RTEMS TI C3x/C4x Applications Supplement

Edition 4.6.0, for RTEMS 4.6.0

30 August 2003

On-Line Applications Research Corporation

On-Line Applications Research Corporation
TEXinfo 2002-11-25.11

COPYRIGHT © 1988 - 2003.
On-Line Applications Research Corporation (OAR).

The authors have used their best efforts in preparing this material. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. No warranty of any kind, expressed or implied, with regard to the software or the material contained in this document is provided. No liability arising out of the application or use of any product described in this document is assumed. The authors reserve the right to revise this material and to make changes from time to time in the content hereof without obligation to notify anyone of such revision or changes.

The RTEMS Project is hosted at `http://www.rtems.com`. Any inquiries concerning RTEMS, its related support components, its documentation, or any custom services for RTEMS should be directed to the contacts listed on that site. A current list of RTEMS Support Providers is at `http://www.rtems.com/support.html`.

Table of Contents

Preface 1

1 CPU Model Dependent Features 3

1.1 Introduction 3
1.2 CPU Model Name 3
1.3 Floating Point Unit 3

2 Calling Conventions 5

2.1 Introduction 5
2.2 Processor Background 5
2.3 Calling Mechanism 5
2.4 Register Usage 5
2.5 Parameter Passing 6
2.5.1 Parameters Passed in Memory 6
2.5.2 Parameters Passed in Registers 6
2.6 User-Provided Routines 7

3 Memory Model 9

3.1 Introduction 9
3.2 Byte Addressable versus Word Addressable 9
3.3 Flat Memory Model 10
3.4 Compiler Memory Models 10
3.4.1 Small Memory Model 10
3.4.2 Large Memory Model 11

4 Interrupt Processing 13

4.1 Introduction 13
4.2 Vectoring of an Interrupt Handler 13
4.2.1 Models Without Separate Interrupt Stacks 13
4.2.2 Models With Separate Interrupt Stacks 13
4.3 Interrupt Levels 14
4.4 Disabling of Interrupts by RTEMS 14
4.5 Interrupt Stack 15

5 Default Fatal Error Processing 17

5.1 Introduction 17
5.2 Default Fatal Error Handler Operations 17

6 Board Support Packages 19

6.1 Introduction 19
6.2 System Reset 19
6.3 Processor Initialization 19

7 Processor Dependent Information Table 21
7.1 Introduction 21
7.2 CPU Dependent Information Table 21

8 Memory Requirements 23
8.1 Introduction 23
8.2 Data Space Requirements 23
8.3 Minimum and Maximum Code Space Requirements 23
8.4 RTEMS Code Space Worksheet 23
8.5 RTEMS RAM Workspace Worksheet 25

9 Timing Specification 27
9.1 Introduction 27
9.2 Philosophy 27
9.2.1 Determinancy 27
9.2.2 Interrupt Latency 28
9.2.3 Context Switch Time 29
9.2.4 Directive Times 29
9.3 Methodology 30
9.3.1 Software Platform 30
9.3.2 Hardware Platform 30
9.3.3 What is measured? 30
9.3.4 What is not measured? 31
9.3.5 Terminology 31

10 BSP_FOR_TIMES Timing Data 33
10.1 Introduction 33
10.2 Hardware Platform 33
10.3 Interrupt Latency 33
10.4 Context Switch 34
10.5 Directive Times 34
10.6 Task Manager 35
10.7 Interrupt Manager 36
10.8 Clock Manager 36
10.9 Timer Manager 36
10.10 Semaphore Manager 37
10.11 Message Manager 37
10.12 Event Manager 38
10.13 Signal Manager 38
10.14 Partition Manager 38
10.15 Region Manager 39
10.16 Dual-Ported Memory Manager 39
10.17 I/O Manager 39
10.18 Rate Monotonic Manager 39

Command and Variable Index 41

Concept Index **43**

Preface

The Real Time Executive for Multiprocessor Systems (RTEMS) is designed to be portable across multiple processor architectures. However, the nature of real-time systems makes it essential that the application designer understand certain processor dependent implementation details. These processor dependencies include calling convention, board support package issues, interrupt processing, exact RTEMS memory requirements, performance data, header files, and the assembly language interface to the executive.

This document discusses the Texas Instrument C3x/C4x architecture dependencies in this port of RTEMS. The C3x/C4x family has a wide variety of CPU models within it. The following CPU model numbers could be supported by this port:

C30 - TMSXXX
C31 - TMSXXX
C32 - TMSXXX
C41 - TMSXXX
C44 - TMSXXX

Initiially, this port does not include full support for C4x models. Primarily, the C4x specific implementations of interrupt flag and mask management routines have not been completed.

It is highly recommended that the RTEMS application developer obtain and become familiar with the documentation for the processor being used as well as the documentation for the family as a whole.

Architecture Documents

For information on the Texas Instruments C3x/C4x architecture, refer to the following documents available from VENDOR (‘`http//www.ti.com/`’):

- *XXX Family Reference, Texas Instruments, PART NUMBER.*

MODEL SPECIFIC DOCUMENTS

For information on specific processor models and their associated coprocessors, refer to the following documents:

- *XXX MODEL Manual, Texas Instruments, PART NUMBER.*
- *XXX MODEL Manual, Texas Instruments, PART NUMBER.*

1 CPU Model Dependent Features

1.1 Introduction

Microprocessors are generally classified into families with a variety of CPU models or implementations within that family. Within a processor family, there is a high level of binary compatibility. This family may be based on either an architectural specification or on maintaining compatibility with a popular processor. Recent microprocessor families such as the SPARC or PA-RISC are based on an architectural specification which is independent or any particular CPU model or implementation. Older families such as the M68xxx and the iX86 evolved as the manufacturer strived to produce higher performance processor models which maintained binary compatibility with older models.

