22

Multiprocessing Manager

22.1 Introduction

In multiprocessor real-time systems, new requirements, such as sharing data and global resources between processors, are introduced. This requires an efficient and reliable communications vehicle which allows all processors to communicate with each other as necessary. In addition, the ramifications of multiple processors affect each and every characteristic of a real-time system, almost always making them more complicated.

RTEMS addresses these issues by providing simple and flexible real-time multiprocessing capabilities. The executive easily lends itself to both tightly-coupled and loosely-coupled configurations of the target system hardware. In addition, **RTEMS** supports systems composed of both homogeneous and heterogeneous mixtures of processors and target boards.

A major design goal of the **RTEMS** executive was to transcend the physical boundaries of the target hardware configuration. This goal is achieved by presenting the application software with a logical view of the target system where the boundaries between processor nodes are transparent. As a result, the application developer may designate objects such as tasks, queues, events, signals, semaphores, and memory blocks as global objects. These global objects may then be accessed by any task regardless of the physical location of the object and the accessing task. **RTEMS** automatically determines that the object being accessed resides on another processor and performs the actions required to access the desired object. Simply stated, **RTEMS** allows the entire system, both hardware and software, to be viewed logically as a single system.

22.2 Background

RTEMS makes no assumptions regarding the connection media or topology of a multiprocessor system. The tasks which compose a particular application can be spread among as many processors as needed to satisfy the application's timing requirements. The application tasks can interact using a subset of the **RTEMS** directives as if they were on the same processor. These directives allow application tasks to exchange data, communicate, and synchronize regardless of which processor they reside upon.

The **RTEMS** multiprocessor execution model is multiple instruction streams with multiple data streams (MIMD). This execution model has each of the processors executing code independent of the other processors. Because of this parallelism, the application designer can more easily guarantee deterministic behavior.

By supporting heterogeneous environments, **RTEMS** allows the systems designer to select the most efficient processor for each subsystem of the application. Configuring **RTEMS** for a heterogeneous environment is no more difficult than for a homogeneous one. In keeping with **RTEMS** philosophy of providing transparent physical node boundaries, the minimal heterogeneous processing required is isolated in the MPCI layer.

22.2.1 Nodes

A processor in a **RTEMS** system is referred to as a node. Each node is assigned a unique non-zero node number by the application designer. **RTEMS** assumes that node numbers are assigned consecutively from one to **maximum_nodes**. The node number, **node**, and the maximum number of nodes, **maximum_nodes**, in a system are found in the **Multiprocessor Configuration Table**. The **maximum_nodes** field and the number of global objects, **maximum_global_objects**, is required to be the same on all nodes in a system.

The node number is used by **RTEMS** to identify each node when performing remote operations. Thus, the **Multiprocessor Communications Interface Layer** (MPCI) must be able to route messages based on the node number.

22.2.2 Global Objects

All **RTEMS** objects which are created with the **GLOBAL** attribute will be known on all other nodes. Global objects can be referenced from any node in the system, although certain directive specific restrictions (e.g. one cannot delete a remote object) may apply. A task does not have to be global to perform operations involving remote objects. The maximum number of global objects is the system is user configurable and can be found in the **maximum_global_objects** field in the **Multiprocessor Configuration Table**. The distribution of tasks to processors is performed during the application design phase. Dynamic task relocation is not supported by **RTEMS**.

22.2.3 Global Object Table

RTEMS maintains two tables containing object information on every node in a multiprocessor system: a local object table and a global object table. The local object table on each node is unique and contains information for all objects created on this node whether those objects are local or global. The global object table contains information regarding all global objects in the system and, consequently, is the same on every node.

Since each node must maintain an identical copy of the global object table, the maximum number of entries in each copy of the table must be the same. The maximum number of entries in each copy is determined by the **maximum_global_objects** parameter in the **Multiprocessor Configuration Table**. This parameter, as well as the **maximum_nodes** parameter, is required to be the same on all nodes. To maintain consistency among the table copies, every node in the system must be informed of the creation or deletion of a global object.

