CSR	MASSACHUSETTS INSTITUTE OF TECHNOLOGY CENTER FOR SPACE RESEARCH CAMBRIDGE, MASSACHUSETTS 02139	
REVISION LOG	**TITLE: Software Detailed Design** **System Startup and Patch Management**	**DOC. NO.** **36-53242 Rev. 01**

Revision	Date (mm/dd/yy)	ECO No.	Page(s) Affected	Reason	Approval
01	4/10/96	36-574	all	Initial version. Incorporated comments from review.	

14.0 System Startup and Patch Management (36-53242 01)

14.1 Purpose

The purpose of the System Startup and Patch Management units are to initialize and patch the instrument software running on the Back End Processor.

14.2 Uses

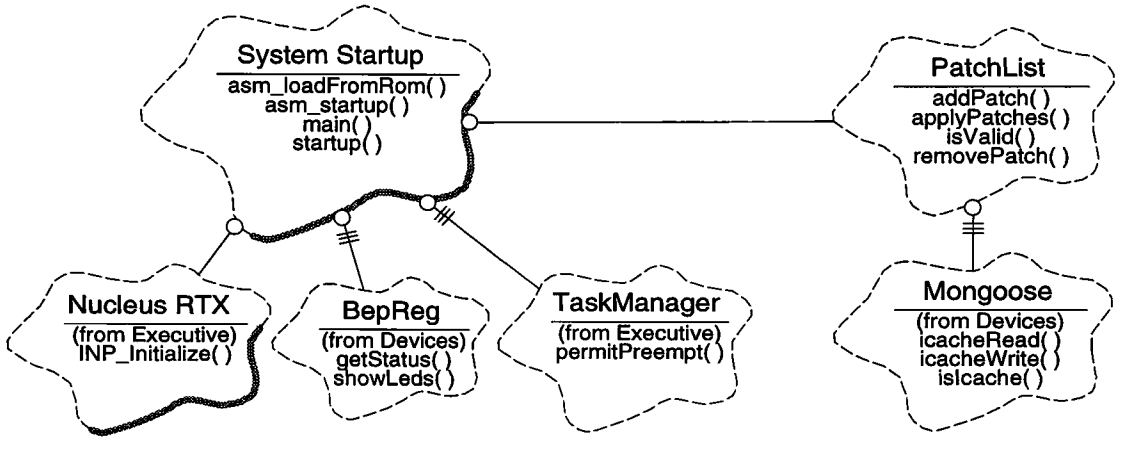
The following lists the use of the System Startup and Patch Management modules and classes:

Use 1:: Copy code and data from the bulk ROM into I-cache and D-cache on the BEP
Use 2:: Initialize processor registers and startup stack
Use 3:: Copy interrupt vector code into I-cache
Use 4:: Apply installed patches
Use 5:: Declare all global objects
Use 6:: Initialize and launch the real-time executive

14.3 Organization

Figure 37 illustrates the class relationships used by the **`System Startup`** routines, and the **`PatchList`** class. Although the **`System Startup`** routines are implemented in assembler or as stand-alone functions, they are represented as member functions of a class-utility in the figure (as indicated by the shadow on the cloud).

FIGURE 37. System Startup and PatchList relationships



System Startup - This is a collection of assembler and C++ subroutines which load and initialize the instrument software. They provide the ability to load code and initialized data sections from the bulk ROM into the Back End Processor's Instruction and Data cache (`asm_loadFromRom`), perform low-level processor initialization (`asm_startup`), perform patch installation, executive initialization and launch the executive (`startup`), and invoke global constructors once the executive is running and enable execution of the remaining tasks in the system (`main`). This class uses the **BepReg** class to determine if the most recent reset was a commanded reset, and if so uses the **PatchList** class to install the system patches.

PatchList - This class is responsible for maintaining the system patch list, and for installing the patches when invoked by startup. It provides functions which append a patch to the system patch list (`addPatch`), remove a patch from the list and compact the list (`removePatch`), determine if the patch list has been corrupted (`isValid`), and apply the set of patches during startup (`applyPatches`).

Nucleus RTX - This represents the collection of routines provided by the Nucleus Real-Time Executive. The startup software uses `INP_Initialize()` to initialize and launch the executive.