RTEMS takes advantage of the similarity of the various models within a CPU family. Although the models do vary in significant ways, the high level of compatibility makes it possible to share the bulk of the CPU dependent executive code across the entire family. Each processor family supported by RTEMS has a list of features which vary between CPU models within a family. For example, the most common model dependent feature regardless of CPU family is the presence or absence of a floating point unit or coprocessor. When defining the list of features present on a particular CPU model, one simply notes that floating point hardware is or is not present and defines a single constant appropriately. Conditional compilation is utilized to include the appropriate source code for this CPU model's feature set. It is important to note that this means that RTEMS is thus compiled using the appropriate feature set and compilation flags optimal for this CPU model used. The alternative would be to generate a binary which would execute on all family members using only the features which were always present.

This chapter presents the set of features which vary across the various implementations of the C3x/C4x architecture that are of importance to rtems. the set of cpu model feature macros are defined in the file cpukit/score/cpu/c4x/rtems/score/c4x.h and are based upon the particular cpu model defined in the bsp's custom configuration file as well as the compilation command line.

1.2 CPU Model Name

The macro `CPU_MODEL_NAME` is a string which designates the name of this cpu model. for example, for the c32 processor, this macro is set to the string "c32".

1.3 Floating Point Unit

The Texas Instruments C3x/C4x family makes little distinction between the various cpu registers. Although floating point operations may only be performed on a subset of the cpu registers, these same registers may be used for normal integer operations. as a result of this, this port of rtems makes no distinction between integer and floating point contexts.

The routine `_CPU_Context_switch` saves all of the registers that comprise a task's context. the routines that initialize, save, and restore floating point contexts are not present in this port.

Moreover, there is no floating point context pointer and the code in `_Thread_Dispatch` that manages the floating point context switching process is disabled on this port.

This not only simplifies the port, it also speeds up context switches by reducing the code involved and reduces the code space footprint of the executive on the Texas Instruments C3x/C4x.

2 Calling Conventions

2.1 Introduction

Each high-level language compiler generates subroutine entry and exit code based upon a set of rules known as the compiler's calling convention. These rules address the following issues:

- register preservation and usage
- parameter passing
- call and return mechanism

A compiler's calling convention is of importance when interfacing to subroutines written in another language either assembly or high-level. Even when the high-level language and target processor are the same, different compilers may use different calling conventions. As a result, calling conventions are both processor and compiler dependent.

The GNU Compiler Suite follows the same calling conventions as the Texas Instruments toolset.

2.2 Processor Background

The TI C3x and C4x processors support a simple yet effective call and return mechanism. A subroutine is invoked via the branch to subroutine (`XXX`) or the jump to subroutine (`XXX`) instructions. These instructions push the return address on the current stack. The return from subroutine (`XXX`) instruction pops the return address off the current stack and transfers control to that instruction. It is important to note that the call and return mechanism for the C3x/C4x does not automatically save or restore any registers. It is the responsibility of the high-level language compiler to define the register preservation and usage convention.

XXX other supplements may have "is is".

2.3 Calling Mechanism

All subroutines are invoked using either a `XXX` or `XXX` instruction and return to the user application via the `XXX` instruction.

2.4 Register Usage

XXX

As discussed above, the `XXX` and `XXX` instructions do not automatically save any registers. Subroutines use the registers **D0**, **D1**, **A0**, and **A1** as scratch registers. These registers are

not preserved by subroutines therefore, the contents of these registers should not be assumed upon return from any subroutine call including but not limited to an RTEMS directive.

The GNU and Texas Instruments compilers follow the same conventions for register usage.

2.5 Parameter Passing

Both the GNU and Texas Instruments compilers support two conventions for passing parameters to subroutines. Arguments may be passed in memory on the stack or in registers.

2.5.1 Parameters Passed in Memory

When passing parameters on the stack, the calling convention assumes that arguments are placed on the current stack before the subroutine is invoked via the `XXX` instruction. The first argument is assumed to be closest to the return address on the stack. This means that the first argument of the C calling sequence is pushed last. The following pseudo-code illustrates the typical sequence used to call a subroutine with three (3) arguments:

```
push third argument
push second argument
push first argument
invoke subroutine
remove arguments from the stack
```

The arguments to RTEMS are typically pushed onto the stack using a `sti` instruction with a pre-incremented stack pointer as the destination. These arguments must be removed from the stack after control is returned to the caller. This removal is typically accomplished by subtracting the size of the argument list in words from the current stack pointer.

With the GNU Compiler Suite, parameter passing via the stack is selected by invoking the compiler with the `-mmemparm XXX` argument. This argument must be included when linking the application in order to ensure that support libraries also compiled assuming parameter passing via the stack are used. The default parameter passing mechanism is XXX.

When this parameter passing mecahanism is selected, the `XXX` symbol is predefined by the C and C++ compilers and the `XXX` symbol is predefined by the assembler. This behavior is the same for the GNU and Texas Instruments toolsets. RTEMS uses these predefines to determine how parameters are passed in to those C3x/C4x specific routines that were written in assembly language.

2.5.2 Parameters Passed in Registers

When passing parameters via registers, the calling convention assumes that the arguments are placed in particular registers based upon their position and data type before the subroutine is invoked via the `XXX` instruction.

The following pseudo-code illustrates the typical sequence used to call a subroutine with three (3) arguments:

```
move third argument to XXX
move second argument to XXX
move first argument to XXX
invoke subroutine
```

With the GNU Compiler Suite, parameter passing via registers is selected by invoking the compiler with the `-mregparm XXX` argument. This argument must be included when linking the application in order to ensure that support libraries also compiled assuming parameter passing via the stack are used. The default parameter passing mechanism is XXX.

When this parameter passing mecahanism is selected, the `XXX` symbol is predefined by the C and C++ compilers and the `XXX` symbol is predefined by the assembler. This behavior is the same for the GNU and Texas Instruments toolsets. RTEMS uses these predefines to determine how parameters are passed in to those C3x/C4x specific routines that were written in assembly language.

2.6 User-Provided Routines

All user-provided routines invoked by RTEMS, such as user extensions, device drivers, and MPCI routines, must also adhere to these calling conventions.

3 Memory Model

3.1 Introduction

A processor may support any combination of memory models ranging from pure physical addressing to complex demand paged virtual memory systems. RTEMS supports a flat memory model which ranges contiguously over the processor's allowable address space. RTEMS does not support segmentation or virtual memory of any kind. The appropriate memory model for RTEMS provided by the targeted processor and related characteristics of that model are described in this chapter.

