21

Configuring a System

21.1 Configuration Table

The **RTEMS Configuration Table** is used to tailor an application for its specific needs. For example, the user can configure the maximum number of tasks for this application. The address of the user-defined **Configuration Table** is passed as an argument to the **initialize_executive** directive, which **MUST** be the first **RTEMS** directive called. The **RTEMS Configuration Table** is defined in the following C structure:

```
typedef struct {
  void                                *work_space_start;
  rtems_unsigned32                     work_space_size;
  rtems_unsigned32                     maximum_tasks;
  rtems_unsigned32                     maximum_timers;
  rtems_unsigned32                     maximum_semaphores;
  rtems_unsigned32                     maximum_queues;
  rtems_unsigned32                     maximum_messages;
  rtems_unsigned32                     maximum_partitions;
  rtems_unsigned32                     maximum_regions;
  rtems_unsigned32                     maximum_ports;
  rtems_unsigned32                     maximum_periods;
  rtems_unsigned32                     maximum_extensions;
  rtems_unsigned32                     microseconds_per_tick;
  rtems_unsigned32                     ticks_per_timeslice;
  rtems_unsigned32                     number_of_initialization_tasks;
  rtems_initialization_tasks_table    *User_initialization_tasks_table;
  rtems_unsigned32                     number_of_device_drivers;
  rtems_driver_address_table          *Device_driver_table;
  rtems_extensions_table              *User_extension_table;
  rtems_multiprocessing_table         *User_multiprocessing_table;
} rtems_configuration_table;
```

work_space_start	is the address of the **RTEMS RAM Workspace**. This area contains items such as the various object control blocks (TCBs, QCBs, ...) and task stacks. If the address is not aligned on a four-word boundary, then **RTEMS** will invoke the fatal error handler during **initialize_executive**.
work_space_size	is the calculated size of the **RTEMS RAM Workspace**. The section **Sizing the RTEMS RAM Workspace** details how to arrive at this number.
maximum_tasks	is the maximum number of tasks that can be concurrently active (created) in the system including initialization tasks.
maximum_timers	is the maximum number of timers that can be concurrently active in the system.
maximum_semaphores	is the maximum number of semaphores that can be concurrently active in the system.
maximum_message_queues	is the maximum number of message queues that can be concurrently active in the system.
maximum_messages	is the maximum number of messages that can be allocated to the application.
maximum_partitions	is the maximum number of partitions that can be concurrently active in the system.
maximum_regions	is the maximum number of regions that can be concurrently active in the system.
maximum_ports	is the maximum number of ports into dual-port memory areas that can be concurrently active in the system.
maximum_extensions	is the maximum number of dynamic user extension sets that can be concurrently active in the system.
microseconds_per_tick	is number of microseconds per clock tick.
ticks_per_slice	is the number of clock ticks for a timeslice.
number_of_initialization_tasks	is the number of initialization tasks configured. At least one initialization task must be configured.
User_initialization_tasks_table	is the address of the **Initialization Task Table**. This table contains the information needed to create and start each of the initialization tasks. The format of this table will be discussed below.

number_of_device_drivers	is the number of device drivers for the system. There should be the same number of entries in the **Device Driver Table**. If this field is zero, then the **User_driver_address_table** entry should be **NULL**.
Device_driver_table	is the address of the **Device Driver Table**. This table contains the entry points for each device driver. If the **number_of_device_drivers** field is zero, then this entry should be **NULL**. The format of this table will be discussed below.
User_extension_table	is the address of the **User Extension Table**. This table contains the entry points for the static set of optional user extensions. If no user extensions are configured, then this entry should be **NULL**. The format of this table will be discussed below.
User_multiprocessing_table	is the address of the **Multiprocessor Configuration Table**. This table contains information needed by **RTEMS** only when used in a multiprocessor configuration. This field must be **NULL** when **RTEMS** is used in a single processor configuration.

21.2 CPU Dependent Information Table

The **CPU Dependent Information Table** is used to describe processor dependent information required by **RTEMS**. This table is generally used to supply **RTEMS** with information only known by the **Board Support Package**. The contents of this table are discussed in the **CPU Dependent Information Table** chapter of the C **Applications Supplement** document for a specific target processor.