14.4 Startup and Patch List Design Issues

14.4.1 Patch List Memory Map and Organization

The system patch list is maintained in the Instruction Cache RAM (I-cache) on the Back End Processor (BEP). Since writing to I-cache must be accomplished via the Mongoose Command Status Interface, corruption of the list due to stray pointers or crashes is unlikely. However, whenever the BEP receives a power-on reset, or if the BEP is reset while removing a patch from the patchlist, there is the potential for the list to become corrupted. As such, the list maintains a checksum of the contents of the list. If the checksum is invalid, patches will not be applied.

Table 15 illustrates the layout of the system patch list. This list grows backwards, from the end of I-cache toward the start of I-cache.

TABLE 15. I-cache Patch List Layout

Region		Address	Byte Size	Description
Patch Area	End of List Pointer	0x800ffffc	4	This points to the next location where a patch may be added
	Checksum	0x800ffff8	4	This is the current 32-bit XOR checksum of the current contents of the patch list.
	Patch Nodes		Variable	These are a collection of patch nodes. The format for each node is as follows:

Name	Offset	Bytes	Description
Patch Id	4*Length +8	4	Unique code to identify the patch
Destination	4*Length +4	4	Points to where to write patch data
Length	4*Length	4	Number of 32-bit words in patch
Data	0	4*Length	Data to write at startup

	Current End of Patch List	End of List Pointer		This is next location after the last patch node in the patch list. This location is pointed to by the End of List Pointer
	...			
	Lowest Patch Address	0x800d7c00		This is the lowest address usable by the patch list. It cannot be moved without a patch.
Bad Pixel/Column Maps		0x800cac00	53248	
Compression Tables		0x800c2c00	32760	
Parameter Blocks		0x800c0400	10240	
System Configuration		0x800c0000	1024	
Code		0x8008000	262144	

14.4.2 Bulk ROM Memory Map and Organization

The ACIS Bulk Flight ROM consists of a simple loading routine, followed by the loadable code and initialized data sections. During system boot, the Back End Processor's Boot ROM jumps to the start of the bulk ROM. The code located at the start of the bulk ROM proceeds to copy code and data sections from the bulk ROM into the BEP's I-cache and D-cache. Once the sections have been copied, the loader jumps to the starting execution address of the loaded code. Table 16 illustrates the layout of the bulk ROM.

TABLE 16. ACIS Flight Software Bulk ROM Layout

Name	Address	Byte Size	Description
Loader Routine	see final memory map	see final memory map	This is the routine which copies the sections from the bulk ROM into the BEP.
Load Sections	0xb800010c	Variable	These are the sections to load into the BEP. Each section has the following format:

Name	Offset	Bytes	Description
Data	8	4*Length	This is the data to copy into the BEP.
Length	4	4	This is the total number of 32-bit words in the section
Destination	0	4	This points to the BEP address to load the data

Checksum	0xb8000108	4	This is a 32-bit wide, XOR checksum of the load image.
Start Address	0xb8000104	4	This contains the address to jump to once the load is complete.
Section Count	0xb8000100	4	This contains the number of sections to be loaded into the BEP.
Jump Vector	0xb8000000	8	This is a code fragment which jumps to the start of the Loader Routine

14.4.3 Executive Configuration

Nucleus RTX is configured using fixed tables at fixed locations. Table 17 lists the items which must be initialized prior to starting the executive. Refer to the Nucleus RTX include file, **in_defs.h**, for the definitions of these structures, and the definition of the END_OF_LIST constant.