3.2 Byte Addressable versus Word Addressable

Processor in the Texas Instruments C3x/C4x family are word addressable. This is in sharp contrast to CISC and RISC processors that are typically byte addressable. In a word addressable architecture, each address points not to an 8-bit byte or octet but to 32 bits.

On first glance, byte versus word addressability does not sound like a problem but in fact, this issue can result in subtle problems in high-level language software that is ported to a word addressable processor family. The following is a list of the commonly encountered problems:

String Optimizations Although each character in a string occupies a single address just as it does on a byte addressable CPU, each character occupies 32 rather than 8 bits. The most significant 24 bytes are of each address are ignored. This in and of itself does not cause problems but it violates the assumption that two adjacent characters in a string have no intervening bits. This assumption is often implicit in string and memory comparison routines that are optimized to compare 4 adjacent characters with a word oriented operation. This optimization is invalid on word addressable processors.

Sizeof The C operation `sizeof` returns very different results on the C3x/C4x than on traditional RISC/CISC processors. The `sizeof(char)`, `sizeof(short)`, and `sizeof(int)` are all 1 since each occupies a single addressable unit that is thirty-two bits wide. On most thirty-two bit processors, `sizeof(char` is one, `sizeof(short)` is two, and `sizeof(int)` is four. Just as software makes assumptions about the sizes of the primitive data types has problems when ported to a sixty-four bit architecture, these same assumptions cause problems on the C3x/C4x.

Alignment Since each addressable unit is thirty-two bit wide, there are no alignment restrictions. The native integer type need only be aligned on a "one unit" boundary not a "four unit" boundary as on numerous other processors.

3.3 Flat Memory Model

XXX check actual bits on the various processor families.

The XXX family supports a flat 32-bit address space with addresses ranging from 0x00000000 to 0xFFFFFFFF (4 gigabytes). Each address is represented by a 32-bit value and is byte addressable. The address may be used to reference a single byte, word (2-bytes), or long word (4 bytes). Memory accesses within this address space are performed in big endian fashion by the processors in this family.

3.4 Compiler Memory Models

The Texas Instruments C3x/C4x processors include a Data Page (`dp`) register that logically is a base address. The `dp` register allows the use of shorter offsets in instructions. Up to 64K words may be addressed using offsets from the `dp` register. In order to address words not addressable based on the current value of `dp`, the register must be loaded with a different value.

The `dp` register is managed automatically by the high-level language compilers. The various compilers for this processor family support two memory models that manage the `dp` register in very different manners. The large and small memory models are discussed in the following sections.

NOTE: The C3x/C4x port of RTEMS has been written so that it should support either memory model. However, it has only been tested using the large memory model.

3.4.1 Small Memory Model

The small memory model is the simplest and most efficient. However, it includes a limitation that make it inappropriate for numerous applications. The small memory model assumes that the application needs to access no more than 64K words. Thus the `dp` register can be loaded at application start time and never reloaded. Thus the compiler will not even generate instructions to load the `dp`.

This can significantly reduce the code space required by an application but the application is limited in the amount of data it can access.

With the GNU Compiler Suite, small memory model is selected by invoking the compiler with either the `-msmall` or `-msmallmemoryXXX` argument. This argument must be included when linking the application in order to ensure that support libraries also compiled for the large memory model are used. The default memory model is XXX.

When this memory model is selected, the `XXX` symbol is predefined by the C and C++ compilers and the `XXX` symbol is predefined by the assembler. This behavior is the same for the GNU and Texas Instruments toolsets. RTEMS uses these predefines to determine the proper handling of the `dp` register in those C3x/C4x specific routines that were written in assembly language.

3.4.2 Large Memory Model

The large memory model is more complex and less efficient than the small memory model. However, it removes the 64K uninitialized data restriction from applications. The `dp` register is reloaded automatically by the compiler each time data is accessed. This leads to an increase in the code space requirements for the application but gives it access to much more data space.

With the GNU Compiler Suite, large memory model is selected by invoking the compiler with either the `-mlarge` or `-mlargememoryXXX` argument. This argument must be included when linking the application in order to ensure that support libraries also compiled for the large memory model are used. The default memory model is XXX.

When this memory model is selected, the `XXX` symbol is predefined by the C and C++ compilers and the `XXX` symbol is predefined by the assembler. This behavior is the same for the GNU and Texas Instruments toolsets. RTEMS uses these predefines to determine the proper handling of the `dp` register in those C3x/C4x specific routines that were written in assembly language.

4 Interrupt Processing

4.1 Introduction

Different types of processors respond to the occurrence of an interrupt in its own unique fashion. In addition, each processor type provides a control mechanism to allow for the proper handling of an interrupt. The processor dependent response to the interrupt modifies the current execution state and results in a change in the execution stream. Most processors require that an interrupt handler utilize some special control mechanisms to return to the normal processing stream. Although RTEMS hides many of the processor dependent details of interrupt processing, it is important to understand how the RTEMS interrupt manager is mapped onto the processor's unique architecture. Discussed in this chapter are the XXX's interrupt response and control mechanisms as they pertain to RTEMS.

4.2 Vectoring of an Interrupt Handler

Depending on whether or not the particular CPU supports a separate interrupt stack, the XXX family has two different interrupt handling models.

4.2.1 Models Without Separate Interrupt Stacks

Upon receipt of an interrupt the XXX family members without separate interrupt stacks automatically perform the following actions:

- To Be Written

4.2.2 Models With Separate Interrupt Stacks

Upon receipt of an interrupt the XXX family members with separate interrupt stacks automatically perform the following actions:

- saves the current status register (SR),
- clears the master/interrupt (M) bit of the SR to indicate the switch from master state to interrupt state,
- sets the privilege mode to supervisor,
- suppresses tracing,
- sets the interrupt mask level equal to the level of the interrupt being serviced,
- pushes an interrupt stack frame (ISF), which includes the program counter (PC), the status register (SR), and the format/exception vector offset (FVO) word, onto the supervisor and interrupt stacks,
- switches the current stack to the interrupt stack and vectors to an interrupt service routine (ISR). If the ISR was installed with the interrupt_catch directive, then the

RTEMS interrupt handler will begin execution. The RTEMS interrupt handler saves all registers which are not preserved according to the calling conventions and invokes the application's ISR.

