CSR

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

REVISION LOG

TITLE: Software Detailed Design
Parameter Block Management

DOC. NO. 36-53229

Revision	Date (mm/dd/yy)	ECO No.	Page(s) Affected	Reason	Approval
01	6/14/95	36-294	all	Initial version for design walkthrough	

17.0 Parameter Block Management (36-53229 01)

17.1 Purpose

The purpose of the Parameter Block Management classes are to maintain sets of science and DEA housekeeping parameter blocks.

This section describes two main classes, the **`PblockList`** class, the **`Pblock`** class, and lists their respective subclasses. Detailed descriptions of each of the subclasses are provided in Appendix TBD.

17.2 Uses

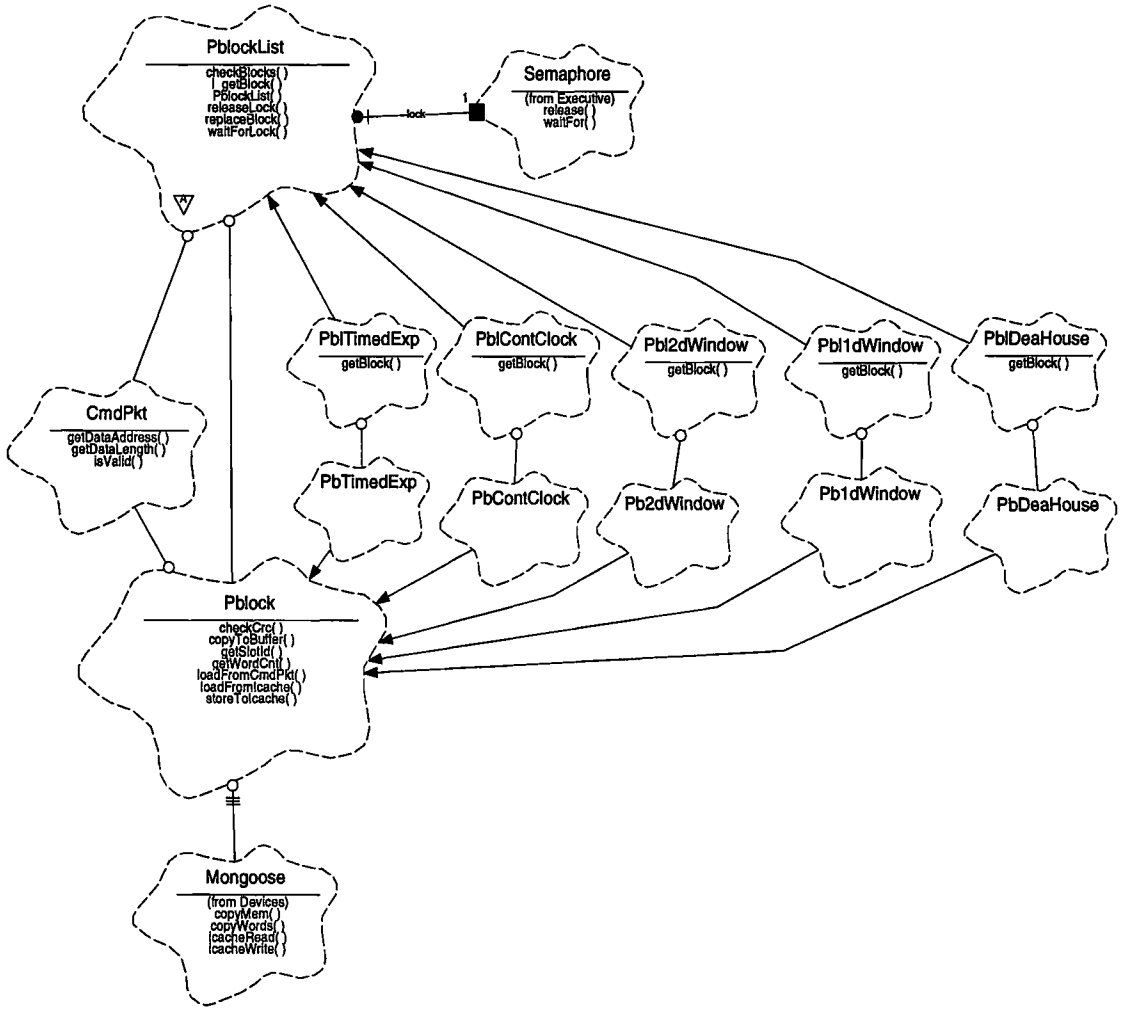
The following lists the primary uses of the Parameter Block Management classes:

Use 1:: Store parameter blocks for use by science and DEA housekeeping
Use 2:: Perform integrity checks on parameter blocks
Use 3:: Supply parameter blocks when needed for a science or DEA housekeeping run

17.3 Organization

Figure 69 illustrates the class relationships used by the Parameter Block Management classes.