TABLE 17. Nucleus RTX Configuration Items

Name	Type	Description
SKD_System_Stack_Ptr	**unsigned***	This must be initialize to point to the start of the system stack space used by the executive.
IN_System_Stack_Size	**unsigned**	This must be initialize to the total number of 32-bit words to use for the system stack.
IN_Last_Memory_Address	**unsigned**	This variable must be initialized to the last memory location to be used by the executive. The executive uses the area between the address of *IN_Last_Address_Used* and the location referenced by this variable for stacks, task control structures, etc.
IN_Last_Address_Used	**unsigned**	This variable must be initialized to reference the first location used by Nucleus RTX for stacks, task control structures, etc.
IN_Fixed_Partitions	**struct IN_FIXED_PARTITION_STRUCT[]**	This is an array of partition definition structures. The first field of entry just after the last used entry must contain the value "END_OF_LIST." This array must contain an initialized entry for each **MemoryPool** instance. The index of the entry serves as the RTX identifier for the pool.
IN_System_Queues	**struct IN_QUEUE_DEFINITION_STRUCT[]**	This is an array of queue definition structures. The first field of the entry just after the last defined queue must contain the value "END_OF_LIST." This array must contain an initialized entry for each **Queue** instance in the system. The index of the entry serves as the RTX identifier for the queue.
IN_System_Event_Groups	**unsigned**	This is the count of event groups used by the instrument. For ACIS, this must be initialized to the total number of tasks in the system.
IN_System_Resources	**unsigned**	This is the count of semaphores used by the instrument. For ACIS, this must be initialized to the total number of **Semaphore** instances used by the system
IN_System_Tasks	**struct IN_TASK_DEFINITION_STRUCT[]**	This is an array of task definition structures. The first field of entry just after the last defined task must contain the value "END_OF_LIST." There must be one entry for each defined **Task** instance within the instrument. The index of an entry serves as the RTX identifier for the task.

14.4.4 Global Constructors

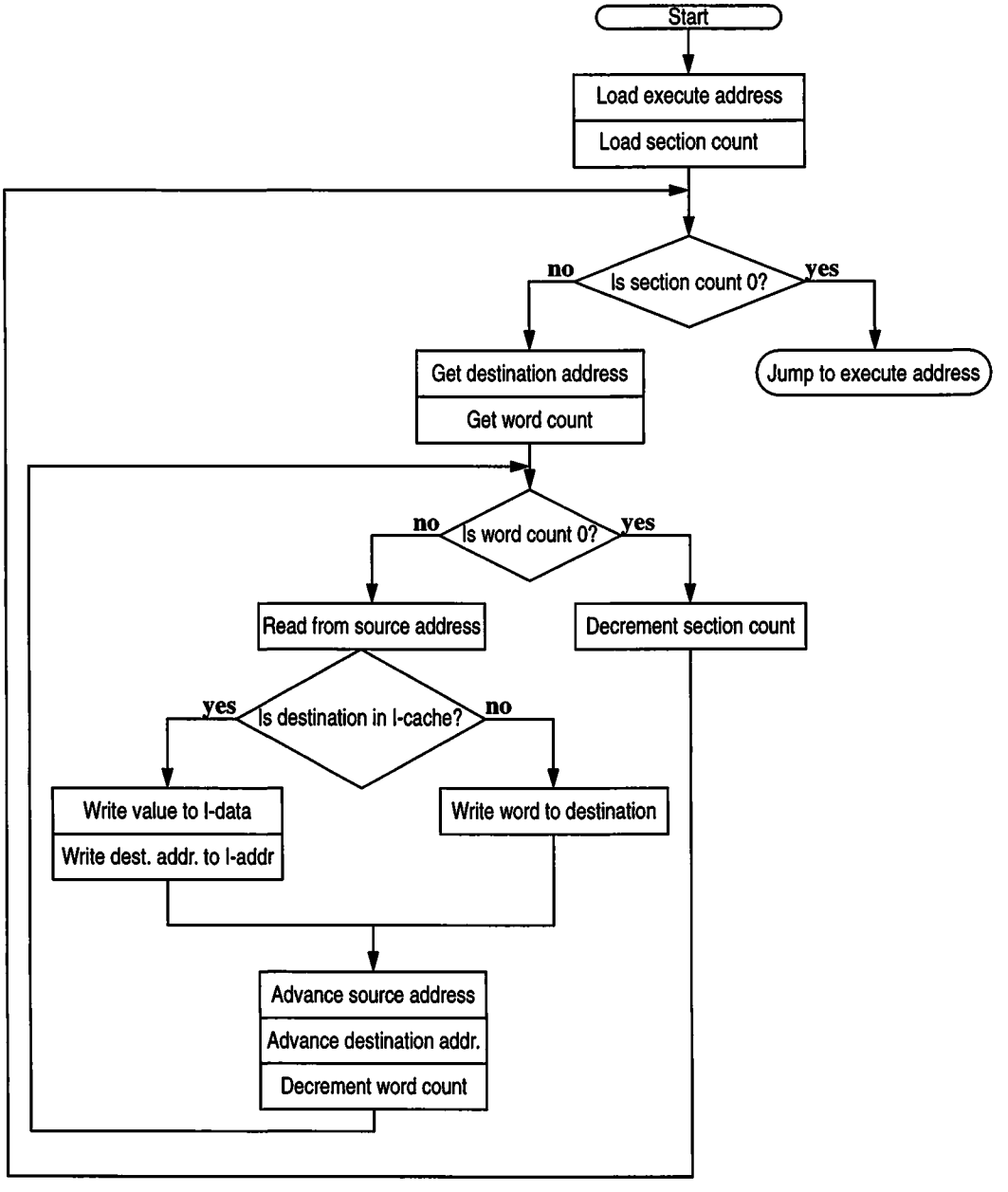
In C++, the constructors for global instances are invoked in the order they appear in an object file. The order in which different object files are processed is undefined by the language. In order to ensure deterministic initialization order, the **System Startup** uses a single source file for all global objects, **globals.C**. This file contains the Nucleus RTX initialization structures, and each global declaration, in the order they are to be initialized.