A nested interrupt is processed similarly by these CPU models with the exception that only a single ISF is placed on the interrupt stack and the current stack need not be switched.

The FVO word in the Interrupt Stack Frame is examined by RTEMS to determine when an outer most interrupt is being exited. Since the FVO is used by RTEMS for this purpose, the user application code MUST NOT modify this field.

The following shows the Interrupt Stack Frame for XXX CPU models with separate interrupt stacks:

Status Register	0x0
Program Counter High	0x2
Program Counter Low	0x4
Format/Vector Offset	0x6

4.3 Interrupt Levels

Eight levels (0-7) of interrupt priorities are supported by XXX family members with level seven (7) being the highest priority. Level zero (0) indicates that interrupts are fully enabled. Interrupt requests for interrupts with priorities less than or equal to the current interrupt mask level are ignored.

Although RTEMS supports 256 interrupt levels, the XXX family only supports eight. RTEMS interrupt levels 0 through 7 directly correspond to XXX interrupt levels. All other RTEMS interrupt levels are undefined and their behavior is unpredictable.

4.4 Disabling of Interrupts by RTEMS

During the execution of directive calls, critical sections of code may be executed. When these sections are encountered, RTEMS disables interrupts to level seven (7) before the execution of this section and restores them to the previous level upon completion of the section. RTEMS has been optimized to insure that interrupts are disabled for less than TBD microseconds on a 20 Mhz XXX with zero wait states. These numbers will vary based the number of wait states and processor speed present on the target board. [NOTE: The maximum period with interrupts disabled is hand calculated. This calculation was last performed for Release 4.0.0.]

Non-maskable interrupts (NMI) cannot be disabled, and ISRs which execute at this level MUST NEVER issue RTEMS system calls. If a directive is invoked, unpredictable results may occur due to the inability of RTEMS to protect its critical sections. However, ISRs that make no system calls may safely execute as non-maskable interrupts.

4.5 Interrupt Stack

RTEMS allocates the interrupt stack from the Workspace Area. The amount of memory allocated for the interrupt stack is determined by the interrupt_stack_size field in the CPU Configuration Table. During the initialization process, RTEMS will install its interrupt stack.

The XXX port of RTEMS supports a software managed dedicated interrupt stack on those CPU models which do not support a separate interrupt stack in hardware.

5 Default Fatal Error Processing

5.1 Introduction

Upon detection of a fatal error by either the application or RTEMS the fatal error manager is invoked. The fatal error manager will invoke the user-supplied fatal error handlers. If no user-supplied handlers are configured, the RTEMS provided default fatal error handler is invoked. If the user-supplied fatal error handlers return to the executive the default fatal error handler is then invoked. This chapter describes the precise operations of the default fatal error handler.

5.2 Default Fatal Error Handler Operations

The default fatal error handler which is invoked by the `rtems_fatal_error_occurred` directive when there is no user handler configured or the user handler returns control to RTEMS. The default fatal error handler disables processor interrupts, places the error code in **XXX**, and executes a `XXX` instruction to simulate a halt processor instruction.

6 Board Support Packages

6.1 Introduction

An RTEMS Board Support Package (BSP) must be designed to support a particular processor and target board combination. This chapter presents a discussion of XXX specific BSP issues. For more information on developing a BSP, refer to the chapter titled Board Support Packages in the RTEMS Applications User's Guide.

6.2 System Reset

An RTEMS based application is initiated or re-initiated when the XXX processor is reset. When the XXX is reset, the processor performs the following actions:

- The tracing bits of the status register are cleared to disable tracing.
- The supervisor interrupt state is entered by setting the supervisor (S) bit and clearing the master/interrupt (M) bit of the status register.
- The interrupt mask of the status register is set to level 7 to effectively disable all maskable interrupts.
- The vector base register (VBR) is set to zero.
- The cache control register (CACR) is set to zero to disable and freeze the processor cache.
- The interrupt stack pointer (ISP) is set to the value stored at vector 0 (bytes 0-3) of the exception vector table (EVT).
- The program counter (PC) is set to the value stored at vector 1 (bytes 4-7) of the EVT.
- The processor begins execution at the address stored in the PC.

6.3 Processor Initialization

The address of the application's initialization code should be stored in the first vector of the EVT which will allow the immediate vectoring to the application code. If the application requires that the VBR be some value besides zero, then it should be set to the required value at this point. All tasks share the same XXX's VBR value. Because interrupts are enabled automatically by RTEMS as part of the initialize executive directive, the VBR MUST be set before this directive is invoked to insure correct interrupt vectoring. If processor caching is to be utilized, then it should be enabled during the reset application initialization code.

In addition to the requirements described in the Board Support Packages chapter of the Applications User's Manual for the reset code which is executed before the call to initialize executive, the XXX version has the following specific requirements:

- Must leave the S bit of the status register set so that the XXX remains in the supervisor state.
- Must set the M bit of the status register to remove the XXX from the interrupt state.
- Must set the master stack pointer (MSP) such that a minimum stack size of MINIMUM_STACK_SIZE bytes is provided for the initialize executive directive.
- Must initialize the XXX's vector table.

Note that the BSP is not responsible for allocating or installing the interrupt stack. RTEMS does this automatically as part of initialization. If the BSP does not install an interrupt stack and – for whatever reason – an interrupt occurs before initialize_executive is invoked, then the results are unpredictable.

7 Processor Dependent Information Table

7.1 Introduction

Any highly processor dependent information required to describe a processor to RTEMS is provided in the CPU Dependent Information Table. This table is not required for all processors supported by RTEMS. This chapter describes the contents, if any, for a particular processor type.

