CSR

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CENTER FOR SPACE RESEARCH
CAMBRIDGE, MASSACHUSETTS 02139

REVISION LOG

TITLE: Software Detailed Design
Command Management Classes

DOC. NO. 36-53213

Revision	Date (mm/dd/yy)	ECO No.	Page(s) Affected	Reason	Approval
—	2/6/95	—	all	Initial version for design walkthrough	
A	5/10/95	36-246	all	Incorporated review comments. ~~Added sol~~ Updated class definitions	5/22/95

13.0 Command Management Classes (36-53213 A)

13.1 Purpose

The purpose of the Command Management system is to receive, execute and log software commands received by the instrument software.

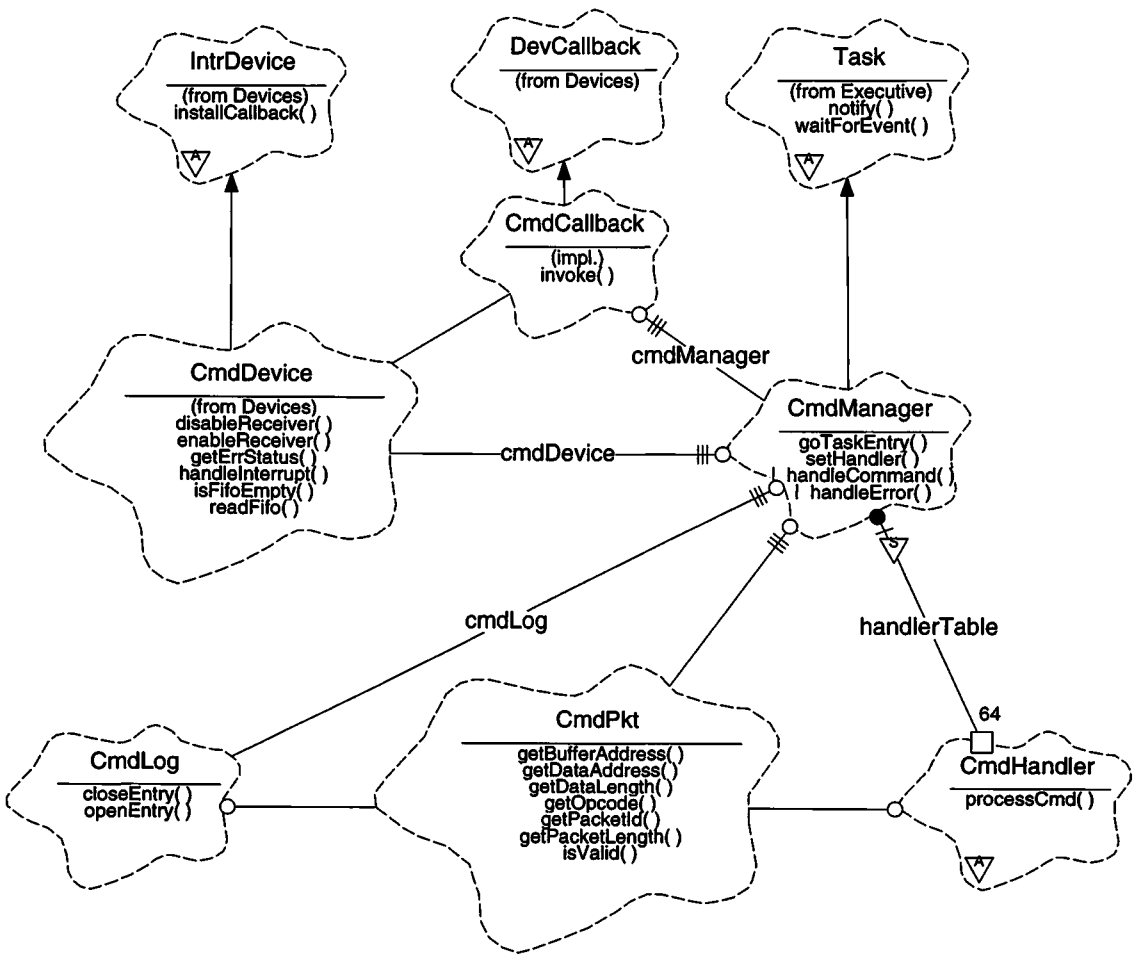
13.2 Uses

Use 1:: Acquire, execute and log commands
Use 2:: Handle command device and protocol errors