14.5 Scenarios

14.5.1 Use 1: Load code and data from the bulk ROM into the BEP

Figure 38 illustrates the sequence of operations which load code and data from the bulk ROM into the Back End Processor RAM.

FIGURE 38. Loading Code and Data



14.5.2 Use 2: Initialize processor registers and startup stack

The code located at the startup execution address is responsible for initializing the some of the Mongoose and R3000's processor registers and setting up a stack for the C++ startup routines. Table 18 lists the register values which must be initialized. The R3000 symbols are defined in **mips.h**, and the Mongoose register symbols are defined in **mongoose.h**. (NOTE: By convention, C and C++ assume that uninitialized data section, .bss, be zeroed during startup, ACIS performs this action after the patch list has been applied. Code prior to and during patching shall not rely on uninitialized data having a value of zero).

TABLE 18. Startup Mongoose and R3000 Register Initialization

Register	Initial Value	Description
C0_SR (C0:$12)	SR_BEV | SR_CU0	R3000 Co-processor 0 Status Register: This register must be initialized to use the Boot Exception Vector, and to have Co-processor 0 enabled. During startup, all R3000 device interrupts are disabled.
C0_CAUSE (C0:$13)	zero	R3000 Co-processor 0 Cause Register: Writing a zero to the cause register ensures that there are no software interrupts pending.
M_CFGREG (Mongoose)	CR_NODMA | CR_WAITST1	Mongoose Configuration Register: This register is initialized to ensure that no Mongoose DMA transfers are underway, and that the processor use 1 wait-state for all bulk memory and devices.
M_MASK (Mongoose)	zero	Mongoose Interrupt Mask Register: Writing a zero to the Mongoose interrupt mask disables all Mongoose device interrupts.
gp (R3000:$28)	_gp	R3000 Global Pointer: This register is initialized to the linker defined value for the global pointer.
sp (R3000:$29)	startup_stack + stack size - 24	R3000 Stack Pointer: The stack pointer is initialized to point at the end of the stack, minus 24 bytes to support the minimum reserved stack space required by functions which call other functions (i.e. non-leaf functions).

Once the registers have been initialized, the startup assembler code jumps to the C++ startup routine, `startup()`.

