REVISIONS

Rev	ECO No.	Description	Checked	Approved	Date
1.0	N/A	Initial Release	PGF	RFG	06/26/96
2.0	N/A	Update for Flight S/W Beta release	PGF	RFG	01/07/97
3.0	N/A	Update for Flight S/W Release 1.0	PGF	RFG	02/07/97
3.1	N/A	Update for Flight S/W Release 1.5	PGF	RFG	06/20/97

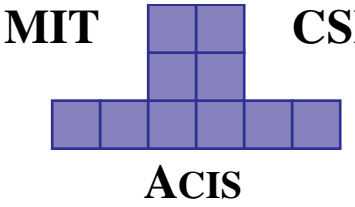
	NAME	DATE
Drawn:	P. Ford	01/13/97
Checked:		
Approved:		
Released:		

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CENTER FOR SPACE RESEARCH

ACIS Software Test Tools

Size	Code Identification No.	Drawing No.	Rev.
T	80230	36-55001	3.1

Scale: NONE



MASSACHUSETTS INSTITUTE OF TECHNOLOGY

CENTER FOR SPACE RESEARCH
CAMBRIDGE, MASSACHUSETTS 02139

ACIS Test Tools

MIT 36-55001 Rev. 3.1

June 20, 1997

CSR	**MASSACHUSETTS INSTITUTE OF TECHNOLOGY** **CENTER FOR SPACE RESEARCH** **CAMBRIDGE, MASSACHUSETTS 02139**	
REVISION LOG	**TITLE:** **ACIS Test Tools**	**DOC. NO.** **36-55001 Rev. 3.1**

Revision	Date mm/dd/yy	ECO No.	Section(s) Affected	Reason	Approval
pre-release 0.6	03/13/96	—	All	Initial Release	—
pre-release 0.7	05/23/96	—	§2.2	Rename pseudo-packets	—
			Tables 3 & 4	Rename fields and add lengths and offsets	
			Table 5	Rename packet classes	
			§3.1	Add description of ***processScience*** & Table 6	
			Table 7	Add `dump`, `reset`, and `stop` commands	
			§3.3	Update ***buildCmds*** examples to reflect IP&CL changes	
			§4.3	Reword parts of ***genPixelImages*** description	
			§7.6	Update frame-buffer function description	
1.0	06/26/96	—	Cover	Update cover sheet to show part number and revision level	RFG 06/29/96
2.0	10/03/96	—	Cover	Added revision log and TBD list	RFG
			§2.1	Added to the description of ***filterServer***, ***filterClient***, and ***shim***	
			§2.6	Added description of hardware commands	
			§3.1	Reorganized ***buildCmds*** description; reworded Tables 7 and 8. Added Table 9.	
			§3.2	Added ***lcmd*** description	
			§3.3	Moved ***ltlm*** description from §5.	
			§3.4	Reworded ***processScience*** description and updated Table 10	
			§3.6	Added ***runacis*** description	
			§5	Removed ***ltlm*** and ***tlmsim*** descriptions	
			§6	Rewritten from ECO 567	
			§9	Rewritten from the contents of *~acis/tools*	
			Appendix A	Updated	

Revision	Date mm/dd/yy	ECO No.	Section(s) Affected	Reason	Approval
3.0	11/22/96	—	Table 5	Changed engineering pseudopacket format	RFG
			Table 6	Added to describe irig-b format	
			§3.4	Replaced *processScience* with *psci*	
			§5	New section describing *psci*	
			§10	Added numerous new manual entries.	
3.1	06/20/97	—	§5	Describe the -B, -T, and -s options.	

Items to be Determined

Table of Contents

List of Figures

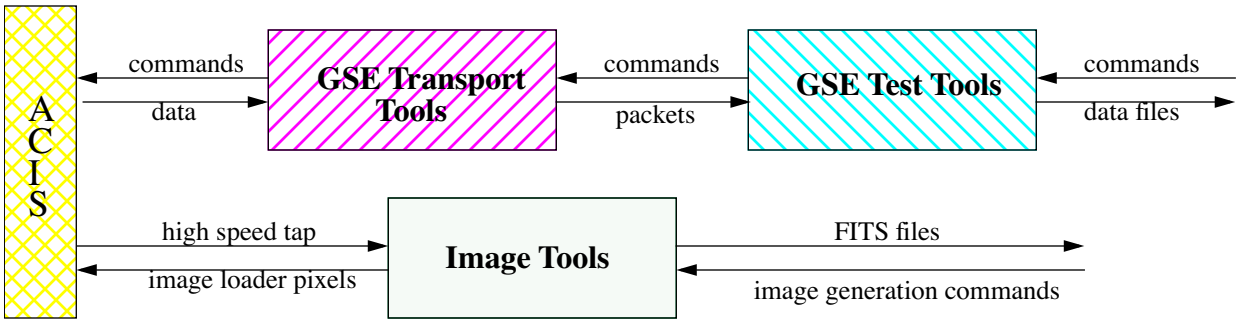
List of Tables

1.0 Introduction

This is a description of the software tools being developed to test ACIS flight software, to support ACIS EGSE, and to perform end-to-end tests of the instrument. It is a joint effort of the ACIS flight software and GSE teams, and includes a detailed description of the command and image definition languages and bit-level descriptions of the several interfaces between sub-components.

The tools are divided into three categories: “GSE Transport Tools” are responsible for sending commands to the instrument and receiving telemetry in reply; “GSE Test Tools” generate the commands from human-readable scripts, interpret the telemetry packets, and analyze their contents; and “Image Tools” are responsible both for reading images from the analog units (DEAs) and for creating and writing images to the digital units (FEPs). The general relationship between these three tool groups is represented in Figure 1; the details are in Figure 2.

FIGURE 1. Test Tool Overview



The interface between the Image Tools and ACIS, the “Image Loader”, is described in Section 8.0 on page 53, and science data formats of use in analyzing the “GSE Test Tools” output are described in Section 9.0. Some alternative image tools are described in Section 10.0. The current status of tool development is shown in Appendix A.