7.2 CPU Dependent Information Table

The XXX version of the RTEMS CPU Dependent Information Table contains the information required to interface a Board Support Package and RTEMS on the XXX. This information is provided to allow RTEMS to interoperate effectively with the BSP. The C structure definition is given here:

```
typedef struct {
  void       (*pretasking_hook)( void );
  void       (*predriver_hook)( void );
  void       (*postdriver_hook)( void );
  void       (*idle_task)( void );
  boolean      do_zero_of_workspace;
  unsigned32   idle_task_stack_size;
  unsigned32   interrupt_stack_size;
  unsigned32   extra_mpci_receive_server_stack;
  void *     (*stack_allocate_hook)( unsigned32 );
  void       (*stack_free_hook)( void* );
  /* end of fields required on all CPUs */

  /* XXX CPU family dependent stuff */
} rtems_cpu_table;
```

`pretasking_hook` is the address of the user provided routine which is invoked once RTEMS APIs are initialized. This routine will be invoked before any system tasks are created. Interrupts are disabled. This field may be NULL to indicate that the hook is not utilized.

`predriver_hook` is the address of the user provided routine that is invoked immediately before the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`postdriver_hook` is the address of the user provided routine that is invoked immediately after the the device drivers and MPCI are initialized. RTEMS initialization is complete but interrupts and tasking are disabled. This field may be NULL to indicate that the hook is not utilized.

`idle_task` is the address of the optional user provided routine which is used as the system's IDLE task. If this field is not NULL, then the RTEMS default IDLE task is not used. This field may be NULL to indicate that the default IDLE is to be used.

`do_zero_of_workspace`
indicates whether RTEMS should zero the Workspace as part of its initialization. If set to TRUE, the Workspace is zeroed. Otherwise, it is not.

`idle_task_stack_size`
is the size of the RTEMS idle task stack in bytes. If this number is less than MINIMUM_STACK_SIZE, then the idle task's stack will be MINIMUM_STACK_SIZE in byte.

`interrupt_stack_size`
is the size of the RTEMS allocated interrupt stack in bytes. This value must be at least as large as MINIMUM_STACK_SIZE.

`extra_mpci_receive_server_stack`
is the extra stack space allocated for the RTEMS MPCI receive server task in bytes. The MPCI receive server may invoke nearly all directives and may require extra stack space on some targets.

`stack_allocate_hook`
is the address of the optional user provided routine which allocates memory for task stacks. If this hook is not NULL, then a stack_free_hook must be provided as well.

`stack_free_hook` is the address of the optional user provided routine which frees memory for task stacks. If this hook is not NULL, then a stack_allocate_hook must be provided as well.

`XXX` is where the CPU family dependent stuff goes.

8 Memory Requirements

8.1 Introduction

Memory is typically a limited resource in real-time embedded systems, therefore, RTEMS can be configured to utilize the minimum amount of memory while meeting all of the applications requirements. Worksheets are provided which allow the RTEMS application developer to determine the amount of RTEMS code and RAM workspace which is required by the particular configuration. Also provided are the minimum code space, maximum code space, and the constant data space required by RTEMS.

8.2 Data Space Requirements

RTEMS requires a small amount of memory for its private variables. This data area must be in RAM and is separate from the RTEMS RAM Workspace. The following illustrates the data space required for all configurations of RTEMS:

- Data Space: 723

8.3 Minimum and Maximum Code Space Requirements

A maximum configuration of RTEMS includes the core and all managers, including the multiprocessing manager. Conversely, a minimum configuration of RTEMS includes only the core and the following managers: initialization, task, interrupt and fatal error. The following illustrates the code space required by these configurations of RTEMS:

- Minimum Configuration: 18,980
- Maximum Configuration: 36,438

8.4 RTEMS Code Space Worksheet

The RTEMS Code Space Worksheet is a tool provided to aid the RTEMS application designer to accurately calculate the memory required by the RTEMS run-time environment. RTEMS allows the custom configuration of the executive by optionally excluding managers which are not required by a particular application. This worksheet provides the included and excluded size of each manager in tabular form allowing for the quick calculation of any custom configuration of RTEMS. The RTEMS Code Space Worksheet is below:

RTEMS Code Space Worksheet

Component	Included	Not Included	Size
Core	12,674	NA	
Initialization	970	NA	
Task	3,562	NA	
Interrupt	54	NA	
Clock	334	NA	
Timer	1,110	184	
Semaphore	1,632	172	
Message	1,754	288	
Event	1,000	56	
Signal	418	56	
Partition	1,164	132	
Region	1,494	160	
Dual Ported Memory	724	132	
I/O	686	00	
Fatal Error	24	NA	
Rate Monotonic	1,212	184	
Multiprocessing	6.952	332	
Total Code Space Requirements			

8.5 RTEMS RAM Workspace Worksheet

The RTEMS RAM Workspace Worksheet is a tool provided to aid the RTEMS application designer to accurately calculate the minimum memory block to be reserved for RTEMS use. This worksheet provides equations for calculating the amount of memory required based upon the number of objects configured, whether for single or multiple processor versions of the executive. This information is presented in tabular form, along with the fixed system requirements, allowing for quick calculation of any application defined configuration of RTEMS. The RTEMS RAM Workspace Worksheet is provided below:

RTEMS RAM Workspace Worksheet

Description	Equation	Bytes Required
maximum_tasks	* 400 =	
maximum_timers	* 68 =	
maximum_semaphores	* 124 =	
maximum_message_queues	* 148 =	
maximum_regions	* 144 =	
maximum_partitions	* 56 =	
maximum_ports	* 36 =	
maximum_periods	* 36 =	
maximum_extensions	* 64 =	
Floating Point Tasks	* 332 =	
Task Stacks	=	
Total Single Processor Requirements		

Description	Equation	Bytes Required
maximum_nodes	* 48 =	
maximum_global_objects	* 20 =	
maximum_proxies	* 124 =	
Total Multiprocessing Requirements		
Fixed System Requirements	8,872	
Total Single Processor Requirements		
Total Multiprocessing Requirements		
Minimum Bytes for RTEMS Workspace		

9 Timing Specification

9.1 Introduction

This chapter provides information pertaining to the measurement of the performance of RTEMS, the methods of gathering the timing data, and the usefulness of the data. Also discussed are other time critical aspects of RTEMS that affect an applications design and ultimate throughput. These aspects include determinancy, interrupt latency and context switch times.

