New Chapters

Edition 4.7.0, for RTEMS 4.7.0

19 January 2006

On-Line Applications Research Corporation

On-Line Applications Research Corporation
TEXinfo 2006-10-04.17

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

1 Stack Bounds Checker

1.1 Introduction

The stack bounds checker is an RTEMS support component that determines if a task has overflowed its run-time stack. The routines provided by the stack bounds checker manager are:

- `Stack_check_Initialize` - Initialize the Stack Bounds Checker
- `Stack_check_Dump_usage` - Report Task Stack Usage

1.2 Background

1.2.1 Task Stack

Each task in a system has a fixed size stack associated with it. This stack is allocated when the task is created. As the task executes, the stack is used to contain parameters, return addresses, saved registers, and local variables. The amount of stack space required by a task is dependent on the exact set of routines used. The peak stack usage reflects the worst case of subroutine pushing information on the stack. For example, if a subroutine allocates a local buffer of 1024 bytes, then this data must be accounted for in the stack of every task that invokes that routine.

Recursive routines make calculating peak stack usage difficult, if not impossible. Each call to the recursive routine consumes n bytes of stack space. If the routine recursives 1000 times, then `1000 * n` bytes of stack space are required.

1.2.2 Execution

The stack bounds checker operates as a set of task extensions. At task creation time, the task's stack is filled with a pattern to indicate the stack is unused. As the task executes, it will overwrite this pattern in memory. At each task switch, the stack bounds checker's task switch extension is executed. This extension checks that the last `n` bytes of the task's stack have not been overwritten. If they have, then a blown stack error is reported.

The number of bytes checked for an overwrite is processor family dependent. The minimum stack frame per subroutine call varies widely between processor families. On CISC families like the Motorola MC68xxx and Intel ix86, all that is needed is a return address. On more complex RISC processors, the minimum stack frame per subroutine call may include space to save a significant number of registers.

Another processor dependent feature that must be taken into account by the stack bounds checker is the direction that the stack grows. On some processor families, the stack grows up or to higher addresses as the task executes. On other families, it grows down to lower addresses. The stack bounds checker implementation uses the stack description definitions provided by every RTEMS port to get for this information.

1.3 Operations

1.3.1 Initializing the Stack Bounds Checker

The stack checker is initialized automatically when its task create extension runs for the first time. When this occurs, the `Stack_check_Initialize` is invoked.

The application must include the stack bounds checker extension set in its set of Initial Extensions. This set of extensions is defined as `STACK_CHECKER_EXTENSION`. If using `<confdefs.h>` for Configuration Table generation, then all that is necessary is to define the macro `STACK_CHECKER_ON` before including `<confdefs.h>` as shown below:

```
#define STACK_CHECKER_ON
  ...
#include <confdefs.h>
```

1.3.2 Reporting Task Stack Usage

The application may dynamically report the stack usage for every task in the system by calling the `Stack_check_Dump_usage` routine. This routine prints a table with the peak usage and stack size of every task in the system. The following is an example of the report generated:

```
     ID      NAME       LOW        HIGH     AVAILABLE      USED
0x04010001   IDLE   0x003e8a60  0x003e9667       2952        200
0x08010002   TA1    0x003e5750  0x003e7b57       9096       1168
0x08010003   TA2    0x003e31c8  0x003e55cf       9096       1168
0x08010004   TA3    0x003e0c40  0x003e3047       9096       1104
0xffffffff  INTR    0x003ecfc0  0x003effbf      12160        128
```

Notice the last time. The task id is 0xffffffff and its name is "INTR". This is not actually a task, it is the interrupt stack.

1.3.3 When a Task Overflows the Stack

When the stack bounds checker determines that a stack overflow has occurred, it will attempt to print a message identifying the task and then shut the system down. If the stack overflow has caused corruption, then it is possible that the message can not be printed.

The following is an example of the output generated:

```
BLOWN STACK!!! Offending task(0x3eb360): id=0x08010002; name=0x54413120
  stack covers range 0x003e5750 - 0x003e7b57 (9224 bytes)
  Damaged pattern begins at 0x003e5758 and is 128 bytes long
```

The above includes the task id and a pointer to the task control block as well as enough information so one can look at the task's stack and see what was happening.