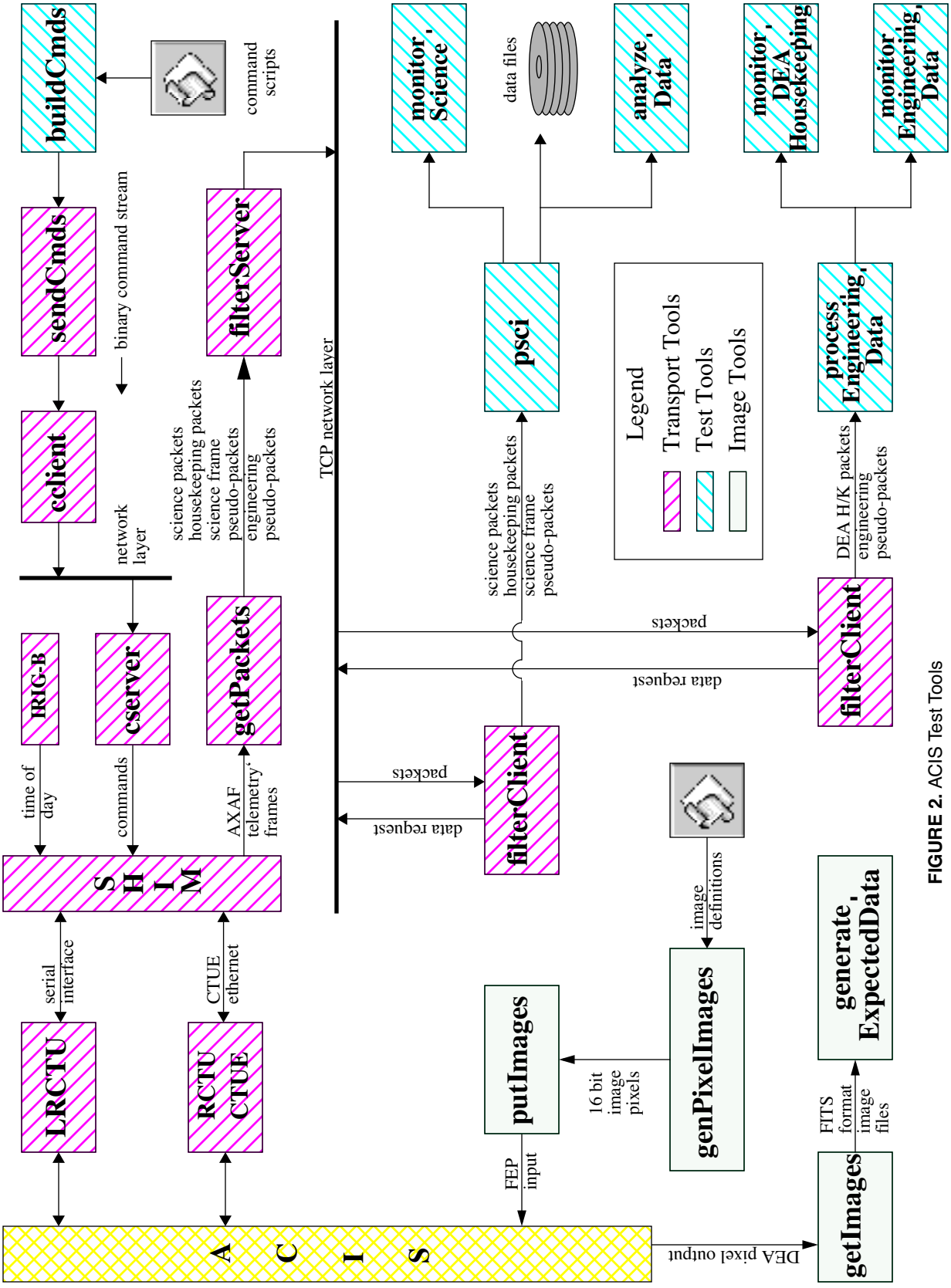


FIGURE 2. ACIS Test Tools

2.0 GSE Transport Tools

These components occupy the top third of Figure 2, except that ***buildCmds*** is a "GSE Test Tool". The ACIS instrument is represented by the box at the top left. It may communicate through as many as 3 simultaneous interfaces, the ***RCTU/CTUE***, representing the interface that will be used at XRCF and closely parallels the AXAF spacecraft interface itself; the ***LRCTU*** which is a simplified replacement for the ***RCTU/CTUE*** developed at MIT; the "High Speed Tap" which samples the digital image pixels being output from one of the ACIS analog boards; and the "Image Loader", an MIT-developed hardware interface that mimics the analog boards and permits test images to be sent directly to one or more FEP boards.

The software consists of the server-client pairs, their input and output interface, and their protocols. There are five GSE Transport programs: ***shim***, ***sendCmds***, ***getPackets***, ***filterServer*** and ***filterClient***. All are being developed by ACIS EGSE personnel, with assistance from the ACIS Flight Software Team. The major external interfaces are between ***buildCmds*** and ***sendCmds***, and between ***getPackets*** and its various clients (via the ***filterServer/filterClient*** combination). Both interfaces may be described as a single stream of binary bytes, with no timing constraints.

2.1 sendCmds

sendCmds receives a binary command stream from ***buildCmds***, containing *type*, *channel*, *data* triplets. It constructs 23-bit, formatted command strings, packages them into 24-bit strings to simplify the output interface, and sends them to ***shim***. This command is described in detail in Section 10.29.

2.2 cclient

cclient makes a TCP connection to a socket previously created by the ***cserver*** program. Once the connection has been established, ***cclient*** copies its standard input to the socket. When it encounters an end-of-file condition on *stdin*, it closes the socket and exits. Its function is therefore to isolate the commands generated by ***sendCmds*** and ***buildCmds*** from ***shim*** and ACIS itself—a series of commands can be issued from separate UNIX processes, even from different host computers, and ***cserver*** will merge them into a single unbroken command stream. This command is described in detail in Section 10.7.

2.3 cserver

cserver creates a socket and listens for connections from ***cclient*** processes. When one is made, it copies the contents of the socket to its standard output, *stdout*. On an end-of-socket or error condition, ***cserver*** closes the connection and waits for another one. This command is described in detail in Section 10.8.

2.4 shim

shim provides a consistent interface between ACIS and all user applications that generate commands or receive telemetry. It can communicate with ACIS either through the ***LRCTU***

(RCTU/CTUE emulator) or through the RCTU/CTUE itself. This command is described in detail in Section 10.30

When sending commands to an ***LRCTU***, ***shim*** drops all High Level Pulse commands because the ***LRCTU*** does not support them. It passes all other Serial Digital Hardware and Software commands to the ***LRCTU*** via a serial interface in the 24 bit format generated by ***sendCmds***. When receiving telemetry from an ***LRCTU*** via the same serial interface, ***shim*** formats the packets into AXAF-I telemetry major frames, adding science frame headers and ACIS and IRIG-B timestamps as appropriate.