9.2 Philosophy

Benchmarks are commonly used to evaluate the performance of software and hardware. Benchmarks can be an effective tool when comparing systems. Unfortunately, benchmarks can also be manipulated to justify virtually any claim. Benchmarks of real-time executives are difficult to evaluate for a variety of reasons. Executives vary in the robustness of features and options provided. Even when executives compare favorably in functionality, it is quite likely that different methodologies were used to obtain the timing data. Another problem is that some executives provide times for only a small subset of directives, This is typically justified by claiming that these are the only time-critical directives. The performance of some executives is also very sensitive to the number of objects in the system. To obtain any measure of usefulness, the performance information provided for an executive should address each of these issues.

When evaluating the performance of a real-time executive, one typically considers the following areas: determinancy, directive times, worst case interrupt latency, and context switch time. Unfortunately, these areas do not have standard measurement methodologies. This allows vendors to manipulate the results such that their product is favorably represented. We have attempted to provide useful and meaningful timing information for RTEMS. To insure the usefulness of our data, the methodology and definitions used to obtain and describe the data are also documented.

9.2.1 Determinancy

The correctness of data in a real-time system must always be judged by its timeliness. In many real-time systems, obtaining the correct answer does not necessarily solve the problem. For example, in a nuclear reactor it is not enough to determine that the core is overheating. This situation must be detected and acknowledged early enough that corrective action can be taken and a meltdown avoided.

Consequently, a system designer must be able to predict the worst-case behavior of the application running under the selected executive. In this light, it is important that a real-time system perform consistently regardless of the number of tasks, semaphores, or other resources allocated. An important design goal of a real-time executive is that all internal

algorithms be fixed-cost. Unfortunately, this goal is difficult to completely meet without sacrificing the robustness of the executive's feature set.

Many executives use the term deterministic to mean that the execution times of their services can be predicted. However, they often provide formulas to modify execution times based upon the number of objects in the system. This usage is in sharp contrast to the notion of deterministic meaning fixed cost.

Almost all RTEMS directives execute in a fixed amount of time regardless of the number of objects present in the system. The primary exception occurs when a task blocks while acquiring a resource and specifies a non-zero timeout interval.

Other exceptions are message queue broadcast, obtaining a variable length memory block, object name to ID translation, and deleting a resource upon which tasks are waiting. In addition, the time required to service a clock tick interrupt is based upon the number of timeouts and other "events" which must be processed at that tick. This second group is composed primarily of capabilities which are inherently non-deterministic but are infrequently used in time critical situations. The major exception is that of servicing a clock tick. However, most applications have a very small number of timeouts which expire at exactly the same millisecond (usually none, but occasionally two or three).

9.2.2 Interrupt Latency

Interrupt latency is the delay between the CPU's receipt of an interrupt request and the execution of the first application-specific instruction in an interrupt service routine. Interrupts are a critical component of most real-time applications and it is critical that they be acted upon as quickly as possible.

Knowledge of the worst case interrupt latency of an executive aids the application designer in determining the maximum period of time between the generation of an interrupt and an interrupt handler responding to that interrupt. The interrupt latency of an system is the greater of the executive's and the applications's interrupt latency. If the application disables interrupts longer than the executive, then the application's interrupt latency is the system's worst case interrupt disable period.

The worst case interrupt latency for a real-time executive is based upon the following components:

- the longest period of time interrupts are disabled by the executive,
- the overhead required by the executive at the beginning of each ISR,
- the time required for the CPU to vector the interrupt, and
- for some microprocessors, the length of the longest instruction.

The first component is irrelevant if an interrupt occurs when interrupts are enabled, although it must be included in a worst case analysis. The third and fourth components are particular to a CPU implementation and are not dependent on the executive. The fourth component is ignored by this document because most applications use only a subset of a microprocessor's instruction set. Because of this the longest instruction actually executed is application dependent. The worst case interrupt latency of an executive is typically defined

as the sum of components (1) and (2). The second component includes the time necessry for RTEMS to save registers and vector to the user-defined handler. RTEMS includes the third component, the time required for the CPU to vector the interrupt, because it is a required part of any interrupt.

Many executives report the maximum interrupt disable period as their interrupt latency and ignore the other components. This results in very low worst-case interrupt latency times which are not indicative of actual application performance. The definition used by RTEMS results in a higher interrupt latency being reported, but accurately reflects the longest delay between the CPU's receipt of an interrupt request and the execution of the first application-specific instruction in an interrupt service routine.

The actual interrupt latency times are reported in the Timing Data chapter of this supplement.

9.2.3 Context Switch Time

An RTEMS context switch is defined as the act of taking the CPU from the currently executing task and giving it to another task. This process involves the following components:

- Saving the hardware state of the current task.
- Optionally, invoking the TASK_SWITCH user extension.
- Restoring the hardware state of the new task.

RTEMS defines the hardware state of a task to include the CPU's data registers, address registers, and, optionally, floating point registers.

Context switch time is often touted as a performance measure of real-time executives. However, a context switch is performed as part of a directive's actions and should be viewed as such when designing an application. For example, if a task is unable to acquire a semaphore and blocks, a context switch is required to transfer control from the blocking task to a new task. From the application's perspective, the context switch is a direct result of not acquiring the semaphore. In this light, the context switch time is no more relevant than the performance of any other of the executive's subroutines which are not directly accessible by the application.

In spite of the inappropriateness of using the context switch time as a performance metric, RTEMS context switch times for floating point and non-floating points tasks are provided for comparison purposes. Of the executives which actually support floating point operations, many do not report context switch times for floating point context switch time. This results in a reported context switch time which is meaningless for an application with floating point tasks.

The actual context switch times are reported in the Timing Data chapter of this supplement.

9.2.4 Directive Times

Directives are the application's interface to the executive, and as such their execution times are critical in determining the performance of the application. For example, an applica-

tion using a semaphore to protect a critical data structure should be aware of the time required to acquire and release a semaphore. In addition, the application designer can utilize the directive execution times to evaluate the performance of different synchronization and communication mechanisms.

The actual directive execution times are reported in the Timing Data chapter of this supplement.

9.3 Methodology

9.3.1 Software Platform

The RTEMS timing suite is written in C. The overhead of passing arguments to RTEMS by C is not timed. The times reported represent the amount of time from entering to exiting RTEMS.