1.4 Routines

This section details the stack bounds checker's routines. A subsection is dedicated to each of routines and describes the calling sequence, related constants, usage, and status codes.

1.4.1 Stack_check_Initialize - Initialize the Stack Bounds Checker

CALLING SEQUENCE:

```
void Stack_check_Initialize( void );
```

STATUS CODES: NONE

DESCRIPTION:

Initialize the stack bounds checker.

NOTES:

This is performed automatically the first time the stack bounds checker task create extension executes.

1.4.2 Stack_check_Dump_usage - Report Task Stack Usage

CALLING SEQUENCE:

```
void Stack_check_Dump_usage( void );
```

STATUS CODES: NONE

DESCRIPTION:

This routine prints a table with the peak stack usage and stack space allocation of every task in the system.

NOTES:

NONE

2 Rate Monotonic Period Statistics

2.1 Introduction

The rate monotonic period statistics manager is an RTEMS support component that maintains statistics on the execution characteristics of each task using a period. The routines provided by the rate monotonic period statistics manager are:

- `Period_usage_Initialize` - Initialize the Period Statistics
- `Period_usage_Reset` - Reset the Period Statistics
- `Period_usage_Update` - Update the Statistics for this Period
- `Period_usage_Dump` - Report Period Statistics Usage

2.2 Background

2.3 Period Statistics

This manager maintains a set of statistics on each period. The following is a list of the information kept:

- `id` is the id of the period.
- `count` is the total number of periods executed.
- `missed_count` is the number of periods that were missed.
- `min_cpu_time` is the minimum amount of CPU execution time consumed on any execution of the periodic loop.
- `max_cpu_time` is the maximum amount of CPU execution time consumed on any execution of the periodic loop.
- `total_cpu_time` is the total amount of CPU execution time consumed by executions of the periodic loop.
- `min_wall_time` is the minimum amount of wall time that passed on any execution of the periodic loop.
- `max_wall_time` is the maximum amount of wall time that passed on any execution of the periodic loop.
- `total_wall_time` is the total amount of wall time that passed during executions of the periodic loop.

The above information is inexpensive to maintain and can provide very useful insights into the execution characteristics of a periodic task loop.

2.3.1 Analysis of the Reported Information

The period statistics reported must be analyzed by the user in terms of what the applications is. For example, in an application where priorities are assigned by the Rate Monotonic Algorithm, it would be very undesirable for high priority (i.e. frequency) tasks to miss their period. Similarly, in nearly any application, if a task were supposed to execute its periodic loop every 10 milliseconds and it averaged 11 milliseconds, then application requirements are not being met.

The information reported can be used to determine the "hot spots" in the application. Given a period's id, the user can determine the length of that period. From that information and the CPU usage, the user can calculate the percentage of CPU time consumed by that periodic task. For example, a task executing for 20 milliseconds every 200 milliseconds is consuming 10 percent of the processor's execution time. This is usually enough to make it a good candidate for optimization.

However, execution time alone is not enough to gauge the value of optimizing a particular task. It is more important to optimize a task executing 2 millisecond every 10 milliseconds (20 percent of the CPU) than one executing 10 milliseconds every 100 (10 percent of the CPU). As a general rule of thumb, the higher frequency at which a task executes, the more important it is to optimize that task.

2.4 Operations

2.4.1 Initializing the Period Statistics

The period statistics manager must be explicitly initialized before any calls to this manager. This is done by calling the `Period_usage_Initialize` service.

2.4.2 Updating Period Statistics

It is the responsibility of each period task loop to update the statistics on each execution of its loop. The following is an example of a simple periodic task that uses the period statistics manager:

```
rtems_task Periodic_task()
{
  rtems_name        name;
  rtems_id          period;
  rtems_status_code status;

  name = rtems_build_name( 'P', 'E', 'R', 'D' );

  (void) rate_monotonic_create( name, &period );

  while ( 1 ) {
    if ( rate_monotonic_period( period, 100 ) == TIMEOUT )
      break;

    /* Perform some periodic actions */

    /* Report statistics */
    Period_usage_Update( period_id );
  }

  /* missed period so delete period and SELF */

  (void) rate_monotonic_delete( period );
  (void) task_delete( SELF );
}
```

2.4.3 Reporting Period Statistics

The application may dynamically report the period usage for every period in the system by calling the `Period_usage_Dump` routine. This routine prints a table with the following information per period:

- period id
- id of the task that owns the period
- number of periods executed
- number of periods missed
- minimum/maximum/average cpu use per period
- minimum/maximum/average wall time per period

The following is an example of the report generated:

```
Period information by period
   ID      OWNER   PERIODS  MISSED    CPU TIME    WALL TIME
0x28010001  TA1       502     0       0/1/ 1.00    0/0/0.00
0x28010002  TA2       502     0       0/1/ 1.00    0/0/0.00
0x28010003  TA3       502     0       0/1/ 1.00    0/0/0.00
0x28010004  TA4       502     0       0/1/ 1.00    0/0/0.00
0x28010005  TA5        10     0       0/1/ 0.90    0/0/0.00
```

2.5 Routines

This section details the rate monotonic period statistics manager's routines. A subsection is dedicated to each of this manager's routines and describes the calling sequence, related constants, usage, and status codes.

2.5.1 Period_usage_Initialize - Initialize the Period Statistics

CALLING SEQUENCE:

```
void Period_usage_Initialize( void );
```

STATUS CODES: NONE

DESCRIPTION:

This routine allocates the table used to contain the period statistics. This table is then initialized by calling the `Period_usage_Reset` service.

NOTES:

This routine invokes the `malloc` routine to dynamically allocate memory.

2.5.2 Period_usage_Reset - Reset the Period Statistics

CALLING SEQUENCE:

```
void Period_usage_Reset( void );
```

STATUS CODES: NONE

DESCRIPTION:

This routine re-initializes the period statistics table to its default state which is when zero period executions have occurred.

NOTES:

NONE

2.5.3 Period_usage_Update - Update the Statistics for this Period

CALLING SEQUENCE:

```
void Period_usage_Update(
  rtems_id   id
);
```

STATUS CODES: NONE

DESCRIPTION:

The `Period_usage_Update` routine must be invoked at the "bottom" of each periodic loop iteration to update the statistics.

NOTES:

NONE

2.5.4 Period_usage_Dump - Report Period Statistics Usage

CALLING SEQUENCE:

```
void Period_usage_Dump( void );
```

STATUS CODES: NONE

DESCRIPTION:

This routine prints out a table detailing the period statistics for all periods in the system.

NOTES:

NONE

3 CPU Usage Statistics

3.1 Introduction

The CPU usage statistics manager is an RTEMS support component that provides a convenient way to manipulate the CPU usage information associated with each task The routines provided by the CPU usage statistics manager are:

- `CPU_usage_Dump` - Report CPU Usage Statistics
- `CPU_usage_Reset` - Reset CPU Usage Statistics

3.2 Background

3.3 Operations

3.4 Report CPU Usage Statistics

3.4.1 Reporting Period Statistics

The application may dynamically report the CPU usage for every task in the system by calling the `CPU_usage_Dump` routine. This routine prints a table with the following information per task:

- task id
- task name
- number of clock ticks executed
- percentage of time consumed by this task

The following is an example of the report generated:

```
CPU Usage by thread
   ID        NAME        TICKS    PERCENT
0x04010001   IDLE           0     0.000
0x08010002   TA1         1203     0.748
0x08010003   TA2          203     0.126
0x08010004   TA3          202     0.126

Ticks since last reset = 1600

Total Units = 1608
```

Notice that the "Total Units" is greater than the ticks per reset. This is an artifact of the way in which RTEMS keeps track of CPU usage. When a task is context switched into the CPU, the number of clock ticks it has executed is incremented. While the task is executing, this number is incremented on each clock tick. Otherwise, if a task begins and completes execution between successive clock ticks, there would be no way to tell that it executed at all.

Another thing to keep in mind when looking at idle time, is that many systems – especially during debug – have a task providing some type of debug interface. It is usually fine to

think of the total idle time as being the sum of the IDLE task and a debug task that will not be included in a production build of an application.

3.5 Reset CPU Usage Statistics

Invoking the `CPU_usage_Reset` routine resets the CPU usage statistics for all tasks in the system.

3.6 Directives

This section details the CPU usage statistics manager's directives. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

3.6.1 CPU_usage_Dump - Report CPU Usage Statistics

CALLING SEQUENCE:

```
void CPU_usage_Dump( void );
```

STATUS CODES: NONE

DESCRIPTION:

This routine prints out a table detailing the CPU usage statistics for all tasks in the system.