When ***shim*** sends commands to an ***RCTU/CTUE***, it includes High Level Pulse commands as well as Serial Digital Hardware and Software commands. It extracts the 23-bit input command words from the 24 bit ***sendCmds*** format and packs them into 48-bit Ground Command Format strings. It then assembles these strings into command blocks, which it sends to the ***RCTU/CTUE*** via a TCP/IP interface. It receives telemetry via the same interface and passes them to its client (***getPackets***) unmodified.

2.5 getPackets

getPackets receives AXAF-I telemetry frames from ***shim***. It extracts ACIS-related information and passes it to its ***filterServer*** client. It must first identify the current telemetry format—either 1 or 2. In the former, ACIS science data is being generated at 512 bits/sec; in the latter at 24 Kbits/sec. In both cases, ***getPackets*** assembles the serial telemetry from ACIS, locates the individual packets by their synch words and lengths, and writes them to ***filterServer*** as separate logical records.

getPackets also sends ***filterServer*** two types of “pseudo-packet”, i.e. records whose format mimics genuine ACIS telemetry packets but whose “type” codes are distinguished from those used by the instrument itself. One type of pseudo-packet contains data from science frame headers and from ACIS timestamps. The other contains ACIS and other AXAF engineering data that was found in the non-science areas of the telemetry frames. This command is described in detail in Section 10.17.

2.6 filterServer

filterServer receives the stream of telemetry packets from *stdin* and listens on an INET socket for network clients to request TCP connections. When this occurs, ***filterServer*** determines the data types requested by the client, and then forks a copy of itself to write those packets to the client. Clients connecting to ***filterServer*** send it a short message that indicates which of the 4 types of telemetry they wish to receive. ***filterServer*** writes all telemetry packets of the requested types to the output socket.

If ***filterServer*** doesn’t understand the data request, it closes the INET socket and writes an error message to *stderr*. Once a connection is made, the packet type cannot be changed. When a client no longer wants packets, it closes the socket—the spawned server process should `exit` after logging the event to *stderr*. This command is described in detail in Section 10.14.

2.7 filterClient

filterClient inspects its argument list to determine the location (host and port number) of a ***filterServer*** process, and which types of packet it is to request. It establishes a TCP connection to the server, sends a request for data, and copies the resulting stream to *stdout*. This command is described in detail in Section 10.13.

2.8 Transport Tool Interfaces

2.8.1 Stdin to sendCmds

sendCmds expects its standard input to consist of pairs of 16-bit words (command type and channel) followed by one or more 16-bit words—the command packet, as described in Table 1 All 16-bit words are assumed to start with their least significant bytes, i.e. little-endian order.

TABLE 1. Command Type and Channel Definition

Command Type		Command Channel		Description
Name	Value	Name	Value	
Serial Digital	2	Software	2	Command used to control the ACIS software
		Hardware	3	Command used to control the ACIS hardware
High Level Pulse	0	Pulse Cmd Channel Number	0-98	PS and MC commands, whose action is determined by the Command Channel value
No-Op	3	TBD	TBD	Potential RCTU/CTUE operation commands

The format and content of command packets are contained in the AXAF IP&CL documents. All packets contain length fields which are extracted by ***sendCmds*** to determine how to read the remainder of the command packet.

High Level Pulse Commands are completely specified by the Command Type and Command Channel pair. Therefore, no command data will follow.

Serial Digital Hardware Commands will consist of a single 16-bit word that will immediately follow the Command Type and Command Channel pair.

Serial Digital Software Commands will consist of from 3 to 256 16-bit words contained in an ACIS software command packet that will immediately follow the Command Type and Command Channel pair. All software command packets contain length fields which are extracted by *sendCmds* to determine how to read the remainder of the command packet. The output, which is produced in 3-byte groups, is described in Table 2. Commands will be passed to ***shim*** as soon as

TABLE 2. *sendCmds* Command Formats

Serial Digital Commands		High Level Pulse Commands	
Bit[1]	Contents	Bit	Contents
0	Unspecified	0	Unspecified

TABLE 2. *sendCmds* Command Formats

Serial Digital Commands		High Level Pulse Commands	
1-2	Command Type	1-2	Command Type
3-18	Command Data	3-14	Unspecified
19-23	Command Channel	15-23	Command Channel

1. bit 0 is the most significant bit and is transmitted/received first

each 16-bit Command Data word is read from *stdin*. Since this is buffered via the *stdio.h* library, it is the responsibility of a program piping commands to ***sendCmds*** to flush the pipe, e.g. with a call to *fflush*(), before any planned inter-command delay. Otherwise, the time delay will be unpredictable.

When ***sendCmds*** reads an illegal packet from *stdin*, it writes an error message to *stderr*. Depending on arguments supplied on its command line, ***sendCmds*** may then decide to continue processing the next command from *stdin*, or abort the run entirely.

2.8.2 Stdout from filterClient

getPackets writes a stream of telemetry packets and pseudo-packets to *stdout*, These are passed through ***filterServer*** to each ***filterClient***, which writes a user-selected sub-set to its *stdout*. The four packet types that may be selected are: “ACIS Science Packets”, “ACIS DEA and Software Housekeeping Packets”, “Science-Frame Pseudo-Packets”, and “Engineering Pseudo-Packets”. Each packet consists of a telemetry header followed by application data, as defined in IP&CL. The header formats and contents are defined in Table 3.

TABLE 3. Header Format and Content

Packet Header Fields	Field Length (bits)	Science or Housekeeping Packet	Science Frame Pseudo-Packet	Engineering Pseudo-Packet
Synch	32	0x736f4166	0x736f4166	0x736f4166
Length	10	Varying[1]	7	Varying[1]
Format Tag	6	Varying	62	61
Sequence Number	16	incremented by 1 for each packet in the telemetry stream	0[2]	0[2]

1. The packet length (number of 32 bit words) varies with the contents.
2. The sequence number of a pseudo-packet is always zero.

All fields are written in “little-endian” format, e.g. the packet synch word, `0x736f4166`, is written as 4 bytes, `0x66`, `0x41`, `0x6f`, and finally `0x73`. The contents of all packets originating within ACIS are defined in IP&CL. The data portion of the Science Frame pseudo-packet is described in Table 4 and that of the Engineering pseudo-packet in Table 5.