14.5.3 Use 3: Copy interrupt vector code into I-cache

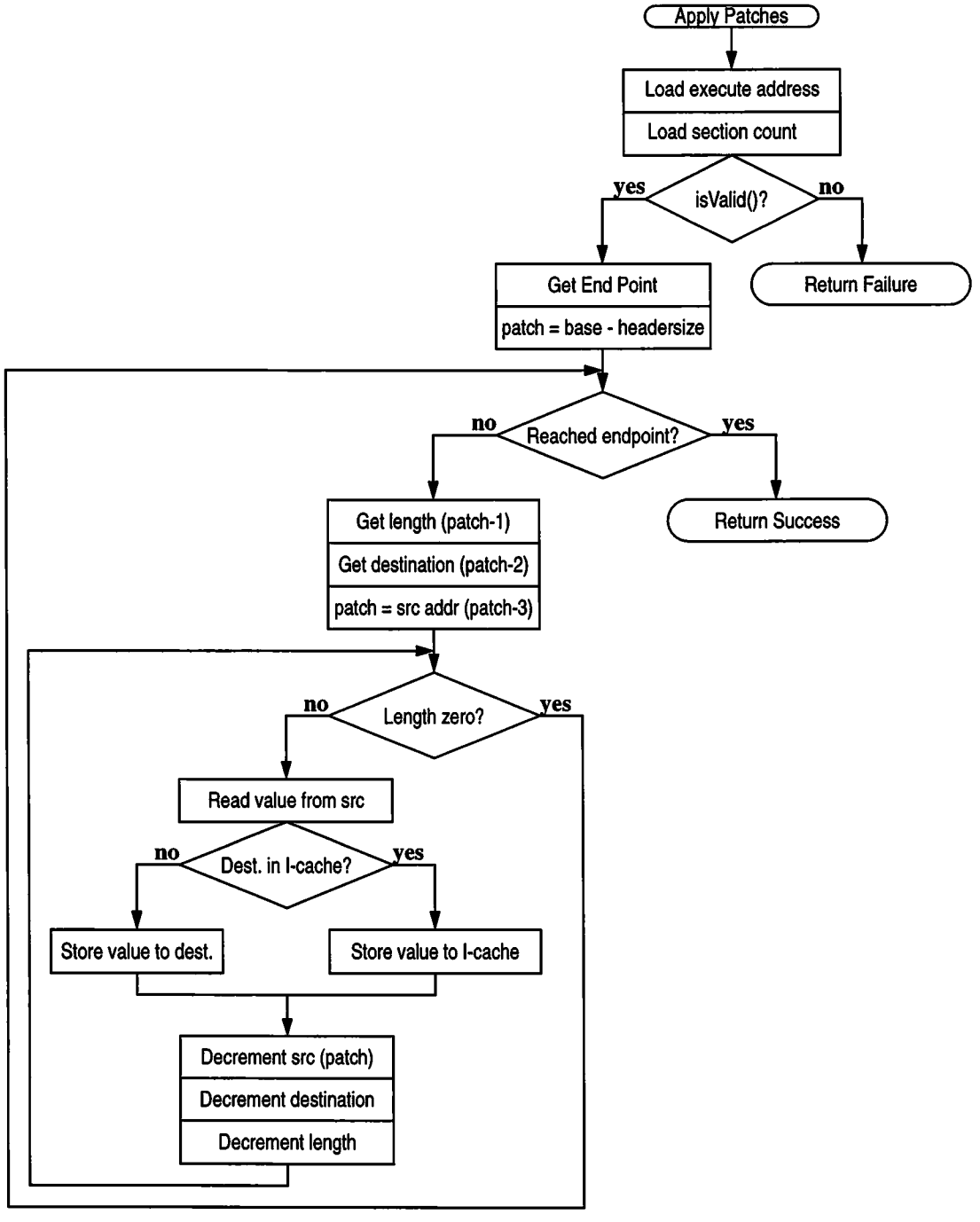
Prior to enabling interrupts on the R3000, the ACIS software must copy a code-fragment which jumps to the ACIS interrupt handler into the General Exception area of I-cache. On ACIS, this is accomplished by providing the appropriate code section in the bulk ROM (see Section 14.4.2 and Section 14.5.3). This causes the interrupt vector code to be loaded into I-cache when all of the other code and initialized data sections are loaded.

During the development process, however, when code is loaded via the ROM monitor across a serial interface, rather than from the instrument's bulk ROM, the startup code is required to copy the interrupt vector code-fragment into I-cache. This is because the ROM monitor uses non-boot interrupts when loading the images. The installed R3000 interrupt handler must then forward exceptions and UART interrupts to the original monitor's handler.

14.5.4 Use 4: Apply installed patches

The first action taken by the C++ startup routine is to determine if the instrument was commanded to reset, and if so, install the patch list. The startup routine uses *bepReg*.`getStatus`() to obtain the value of the Back End Processor's status register. If processor was commanded to reset (rather than watchdog reset, or power-on reset), startup calls *patchList*.`applyPatches`() to install the contents of the patch list. Figure 39 illustrates the process used to install the patch list.

