Card Support Package, 72
PC/ 104 Systems, supported, 22-23
PC/ AT compatible
  BIOS extension method, 1-52
  Boards, 24, 26, 51
  devices detected at initialization, 56

PC/ AT compatible, *cont.*
EkCustom functions, 54, 105
hardware configuration, vii-viii, 3-4
kernel build options, 45-48
RAM-based applications, 21-22
replaceable code, 54-58, 65
ROM-based applications, 21-24
VSB template, 25-28
PCI Bus APIs, 103
PCI Systems, 24
PCMCIA devices. *See* PC Card devices
performance, measuring, 111-115
Phar Lap 386|ASM assembler, 117
Phar Lap software, x-xi
World's Smallest Web Server, x-xi, 4-5, 41
plug-ins, ETS MicroWeb Server, 81
porting APIs, 104-105
ports, direct I/ O, 36
POST processing, 51
PPP connections, 5, 63, 77. *See also* network programming
priority
inversion avoidance, 9, 73-74
of threads, 3, 8-11
process-related APIs, 95
processor exceptions. *See* exceptions
processors, supported, viii
program load method, 22-23
programming environment. *See also* embedded programs
C run-time library support, 7, 61-62, 107-109
device drivers, 71-72
host command line and environment, 64-65
host/ local file system, 59, 62-63, 74-75
I/ O with host, 61-62
interrupts and exceptions, 67-71
memory mapped devices, 64
memory usage, 74-75
networking, 4-5, 63, 77-81
overview, 59-60
protected-mode environment, 3-4, 7-8, 54-58, 60-61
programming environment, *cont.*
replaceable code, 54-58, 65, 104-105
scalable, 60
structured exception handling, 70-71
protected mode, 3-4, 7-8, 54-58
protocols, application and network, 63, 77, 78

R

RAM memory usage, 74-75
RAM-based applications, 21-22
real-mode initialization, 54-58
Realtime ETS Kernel API, 83-105
replaceable modules, 54-58, 65, 104-105
Win32 API functions, 1, 8, 83-105
WinSock 1.1 APIs, 97-98
Realtime ETS Kernel, features, viii, 7-12
realtime program, example, 12-14
redistribution information, x
replaceable code, 6, 54-58, 65, 104-105
ROM
applications, typical development cycle, 21-24
BIOS extension boot method, 1-52
booting from, 22-23, 50-52
loading embedded program into, 22-23
memory usage, 74-75
RTOS (realtime operating system), vii, 1
run mode, 22-23, 48
Run-Time License, x
running with Realtime ETS Kernel, 20-21

S

scheduler, 3, 9-11, 73-74. *See also* multithreaded applications
screen drivers, 4, 71-72
screen I/ O, 61-62, 71-72
semaphores, Realtime ETS Kernel support, 11
serial drivers, 71-72

serial communications, 64. *See also* host communications
  kernel options, 45
  system requirements, ix-x
  used in typical development cycle, 22-23
serial modems, 72
servers, network, 78. *See also* network programming
Simple Mail Transfer Protocol (SMTP), 63, 78
SLIP connections, 5, 63, 77. *See also* network programming
smallest .pharlap.com, x-xi, 4-5, 41
sockets, 77. *See also* network programming
software interrupts, 67
stack size, 13, 74-75
STD-32 Systems, 24
structured exception handling, 70-71
SYMDEB debugger,53
synchronization objects and threads, 8-11
System Information, Target, 37
system requirements, hardware and software, ix-x, 3-4

T

target. *See also* host communications
  bootstrap process, 7-8, 22-23, 54-58
  file system, 62-63
  keyboard and screen drivers, 71-72
  supported hardware, 23
  Target Port Input/ Output, 36
  Target System Information, 37
TCP/ IP support, 3, 5, 63, 77-78, 79. *See also* network programming
  device driver configuration APIs, 91
templates, Visual System Builder (VSB), 25-28
threads
  APIs, 85-87
  debugging multithreaded programs, 38-42
  ETS Multithread Library, debug version, 40-41
  EtsDumpThreads(), 41-42
threads, *cont.*
  event logging system, 39, 42-44
  hardware interrupts and, 11-12
  priority inversion avoidance, 9, 73-74
  priority, 3, 8-11
  programming concepts and, 8-11
  scheduling, 9-11
  ThreadId field, event logging, 39, 42-44
time APIs, 90
time slice, 9-10
timer, 4, 9-11, 61, 71-72
timing. *See* performance
Turbo Debugger, 117

U-V

UART, supported, 77
video memory, 64
Visual System Builder (VSB)
  choosing templates, 25-28
  linker command files, 5-6, 25-28
  Property Sheets, 26-28

W-X

WaitHost mode, 8, 21-23, 45-48
web server
  ETS MicroWeb Server, 79-81
  sample programs, 81
Win32 API support, 1, 8, 83-105
Win32 tools, supported, vii
WinSock, 3, 4, 63, 77, 97-98. *See also* network programming
World Wide Web connections, thread for, 42
World's Smallest Web Server, x-xi, 4-5, 41
x86 CPU Evaluation Boards, 24