Each packet will be written to the ***getPackets*** standard output stream as soon as the last data byte that contributes to it is read from ***shim***. A Science Frame Pseudo-Packet will be written after

TABLE 4. Science Frame Pseudo-Packet Format and Content

Field Name	Source			*filterClient* Output Format	Description
	Location	Start	Length		
`format`	Virtual Channel ID	bit 10	3 bits	unsigned int	Frame format identifier, either `1` (signifying 512 bps) or `2` (24 kbps), *i.e.* the AXAF *tlm* code + 1.
`majorFrameId`	CCSDS Header	bit 16	17 bits	unsigned int	Virtual Channel Data Unit Major Frame Count (0 to 131071)
`minorFrameId`	CCSDS Header	bit 33	7 bits	unsigned short	Virtual Channel Data Unit Major Frame Count (0 to 127)
`irigb`	Science Header	byte 32	6 bytes	unsigned short [3]	Time (msec) from the IRIG-B interface
`bepSciTime`	Science Data	byte 56	4 bytes	unsigned int	Latched version of the BEP science pulse 1 MHz timestamp
	Next-in-line Data	TBD			

TABLE 5. Engineering Pseudo-Packet Format and Content

Field Name	Source			*filterClient* Output Format	Description
	Location	Start	Length		
`format`	Virtual Channel ID	bit 10	3 bits	unsigned int	Frame format identifier, either `1` (signifying 512 bps) or `2` (24 kbps), *i.e.* the AXAF *tlm* code + 1.
`majorFrameId`	CCSDS Header	bit 16	17 bits	unsigned int	Major Frame Counter (0 to 131071)
followed by an array of one or more elements, each consisting of the following fields					
`data`	Variable location within major frame	var	8 bits	unsigned char	Engineering data
`minorFrameId`	CCSDS Header	bit 33	7 bits	unsigned char	Virtual Channel Data Unit Frame Counter (0 to 127)
`minorFrameByte`				unsigned short	Byte number in the minor frame (0 to 1024)

reading the last byte of each complete minor frame containing a science frame header. An Engineering Pseudo-Packet will be written after the last byte of each complete major frame is read.

The 6-byte IRIG-B timestamp in the AXAF-I minor frame is the result of packing 4 separate bit fields into a 48-bit string. However, ***getPackets*** treats the 6 bytes of the IRIG-B timestamp as 3 unsigned 16-bit integers. It copies them into the Engineering Pseudo-Packet and converts them to little-endian format, which is the ACIS standard. Table 4 describes how to decipher the Engineering Pseudo-Packet's `irigb` field.

TABLE 6. IRIG-B Field Format and Contents

Field Name	Bit Length	Byte	Word
Julian Day	11	0,1	0
Seconds	17	1,2,3	0,1
Milliseconds	10	3,4	1,2
Microseconds (always zero)	10	4,5	2

2.8.3 filterClient arguments

The content of ***filterClient***'s *stdout* stream and the location and port number of the ***filterServer*** are selected by UNIX runtime arguments, as described in Section 10.14 and Section 10.13. The telemetry packet selections will be determined by the presence of one or more of the mnemonics listed in Table 7.

TABLE 7. filterClient packet classes

Class	Packets Selected	Description
SCI	ACIS Science	ACIS science packets corresponding to the format numbers 1 through 9 and 12 through 37
HKP	DEA & Software Housekeeping	ACIS packets corresponding to the two formats numbered 10 (TTAG_SW_HOUSE) and 11 (TTAG_DEA_HOUSE)
HDR	Science Frame Pseudo-Packet	Pseudo-Packets generated within `getPackets` containing science frame and BEP science timestamp information
ENG	Engineering Pseudo-Packet	Pseudo-Packets generated within getPackets containing ACIS and other spacecraft engineering channel readouts

3.0 GSE Test Tools

These consist of two groups of UNIX commands—primitive programs that translate between binary ACIS representations (commands and telemetry packets) and their human-readable ASCII equivalents; and analysis programs that perform various higher-level functions. All programs are being developed by the ACIS flight software team, except were noted.

3.1 buildCmds

This program, described in detail in Section 10.6, reads an ASCII command script from *stdin* and writes a binary command stream to *stdout*. The binary output format is described in “Stdin to sendCmds” on page 5.

Each ACIS command must appear on a separate input line in the standard input to ***buildCmds***. The only exception is that in-line parameter blocks may contain newline characters within enclosing braces. The full list of ***buildCmds*** commands are shown in the following tables.

TABLE 8. Serial Commands to BEP Software

Command[1]	Arguments	Description
add	*id* cc badColumn *file* *id* cc badColumn *paramBlock*	Add entries to the continuous clocking bad column block from binary *file* or in-line *paramblock*
	id te badColumn *file* *id* te badColumn *paramBlock*	Add entries to the timed exposure bad column block from binary *file* or in-line *paramblock*
	id badPixel *file* *id* badPixel *paramBlock*	Add entries to the timed exposure bad pixel map block from binary *file* or in-line *paramblock*
	id patch patchId *file*	Uplink a software patch from a binary file
change	*id* systemConfig *file* *id* systemConfig *paramBlock*	Change entries in the system configuration parameter block from binary *file* or in-line *paramblock*
continue	*id* upLink *file*	Continue uplink boot from a binary file
dump	*id* badPixel	Dump the bad pixel map block
	id cc	Dump all continuous clocking parameter blocks
	id cc badColumn	Dump the bad column map block used by continuous clocking science modes
	id dea	Dump the DEA housekeeping monitor parameter block
	id huffman	Dump all Huffman data compression tables
	id patchList	Dump the patch list
	id systemConfig	Dump the system configuration table
	id te	Dump all timed exposure parameter blocks
	id te badColumn	Dump the bad column map used by timed exposure science modes
	id window1D	Dump all 1-dimensional window blocks
	id window2D	Dump all 2-dimensional window blocks

TABLE 8. Serial Commands to BEP Software (Continued)