13.3 Organization

Figure 39 illustrates the top-level classes and their relationships involved in processing commands

FIGURE 39. Command Management Class Relationships



CmdManager- The **CmdManager** class is a subclass of ***Executive***::**Task** (see Section 12.0). There is only one instance of this class, called *cmdManager*. This class is responsible for actively waiting for commands or errors from the command device, and for executing the commands, and recovering from errors. This class uses the ***Devices***::**CmdDevice** class to provide an interface to the command hardware, the **CmdCallback** class to install an interrupt callback in the **CmdDevice**, the **CmdLog** class to log and echo processed commands, and the **CmdPkt** class to contain the body of a command packet. The **CmdManager** also contains a table of 64 pointers to subclass of **CmdHandler**. It uses the *opcode* field of a given command packet to lookup the corresponding handler from this table.

CmdHandler- This class is an abstract class which represents an object responsible for executing a particular command or group of commands. Every subclass of **CmdHandler** is responsible for implementing a command-specific `processCmd()` member function. This function verifies the command arguments, rejecting illegal commands, and processing accepted commands. Redundant command argument checking by other classes should be minimized.

CmdPkt- This class is used to hold the contents of a command packet, and to provide access to the packet's header information, and to its data.

CmdCallback- This class is a subclass of ***Devices***::**DevCallback**. It is used by the **CmdManager** to obtain control during processing of command interrupts.

CmdLog- This class is responsible for acknowledging the receipt and execution of commands. It implements both the command logging features and command echo features. This class is not described in detail in this section. The details of this class are described in Section TBD.

Task - This class is defined by the ***Executive*** class category, and is described in Section 12.0.

CmdDevice - This class is a subclass of **IntrDevice**, and is defined by the ***Devices*** class category, and is described in Section 8.0.

IntrDevice - This class is defined by the ***Devices*** class category, and is described in Section 6.0.

DevCallback - This class is defined by the ***Devices*** class category, and is described in Section 6.0.

13.4 Command Processing Assumptions and Restrictions

13.4.1 Packet Format

The ACIS instrument software receives commands in the form of a series of 16-bit command words, known as a command packet. Each packet starts with a header, which is followed with any operation-specific data. The instrument operation selected by the command is specified using a Command Opcode field in the header.

The command manager assumes that all command packets appear as a series of 16-bit words, starting with the following header:

TABLE 14. Command Packet Header

Word	Field	Description	Min. Value	Max. Value
0	Packet Length	This field contains the length of the packet in 16-bit words.	3	256
1	Packet Identifier	This field contains an arbitrary number which the instrument uses solely for logging purposes.	0	65,535
2	Command Opcode	This field identifies which command to execute.	0	63
3 - (Packet Length - 1)	Data	The remainder of the packet contains opcode-specific data.		

This format is encapsulated using the **`CmdPkt`** class. The **`CmdPkt`** class provides access functions to the length (`getPacketLength`), identifier (`getPacketId`) and opcode fields (`getOpcode`), and also provides functions which return a pointer to the data area (`getDataAddress`), and the length of the contained data (`getDataLength`). In order to support echoing of commands, the **`CmdPkt`** class also provides access functions which return a pointer to the raw command buffer (`getBufferAddress`). NOTE: The length of the packet within the buffer can be determined using `getPacketLength`.

13.4.2 Processing Time

In order to eliminate the need to have interrupt handlers drain the command FIFO, the instrument software assumes that commands will not be sent faster than one command packet per Minor Telemetry Frame (i.e. about 4 commands per second), and will accept the real-time requirement that the **`CmdManager`** class must be capable of reading a command from the **`CmdDevice`** within 1 Minor Telemetry Frame (~250ms) of being written into the FIFO (including all interrupt overhead).

NOTE: The baseline requirement is to ensure that all commands be disposed of by the **`CmdManager`** in under 250ms. If a certain command causes the **`CmdManager`** to miss this deadline, then the additional delay required by the command shall be published in the ACIS Software Operators Manual.