FIGURE 39. Install patches



14.5.5 Use 5: Declare all global objects

In order to ensure a deterministic initialization sequence, the **System Startup** unit declares all global class instances in one file, **globals.C**. In general, system is initialized from the bottom up. The device classes are declared first, followed by the executive classes, followed by command handlers, and other protocols classes. Finally, the application classes, are declared. The exact order of the global class declarations (and subsequent initialization) is determined by the final implementation, and shall be provided in the AS-BUILT Detailed Design (MIT 36-53200).

14.5.6 Use 6: Initialize and launch the real-time executive

The **System Startup** uses patchable, initialized Nucleus RTX Configuration structures to configure the system's tasks, semaphores, and event groups. Since the memory partitions and queues are coupled to and rely entirely on the allocation of telemetry buffers in the Back End Processor's bulk memory, the **System Startup** uses a telemetry buffer configuration table to determine which telemetry buffer pools are required, the size of the packets maintained by each pool, and the number of buffers in each pool. The main initialization routine, `startup()`, calls `setupRtx()` to initialize the RTX memory partition and queue structures corresponding to each telemetry buffer pool, and establish the total number of telemetry packets in the instrument. This total is then used to initialize the RTX queue used for the telemetry manager's (**TlmManager**) telemetry packet buffer transmission queue.

14.6 System Startup Routines

Documentation:

This is a collection of assembler and C++ subroutines which initialize the instrument software, apply the patchlist, and launch the real-time executive.

Export Control: Public

Uses:

PatchList
Nucleus RTX
BepReg
TaskManager

Interface:

Operations:

`asm_loadFromRom()`
`asm_startup()`
`main()`
`setupRtx()`
`startup()`

Concurrency: Sequential

Persistence: Transient

14.6.1 asm_loadFromRom()

Public member of: **System Startup**

Return Class: **void**

Documentation:

This is an R3000 assembler routine which runs directly from the bulk ROM. This routine copies the code and data sections from the bulk ROM into the Back End Processor's Instruction and Data caches.

Concurrency: Sequential

14.6.2 asm_startup()

Public member of: **System Startup**

Return Class: **void**

Documentation:

This is an R3000 assembler routine which initializes the R3000 processor registers, zeros the uninitialized data section of memory, .bss (beginning at the address indicated by the linker symbol "_bss" and ending at the address indicated by "_end"), sets up a stack for use during startup, installs the interrupt vector code at the start of I-cache, and branches to the main C++ startup code.

Concurrency: Sequential

14.6.3 main()

Public member of: **System Startup**

Return Class: **void**

Documentation:

This is the routine installed to be the first routine invoked when the executive starts. It is established as the system initialization thread by startup as a low-priority task, which executes upon starting the executive, and has preemption disabled. Once the executive starts, it invokes this task. The C++ compiler embeds a call to `__main()` within this function, which invokes all of the global constructor routines. Once the constructors have been invoked, this routine tells the *taskManager* to allow preemption. At this point the higher priority tasks are run. In case all tasks suspend, this function enters an infinite loop to prevent the task from returning to the executive.

Concurrency: Synchronous

14.6.4 setupRtx()

Public member of: **System Startup**

Return Class: **void**

Documentation:

This function is responsible for setting up the Nucleus RTX configuration tables. The function iterates through the *tlmPoolConfig* table, declared in **globals.C**. For each entry in the table, it initializes the corresponding entry in the *IN_Fixed_Partitions* and *IN_System_Queues* tables, also declared in **globals.C**. It then adds the number of packets in the entry's pool to *totalPacketCount*. Once the entry table has been processed, it initializes remaining entry in *IN_System_Queues* for the telemetry manager's (*tlmManager*) packet transmission queue.

Concurrency: Sequential

14.6.5 startup()

Public member of: **System Startup**

Return Class: **void**

Documentation:

This is the main C++ startup routine. This function installs patches, initializes the real-time executive, and launches the executive. This routine uses *bepReg*.getStatus() to obtain the contents of the Back End Register's Status Register. If the status indicates a commanded reset, it calls *patchList*.applyPatches() to install the patch list. The function then calls setupRtx() to configure the telemetry packet buffer partitions and queues. Finally, startup() calls INP_Initialize() to initialize ***Nucleus RTX*** and start the executive. Once the executive is running, the only task enabled to run, main(), is invoked.