NOTES:

NONE

3.6.2 CPU_usage_Reset - Reset CPU Usage Statistics

CALLING SEQUENCE:

```
void CPU_usage_Reset( void );
```

STATUS CODES: NONE

DESCRIPTION:

This routine re-initializes the CPU usage statistics for all tasks in the system to their initial state. The initial state is that a task has not executed and thus has consumed no CPU time. default state which is when zero period executions have occurred.

NOTES:

NONE

4 Error Reporting Support

4.1 Introduction

These error reporting facilities are an RTEMS support component that provide convenient facilities for handling error conditions in an RTEMS application. of each task using a period. The services provided by the error reporting support component are:

- `rtems_error` - Report an Error
- `rtems_panic` - Report an Error and Panic
- `rtems_status_text` - ASCII Version of RTEMS Status

4.2 Background

4.2.1 Error Handling in an Embedded System

Error handling in an embedded system is a difficult problem. If the error is severe, then the only recourse is to shut the system down in a safe manner. Other errors can be detected and compensated for. The error reporting routines in this support component – `rtems_error` and `rtems_panic` assume that if the error is severe enough, then the system should be shutdown. If a simple shutdown with some basic diagnostic information is not sufficient, then these routines should not be used in that particular system. In this case, use the `rtems_status_text` routine to construct an application specific error reporting routine.

4.3 Operations

4.3.1 Reporting an Error

The `rtems_error` and `rtems_panic` routines can be used to print some diagnostic information and shut the system down. The `rtems_error` routine is invoked with a user specified error level indicator. This error indicator is used to determine if the system should be shutdown after reporting this error.

4.4 Routines

This section details the error reporting support compenent's routine. A subsection is dedicated to each of this manager's routines and describes the calling sequence, related constants, usage, and status codes.

4.4.1 rtems_status_text - ASCII Version of RTEMS Status

CALLING SEQUENCE:

```
const char *rtems_status_text(
  rtems_status_code status
);
```

STATUS CODES:

Returns a pointer to a constant string that describes the given RTEMS status code.

DESCRIPTION:

This routine returns a pointer to a string that describes the RTEMS status code specified by `status`.

NOTES:

NONE

4.4.2 rtems_error - Report an Error

CALLING SEQUENCE:

```
int rtems_error(
  int         error_code,
  const char *printf_format,
  ...
);
```

STATUS CODES:

Returns the number of characters written.

DESCRIPTION:

This routine prints the requested information as specified by the `printf_format` parameter and the zero or more optional arguments following that parameter. The `error_code` parameter is an error number with either `RTEMS_ERROR_PANIC` or `RTEMS_ERROR_ABORT` bitwise or'ed with it. If the `RTEMS_ERROR_PANIC` bit is set, then then the system is system is shutdown via a call to `_exit`. If the `RTEMS_ERROR_ABORT` bit is set, then then the system is system is shutdown via a call to `abort`.

NOTES:

NONE

4.4.3 rtems_panic - Report an Error and Panic

CALLING SEQUENCE:

```
int rtems_panic(
   const char *printf_format,
   ...
);
```

STATUS CODES:

Returns the number of characters written.

DESCRIPTION:

This routine is a wrapper for the `rtems_error` routine with an implied error level of `RTEMS_ERROR_PANIC`. See `rtems_error` for more information.

NOTES:

NONE

5 Monitor Task

5.1 Introduction

The monitor task is a simple interactive shell that allows the user to make inquries about he state of various system objects. The routines provided by the monitor task manager are:

- `rtems_monitor_init` - Initialize the Monitor Task
- `rtems_monitor_wakeup` - Wakeup the Monitor Task

5.2 Background

There is no background information.

5.3 Operations

5.3.1 Initializing the Monitor

The monitor is initialized by calling `rtems_monitor_init`. When initialized, the monitor is created as an independent task. An example of initializing the monitor is shown below:

```
#include <rtems/monitor.h>
        ...
rtems_monitor_init(0);
```

The "0" parameter to the `rtems_monitor_init` routine causes the monitor to immediately enter command mode. This parameter is a bitfield. If the monitor is to suspend itself on startup, then the `RTEMS_MONITOR_SUSPEND` bit should be set.