21.3 Initialization Task Table

The **Initialization Task Table** is used to describe each of the user initialization tasks to the Initialization Manager. The table contains one entry for each initialization task the user wishes to create and start. The fields of this data structure directly correspond to arguments to the **task_create** and **task_start** directives. The number of entries is found in the **number_of_initialization_tasks** entry in the **Configuration Table**. The format of each entry in the **Initialization Task Table** is defined in the following C structure:

```
typedef struct {
  rtems_name           name;
  rtems_unsigned32     stack_size;
  rtems_task_priority  initial_priority;
  rtems_attribute      attribute_set;
  rtems_task_entry     entry_point;
  rtems_mode           mode_set;
  rtems_task_argument  argument;
} rtems_initialization_tasks_table;
```

name	is the name of this initialization task.
stack_size	is the size of the stack for this initialization task.
initial_priority	is the priority of this initialization task.
attribute_set	is the attribute set used during creation of this initialization task.
entry_point	is the address of the entry point of this initialization task.
mode_set	is the initial execution mode of this initialization task.
argument	is the initial argument for this initialization task.

A typical declaration for an **Initialization Task Table** might appear as follows:

```
rtems_initialization_tasks_table Initialization_tasks[2] = {
   { INIT_1_NAME,
     1024,
     1,
     DEFAULT_ATTRIBUTES,
     Init_1,
     DEFAULT_MODES,
     1
   },
   { INIT_2_NAME,
     1024,
     250,
     FLOATING_POINT,
     Init_2,
     INTERRUPT_LEVEL(0)|NO_PREEMPT,
     2
   }
};
```

21.4 Driver Address Table

The **Device Driver Table** is used to inform the I/O Manager of the set of entry points for each device driver configured in the system. The table contains one entry for each device driver required by the application. The number of entries is defined in the **number_of_device_drivers** entry in the **Configuration Table**. The format of each entry in the **Device Driver Table** is defined in the following C structure :

```
typedef struct {
  rtems_device_driver_entry initialization;
  rtems_device_driver_entry open;
  rtems_device_driver_entry close;
  rtems_device_driver_entry read;
  rtems_device_driver_entry write;
  rtems_device_driver_entry control;
} rtems_driver_address_table;
```

initialization	is the address of the entry point called by **io_initialize** to initialize a device driver and its associated devices.
open	is the address of the entry point called by **io_open**.
close	is the address of the entry point called by **io_close**.
read	is the address of the entry point called by **io_read**.
write	is the address of the entry point called by **io_write**.
control	is the address of the entry point called by **io_control**.

Driver entry points configured as **NULL** will always return a status code of **SUCCESSFUL**. No user code will be executed in this situation.

A typical declaration for a **Device Driver Table** might appear as follows:

```
rtems_driver_address_table Driver_table[2] = {
   { tty_initialize, tty_open,  tty_close,  /* major = 0 */
     tty_read,       tty_write, tty_control
   },
   { lp_initialize, lp_open,    lp_close,   /* major = 1 */
     NULL,          lp_write,   lp_control
   }
};
```

More information regarding the construction and operation of device drivers is provided in the **I/O Manager** chapter.

21.5 User Extensions Table

The **User Extensions Table** is used to inform **RTEMS** of the optional user-supplied static extension set. This table contains one entry for each possible extension. The entries are called at critical times in the life of the system and individual tasks. The application may create dynamic extensions in addition to this single static set. The format of each entry in the **User Extensions Table** is defined in the following C structure:

```
typedef struct {
  rtems_tasks_create_extension  task_create;
  rtems_tasks_start_extension   task_start;
  rtems_tasks_restart_extension task_restart;
  rtems_tasks_delete_extension  task_delete;
  rtems_tasks_switch_extension  task_switch;
  rtems_tasks_exitted_extension task_exitted;
  rtems_fatal_extension         fatal;
} rtems_extension_table;
```

task_create is the address of the user-supplied subroutine for the TASK_CREATE extension. If this extension for task creation is defined, it is called from the **task_create** directive. A value of **NULL** indicates that no extension is provided.

task_start is the address of the user-supplied subroutine for the TASK_START extension. If this extension for task initiation is defined, it is called from the **task_start** directive. A value of **NULL** indicates that no extension is provided.

task_restart is the address of the user-supplied subroutine for the TASK_RESTART extension. If this extension for task re-initiation is defined, it is called from the **task_restart** directive. A value of **NULL** indicates that no extension is provided.