22.2.4 Remote Operations

When an application performs an operation on a remote global object, **RTEMS** must generate a Remote Request (RQ) message and send it to the appropriate node. After completing the requested operation, the remote node will build a Remote Response (RR) message and send it to the originating node. Messages generated as a side-effect of a directive (such as deleting a global task) are known as Remote Processes (RP) and do not require the receiving node to respond.

Other than taking slightly longer to execute directives on remote objects, the application is unaware of the location of the objects it acts upon. The exact amount of overhead required for a remote operation is dependent on the media connecting the nodes and, to a lesser degree, on the efficiency of the user-provided MPCI routines.

The following shows the typical transaction sequence during a remote application:

1. *The application issues a directive accessing a remote global object.*
2. ***RTEMS** determines the node on which the object resides.*

3. ***RTEMS** calls the user-provided MPCI routine GET_PACKET to obtain a packet in which to build a RQ message.*
4. *After building a message packet, **RTEMS** calls the user-provided MPCI routine SEND_PACKET to transmit the packet to the node on which the object resides (referred to as the destination node).*
5. *The calling task is blocked until the RR message arrives, and control of the processor is transferred to another task.*
6. *The MPCI layer on the destination node senses the arrival of a packet (commonly in an ISR), and calls the **multiprocessing_announce** directive. This directive readies the **Multiprocessing Server**.*
7. *The **Multiprocessing Server** calls the user-provided MPCI routine RECEIVE_PACKET, performs the requested operation, builds an RR message, and returns it to the originating node.*
8. *The MPCI layer on the originating node senses the arrival of a packet (typically via an interrupt), and calls the **RTEMS multiprocessing_announce** directive. This directive readies the **Multiprocessing Server**.*
9. *The **Multiprocessing Server** calls the user-provided MPCI routine RECEIVE_PACKET, readies the original requesting task, and blocks until another packet arrives. Control is transferred to the original task which then completes processing of the directive.*

If an uncorrectable error occurs in the user-provided MPCI layer, the fatal error handler should be invoked. **RTEMS** assumes the reliable transmission and reception of messages by the MPCI and makes no attempt to detect or correct errors.

22.2.5 Proxies

A **proxy** is an **RTEMS** data structure which resides on a remote node and is used to represent a task which must block as part of a remote operation. This action can occur as part of the **semaphore_obtain** and **message_queue_receive** directives. If the object were local, the task's control block would be available for modification to indicate it was blocking on a message queue or semaphore. However, the task's control block resides only on the same node as the task. As a result, the remote node must allocate a proxy to represent the task until it can be readied.

The maximum number of proxies is defined in the **Multiprocessor Configuration Table**. Each node in a multiprocessor system may require a different number of proxies to be configured. The distribution of proxy control blocks is application dependent and is different from the distribution of tasks.

22.2.6 Multiprocessor Configuration Table

The **Multiprocessor Configuration Table** contains information needed by **RTEMS** when used in a multiprocessor system. This table is discussed in detail in the section **Multiprocessor Configuration Table** of the **Configuring a System** chapter.

22.3 Multiprocessor Communications Interface Layer

The **Multiprocessor Communications Interface Layer** (MPCI) is a set of user-provided procedures which enable the nodes in a multiprocessor system to communicate with one another. These routines are invoked by **RTEMS** at various times in the preparation and processing of remote requests. Interrupts are enabled when an MPCI procedure is invoked. It is assumed that if the execution mode and/or interrupt level are altered by the MPCI layer, that they will be restored prior to returning to **RTEMS**.

The MPCI layer is responsible for managing a pool of buffers called **packets** and for sending these packets between system nodes. Packet buffers contain the messages sent between the nodes. Typically, the MPCI layer will encapsulate the packet within an envelope which contains the information needed by the MPCI layer. The number of packets available is dependent on the MPCI layer implementation.