FIGURE 69. Parameter Block Management Classes



Pblock- This abstract class is responsible for interpreting the contents of a single parameter block, and for moving parameter data into and out of storage in Instruction Cache RAM. It provides functions to load the contents of its internal buffer from the contents of a command packet (`loadFromCmdPkt`), to copy its contents into a client supplied buffer (`copyToBuffer`), to store and retrieve the contents of its internal buffer to and from a region of instruction cache memory (`storeToIcache` and `loadFromIcache`), to verify the integrity of the block using its CRC value (`checkCrc`), and to retrieve the slot id and number of words contained within the block (`getBlockId` and `getWordCnt`). This class uses the services provided by the ***Devices***::**Mongoose** device class to copy data, and to store and load data from instruction cache RAM.

PblockList- This abstract class is responsible for maintaining a fixed size collection of homogenous parameter blocks (**Pblock** or one of its subclasses). This class provides functions to replace the contents of one its parameter blocks with the contents of a command packet (`replaceBlock`), to check the integrity of all of its blocks (`checkBlocks`) and to retrieve the contents of one of its blocks (`getBlock`). It also

provides functions to obtain and release a semaphore associated with the collection (`waitForLock` and `releaseLock`). This class uses the services provided by the **`Pblock`** class to manage the contents of a single parameter block, and contains a ***`Executive`***`::`**`Semaphore`** instance associated with its collection of parameter blocks.

The following lists the various subclasses of **`Pblock`**. Each of these classes are responsible for providing functions with return the values of the various fields within the parameter block. The functions belonging to each of these classes are produced using a code-generator directly from the Instrument Program and Command List specification. The functions belonging to each of these classes are described in Appendix TBD:

`PbTimedExp` - Timed Exposure Parameter Block

`PbContClock` - Continuous Clocking Parameter Block

`Pb2dWindow` - 2D Window List Parameter Block

`Pb1dWindow` - 1D Window List Parameter Block

`PbDeaHouse` - DEA Housekeeping Parameter Block

The following lists the various subclasses of **`PblockList`**. There is one global instance of each of these classes within the instrument. Each of these classes is responsible for managing the collection of parameter blocks of the indicated type, and, in addition to the member functions provided by **`PblockList`**, overload the `getBlock()` function with a public member function. Rather than using the generic **`Pblock`** type as an argument, the overloaded functions require the correct parameter block type for their input arguments.

`Pbl``TimedExp` -Manage all Timed Exposure Parameter Blocks (**`PbTimedExp`**)

`Pbl``ContClock` - Manage all Continuous Clocking Parameter Blocks (**`PbContClock`**)

`Pbl``2dWindow` -Manage all 2D Window List Parameter Blocks (**`Pb2dWindow`**)

`Pbl``1dWindow`- Manage all 1D Window List Parameter Blocks (**`Pb1dWindow`**)

`Pbl``DeaHouse` - Manage all DEA Housekeeping Parameter Blocks (**`PbDeaHouse`**)

17.4 Parameter Block Storage

In order to reduce the opportunity that writes through a corrupted data pointer will corrupt parameter blocks intended for use for longer periods of time, parameter blocks within ACIS are maintained in Instruction Cache RAM. The **`Pblock`** class and its various subclasses are used as temporary objects which move data into and out of I-cache slots, and provide functions which retrieve specific fields of a given type of parameter block. Each parameter block slot in I-cache is sized to hold the maximum size allowed for a parameter block (i.e. one command packet). The number of slots reserved for each type of block is TBD. Upon power-on, this area of I-cache is loaded with default parameter blocks from the main instrument ROM. Subsequent resets do not modify the slot contents. Assuming, for now, that the instrument supports 4 blocks of each type, Table 17 illustrates the overall layout of parameter block slots within I-cache RAM, where the addresses and sizes are gross approximations. The first two slots of each type of block contain the default parameter blocks for Imaging and Spectroscopy observations.

TABLE 17. I-cache Parameter Block Layout (TBD)