task_delete is the address of the user-supplied subroutine for the TASK_DELETE extension. If this **RTEMS** extension for task deletion is defined, it is called from the **task_delete** directive. A value of **NULL** indicates that no extension is provided.

task_switch is the address of the user-supplied subroutine for the task context switch extension. This subroutine is called from **RTEMS** dispatcher after the current task has been swapped out but before the new task has been swapped in. A value of **NULL** indicates that no extension is provided. As this routine is invoked after saving the current task's context and before restoring the heir task's context, it is not necessary for this routine to save and restore any registers.

task_begin is the address of the user-supplied subroutine which is invoked immediately before a task begins execution. It is invoked in the context of the beginning task. A value of **NULL** indicates that no extension is provided.

task_exitted is the address of the user-supplied subroutine which is invoked when a task exits. This procedure is responsible for some action which will allow the system to continue execution (i.e. delete or restart the task) or to terminate with a fatal error. If this field is set to **NULL**, the default **RTEMS** task_exitted handler will be invoked.

fatal is the address of the user-supplied subroutine for the FATAL extension. This **RTEMS** extension of fatal error handling is called from the **fatal_error_occurred** directive. If the user's fatal error handler returns or if this entry is **NULL** then the default **RTEMS** fatal error handler will be executed.

A typical declaration for a **User Extension Table** which defines the TASK_CREATE, TASK_DELETE, TASK_SWITCH, and FATAL extension might appear as follows:

```
rtems_extensions_table User_extensions = {
   task_create_extension,
   NULL,
   NULL,
   task_delete_extension,
   task_switch_extension,
   NULL,
   fatal_extension
};
```

More information regarding the user extensions is provided in the **User Extensions** chapter.

21.6 Multiprocessor Configuration Table

The **Multiprocessor Configuration Table** contains information needed when using **RTEMS** in a multiprocessor configuration. Many of the details associated with configuring a multiprocessor system are dependent on the multiprocessor communications layer provided by the user. The address of the **Multiprocessor Configuration Table** should be placed in the **User_multiprocessing_table** entry in the primary **Configuration Table**. Further details regarding many of the entries in the **Multiprocessor Configuration Table** will be provided in the **Multiprocessing** chapter. The format of the **Multiprocessor Configuration Table** is defined in the following C structure:

```
typedef struct {
  rtems_unsigned32  node;
  rtems_unsigned32  maximum_nodes;
  rtems_unsigned32  maximum_global_objects;
  rtems_unsigned32  maximum_proxies;
  rtems_mpci_table *User_mpci_table;
} rtems_multiprocessing_table;
```

node is a unique processor identifier and is used in routing messages between nodes in a multiprocessor configuration. Each processor must have a unique node number. **RTEMS** assumes that node numbers start at one and increase sequentially. This assumption can be used to advantage by the user-supplied MPCI layer. Typically, this requirement is made when the node numbers are used to calculate the address of inter-processor communication links. Zero should be avoided as a node number because some MPCI layers use node zero to represent broadcasted packets. Thus, it is recommended that node numbers start at one and increase sequentially.

maximum_nodes is the number of processor nodes in the system.

maximum_global_objects is the maximum number of global objects which can exist at any given moment in the entire system. If this parameter is not the same on all nodes in the system, then a fatal error is generated to inform the user that the system is inconsistent.

maximum_proxies is the maximum number of proxies which can exist at any given moment on this particular node. A **proxy** is a substitute task control block which represent a task residing on a remote node when that task blocks on a remote object. Proxies are used in situations in which delayed interaction is required with a remote node.

User_mpci_table is the address of the **Multiprocessor Communications Interface Table**. This table contains the entry points of user-provided functions which constitute the multiprocessor communications layer. This table must be provided in multiprocessor configurations with all entries configured. The format of this table and details regarding its entries can be found in the next section.