Command[1]	Arguments	Description
exec	*id address* [*args*]	Execute the function located in the BEP memory at the specified *address* (a multiple of 4), with optional 32-bit *arguments*
	id fep *fepId address* [*args*]	Execute the function located at the specified *address* (a multiple of 4) of FEP number *fepId*, with optional 32-bit *arguments*
load	*id* cc *slotId file* *id* cc *slotId paramBlock*	Load a continuous clocking parameter block at the specified *slotId* from binary *file* or in-line *paramblock*
	id dea *slotId file* *id* dea *slotId paramBlock*	Load a DEA housekeeping parameter block at the specified *slotId* from binary *file* or in-line *paramblock*
	id te *slotId file* *id* te *slotId paramBlock*	Load a timed exposure parameter block at the specified *slotId* from binary *file* or in-line *paramblock*
	id window1D *slotId file* *id* window1D *slotId paramBlock*	Load a one-dimensional window parameter block at the specified *slotId* from binary *file* or in-line *paramblock*
	id window2D *slotId file* *id* window2D *slotId paramBlock*	Load a two-dimensional window parameter block at the specified *slotId* from binary *file* or in-line *paramblock*
read	*id address length*	Read *length* 32–bit words of BEP memory starting at the specified *address*
	id fep *fepId address length*	Read *length* 32–bit words of memory from FEP number *fepId*, starting at the specified *address*
	id pram *ccdId address length*	Read *length* 16–bit words of PRAM memory from DEA number *ccdId*, starting at the specified 16-bit word *address*
	id sram *ccdId address length*	Read *length* 16–bit words of SRAM memory from DEA number *ccdId*, starting at the specified 16-bit word *address*
remove	*id* patch *patchId*	Remove *patchId* from the software patch list
reset	*id* badPixel	Remove all entries from the bad pixel map
	id cc badColumn	Remove all entries from the continuous clocking bad column map
	id te badColumn	Remove all entries from the timed exposure bad column map
start	*id* cc *slotId*	Start a continuous clocking science run with parameters from *slotId*
	id cc bias *slotId*	Start a continuous clocking bias calculation with parameters from *slotId*
	id dea *slotId*	Start the DEA housekeeping monitor with parameters from *slotId*
	id te *slotId*	Start a timed exposure science run with parameters from *slotId*
	id te bias *slotId*	Start a timed exposure bias calculation with parameters from *slotId*
	id upLink *file*	Start an uplink boot from binary data in *file*
stop	*id* dea	Stop the currently executing DEA housekeeping monitor
	id science	Stop the currently executing science run
wait	*id seconds*	Suspend command output an integral number of *seconds*

TABLE 8. Serial Commands to BEP Software (Continued)

Command[1]	Arguments	Description
write	*id address file*	Write the binary contents of *file* to BEP memory, starting at *address* (divisible by 4)
	id fep *fepId address file*	Write the binary contents of *file* to FEP number *fepId*, memory starting at *address* (divisible by 4)
	id pram *ccdId address file*	Write the binary contents of *file* to the PRAM memory of DEA number *ccdId*, starting at 16-bit word *address*
	id sram *ccdId address file*	Write the binary contents of *file* to the SRAM memory of DEA number *ccdId*, starting at 16-bit word *address*

1. all ACIS Software Serial Commands are assigned the mnemonic 1SWSDICL in the AXAF IP&CL tables.

TABLE 9. Serial commands to BEP Hardware

Command	Arguments	Mnemonic[1]	Description
halt	bep	1RSETIRT	Reset (i.e. halt) the BEP processor
run	bep	1RSETIRT	Run the BEP processor. This is the default on BEP power up
select	bep *bepId*	1BSELICL	Select which BEP to use (0 for BEP A, 1 for BEP B). The default on power up is 0 (BEP A)
	eeprom *mode*[2]	TBD	Select the EEPROM mode: either PROGRAMMING for programmer data readout or TELEMETRY for software bi-level telemetry. The default on power up is TELEMETRY
set	bootmodifier *mode*	1BMODIBM	Set the boot modifier mode: either ON to boot from uplink, or OFF to boot from ROM. The default on BEP power up is OFF
	radiationmonitor *mode*	1RMONIRM	Set the radiation mode to HIGH or LOW. The default on DEA power up is LOW
	warmboot *mode*	1SBYISB	Set the warm boot flag either ON or OFF

1. command mnemonics are defined in the AXAF IP&CL tables.
2. EEPROM commands cannot be executed on flight hardware after SI integration.

TABLE 10. Pulse Commands to PSMC Hardware

Command	Arguments[1]	Mnemonic[2]	Description[1]
close	door *id*	1MCDR*ON	close door *id*
	vent *id*	1VVCC*ON	close vent valve *id*
	relief *id*	1LVCC*ON	close little vent valve *id*
closeabort	door *id*	1MCDR*OF	stop closing door drive *id*
	vent *id*	1VVCC*OF	stop closing vent valve *id*

TABLE 10. Pulse Commands to PSMC Hardware (Continued)

Command	Arguments[1]	Mnemonic[2]	Description[1]
disable	daBake *id*	1HBO*DS	disable commands to bakeout heater *id*
	daHeater *id*	1HHTR*DS	disable commands to housing heater *id*
	dea *id*	1DEPS*DS	disable commands to DEA power supply *id*
	door *id*	1MCMD*DS	disable commands to door mechanism drive *id*
	dpa *id*	1DPPS*DS	disable commands to DPA power supply *id*
	pressure *id*	1PRES*DS	disable pressure sensor *id*
	relief *id*	1LVC*DS	disable commands to little vent valve *id*
	vent *id*	1VVC*DS	disable commands to side vent valve *id*
enable	daBake *id*	1HBO*EN	enable commands to bakeout heater *id*
	daHeater *id*	1HHTR*EN	enable commands to housing heater *id*
	dea *id*	1DEPS*EN	enable commands to DEA power supply *id*
	door *id*	1MCMD*EN	enable commands to door mechanism drive *id*
	dpa *id*	1DPPS*EN	enable commands to DPA power supply *id*
	pressure *id*	1PRES*EN	enable pressure sensor *id*
	relief *id*	1LVC*EN	enable commands to little vent valve *id*
	vent *id*	1VVC*EN	enable commands to side vent valve *id*
open	door *id*	1MODR*ON	open door *id*
	vent *id*	1VVCO*ON	open vent valve *id*
	relief *id*	1LVCO*ON	open little vent valve *id*
openabort	door *id*	1MODR*OF	stop opening door drive *id*
	vent *id*	1VVCO*OF	stop opening vent valve *id*
poweroff	daBake *id*	1HBO*OF	power off bakeout heater *id*
	daHeater *id*	1HHTR*OF	power off housing heater *id*
	dea *id*	1DEPS*OF	power off DEA power supply *id*
	dpa *id*	1DPPS*OF	power off DPA power supply *id*
poweron	daBake *id*	1HBO*ON	power on bakeout heater *id*
	daHeater *id*	1HHTR*ON	power on housing heater *id*
	dea *id*	1DEPS*ON	power on DEA power supply *id*
	dpa *id*	1DPPS*ON	power on DPA power supply *id*