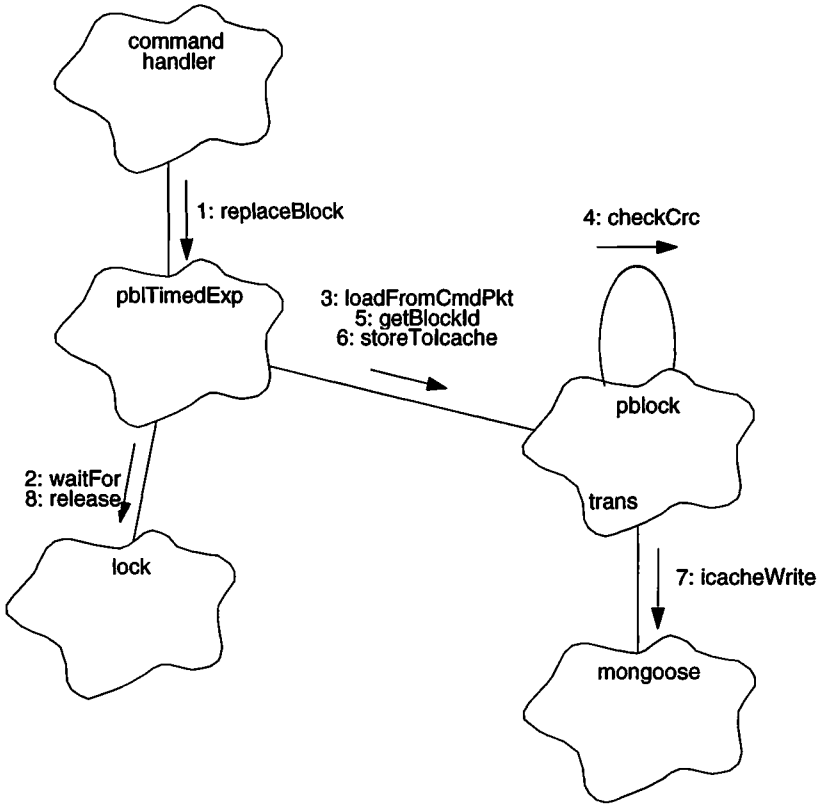
Region	Address	Byte Size	Description	Slot #
Patch Area (grow down from 0x800fffff)	0x800fffff 0x800cfc00	0x30400		
Bad Pixel and Column Maps	0x800c2c00	0xd000		
DEA Housekeeping Blocks	0x800c2a00	0x200	General Purpose	3
	0x800c2800	0x200	General Purpose	2
	0x800c2600	0x200	General Purpose	1
	0x800c2400	0x200	Default	0
1D Window Blocks	0x800c2200	0x200	General Purpose	3
	0x800c2000	0x200	General Purpose	2
	0x800c1e00	0x200	Default Spectroscopy	1
	0x800c1c00	0x200	Default Imaging	0
2D Window Blocks	0x800c1a00	0x200	General Purpose	3
	0x800c1800	0x200	General Purpose	2
	0x800c1600	0x200	Default Spectroscopy	1
	0x800c1400	0x200	Default Imaging	0
Continuous Clocking Blocks	0x800c1200	0x200	General Purpose	3
	0x800c1000	0x200	General Purpose	2
	0x800c0e00	0x200	Default Spectroscopy	1
	0x800c0c00	0x200	Default Imaging	0
Timed Exposure Blocks	0x800c0a00	0x200	General Purpose	3
	0x800c0800	0x200	General Purpose	2
	0x800c0600	0x200	Default Spectroscopy	1
	0x800c0400	0x200	Default Imaging	0
System Configuration	0x800c0000	0x400		
Code	0x80080000	0x40000		

17.5 Scenarios

17.5.1 Use 1: Store parameter blocks

Figure 70 illustrates the steps used to store Timed Exposure Parameter blocks. The steps used to store Continuous Clocking, 2D Window, 1D Window and DEA Housekeeping parameter blocks are identical except for the identities of the parameter block list objects.

FIGURE 70. Load Parameter Block Scenario



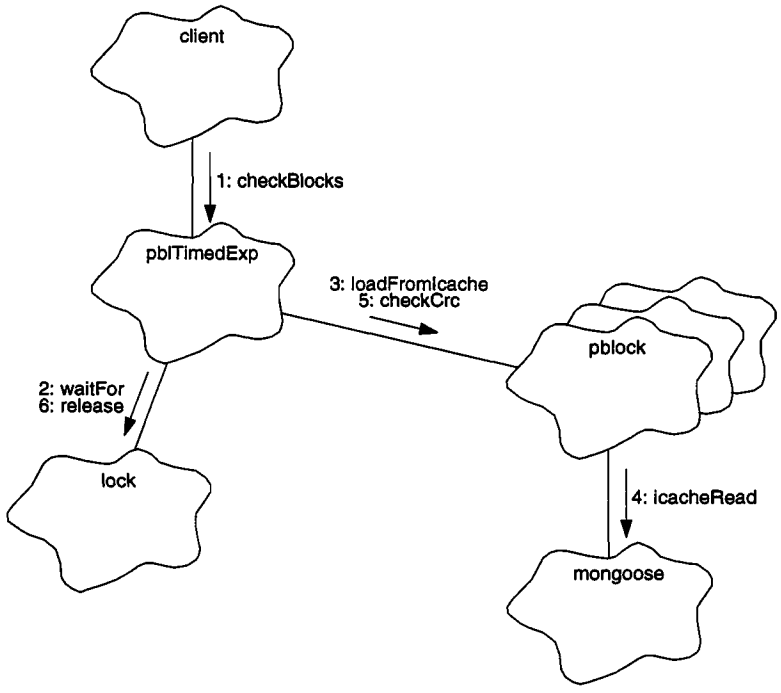
1. The client *command handler* receives a command to overwrite a parameter block, and instructs the Timed Exposure Parameter Block List object to store the block using *pblTimedExp*.`replaceBlock()`.
2. `replaceBlock()` suspends the current task until it has obtained its semaphore, indicating that it has exclusive access to the parameter block list using *lock*.`waitFor()`. If the wait time-out expires, the attempt to replace the block is aborted. For the purposes of this scenario, assume that the *lock* is obtained successfully.
3. `replaceBlock()` declares an instance of a Timed Exposure Parameter Block, *pblock*, and instructs it to load its contents from the command packet, passed in from the handler using *pblock*.`loadFromCmdPkt()`. At this point, *pblock* has a copy of the parameter block data.