Concurrency: Sequential

14.7 Class PatchList

Documentation:

This class represents the list of ACIS patches.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **none**

Implementation Uses:

Mongoose

Public Interface:

Operations:
```
addPatch()
applyPatches()
isValid()
removePatch()
```

Protected Interface:

Operations:
```
computeChecksum()
findPatch()
updateChecksum()
```

Private Interface:

Has-A Relationships:

unsigned* const *patchBase*: This is the base address of the patch area in I-cache.

Concurrency: Guarded

Persistence: Persistent

14.7.1 addPatch()

Public member of: **PatchList**

Return Class: **Boolean**

Arguments:

unsigned *patchId*
unsigned* *dstAddress*
unsigned *datalen*
const unsigned* *patchData*

Documentation:

This function adds a patch to the system patch list. *patchId* is the identifier for the patch to add. *dstAddress* is the destination address that the patch is written to when applied the next time the instrument receives a commanded reset. *datalen* is the number of 32-bit data words in the patch, and *patchData* points to the data belonging to the patch. If a patch already exists with the same *patchId*, the function returns *BoolFalse* and the patch is not installed. If no conflicting patch is found, the new patch is appended to the end of the patch list and the function returns *BoolTrue*.

Semantics:

This function concatenates the patch to the list, updates the checksum of the list and then advances the list endpoint. If a reset occurs before the checksum is stored and the endpoint is advanced, the patch is not installed. If a reset occurs after the checksum is stored, but before the endpoint is advanced, the entire list will be flagged as invalid.

Concurrency: Guarded

14.7.2 applyPatches()

Public member of: **PatchList**

Return Class: **Boolean**

Documentation:

This function checks the checksum of the patch list. If the list is valid, it traverses the list and applies each patch. If successful, the function returns *BoolTrue*. If the list is invalid, it returns *BoolFalse*. This function is implemented as a static member function to allow patches to be applied during startup before any of the C++ constructors are invoked. See Section 14.5.4 for a detailed description of the operation of this function.

Concurrency: Sequential

14.7.3 computeChecksum()

Protected member of: **PatchList**

Return Class: **unsigned**

Documentation:

This function computes the checksum of the current patchlist, and returns the computed value. This function uses mongoose.icacheRead() to obtain the current end of the patch list. It then sets the initial sum value to all 1s, and iterates through each word in the patchlist and XORs the word with the current sum. Once the list has been processed, the function returns the sum.

Concurrency: Guarded

14.7.4 findPatch()

Protected member of: **PatchList**

Return Class: **unsigned***

Arguments:

unsigned *patchId*

Documentation:

This function searches the patchlist for the patch entry associated with *patchId* and returns a pointer to the start of the patch. If no such patch is found, the function returns 0. The function uses *mongoose*.icacheRead() to obtain the end point of the list. It then traverses the list until it reaches the end point, or it finds a patch whose identifier matches *patchId*. If it finds a match, the function returns the address of the matching patch. If not, it returns 0.

Concurrency: Guarded

14.7.5 isValid()

Public member of: **PatchList**

Return Class: **Boolean**

Documentation:

This function checks the checksum of the patch list. If the list is valid, it returns *BoolTrue*. If the patch list has been corrupted, it returns *BoolFalse*. This function is implemented as a static member function to allow it to be used during startup, before any of the C++ constructors have been invoked. This function calls computeChecksum() to compute and return the checksum of the data currently stored, and compares the result with the value located in the patch list header. If the values match, the function returns *BoolTrue*, otherwise, it returns *BoolFalse*.

Concurrency: Guarded

14.7.6 removePatch()

Public member of: **PatchList**

Return Class: **Boolean**

Arguments:

unsigned *patchId*

Documentation:

This function searches the patch list for a patch containing patchId. If found, the function removes the patch from the list and returns *BoolTrue*. If the patch is not found, it returns *BoolFalse*.

Semantics:

Search until the node is found, then copy the remaining patch area on top of the removed patch. Update the checksum and adjust the end of list pointer. If a reset occurs during the compacting operation, the checksum will be invalid and the list will be flagged invalid.

Concurrency: Guarded

14.7.7 updateChecksum()

Protected member of: **PatchList**

Return Class: **void**

Documentation:

This function computes and stores the checksum of the current patch list. The new checksum is computed using computeChecksum(), and is stored using *mongoose*.icacheWrite().

Concurrency: Guarded