13.4.3 Delays between groups of packets

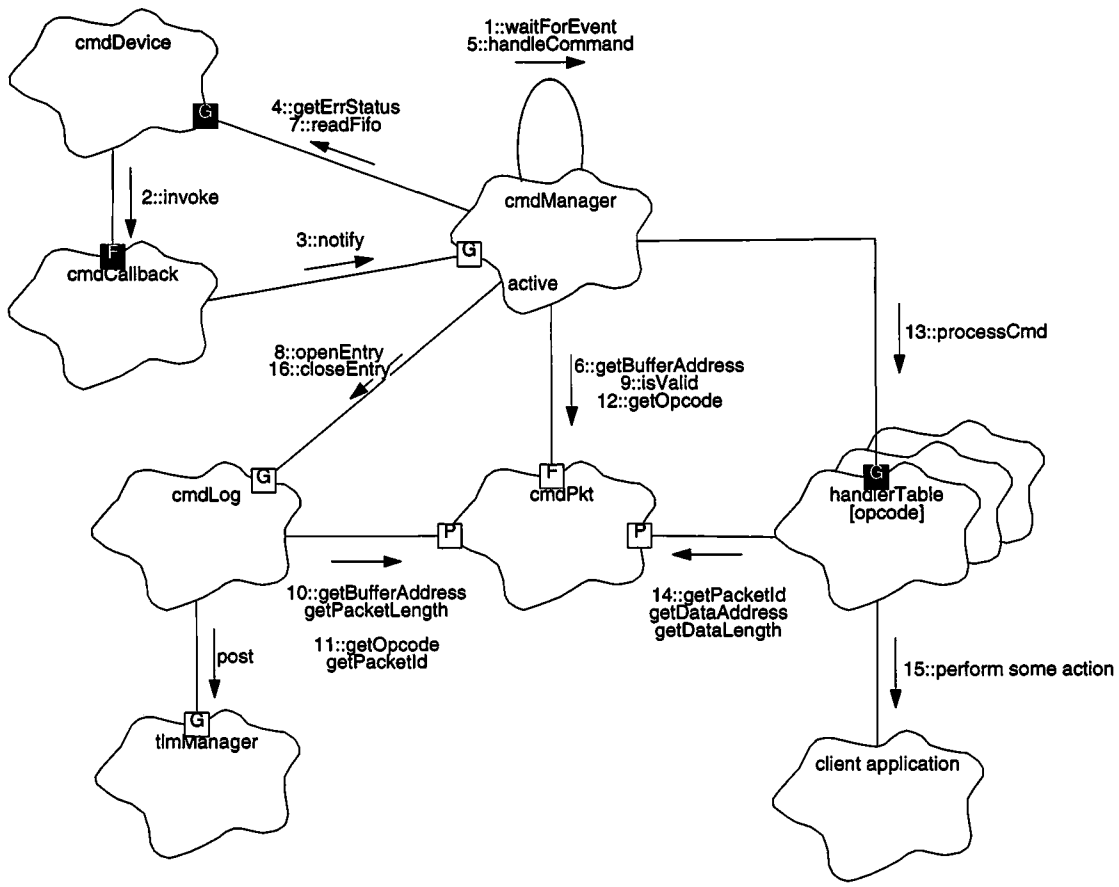
The ACIS instrument software uses a time-based method of grouping sets of related command packets. If an error occurs while receiving a particular command, this delay is used to determine the start of the next unrelated command packet. After an error is detected, all packets which arrive within this time limit from one another will be discarded. This delay time is approximately 1 second.

13.5 Scenarios

13.5.1 Use 1::Read, log and execute commands

Figure 40 illustrates the overall command processing scenario.

FIGURE 40. Command Processing Scenario



1. The *cmdManager* starts up and proceeds to wait for an event indicating activity from the *cmdDevice* (inherited **Task**::waitForEvent()).
2. Meanwhile, a command packet is acquired by the hardware and generates a command interrupt. The *cmdDevice* handles the interrupt and, as part of its interrupt processing, invokes the installed *cmdCallback*'s invoke() member function.
3. *cmdCallback* then invokes *cmdManager*.notify() to wake up the *cmdManager* task. The *cmdManager*'s is then woken up (appearing as a return from waitForEvent()).
4. The *cmdManager* then tests for any errors on the *cmdDevice* using **CmdDevice**::getErrStatus(). If an error is detected, the *cmdManager* invokes its recovery routine, handleError() (not shown). For the purposes of this scenario, assume that the device reported no errors.