4. *pblock*.loadFromCmdPkt() copies the parameter data from the command buffer, and then checks the integrity of the copied data by calling checkCrc(). If the data has been corrupted, loadFromCmdPkt() returns an error. Assume for the purposes of this scenario that the parameters are intact.
5. Once *pblock* has copied the parameters, *pblTimedExp*.replaceBlock() obtains the slot identifier from the block using *pblock*.getBlockId().
6. replaceBlock() uses the returned slot id to select the region in Instruction Cache RAM to store the block, and copies the block data into I-cache using *pblock*.storeToIcache().
7. *pblock*.storeToIcache() uses *mongoose*.icacheWrite() to copy its parameter block data to the selected region.
8. Once the parameter block has been stored into the appropriate slot in I-cache, replaceBlock() releases its semaphore, using *lock*.release().

17.5.2 Use 2: Perform integrity checks

Figure 71 illustrates the steps used to check the integrity of the collection of Timed Exposure Parameter blocks. The steps used to check other types of blocks are identical, except for the identities of the parameter block list objects.

FIGURE 71. Check integrity of stored Parameter Blocks

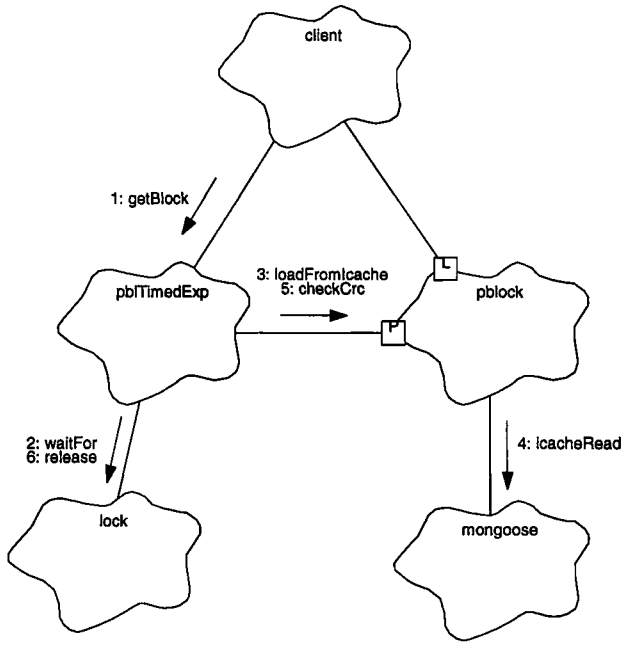


1. The *client* instructs the Timed Exposure Parameter Block List object to verify the integrity of each of its blocks using *pblTimedExp*.checkBlocks().
2. checkBlocks() prevents other tasks from overwriting the contents of the blocks by obtaining its semaphore using *lock*.waitFor(). If the wait time-out expires, the attempt to check the block is aborted. Assume that the *lock* is obtained successfully.
3. checkBlocks() iterates through each parameter slot that it maintains. On each iteration, it declares a Timed Exposure Parameter block instance, *pblock*, and instructs it to load its contents from the parameter slot maintained in I-cache RAM, using *pblock*.loadFromIcache().
4. *pblock*.loadFromIcache() uses *mongoose*.icacheRead() to copy the parameter block data from I-cache RAM into its private buffer.
5. *pblTimedExp*.checkBlocks() instructs the block to compute its CRC, compare the value with the one contained within the parameter block data, and return the result of the comparison, using *pblock*.checkCrc().
6. *pblTimedExp*.checkBlocks() repeats these steps for each of its parameter block slots. Once all of the blocks have been checked, checkBlocks() releases the lock, using *lock*.release(). It then returns whether or not any of its parameter blocks have been corrupted.

17.5.3 Use 3: Supply parameter blocks when needed

Figure 72 illustrates the steps used to retrieve the contents of a Timed Exposure Parameter block. The steps used to obtain Continuous Clocking, 2D Window, 1D Window and DEA Housekeeping parameter blocks are identical except for the identities of the parameter block list objects.