TABLE 10. Pulse Commands to PSMC Hardware (Continued)

Command	Arguments[1]	Mnemonic[2]	Description[1]
turnoff	daBake	1HBOAOF 1HBOBOF 1HBOADS 1HBOBDS	power off and disable both bakeout heaters
	daBake *id*	1HBO*OF 1HBO*DS	power off and disable bakeout heater *id*
	daHeater	1HHTRAOF 1HHTRBOF 1HHTRADS 1HHTRBDS	power off and disable both housing heaters
	daHeater *id*	1HHTR*OF 1HHTR*DS	power off and disable housing heater *id*
	dea	1DEPSAOF 1DEPSBOF 1DEPSADS 1DEPSBDS	power off and disable both DEAs
	dea *id*	1DEPS*OF 1DEPS*DS	power off and disable DEA *id*
	dpa	1DPPSAOF 1DPPSBOF 1DPPSADS 1DPPSBDS	power off and disable both DPAs
	dpa *id*	1DPPS*OF 1DPPS*DS	power off and disable DPA *id*
turnon	daBake *id*	1HBO*EN 1HBO*ON	enable and power on bakeout header *id*
	daHeater *id*	1HHTR*EN 1HHTR*ON	enable and power on housing heater *id*
	dea *id*	1DEPS*EN 1DEPS*ON	enable and power on DEA *id*
	dpa *id*	1DPPS*EN 1DPPS*ON	enable and power on DPA *id*

1. the *id* field represents the hardware redundancy, either 0 for the A-side or 1 for the B-side.
2. command mnemonics are defined by AXAF IP&CL tables. '*' represents the hardware redundancy, either 'A' or 'B'.

3.1.1 buildCmds Examples

- The following UNIX pipe commands the ACIS instrument to start executing stored timed exposure parameter block 1:

```
echo 'start 22 te 1' | buildCmds | sendCmds
```

- Dump 600 words from FEP number 2, starting at word offset 45678:

```
echo 'read 4 fep 2 45678 600' | buildCmds | sendCmds
```

- Start a timed exposure using the parameter block in slot #2. Give this command the identifier '11'.

```
echo start 11 te 2 | buildCmds | sendCmds
```

- Stop a science run—give this command the identifier '5'.

```
stop 5 science | buildCmds | sendCmds
```

- Load a 1-dimensional window block. The order of keywords within each window structure is significant—if a keyword is omitted, the most recent value will be used (zero if it has not yet been used within the block).

```
load 33 window1D 4 {
  parameterBlockName        = window1D
  windowBlockId             = 45
  arrayDim                  = 3             ⇐ number of structures to follow
  ccdId                     = 1             ⇐ first window structure
  ccdColumn                 = 2
  width                     = 4
  sampleCycle               = 6
  lowerEventAmplitude       = 7
  eventAmplitudeRange       = 8
  ccdId                     = 2             ⇐ second window structure
  ccdColumn                 = 3
  width                     = 6
  sampleCycle               = 8
  lowerEventAmplitude       = 10
  eventAmplitudeRange       = 20
  ccdId                     = 3             ⇐ third window structure
  ccdColumn                 = 20
  width                     = 40
  sampleCycle               = 2
  lowerEventAmplitude       = 70
  eventAmplitudeRange       = 80
}
```

- Load a Continuous Clocking parameter block. The parameter block is to be stored in slot #3 and the command is given the identifier '22'. The keywords must appear in the order shown. If omitted, a zero value will be assumed.

```
load 22 cc 3 {
  paramBlockName            = ccBlock
  parameterBlockId          = 2030
  fepCcdSelect              = 0 1 2 3 4 5
  fepMode                   = 1
  bepPackingMode            = 1
  ignoreBadColumnMap        = 0
  recomputeBias             = 1
  trickleBias               = 1
  rowSum                    = 4
  columnSum                 = 5
  overclockPairsPerNode     = 8
  outputRegisterMode        = 2
  CcdVideoResponse          = 1 1 1 1 1 1
  fep0EventThreshold        = 100 100 100 100
  fep1EventThreshold        = 100 100 100 100
  fep2EventThreshold        = 100 100 100 100
  fep3EventThreshold        = 100 100 100 100
  fep4EventThreshold        = 100 100 100 100
```

```
  fep5EventThreshold         = 100 100 100 100
  fep0SplitThreshold         = 80 80 80 80
  fep1SplitThreshold         = 80 80 80 80
  fep2SplitThreshold         = 80 80 80 80
  fep3SplitThreshold         = 80 80 80 80
  fep4SplitThreshold         = 80 80 80 80
  fep5SplitThreshold         = 80 80 80 80
  lowerEventAmplitude        = 800
  eventAmplitudeRange        = 3000
  gradeSelections            = 0xf[1]
  windowSlotIndex            = 2
  rawCompressionSlotIndex    = 1
  ignoreInitialFrames        = 2
  biasAlgorithmId            = 1 1 1 1 1 1
  biasRejection              = 2 2 2 2 2 2
  fep0VideoOffset            = 1000 1000 1000 1000
  fep1VideoOffset            = 1000 1000 1000 1000
  fep2VideoOffset            = 1000 1000 1000 1000
  fep3VideoOffset            = 1000 1000 1000 1000
  fep4VideoOffset            = 1000 1000 1000 1000
  fep5VideoOffset            = 1000 1000 1000 1000
  deaLoadOverride            = 0
  fepLoadOverride            = 0
}
```

3.2 lcmd

This program, described in detail in Section 10.18, reads a binary command file, e.g. one that was generated by ***buildCmds***, and writes its contents to *stdout* in ASCII, e.g.

```
echo 'read 4 fep 2 45678 600' | buildCmds | lcmd
```

generates the following output:

```
readFep[0] = {
  commandLength              = 8
  commandIdentifier          = 4
  commandOpcode              = CMDOP_READ_FEP (4)
  fepId                      = 2
  readAddress                = 0x0000b26e
  wordCount                  = 600
}
```

3.3 ltlm

This program, described in detail in Section 10.23, reads a stream of ACIS telemetry packets, e.g. one that was generated by ***getPackets*** or ***filterClient***, and writes its contents to *stdout* in ASCII, e.g.

```
filterclient -m ps | ltlm
```

might generate the following output:

```
scienceFramePseudo[0] = {
  synch                      = 0x736f4166
```