The tests are based upon one of two execution models: (1) single invocation times, and (2) average times of repeated invocations. Single invocation times are provided for directives which cannot easily be invoked multiple times in the same scenario. For example, the times reported for entering and exiting an interrupt service routine are single invocation times. The second model is used for directives which can easily be invoked multiple times in the same scenario. For example, the times reported for semaphore obtain and semaphore release are averages of multiple invocations. At least 100 invocations are used to obtain the average.

9.3.2 Hardware Platform

Since RTEMS supports a variety of processors, the hardware platform used to gather the benchmark times must also vary. Therefore, for each processor supported the hardware platform must be defined. Each definition will include a brief description of the target hardware platform including the clock speed, memory wait states encountered, and any other pertinent information. This definition may be found in the processor dependent timing data chapter within this supplement.

9.3.3 What is measured?

An effort was made to provide execution times for a large portion of RTEMS. Times were provided for most directives regardless of whether or not they are typically used in time critical code. For example, execution times are provided for all object create and delete directives, even though these are typically part of application initialization.

The times include all RTEMS actions necessary in a particular scenario. For example, all times for blocking directives include the context switch necessary to transfer control to a new task. Under no circumstances is it necessary to add context switch time to the reported times.

The following list describes the objects created by the timing suite:

- All tasks are non-floating point.
- All tasks are created as local objects.
- No timeouts are used on blocking directives.
- All tasks wait for objects in FIFO order.

In addition, no user extensions are configured.

9.3.4 What is not measured?

The times presented in this document are not intended to represent best or worst case times, nor are all directives included. For example, no times are provided for the initialize executive and fatal_error_occurred directives. Other than the exceptions detailed in the Determinancy section, all directives will execute in the fixed length of time given.

Other than entering and exiting an interrupt service routine, all directives were executed from tasks and not from interrupt service routines. Directives invoked from ISRs, when allowable, will execute in slightly less time than when invoked from a task because rescheduling is delayed until the interrupt exits.

9.3.5 Terminology

The following is a list of phrases which are used to distinguish individual execution paths of the directives taken during the RTEMS performance analysis:

another task The directive was performed on a task other than the calling task.

available A task attempted to obtain a resource and immediately acquired it.

blocked task The task operated upon by the directive was blocked waiting for a resource.

caller blocks The requested resoure was not immediately available and the calling task chose to wait.

calling task The task invoking the directive.

messages flushed One or more messages was flushed from the message queue.

no messages flushed No messages were flushed from the message queue.

not available A task attempted to obtain a resource and could not immediately acquire it.

no reschedule The directive did not require a rescheduling operation.

NO_WAIT A resource was not available and the calling task chose to return immediately via the NO_WAIT option with an error.

obtain current The current value of something was requested by the calling task.

preempts caller The release of a resource caused a task of higher priority than the calling to be readied and it became the executing task.

ready task The task operated upon by the directive was in the ready state.

reschedule The actions of the directive necessitated a rescheduling operation.

returns to caller The directive succeeded and immediately returned to the calling task.

returns to interrupted task
The instructions executed immediately following this interrupt will be in the interrupted task.

returns to nested interrupt
The instructions executed immediately following this interrupt will be in a previously interrupted ISR.

returns to preempting task
The instructions executed immediately following this interrupt or signal handler will be in a task other than the interrupted task.

signal to self The signal set was sent to the calling task and signal processing was enabled.

suspended task The task operated upon by the directive was in the suspended state.

task readied The release of a resource caused a task of lower or equal priority to be readied and the calling task remained the executing task.

yield The act of attempting to voluntarily release the CPU.

10 BSP_FOR_TIMES Timing Data

10.1 Introduction

The timing data for the XXX version of RTEMS is provided along with the target dependent aspects concerning the gathering of the timing data. The hardware platform used to gather the times is described to give the reader a better understanding of each directive time provided. Also, provided is a description of the interrupt latency and the context switch times as they pertain to the XXX version of RTEMS.

10.2 Hardware Platform

All times reported except for the maximum period interrupts are disabled by RTEMS were measured using a Motorola BSP_FOR_TIMES CPU board. The BSP_FOR_TIMES is a 20Mhz board with one wait state dynamic memory and a XXX numeric coprocessor. The Zilog 8036 countdown timer on this board was used to measure elapsed time with a one-half microsecond resolution. All sources of hardware interrupts were disabled, although the interrupt level of the XXX allows all interrupts.

The maximum period interrupts are disabled was measured by summing the number of CPU cycles required by each assembly language instruction executed while interrupts were disabled. The worst case times of the XXX microprocessor were used for each instruction. Zero wait state memory was assumed. The total CPU cycles executed with interrupts disabled, including the instructions to disable and enable interrupts, was divided by 20 to simulate a 20Mhz XXX. It should be noted that the worst case instruction times for the XXX assume that the internal cache is disabled and that no instructions overlap.

10.3 Interrupt Latency

The maximum period with interrupts disabled within RTEMS is less than TBD microseconds including the instructions which disable and re-enable interrupts. The time required for the XXX to vector an interrupt and for the RTEMS entry overhead before invoking the user's interrupt handler are a total of 9 microseconds. These combine to yield a worst case interrupt latency of less than TBD + 9 microseconds at 20Mhz. [NOTE: The maximum period with interrupts disabled was last determined for Release 4.0.0.]

It should be noted again that the maximum period with interrupts disabled within RTEMS is hand-timed and based upon worst case (i.e. CPU cache disabled and no instruction overlap) times for a 20Mhz XXX. The interrupt vector and entry overhead time was generated on an BSP_FOR_TIMES benchmark platform using the Multiprocessing Communications registers to generate as the interrupt source.

10.4 Context Switch

The RTEMS processor context switch time is 35 microseconds on the BSP_FOR_TIMES benchmark platform when no floating point context is saved or restored. Additional execution time is required when a TASK_SWITCH user extension is configured. The use of the TASK_SWITCH extension is application dependent. Thus, its execution time is not considered part of the raw context switch time.