FIGURE 72. Retrieve contents of Parameter Block



1. The `client` declares an instance of a Timed Exposure Parameter Block object, `pblock`, and passes the block slot id and the instance to the Timed Exposure Parameter Block list to be loaded to *`pblTimedExp`*`.getBlock()`.
2. *`pblTimedExp`*`.getBlock()` prevents overwrites of the list while performing the retrieval using *`lock`*`.waitFor()`. If the wait's time-out expires, the fetch is aborted. Assume for the purposes of this scenario that the lock is obtained.
3. `getBlock()` instructs the block to load its contents from I-cache RAM, using *`pblock`*`.loadFromIcache()`.
4. `loadFromIcache()` uses *`mongoose`*`.icacheRead()` to copy the parameter data into its local buffer.
5. `getBlock()` instructs the block to check its CRC, using *`pblock`*`.checkCrc()`.
6. `getBlock()` releases the semaphore, using *`lock`*`.release()`. It then returns to the client, informing it whether or not the loaded block's CRC was valid or not.

17.6 Class PblockList

Documentation:

This class represents a collection of parameter blocks. It defines the common functions defined for all types of parameter blocks.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Uses:

Pblock

Public Interface:

Operations:
```
PblockList()
checkBlocks()
releaseLock()
replaceBlock()
waitForLock()
```

Protected Interface:

Operations:
```
getBlock()
```

Private Interface:

Has-A Relationships:

Semaphore *lock*: This is used lock access to the parameter blocks.

unsigned* const *slotBase*: This points to the starting location within I-cache RAM of the array of parameter block slots.

const unsigned *slotCnt*: This is the number of parameter blocks.

const unsigned *slotWordCnt*: This specifies the number of 32-bit words in each parameter block slot maintained by this instance.

Concurrency: Guarded

Persistence: Persistent

17.6.1 PblockList()

Public member of: **PblockList**

Arguments:

unsigned *semid*
unsigned* *base*
unsigned *nSlots*
unsigned *wordCnt*

Documentation:

This is the constructor for a parameter block list. *semid* is the RTX id for the semaphore the instance uses to synchronize access to the parameters, *base* is the starting address within I-cache used to maintain the list of blocks, *nSlots* is the number of parameter block slots maintained by the instance, and *wordCnt* is the number of 32-bit words in each parameter block slot.

Semantics:

Construct *lock* using *semid*. Copy *base to* slotBase, *nSlots* to *slotCnt* and *wordCnt* to *slotWordCnt*.

Concurrency: Sequential

17.6.2 checkBlocks()

Public member of: **PblockList**

Return Class: **Boolean**

Documentation:

This function checks the CRC of all parameter blocks in the collection. It returns *BoolTrue* if all are ok, and *BoolFalse* if there is an error in one or more of the blocks.

Semantics:

Call waitForLock() to prevent overwrites. If it times out, return *BoolFalse*. For each slot, declare *pblock*, instruct it to load from i-cache address *slotBase* + (slot counter * *slotWordCnt*), using *pblock*.loadFromIcache(), then call *pblock*.checkCrc() and set a flag it if fails. Once all slots have been checked, release the lock using releaseLock(). If any CRC fails, return *BoolFalse*. If all of the CRC checks pass, return *BoolTrue*.

Concurrency: Synchronous

17.6.3 getBlock()

Protected member of: **PblockList**

Return Class: **Boolean**

Arguments:

unsigned *slotid*
Pblock& *pblock*

Documentation:

This function fills *pblock* with the parameter block data stored in the location identified by *slotid*. If successful, this function returns *BoolTrue*. If *slotid* is invalid, the wait for the lock times-out, or if the CRC of the requested block is invalid, this function returns *BoolFalse*.

Semantics:

Call waitForLock() to prevent overwrites. If it times out, return *BoolFalse*. Instruct *pblock* to load its contents from I-cache address *slotBase* + (*slotid* * *slotWordCnt*) using *pblock*.loadFromIcache(), then call *pblock*.checkCrc(). Release the lock using releaseLock(). If the CRC failed, return *BoolFalse*, otherwise return *BoolTrue*.

Concurrency: Synchronous

17.6.4 releaseLock()

Public member of: **PblockList**

Return Class: **void**

Documentation:

This function releases the parameter block list, by calling *lock*.release(), allowing other tasks to use this instance.

Concurrency: Synchronous

17.6.5 replaceBlock()

Public member of: **PblockList**

Return Class: **Boolean**

Arguments:

const CmdPkt* *cmdpkt*

Documentation:

This function uses the *cmdpkt* to replace the contents of the block specified by the block id in the command. This function returns *BoolTrue* if the replacement succeeds, and *BoolFalse* if it fails.

Semantics:

Call waitForLock() to prevent overwrites. If it times out, return *BoolFalse*. Declare a parameter block, *pblock*, and instruct it to load its contents from the command packet, using *pblock*.loadFromCmdPkt(). Check its CRC using *pblock*.checkCrc(). If the CRC is valid, retrieve the parameter block's slot id using *pblock*.getSlotId(), and store its contents into I-cache using *pblock*.storeToIcache(). Release the semaphore using releaseLock(). If the loaded parameter block's CRC check failed return *BoolFalse* without storing the block. If the block is intact and stored, the function returns *BoolTrue*.