1. The continuous clocking `gradeSelection` value consists of an optional '0x' followed by a hexadecimal digit (0–9, a–f).

```
  telemetryLength            = 7
  formatTag                  = TTAG_PSEUDO_SCIENCE (62)
  sequenceNumber             = 0
  format                     = 2
  majorFrameId               = 0
  minorFrameId               = 0
  irigBdays                  = 935
  irigBsecs                  = 72260
  irigBmsecs                 = 0
  irigBusecs                 = 0
  bepSciTime                 = 0xa5997aff
}
bepStartupMessage[0] = {
  synch                      = 0x736f4166
  telemetryLength            = 7
  formatTag                  = TTAG_STARTUP (8)
  sequenceNumber             = 0
  bepTickCounter             = 0x00000237
  version                    = 2147483647
  lastFatalCode              = 0
  lastFatalValue             = 0
  watchdogFlag               = 0
  patchValidFlag             = 1
  configFlag                 = 1
  parametersFlag             = 1
}
```

3.4 psci

This program, described in detail in Section 5.0, reads telemetry packets from *stdin* (e.g. the output of ***filterClient***), outputs a continuous packet summary to *stdout*, and writes selected science data to disk files or UNIX pipes. The summaries always include the following information from all packets to verify the ACIS execution sequence:

- packet name
- sequence number

They also include detailed information that depends on the telemetry packet type for the purpose of verifying some detail of ACIS command execution. The information is written to *stdout* in ASCII characters, one item per line.

In addition to its monitoring function, ***psci*** may also be commanded to extract specific data fields from telemetry packets and write them to disk files or pipes. ***psci*** generates several sets of log files, each one corresponds to a group of telemetry packets. The file content is identified by its name, examples of which are shown in Table 11.

3.5 analyzeData

This is a suite of programs to perform various data analysis functions, including those described in “Analysis Procedure” on page 62.

3.6 runacis

This is a UNIX Bourne shell script that executes the BEP software simulator, ***acisBepUnix***, described in Section 10.2, and a single copy of the FEP software simulator, ***acisFepUnix***, described in Section 10.3, on a remote host. Its standard input stream, *stdin*, should consist of a binary ACIS command stream, e.g. as output by ***buildCmds***. Its standard output stream, *stdout*, will consist of a stream of ACIS packets such as those generated by ***getPackets*** and ***filterClient***. The user may specify as command line option the name of a shell script that generates a pixel stream to be read by ***acisFepUnix***. ***runacis*** is described in detail in Section 10.27.

3.7 monitorDeaHousekeeping

This program reads ASCII-format DEA housekeeping summaries from *stdin* and displays their contents on a graphical interface. This program is being developed by ACIS EGSE personnel

3.8 monitorEngineeringData

This program reads ASCII-format engineering summaries from *stdin* and displays their contents on a TBD graphical interface. This program is being developed by ACIS EGSE personnel

3.9 monitorScience

This program reads ASCII-format packet summaries from *stdin* and displays their contents in a Tcl/Tk interface. It is described in detail in Section 10.24.

3.10 processEngineeringData

This program reads packets from *stdin*, inspects all engineering telemetry packets (ignoring the remainder), and writes to *stdout* a summary of these packets in a TBD format. This program is being developed by ACIS EGSE personnel.

4.0 Image Tools

Image operations fall into two categories: (a) simulating DEA output and feeding it into one or more Front End Processors (FEPs), and (b) intercepting real DEA output for off-line examination. The first task is accomplished by means of a “Frame Buffer”, a dedicated hardware interface that down-loads pixels to the FEPs at a measured rate, and the latter by a “High-Speed Tap”, essentially the same process in reverse.

4.1 getImages

The ***getImages*** command instructs an attached ARIEL signal processor to capture the output of a DEA controller via its high speed tap and to write it to a series of disk files. The format is described in Section 9.2 on page 57.

4.2 putImages

putImages reads a stream of 16-bit pixels from ***stdin*** and writes them to an attached Frame Buffer for transmittal to one or more FEPs. The process is described in detail in Section 8.0.

4.3 genPixelImages

genPixelImages reads input commands from *stdin* and writes images to *stdout* in a format suitable for loading into the “Frame Buffer” described in Section 8.0. This format consists of 16 bit-words containing frame-buffer directives, FEP synchronization codes, and pixel and overclock values. Each image begins with four VSYNC codes and may contain from 1 to 1024 “rows”, each beginning with four HSYNC codes. Each row may contain between 4 and 1024 “columns”, divided into “nodes” (four in “ABCD” mode, two in either “AC” or “BD” mode), and followed by 0 to 15 pairs of overclocks per node. Note that the fourth, diagnostic, clocking mode generates no pixel values. It is therefore simulated by “ABCD” mode and no separate ***genPixelImages*** option is required. This command is described in detail in Section 10.16

4.4 loadFitsImage

This is an alternative to ***genPixelImages*** for creating binary pixel streams for the Frame Buffer (Section 8.0 on page 53) from an existing 2-D FITS image such those created by ***putImages***. This command is fully described in Section 10.21.

4.5 genObjectImage

This is an alternative to ***genPixelImages*** and ***loadFitsImage***. Instead of defining the pixels row-by-row, its command language (read from *stdin*) defines the characteristics of the output image, of each of its output nodes, and of various objects (“events” and “blobs”), which are then located at various row and column addresses in the image. This command is fully described in Section 10.15.

4.6 generateExpectedData

This is a library of UNIX programs that duplicate the steps used by ACIS flight software in processing DEA pixel streams into raw pixel images, histograms, and photon event lists. Each step is implemented as a filter, permitting multiple to be applied in sequence to the same input data. The purpose of these programs is to repeat in a UNIX environment the operations executed by ACIS flight software in order to verify their correctness.

4.7 Image Tool Interfaces

The major external interfaces are between ***genPixelImages*** and ***putImages***, and between ***getImages*** and various image display programs. Both interfaces may be described as a single stream of binary bytes, with no timing constraints.

4.7.1 stdin to putImages

The data stream sent from ***genPixelImages*** to ***putImages*** consists of 16-bit words, containing numeric data in their 12 least significant bits and codes in their 4 most significant bits. Some words are interpreted as local "directive functions" within the Frame Buffer (see Section 8.0 on page 53). All other words are passed along to the FEPs, where their codes determine whether their 12-bit values are to be interpreted as CCD pixel data or overclocks, or whether the word is a no-op or FEP control code (HSYNC or VSYNC).