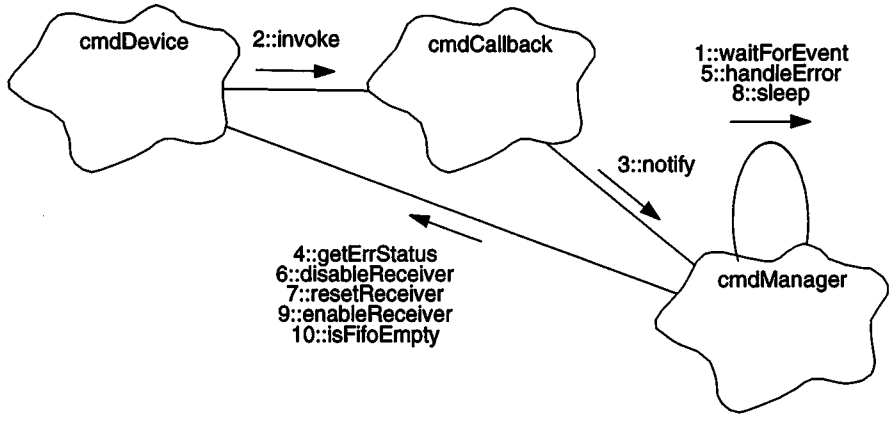
5. If no errors are present, the *cmdManager* invokes its handleCommand() function to read and execute the command.
6. handleCommand() then queries a local **CmdPkt** for its packet buffer address, using **CmdPkt**::getBufferAddress().
7. handleCommand() then reads the packet length and data from the *cmdDevice* into the acquired packet buffer using **CmdDevice**::readFifo().
8. The *cmdManager* then asks the *cmdLog* to open an entry for the packet, using **CmdLog**::openEntry().
9. The *cmdManager* then asks the now filled *cmdPkt* to perform a simple sanity check on its contents, using **CmdPkt**::isValid(). If the packet is invalid, the *cmdManager* drops into its command interface error handling code, handleError() (not shown). Assume for the purposes of this scenario, that the read packet data is valid.
10. If in verbose mode, the *cmdLog* uses **CmdPkt**::getBufferAddress(), and **CmdPkt**::getPacketLength() to obtain the raw packet buffer address and length from the passed command packet. It then proceeds to form and post a command echo packet to the telemetry manager (not shown). NOTE: If we eliminate verbose mode, the sending of the command packet will occur AFTER the command has been executed, via the **CmdLog**::closeEntry() member function.
11. If terse mode is supported, the cmdLog uses **CmdPkt**::getOpcode() and **CmdPkt**::getPacketId() to obtain the passed packet's command code and identifier. It then records the information in its log table (not shown).
12. Once the log entry has been opened, the *cmdManager* queries the packet for its opcode using **CmdPkt**::getOpcode().
13. The *cmdManager* then uses the read opcode as index into a table of pointers to command handlers. It then invokes the handler's processCmd() member function, passing a pointer to the *cmdPkt* packet.
14. The handler then obtains the packet identifier (getPacketId), data address (getDataAddress), and data length (getDataLength) to retrieve any information it needs from the packet.
15. The handler then performs command-specific operations, some of which may invoke member functions of other classes within the system.
16. Once the handler returns, the *cmdManager* passes the returned code to *cmdLog*.closeEntry() to close out the entry for the command and record the disposition of the command.

NOTE: When we eliminate terse mode, **CmdLog**::closeEntry() will be responsible for echoing the command packet, along with the result code provided by the command handler.

13.5.2 Use 2:: Recover from command errors

In order to locate the start of a command after detecting a command device or header format error, the **CmdManager** class uses an approach which relies on a minimum time-delay between decoupled command packets. Figure 41 illustrates this approach.

FIGURE 41. Command Recovery Scenario



1. The *cmdManager* is in its main loop, waiting for something to happen, using the inherited function, **Task**::waitForEvent().
2. An error occurs on the *cmdDevice* causing an interrupt. Its interrupt handler then invokes the installed callback *cmdCallback*.invoke().
3. The callback then wakes up the *cmdManager* task using *cmdManager*.notify(). The *cmdManager* then returns from its waitForEvent() call.
4. The *cmdManager* tests the *cmdDevice* for an error, using *cmdDevice*.getErrStatus(). In this scenario, an error is present.
5. The *cmdManager*, upon detecting an error from the *cmdDevice*, invokes its handleError() function.
6. handleError() then enters a loop, where it disables the *cmdDevice* interrupt generation logic using **CmdDevice**::disableReceiver(). NOTE: Command words received by the instrument while the receiver is disabled are still written into the FIFO, but will not cause a command interrupt).
7. handleError() resets the error condition and clears the FIFO using **CmdDevice**.resetReceiver()
8. The error loop then sleeps (**Task**::sleep()) for a period of time, determined by the **CmdManager** variable, *pktDelay* (not shown).
9. handleError() then re-enables the receiver using **CmdDevice**::enableReceiver().
10. handleError() then tests the command FIFO using *cmdDevice*.isFifoEmtpy(). If the FIFO is not empty, or if the error condition persists (*cmdDevice*.getErrStatus()), handleError() repeats the process from