Concurrency: Synchronous

17.6.6 waitForLock()

Public member of: **PblockList**

Return Class: **Boolean**

Arguments:

unsigned *timeout*

Documentation:

This function waits for the parameter block list to become available, and locks it by calling *lock*.waitFor(). If successful, this function returns *BoolTrue*. If *timeout* is reached first, it returns *BoolFalse*.

Concurrency: Synchronous

17.7 Class Pblock

Documentation:

This class is used to define a common top-level interface for all parameter blocks loaded into the instrument. This class provides mechanisms to load parameter block data from command packets, to store and retrieve parameter data from I-cache. Subclasses of this class provide functions which retrieve block-specific fields.

Export Control: Public

Cardinality: n

Hierarchy:

Superclasses: **none**

Public Uses:

CmdPkt

Implementation Uses:

Mongoose

Public Interface:

Operations:

```
checkCrc()
copyToBuffer()
getSlotId()
getWordCnt()
loadFromCmdPkt()
loadFromIcache()
storeToIcache()
```

Protected Interface:

Has-A Relationships:

unsigned *databuf*[128]: This is the data buffer which contains the parameter block.

Concurrency: Guarded

Persistence: Transient

17.7.1 checkCrc()

Public member of: **Pblock**

Return Class: **Boolean**

Documentation:

This function computes the CRC of the current block and compares it with the CRC contained within the block. If the CRC is invalid, the function returns *BoolFalse*, else it returns *BoolTrue*.

Preconditions:

The block contents must have been loaded either via `loadFromCmdPkt()` or `loadFromIcache()`.

Concurrency: Guarded

17.7.2 copyToBuffer()

Public member of: **Pblock**

Return Class: **void**

Arguments:

unsigned* *dstptr*

Documentation:

This function copies the body of the parameter block to the buffer pointed to by *dstptr*. The caller must ensure that the destination buffer is large enough to hold the contents of the parameter block.

Preconditions:

The block contents must have been loaded either via `loadFromCmdPkt()` or `loadFromIcache()`.

Concurrency: Guarded

17.7.3 getSlotId()

Public member of: **Pblock**

Return Class: **unsigned**

Documentation:

This function returns the block id of the parameter block.

Preconditions:

The block contents must have been loaded either via `loadFromCmdPkt()` or `loadFromIcache()`.

Concurrency: Guarded

17.7.4 getWordCnt()

Public member of: **Pblock**

Return Class: **unsigned**

Documentation:

This function returns the length of the parameter block, in 32-bit words.

Preconditions:

The block contents must have been loaded either via `loadFromCmdPkt()` or `loadFromIcache()`.

Concurrency: Guarded

17.7.5 loadFromCmdPkt()

Public member of: **Pblock**

Return Class: **void**

Arguments:

const CmdPkt* *cmdpkt*

Documentation:

This function loads the parameter block contents from the command packet pointed to by *cmdpkt*.

Semantics:

Get the packet's data buffer address and length using *cmdpkt*->getDataAddress() and *cmdpkt*->getDataLength() respectively. Copy the data from the packet into the local data buffer using *mongoose*.copyMem().

Concurrency: Guarded

17.7.6 loadFromIcache()

Public member of: **Pblock**

Return Class: **void**

Arguments:

const unsigned* *srcptr*

Documentation:

This function loads the parameter block from data store in I-cache at the address, *srcptr* into its data buffer, using *mongoose*.icacheRead().

Concurrency: Guarded

17.7.7 storeToIcache()

Public member of: **Pblock**

Return Class: **void**

Arguments:

unsigned* *dstptr*

Documentation:

This function stores the parameter block information at the I-cache address *dstptr*, using *mongoose*.icacheWrite().

Preconditions:

The block contents must have been loaded either via loadFromCmdPkt() or loadFromIcache().