5.4 Routines

This section details the monitor task manager's routines. A subsection is dedicated to each of this manager's routines and describes the calling sequence, related constants, usage, and status codes.

5.4.1 rtems_monitor_init - Initialize the Monitor Task

CALLING SEQUENCE:

```
void rtems_monitor_init(
  unsigned32 monitor_flags
);
```

STATUS CODES: NONE

DESCRIPTION:

This routine initializes the RTEMS monitor task. The `monitor_flags` parameter indicates how the server task is to start. This parameter is a bitfield and has the following constants associated with it:

- **RTEMS_MONITOR_SUSPEND** - suspend monitor on startup
- **RTEMS_MONITOR_GLOBAL** - monitor should be global

If the `RTEMS_MONITOR_SUSPEND` bit is set, then the monitor task will suspend itself after it is initialized. A subsequent call to `rtems_monitor_wakeup` will be required to activate it.

NOTES:

The monitor task is created with priority 1. If there are application tasks at priority 1, then there may be times when the monitor task is not executing.

5.4.2 rtems_monitor_wakeup - Wakeup the Monitor Task

CALLING SEQUENCE:

```
void rtems_monitor_wakeup( void );
```

STATUS CODES: NONE

DESCRIPTION:

This routine is used to activate the monitor task if it is suspended.

NOTES:

NONE

5.5 Monitor Interactive Commands

The following commands are supported by the monitor task:

- `help` - Obtain Help
- `pause` - Pause Monitor for a Specified Number of Ticks
- `exit` - Invoke a Fatal RTEMS Error
- `symbol` - Show Entries from Symbol Table
- `continue` - Put Monitor to Sleep Waiting for Explicit Wakeup
- `config` - Show System Configuration
- `itask` - List Init Tasks
- `mpci` - List MPCI Config
- `task` - Show Task Information
- `queue` - Show Message Queue Information
- `extension` - User Extensions
- `driver` - Show Information About Named Drivers
- `dname` - Show Information About Named Drivers
- `object` - Generic Object Information
- `node` - Specify Default Node for Commands That Take IDs

5.5.1 help - Obtain Help

The `help` command prints out the list of commands. If invoked with a command name as the first argument, detailed help information on that command is printed.

5.5.2 pause - Pause Monitor for a Specified Number of Ticks

The `pause` command cause the monitor task to suspend itself for the specified number of ticks. If this command is invoked with no arguments, then the task is suspended for 1 clock tick.

5.5.3 exit - Invoke a Fatal RTEMS Error

The `exit` command invokes `rtems_error_occurred` directive with the specified error code. If this command is invoked with no arguments, then the `rtems_error_occurred` directive is invoked with an arbitrary error code.

5.5.4 symbol - Show Entries from Symbol Table

The `symbol` command lists the specified entries in the symbol table. If this command is invoked with no arguments, then all the symbols in the symbol table are printed.

5.5.5 continue - Put Monitor to Sleep Waiting for Explicit Wakeup

The `continue` command suspends the monitor task with no timeout.

5.5.6 config - Show System Configuration

The `config` command prints the system configuration.

5.5.7 itask - List Init Tasks

The `itask` command lists the tasks in the initialization tasks table.

5.5.8 mpci - List MPCI Config

The `mpci` command shows the MPCI configuration information

5.5.9 task - Show Task Information

The `task` command prints out information about one or more tasks in the system. If invoked with no arguments, then information on all the tasks in the system is printed.

5.5.10 queue - Show Message Queue Information

The `queue` command prints out information about one or more message queues in the system. If invoked with no arguments, then information on all the message queues in the system is printed.

5.5.11 extension - User Extensions

The `extension` command prints out information about the user extensions.

5.5.12 driver - Show Information About Named Drivers

The `driver` command prints information about the device driver table.

5.5.13 dname - Show Information About Named Drivers

The `dname` command prints information about the named device drivers.

5.5.14 object - Generic Object Information

The `object` command prints information about RTEMS objects.

5.5.15 node - Specify Default Node for Commands That Take IDs

The `node` command sets the default node for commands that look at object ID ranges.

Command and Variable Index

There are currently no Command and Variable Index entries.

Concept Index

There are currently no Concept Index entries.