Since RTEMS was designed specifically for embedded missile applications which are floating point intensive, the executive is optimized to avoid unnecessarily saving and restoring the state of the numeric coprocessor. The state of the numeric coprocessor is only saved when an FLOATING_POINT task is dispatched and that task was not the last task to utilize the coprocessor. In a system with only one FLOATING_POINT task, the state of the numeric coprocessor will never be saved or restored. When the first FLOATING_POINT task is dispatched, RTEMS does not need to save the current state of the numeric coprocessor.

The exact amount of time required to save and restore floating point context is dependent on whether an XXX or XXX is being used as well as the state of the numeric coprocessor. These numeric coprocessors define three operating states: initialized, idle, and busy. RTEMS places the coprocessor in the initialized state when a task is started or restarted. Once the task has utilized the coprocessor, it is in the idle state when floating point instructions are not executing and the busy state when floating point instructions are executing. The state of the coprocessor is task specific.

The following table summarizes the context switch times for the BSP_FOR_TIMES benchmark platform:

No Floating Point Contexts	35
Floating Point Contexts	
restore first FP task	39
save initialized, restore initialized	66
save idle, restore initialized	66
save idle, restore idle	68

10.5 Directive Times

This sections is divided into a number of subsections, each of which contains a table listing the execution times of that manager's directives.

10.6 Task Manager

TASK_CREATE	148
TASK_IDENT	350
TASK_START	76
TASK_RESTART	
calling task	95
suspended task – returns to caller	89
blocked task – returns to caller	124
ready task – returns to caller	92
suspended task – preempts caller	125
blocked task – preempts caller	149
ready task – preempts caller	142
TASK_DELETE	
calling task	170
suspended task	138
blocked task	143
ready task	144
TASK_SUSPEND	
calling task	71
returns to caller	43
TASK_RESUME	
task readied – returns to caller	45
task readied – preempts caller	67
TASK_SET_PRIORITY	
obtain current priority	31
returns to caller	64
preempts caller	106
TASK_MODE	
obtain current mode	14
no reschedule	16
reschedule – returns to caller	23
reschedule – preempts caller	60
TASK_GET_NOTE	33
TASK_SET_NOTE	33
TASK_WAKE_AFTER	
yield – returns to caller	16
yield – preempts caller	56
TASK_WAKE_WHEN	117

10.7 Interrupt Manager

It should be noted that the interrupt entry times include vectoring the interrupt handler.

Interrupt Entry Overhead	
returns to nested interrupt	12
returns to interrupted task	9
returns to preempting task	9
Interrupt Exit Overhead	
returns to nested interrupt	¡1
returns to interrupted task	8
returns to preempting task	54

10.8 Clock Manager

CLOCK_SET	86
CLOCK_GET	1
CLOCK_TICK	17

10.9 Timer Manager

TIMER_CREATE	28
TIMER_IDENT	343
TIMER_DELETE	
inactive	43
active	47
TIMER_FIRE_AFTER	
inactive	58
active	61
TIMER_FIRE_WHEN	
inactive	88
active	88
TIMER_RESET	
inactive	54
active	58
TIMER_CANCEL	
inactive	31
active	34

10.10 Semaphore Manager

SEMAPHORE_CREATE	60
SEMAPHORE_IDENT	367
SEMAPHORE_DELETE	58
SEMAPHORE_OBTAIN	
available	38
not available – NO_WAIT	38
not available – caller blocks	109
SEMAPHORE_RELEASE	
no waiting tasks	44
task readied – returns to caller	66
task readied – preempts caller	87

10.11 Message Manager

MESSAGE_QUEUE_CREATE	200
MESSAGE_QUEUE_IDENT	341
MESSAGE_QUEUE_DELETE	80
MESSAGE_QUEUE_SEND	
no waiting tasks	97
task readied – returns to caller	101
task readied – preempts caller	123
MESSAGE_QUEUE_URGENT	
no waiting tasks	96
task readied – returns to caller	101
task readied – preempts caller	123
MESSAGE_QUEUE_BROADCAST	
no waiting tasks	53
task readied – returns to caller	111
task readied – preempts caller	133
MESSAGE_QUEUE_RECEIVE	
available	79
not available – NO_WAIT	43
not available – caller blocks	114
MESSAGE_QUEUE_FLUSH	
no messages flushed	29
messages flushed	39

10.12 Event Manager

EVENT_SEND	
no task readied	24
task readied – returns to caller	60
task readied – preempts caller	84
EVENT_RECEIVE	
obtain current events	1
available	28
not available – NO_WAIT	23
not available – caller blocks	84

10.13 Signal Manager

SIGNAL_CATCH	15
SIGNAL_SEND	
returns to caller	37
signal to self	55
EXIT ASR OVERHEAD	
returns to calling task	37
returns to preempting task	54

10.14 Partition Manager

PARTITION_CREATE	70
PARTITION_IDENT	341
PARTITION_DELETE	42
PARTITION_GET_BUFFER	
available	35
not available	33
PARTITION_RETURN_BUFFER	33

10.15 Region Manager

REGION_CREATE	63
REGION_IDENT	348
REGION_DELETE	39
REGION_GET_SEGMENT	
available	52
not available – NO_WAIT	49
not available – caller blocks	123
REGION_RETURN_SEGMENT	
no waiting tasks	54
task readied – returns to caller	114
task readied – preempts caller	136

10.16 Dual-Ported Memory Manager

PORT_CREATE	35
PORT_IDENT	340
PORT_DELETE	39
PORT_INTERNAL_TO_EXTERNAL	26
PORT_EXTERNAL_TO_INTERNAL	27

10.17 I/O Manager

IO_INITIALIZE	4
IO_OPEN	2
IO_CLOSE	1
IO_READ	2
IO_WRITE	3
IO_CONTROL	2

10.18 Rate Monotonic Manager

RATE_MONOTONIC_CREATE	32
RATE_MONOTONIC_IDENT	341
RATE_MONOTONIC_CANCEL	39
RATE_MONOTONIC_DELETE	
active	51
inactive	48
RATE_MONOTONIC_PERIOD	
initiate period – returns to caller	54
conclude period – caller blocks	74
obtain status	31

Command and Variable Index

There are currently no Command and Variable Index entries.

Concept Index

There are currently no Concept Index entries.