The entry points to the routines in the user's MPCI layer should be placed in the **Multiprocessor Communications Interface Table**. The user must provide entry points for each of the following table entries in a multiprocessor system:

initialization	initialize the MPCI
get_packet	obtain a packet buffer
return_packet	return a packet buffer
send_packet	send a packet to another node
receive_packet	called to get an arrived packet

A packet is sent by **RTEMS** in each of the following situations:

- *an RQ is generated on an originating node;*
- *an RR is generated on a destination node;*

- *a global object is created;*
- *a global object is deleted;*
- *a local task blocked on a remote object is deleted;*
- *during system initialization to check for system consistency.*

If the target hardware supports it, the arrival of a packet at a node may generate an interrupt. Otherwise, the real-time clock ISR can check for the arrival of a packet. In any case, the **multiprocessing_announce** directive **must** be called to announce the arrival of a packet. After exiting the ISR, control will be passed to the **Multiprocessing Server** to process the packet. The **Multiprocessing Server** will call the **get_packet** entry to obtain a packet buffer and the **receive_entry** entry to copy the message into the buffer obtained.

22.3.1 INITIALIZATION

The INITIALIZATION component of the user-provided MPCI layer is called as part of the **initialize_executive** directive to initialize the MPCI layer and associated hardware. It is invoked immediately after all of the device drivers have been initialized. This component should be adhere to the following specification:

```
procedure user_mpci_initialization(
  configuation          : in     rtems.configuration_table;
  cpu_table             : in     rtems.cpu_table_pointer;
  multiprocessing_table : in     rtems.multiprocessing_table_pointer;
);
```

where **configuration** is the address of the user's **Configuration Table**. Operations on global objects cannot be performed until this component is invoked. The INITIALIZATION component is invoked only once in the life of any system. If the MPCI layer cannot be successfully initialized, the fatal error manager should be invoked by this routine.

One of the primary functions of the MPCI layer is to provide the executive with packet buffers. The INITIALIZATION routine must create and initialize a pool of packet buffers. There must be enough packet buffers so **RTEMS** can obtain one whenever needed.

22.3.2 GET_PACKET

The GET_PACKET component of the user-provided MPCI layer is called when **RTEMS** must obtain a packet buffer to send or broadcast a message. This component should be adhere to the following prototype:

```
procedure user_mpci_get_packet(
  packet :    out rtems.prefix_pointer
);
```

where **packet** is the address of a pointer to a packet. This routine always succeeds and, upon return, **packet** will contain the address of a packet. If for any reason, a packet cannot be successfully obtained, then the fatal error manager should be invoked.

RTEMS has been optimized to avoid the need for obtaining a packet each time a message is sent or broadcast. For example, **RTEMS** sends response messages (RR) back to the originator in the same packet in which the request message (RQ) arrived.

22.3.3 RETURN_PACKET

The RETURN_PACKET component of the user-provided MPCI layer is called when **RTEMS** needs to release a packet to the free packet buffer pool. This component should be adhere to the following prototype:

```
procedure user_mpci_return_packet(
  packet : in     rtems.prefix_pointer
);
```

where **packet** is the address of a packet. If the packet cannot be successfully returned, the fatal error manager should be invoked.

22.3.4 RECEIVE_PACKET

The RECEIVE_PACKET component of the user-provided MPCI layer is called when **RTEMS** needs to obtain a packet which has previously arrived. This component should be adhere to the following prototype:

```
procedure user_mpci_receive_packet(
  packet :    out rtems.prefix_pointer
);
```

where **packet** is a pointer to the address of a packet to place the message from another node. If a message is available, then **packet** will contain the address of the message from another node. If no messages are available, this entry **packet** should contain **NULL**.