21.7 Multiprocessor Communications Interface Table

The format of this table is defined in the following C structure:

```
typedef struct {
  rtems_mpci_initialization_entry initialization;
  rtems_mpci_get_packet_entry     get_packet;
  rtems_mpci_return_packet_entry  return_packet;
  rtems_mpci_send_entry           send;
  rtems_mpci_receive_entry        receive;
} rtems_mpci_table;
```

default_timeout is the default maximum length of time a task should block waiting for a response to a directive which results in communication with a remote node. The maximum length of time is a function the user supplied multiprocessor communications layer and the media used. This timeout only applies to directives which would not block if the operation were performed locally.

initialization is the address of the entry point for the initialization procedure of the user supplied multiprocessor communications layer.

get_packet is the address of the entry point for the procedure called by **RTEMS** to obtain a packet from the user supplied multiprocessor communications layer.

return_packet is the address of the entry point for the procedure called by **RTEMS** to return a packet to the user supplied multiprocessor communications layer.

send	is the address of the entry point for the procedure called by **RTEMS** to send an envelope to another node. This procedure is part of the user supplied multiprocessor communications layer.
receive	is the address of the entry point for the procedure called by **RTEMS** to retrieve an envelope containing a message from another node. This procedure is part of the user supplied multiprocessor communications layer.

More information regarding the required functionality of these entry points is provided in the **Multiprocessor** chapter.

21.8 Determining Memory Requirements

Since memory is a critical resource in many real-time embedded systems, **RTEMS** was specifically designed to allow unused managers to be excluded from the run-time environment. This allows the application designer the flexibility to tailor **RTEMS** to most efficiently meet system requirements while still satisfying even the most stringent memory constraints. As result, the size of the **RTEMS** executive is application dependent. A **Memory Requirements** worksheet is provided in the C Applications Supplement document for a specific target processor. This worksheet can be used to calculate the memory requirements of a custom **RTEMS** run-time environment. To insure that enough memory is allocated for future versions of **RTEMS**, the application designer should round these memory requirements up. The following managers may be optionally excluded:

- *signal*
- *partition*
- *region*
- *timer*
- *dual ported memory*
- *semaphore*
- *I/O*
- *message*
- *event*
- *rate monotonic*
- *multiprocessing*
- *user extensions*

RTEMS based applications must somehow provide memory for **RTEMS**' code and data space. Although **RTEMS**' data space must be in RAM, its code space can be located in either ROM or RAM. In addition, the user must allocate RAM for the **RTEMS RAM Workspace**. The size of this area is application dependent and can be calculated using the formula provided in the **Memory Requirements** chapter of the C Applications Supplement document for a specific target processor.

All **RTEMS** data variables and routine names used by **RTEMS** begin with the underscore (_) character followed by an upper-case letter. If **RTEMS** is linked with an application, then the application code should **NOT** contain any symbols which begin with the underscore character and followed by an upper-case letter to avoid any naming conflicts. All **RTEMS** directive names should be treated as reserved words.

21.9 Sizing the RTEMS RAM Workspace

The **RTEMS RAM Workspace** is a user-specified block of memory reserved for use by **RTEMS**. The application should **NOT** modify this memory. This area consists primarily of the **RTEMS** data structures whose exact size depends upon the values specified in the **Configuration Table**. In addition, task stacks and floating point context areas are dynamically allocated from the **RTEMS RAM Workspace**.

The starting address of the **RTEMS RAM Workspace** must be aligned on a four-byte boundary. Failure to properly align the workspace area will result in the **fatal_error_occurred** directive being invoked with the **INVALID_ADDRESS** error code.

A worksheet is provided in the **Memory Requirements** chapter of the C Applications Supplement document for a specific target processor to assist the user in calculating the minimum size of the **RTEMS RAM Workspace** for each application. The value calculated with this worksheet is the minimum value that should be specified as the **work_space_size** parameter of the **Configuration Table**. The user is cautioned that future versions of **RTEMS** may not have the same memory requirements per object. Although the value calculated is sufficient for a particular target processor and release of **RTEMS**, memory usage is subject to change across versions and target processors. The user is advised to allocate somewhat more memory than the worksheet recommends to insure compatibility with future releases for a specific target processor and other target processors. To avoid problems, the user should recalculate the memory requirements each time one of the following events occurs:

- *a configuration parameter is modified,*
- *task or interrupt stack requirements change,*
- *task floating point attribute is altered,*
- ***RTEMS** is upgraded, or*
- *the target processor is changed.*

Failure to provide enough space in the **RTEMS RAM Workspace** will result in the **fatal_error_occurred** directive being invoked with the **UNSATISFIED** error code.