the point of disabling the receiver. If the FIFO is empty, and the error condition is cleared, `handleError()` returns and the *`cmdManager`* proceeds to wait for the next command. NOTE: All software housekeeping reports in this scenario are TBD.

13.6 Class CmdManager

Documentation:

The Command Manager is responsible for reading commands from the Command Device and for executing the received commands.

Export Control: Public

Cardinality: 1

Hierarchy:

Superclasses: **`Task`**

Implementation Uses:

`CmdDevice`
`CmdPkt`
`CmdLog`

Public Interface:

Operations:
`CmdManager()`
`goTaskEntry()`
`setHandler()`

Constants:

RECOVER_SECONDS=1 second

Protected Interface:

Has-A Relationships:

`CmdHandler` *`handlerTable`*[]: This table is an array of pointers to Command Handler instances. The array is indexed by command opcode.

`const unsigned` *`pktDelay`*: This read-only variable contains the number of clock ticks (1/10 second) to wait after an error to ensure the start of a subsequent packet. This value is set to RECOVER_SECONDS converted into clock ticks (i.e. multiplied by 10).

Operations:
`handleCommand()`
`handleError()`

Concurrency: Active

Persistence: Persistent

13.6.1 CmdManager()

Public member of: **CmdManager**

Arguments:

unsigned *taskId*

Documentation:

This constructor initializes the task information. *taskId* is the ***Nucleus RTX*** task identifier for the **CmdManager**.

Semantics:

Invoke the parent's **Task**::Task() constructor, passing *taskId* as the argument and initialize the read-only *pktDelay* variable to 1 second (10 timer ticks).

Concurrency: Sequential

13.6.2 goTaskEntry()

Public member of: **CmdManager**

Return Class: **void**

Documentation:

This function contains the main loop of the Command Manager task. Its loop must iterate at least once per 250ms in order to never miss a command. This function NEVER returns.

Semantics:

Within a FOREVER loop, wait for an event (using **Task**::waitForEvent()). If a task monitor query is present, respond to the query. If the command device wants attention, test for errors using *cmdDevice*.getErrStatus(). If an error is present, call handleError(), otherwise, call handleCommand() to process any pending commands

Time complexity: When commands are present, must be able to iterate in under 250ms

Concurrency: Synchronous

13.6.3 handleCommand()

Protected member of: **CmdManager**

Return Class: **void**

Documentation:

This function reads and executes a command from the Command Device.

Semantics:

Read the packet length from the command FIFO (*cmdDevice*.readFifo()). If length invalid, call handleError() and return. If length valid, get packet buffer address, store length and read remainder of packet. Test the command header for legal values (*cmdDevice*.isValid()). If header invalid, call handleError() and return. If header ok, open the command log entry (*cmdLog*.openEntry()). Get the packet opcode and index the handler table. If the table entry is 0, log command un-implemented. If not 0, invoke the handler (*handler*->processCmd()), and log (*cmdLog*.closeEntry()) the returned result.

Time complexity: Must execute in under 250ms

Concurrency: Synchronous

13.6.4 handleError()

Protected member of: **CmdManager**

Return Class: **void**

Documentation:

This function attempts to recover from errors in the Command Device.

Semantics:

Do the following until there are no command errors reported and until the FIFO remains empty: disable the receive logic and reset the FIFO, sleep for the packet delay time, re-enable the receive logic.

Concurrency: Synchronous

13.6.5 setHandler()

Public member of: **CmdManager**

Return Class: **void**

Arguments:

CmdOpcode *opcode*
CmdHandler* *handler*

Documentation:

This function writes the *handler* pointer into the *handlerTable* slot indexed by *opcode*, overwriting any previous handler pointer occupying that slot. If *handler* is 0, then commands corresponding to *opcode* are rejected by the *cmdManager*.