Concurrency: Guarded

```
//   /acis/h3/acisfs/configcntl/models/filesprotocols/pblocklist.H,v
//   pblocklist.H,v 1.1 1995/06/15 17:17:47 jimf Exp
//   pblocklist.H,v
 * Revision 1.1  1995/06/15  17:17:47  jimf
 * Initial versions for design walkthrough. pblock.C needs to be fixed
 * to bring in files generated directly from IPCL.
 *

//
//       Copyright Massachusetts of Technology, 1995
//
// Module Specification PblockList (Package)
//
// Subsystem: filesProtocols
// File: filesprotocols/pblocklist.H
//
#ifndef pblocklist_h
#define pblocklist_h 1
// Additional Includes:
#include "acis.h"


// Semaphore
#include "filesexecutive/semaphore.H"
// Pblock
#include "filesprotocols/pblock.H"


// Class PblockList
//
// Documentation:
//        This class represents a collection of parameter
//        blocks. It defines the common functions defined for
//        all types of parameter blocks.
// Concurrency:   Guarded
// Persistence:   Persistent
// Cardinality: n
//
class PblockList
{
  public:
    PblockList(unsigned semid, unsigned*  base, unsigned nSlots, unsigned wordCnt);
    virtual Boolean checkBlocks();
    virtual Boolean replaceBlock(const CmdPkt*cmdpkt);
    virtual Boolean waitForLock(unsigned timeout);
    virtual void releaseLock();


  protected:
    virtual Boolean getBlock(unsigned slotid, Pblock&  pblock);


  private:


  private:  // implementation

    Semaphore lock;


    unsigned* const slotBase;


    const unsigned slotCnt;


    const unsigned slotWordCnt;


    const unsigned lockTimeout;

};
#endif
```

```
//   /acis/h3/acisfs/configcntl/models/filesprotocols/pblocklist.C,v
//   pblocklist.C,v 1.1 1995/06/15 17:17:52 jimf Exp
//   pblocklist.C,v
 * Revision 1.1  1995/06/15  17:17:52  jimf
 * Initial versions for design walkthrough. pblock.C needs to be fixed
 * to bring in files generated directly from IPCL.
 *

//
//       Copyright Massachusetts Institute of Technology, 1995
//
// Module Body PblockList (Package)
//
// Subsystem: filesProtocols
// File: filesprotocols/pblocklist.C
//


// PblockList
#include "filesprotocols/pblocklist.H"


// Class PblockList
// Constructors
// ----------------------------------------------------
PblockList::PblockList(unsigned semid, unsigned*  base, unsigned nSlots, unsigned wordCn
t)
// ----------------------------------------------------



    :
    lock(semid),
    slotBase(base),
    slotCnt(nSlots),
    slotWordCnt(wordCnt),
    lockTimeout(2)

{
    DebugProbe probe;

#include "filesdevices/mongoose.H"
    assert (mongoose.isIcache(base));
    assert (mongoose.isIcache(base+(nSlots*wordCnt)));
    assert (wordCnt >= 128);

}
// Other Operations
// ----------------------------------------------------
Boolean PblockList::checkBlocks()
// ----------------------------------------------------
{
    DebugProbe probe;

    // ---- Prevent overwrites ----
    if (waitForLock(lockTimeout) == BoolFalse)
    {
        return BoolFalse;
    }
    Boolean retval = BoolTrue;  // Return value, start with all ok
    const unsigned* addr = slotBase;
    for (unsigned blockid = 0;
         blockid < slotCnt;
         blockid++, addr += slotWordCnt)
    {
        // --- Get block and check it's CRC ---
        Pblock pblock;
        pblock.loadFromIcache (addr);
        if (pblock.checkCrc() != BoolTrue)
        {
            retval = BoolFalse;
        }
    }
    // ---- Return if any of the blocks were corrupted ----
    releaseLock();
    return retval;

}
// ----------------------------------------------------
Boolean PblockList::replaceBlock(const CmdPkt*cmdpkt)
// ----------------------------------------------------
{
    DebugProbe probe;

    assert (cmdpkt != 0);
    // ---- Prevent overwrites ----
    if (waitForLock(lockTimeout) == BoolFalse)
    {
        return BoolFalse;
    }
    // ---- Check block id ----
    Boolean retval = BoolFalse; // Assume invalid block id
    // ---- Get block ptr and tell it to load from cmdPkt ----
    Pblock pblock;
    pblock.loadFromCmdPkt (cmdpkt);
    if (pblock.checkCrc () == BoolTrue)
    {
        unsigned slot = pblock.getSlotId();
        pblock.storeToIcache (slotBase + (slot * slotWordCnt));
    }
    // ---- Return result of block id check ----
    releaseLock ();
    return retval;

}
// ----------------------------------------------------
Boolean PblockList::getBlock(unsigned slotid, Pblock&  pblock)
// ----------------------------------------------------
{
    DebugProbe probe;

    // ---- Prevent overwrites ----
    if (waitForLock(lockTimeout) == BoolFalse)
    {
        return BoolFalse;
    }
    Boolean retval = BoolFalse;
    if (slotid < slotCnt)
    {
        pblock.loadFromIcache (slotBase + (slotid * slotWordCnt));
        if (pblock.checkCrc () == BoolTrue)
        {
            retval = BoolTrue;
        }
    }
    // ---- Return True if all ok, else return False ----
    releaseLock ();
```

```
    return retval;

}
// ---------------------------------------------------
Boolean PblockList::waitForLock(unsigned timeout)
// ---------------------------------------------------
{
    DebugProbe probe;

    Boolean retval = lock.waitFor (timeout);
    return retval;

}
// ---------------------------------------------------
void PblockList::releaseLock()