22.3.5 SEND_PACKET

The SEND_PACKET component of the user-provided MPCI layer is called when **RTEMS** needs to send a packet containing a message to another node. This component should be adhere to the following prototype:

```
procedure user_mpci_send_packet(
  destination   : in     rtems.unsigned32;
  the_packet    : in     rtems.prefix_pointer
);
```

where **node** is the node number of the destination, **packet** is the address of a packet which containing a message, and **packet_length** is the length of the message in bytes. If the packet cannot be successfully sent, the fatal error manager should be invoked.

If **node** is set to zero, the packet is to be broadcasted to all other nodes in the system. Although some MPCI layers will be built upon hardware which support a broadcast mechanism, others may be required to generate a copy of the packet for each node in the system.

Many MPCI layers use the **packet_length** to avoid sending unnecessary data. This is especially useful if the media connecting the nodes is relatively slow.

22.3.6 Supporting Heterogeneous Environments

Developing an MPCI layer for a heterogeneous system requires a thorough understanding of the differences between the processors which comprise the system. One difficult problem is the varying data representation schemes used by different processor types. The most pervasive data representation problem is the order of the bytes which compose a data entity. Processors which place the least significant byte at the smallest address are classified as **little endian** processors. Little endian byte-ordering is shown below:

Byte 3	**Byte 2**	**Byte 1**	**Byte 0**

Conversely, processors which place the most significant byte at the smallest address are classified as **big endian** processors. Big endian byte-ordering is shown below:

Byte 0	**Byte 1**	**Byte 2**	**Byte 3**

Unfortunately, sharing a data structure between big endian and little endian processors requires translation into a common endian format. An application designer typically chooses the common endian format to minimize conversion overhead.

Another issue in the design of shared data structures is the alignment of data structure elements. Alignment is both processor and compiler implementation dependent. For example, some processors allow data elements to begin on any address boundary, while others impose restrictions. Common restrictions are that data elements must begin on either an even address or on a long word boundary. Violation of these restrictions may cause an exception or impose a performance penalty.

Other issues which commonly impact the design of shared data structures include the representation of floating point numbers, bit fields, decimal data, and character strings. In addition, the representation method for negative integers could be one's or two's complement. These factors combine to increase the complexity of designing and manipulating data structures shared between processors.

RTEMS addressed these issues in the design of the packets used to communicate between nodes. The **RTEMS** packet format is designed to allow the MPCI layer to perform all necessary conversion without burdening the developer with the details of the **RTEMS** packet format. As a result, the MPCI layer must be aware of the following:

- *All packets must begin on a four byte boundary.*
- *Packets are composed of both **RTEMS** and application data. All **RTEMS** data is treated as thirty-two (32) bit unsigned quantities and is in the first **MINIMUM_UNSIGNED32S_TO_CONVERT** thirty-two (32) quantities of the packet.*
- *The **RTEMS** data component of the packet must be in native endian format. Endian conversion may be performed by either the sending or receiving MPCI layer.*
- ***RTEMS** makes no assumptions regarding the application data component of the packet.*

22.4 Operations

22.4.1 Announcing a Packet

The **multiprocessing_announce** directive is called by the MPCI layer to inform **RTEMS** that a packet has arrived from another node. This directive can be called from an interrupt service routine or from within a polling routine.

22.5 Directives

This section details the additional directives required to support **RTEMS** in a multiprocessor configuration. A subsection is dedicated to each of this manager's directives and describes the calling sequence, related constants, usage, and status codes.

22.5.1 MULTIPROCESSING_ANNOUNCE - Announce the arrival of a packet

CALLING SEQUENCE:

```
procedure multiprocessing_announce;
```

DIRECTIVE STATUS CODES:

NONE

DESCRIPTION:

This directive informs **RTEMS** that a multiprocessing communications packet has arrived from another node. This directive is called by the user-provided **MPCI**, and is only used in multiprocessor configurations.

NOTES:

This directive is typically called from an ISR.

This directive will almost certainly cause the calling task to be preempted.

This directive does not generate activity on remote nodes.