Concurrency: Sequential

13.7 Class CmdPkt

Documentation:

Command Packets are used to instruct the ACIS software to perform some action. The top-level command packet acts as the transport layer into the ACIS software.

All Command Packets have the following, overall format:

Word 0: Length
Word 1: Packet Identifier
Word 2: Command Opcode
Words[Length-3]: Command opcode-specific data

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Interface:

Operations:
```
getBufferAddress()
getDataAddress()
getDataLength()
getOpcode()
getPacketId()
getPacketLength()
isValid()
```

Private Interface:

Has-A Relationships:

unsigned short *pktBuffer*[256]: This buffer contains the list of 16-bit words from the command packet.

Concurrency: Guarded

Persistence: Transient

13.7.1 getBufferAddress()

Public member of: **CmdPkt**

Return Class: **unsigned short***

Documentation:

This function returns a pointer to the packet buffer. This buffer is at least 256 16-bit words in length.

Concurrency: Guarded

13.7.2 getDataAddress()

Public member of: **CmdPkt**

Return Class: **const unsigned short***

Documentation:

If the packet contains data, this function returns a read-only pointer to the command packet's data area. The length of this area is determined by calling `getDataLength()`. If the packet's length does not support any data, this function returns 0.

Concurrency: Guarded

13.7.3 getDataLength()

Public member of: **CmdPkt**

Return Class: **unsigned**

Documentation:

This function returns the number of 16-bit words in the packet's data area. The address of this area is obtained using `getDataAddress()`.

Concurrency: Guarded

13.7.4 getOpcode()

Public member of: **CmdPkt**

Return Class: **enum CmdOpcode**

Documentation:

If the packet length is valid, this function returns the opcode contained within the command packet, otherwise it returns CMDOP_INVALID.

Concurrency: Guarded

13.7.5 getPacketId()

Public member of: **CmdPkt**

Return Class: **unsigned**

Documentation:

If the packet's length is valid, this function returns the packet identifier contained within the command packet, otherwise it returns 0xffffffff.

Concurrency: Guarded

13.7.6 getPacketLength()

Public member of: **CmdPkt**

Return Class: **unsigned**

Documentation:

This function returns the total number of 16-bit words delivered by the command packet. The address of the raw packet buffer is provided by `getBufferAddress()`.

Concurrency: Guarded

13.7.7 isValid()

Public member of: **`CmdPkt`**

Return Class: **`Boolean`**

Documentation:

This function performs a top level sanity check on its contents. It returns:
`BoolTrue` - Sanity check passed
`BoolFalse` - Sanity check failed

Concurrency: Guarded

13.8 Class CmdHandler

Documentation:

A command handler is responsible for handling a given command from the ground. This class is an abstract class, and is intended to serve as a top level command handler template. All command handlers must be a subclass of this type.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Uses:

CmdPkt
CmdLog

Public Interface:

Operations: `processCmd()`

Protected Interface:

Operations: `getPktInfo()`

Concurrency: Synchronous

Persistence: Persistent

13.8.1 getPktInfo()

Protected member of: **CmdHandler**

Return Class: **enum CmdResult**

Arguments:

CmdPkt* *cmdpkt*
unsigned& *cmdid*
unsigned& *datacnt*
unsigned short** *dataaddr*
unsigned *mincnt*

Documentation:

This function obtains and checks basic information from the passed command packet. *cmdpkt* points to the packet in question, *cmdid* is the command packet identifier field, *datacnt* is a reference to a packet data word count, *dataaddr* is a reference to a packet data pointer, and *mincnt* specifies the minimum number of packet data words. If the packet data word count is greater than or equal to *mincnt*, less than or equal to (256-3), and the packet data address is valid, this function will return CMDRESULT_OK. If the word count is too low or too high, it will return CMDRESULT_INVALID_DATACNT. If the data address is invalid (should never happen), it will return CMDRESULT_INVALID_DATAPTR.

Concurrency: Synchronous

13.8.2 processCmd()

Public member of: **CmdHandler**

Return Class: **enum CmdResult**

Arguments:

CmdPkt* *pkt*

Documentation:

This is an abstract virtual function. Subclasses of **CmdHandler** must implement this function to execute the command indicated by the referenced command packet, and must return the appropriate **CmdResult** code indicating the disposition of the command.

Time complexity: All `processCmd()` implementations must execute in under 250ms

Concurrency: Synchronous