// ---------------------------------------------------
{
    DebugProbe probe;

    lock.release ();

}
// Get and Set Operations for Has Relationships
// Additional Declarations
```

```
//   /acis/h3/acisfs/configcntl/models/filesprotocols/pblock.H,v
//   pblock.H,v 1.1 1995/06/15 17:17:56 jimf Exp
//   pblock.H,v
 * Revision 1.1  1995/06/15  17:17:56  jimf
 * Initial versions for design walkthrough. pblock.C needs to be fixed
 * to bring in files generated directly from IPCL.
 *

//
//       Copyright Massachusetts of Technology, 1995
//
// Module Specification Pblock (Package)
//
// Subsystem: filesProtocols
// File: filesprotocols/pblock.H
//
#ifndef pblock_h
#define pblock_h 1
// Additional Includes:
#include "acis.h"


// Mongoose
#include "filesdevices/mongoose.H"
// CmdPkt
#include "filesprotocols/cmdpkt.H"


// Class Pblock
//
// Documentation:
//        This class is used to define a common top-level
//        interface for all parameter blocks loaded into the
//        instrument.
// Concurrency:   Sequential
// Persistence:   Transient
// Cardinality: n
//
class Pblock
{
  public:
    unsigned getWordCnt() const;
    unsigned getSlotId() const;
    Boolean checkCrc() const;
    void copyToBuffer(unsigned*dstptr) const;
    virtual void loadFromCmdPkt(const CmdPkt*cmdpkt);
    virtual void storeToIcache(unsigned*dstptr);
    virtual void loadFromIcache(const unsigned*srcptr);


  protected:


  private:


  private:  // implementation

    unsigned  databuf[128];



};
#endif
```

```
//   /acis/h3/acisfs/configcntl/models/filesprotocols/pblock.C,v
//   pblock.C,v 1.1 1995/06/15 17:18:01 jimf Exp
//   pblock.C,v
 * Revision 1.1  1995/06/15  17:18:01  jimf
 * Initial versions for design walkthrough. pblock.C needs to be fixed
 * to bring in files generated directly from IPCL.
 *

//
//       Copyright Massachusetts Institute of Technology, 1995
//
// Module Body Pblock (Package)
//
// Subsystem: filesProtocols
// File: filesprotocols/pblock.C
//


// Pblock
#include "filesprotocols/pblock.H"

// TBD:: Move this to a format file emitted by an enhanced code-generator
// ---- IPCL2C++ - IPCL to C++ Converter ----
// EMITC++ - C++ Code Generator
class Parameter_Block_Header
{
public:
        unsigned get_Slot_Index (const unsigned short* inputData)  const;
        unsigned get_CRC (const unsigned short* inputData)  const;
};
// -----------------------------------------------------
inline unsigned Parameter_Block_Header::get_Slot_Index (const unsigned short* inputData)
  const
// -----------------------------------------------------
{
        return inputData[0];
}
// -----------------------------------------------------
inline unsigned Parameter_Block_Header::get_CRC (const unsigned short* inputData)  const
// -----------------------------------------------------
{
        return inputData[1];
}

// Class Pblock
// Other Operations
// -----------------------------------------------------
unsigned Pblock::getWordCnt() const
// -----------------------------------------------------
{
    DebugProbe probe;

    return databuf[0];

}
// -----------------------------------------------------
unsigned Pblock::getSlotId() const
// -----------------------------------------------------
{
    DebugProbe probe;

    Parameter_Block_Header parser;
    unsigned slotId = parser.get_Slot_Index((unsigned short*) (databuf+1));
    return slotId;

}
// -----------------------------------------------------
Boolean Pblock::checkCrc() const
// -----------------------------------------------------
{
    DebugProbe probe;

extern unsigned computeCrc (const unsigned* buf);
    unsigned computedCrc = computeCrc (databuf+1);
    Parameter_Block_Header parser;
    unsigned storedCrc = parser.get_CRC((unsigned short*) (databuf+1));
    if (storedCrc != computedCrc)
    {
        return BoolFalse;
    }
    return BoolTrue;

}
// -----------------------------------------------------
void Pblock::copyToBuffer(unsigned*dstptr) const
// -----------------------------------------------------
{
    DebugProbe probe;

    assert (dstptr != 0);
    mongoose.copyWords (databuf+1, dstptr, databuf[0]);

}
// -----------------------------------------------------
void Pblock::loadFromCmdPkt(const CmdPkt*cmdpkt)
// -----------------------------------------------------
{
    DebugProbe probe;

    assert (cmdpkt != 0);
    assert (cmdpkt->isValid());
    // ---- Copy data into buffer ----
    const unsigned short* srcptr = cmdpkt->getDataAddress();
    unsigned len = cmdpkt->getDataLength();
    databuf[0] = len * (sizeof(*srcptr)/sizeof(databuf[0]));
    mongoose.copyMem(srcptr, databuf+1, databuf[0]);

}
// -----------------------------------------------------
void Pblock::storeToIcache(unsigned*dstptr)
// -----------------------------------------------------
{
    DebugProbe probe;

    assert (dstptr != 0);
    mongoose.icacheWrite (databuf, dstptr, databuf[0] + 1);

}
// -----------------------------------------------------
void Pblock::loadFromIcache(const unsigned*srcptr)
// -----------------------------------------------------
{
    DebugProbe probe;

    assert (srcptr != 0);
    mongoose.icacheRead (srcptr, databuf, WORDCNT(databuf));
```

```
}
// Get and Set Operations for Has Relationships
// Additional Declarations
```