4.7.2 Output from getImages

The output from *getImages* consists of one or more files on magnetic disk. The files are written in FITS image format. Each set of VSYNC codes starts a new file. Within each image, each set of HSYNC codes begins a new row. The 4 most significant bits in each 16-bit data or overclock pixel are filled with zeroes. All other pixel types are ignored, i.e. they are not included in the output. The FITS headers are described in Section 9.2 on page 57. The files are created with a user-specified base name, followed by a sequentially increasing frame number, followed by a file extension of "`.fits`".

5.0 The psci Command

This program reads a stream of ACIS packets, verifies their format and internal consistency, and optionally, sorts, reformats, and writes them to a series of data streams and disk files, as detailed in Table 11. The UNIX command syntax is as follows:

```
psci [-BDTVacmpqsuv] [-h name] [-l name] [file]
```

Packets are read from the input *`file`* named on the ***psci*** command line, or, if omitted, from the standard input stream, *stdin*. They are subjected to a variety of tests, as detailed below. If the `–l` option is specified, their headers are translated into ASCII and written to log files. If `–m` is specified, a one-line description is written to *stdout*, suitable for display by ***monitorScience*** (*q.v.*) Most packets are then discarded, and ***psci*** reads the next one, but some are retained, as follows:

- the most recent exposure header packet from each FEP,
- all event data packets, until a corresponding exposure header packet is encountered,
- multi-packet memory read-out packets originating from a single BEP command
- the most recent `dumped*Block` and `dumpedHuffman` packets.

5.1 Packet Field Verification

psci has been compiled with tables derived directly from the IP&CL Structures database. Packets with unrecognized TTAG codes (as defined in the *"acis_h/interface.h"* file) cause warning messages to be written to *stderr*, and are ignored. All fields in recognized packets and pseudopackets[1] are then checked against their IP&CL limits—bit fields are expanded to "`unsigned long int`" values unless their minimum permissible values (column 15 in the IP&CL structure tables) are negative, in which case, ***psci*** treats them as twos-complement signed integers and expands them to "`long int`". If a field is discovered to be out of range, ***psci*** writes a message to *stderr*, e.g.

```
file: packet[ntotal,ncount].field[index] above maximum (val > maxval)
file: packet[ntotal,ncount].field[index] below minimum (val < minval)
```

This example illustrates several features of ***psci***. All *stderr* messages begin with a *`file:`* argument; for errors and warnings, this is the name of the input file (or "*stdin*"); for informatory messages, it is usually the name of an output file. Packets are designated by their IP&CL names[2], *e.g.*, `exposureTeRaw`, followed by *`ntotal`*, the sequence number of the packet within the input stream, and *`ncount`*, the sequence number among packets of this particular type. Both counts start at zero, so the first packet is `[0,0]`. Multi-dimensional fields within packets are followed by an array *`index`*, which also starts at `0`. The *`value`* of the field is displayed as a decimal integer.

1. Since the format of pseudopackets is not governed by IP&CL, their fields are described to ***psci*** at compile time in a file named *pseudo.map*, in an identical format to the intermediate *cmd.map* and *tlm.map"* files used to construct C structures from IP&CL tables, as described in §5.12, below.

2. Many field names in the IP&CL tables contain spaces. ***psci*** treats them consistently as single words by (a) capitalizing all words but the first, and (b) removing the spaces. Thus, "Command Opcode" becomes `commandOpcode`, not `CommandOpcode` or `commandOpCode`.

TABLE 11. Output files and streams generated by *psci*

File or Stream	Contents	Remarks
stdout	One line per input packet, containing the packet name followed by one or more `keyword=value` fields.	For use by ***monitorScience***. Only generated when the `–m` option is specified on the ***psci*** command line. For an example, see Table 13.
stderr	Error, warning, and informatory messages.	Warning messages will be suppressed by including `–q` on the ***psci*** command line; informatory messages are only generated if `–v` is specified.
Files containing formatted listings of packet headers		
name`.`*s*`.bias.log`	Bias packets from science run *s*.	
name`.command.log`	`CommandEcho` packets.	Details all commands received and echoed by the ACIS BEP.
name`.deahk.log`	`DeaHousekeeping` packets.	
name`.`*s*`.science.log`	Science packets from run *s*.	Includes the contents of all `load*Block` commands within `dumped*Block` packets.
name`.packet.log`	Miscellaneous packets.	Describes all packets not logged in one of the other files.
name`.pseudo.log`	Pseudopackets.	Only generated if the `–p` flag is specified on the ***psci*** command line.
name`.swhk.log`	`SwHousekeeping` packets.	Details all messages received from the BEP software housekeeper.
Files containing data in various other formats		
name`.`*s*`.`*n*`.erv.txt`	Events in "extended RV" format from FEP number *n* from science run *s*. `*.txt` if ASCII; `*.dat` if binary.	If the `–a` flag is specified on the ***psci*** command line, the file will be written in ASCII (see Table 15); else it will contain 36-byte binary records (Table 14).
name`.`*s*`.`*n*`.erv.dat`		
name`.`*s*`.`*n*`.`*i*`–`*j*`.hist.txt`	Histograms from exposures *i* through *j* of FEP number *n* from science run *s*. `*.txt` if ASCII; `*.fits` if binary.	If the `–a` flag is specified, contains columns of ASCII values (see Table 17), one per CCD output node; otherwise writes FITS files containing 4096 samples of 32-bit binary integers per output node.
name`.`*s*`.`*n*`.`*i*`–`*j*`.hist.fits`		
name`.`*s*`.packet.`*n*`.txt`	The contents of the *n*'th instance of a memory read-back *packet*—multiple packets generated by the same ACIS command will be concatenated.	If `–a` is specified on the ***psci*** command line, the data are written in groups of hexadecimal ASCII words to `*.txt` (except Huffman blocks which are formatted as shown in Table 19.) Without `–a`, they are written in binary to `*.dat`.
name`.`*s*`.packet.`*n*`.dat`		
name`.`*s*`.`*n*`.bias.fits`	A bias map from FEP number *n* from science run *s*, in FITS format.	Continuously clocked bias maps are replicated to 512 rows by 1024 columns. Timed exposure maps contain 1024 rows of 1024 columns (see Table 18).
name`.`*s*`.`*n*`.`*m*`.raw.fits`	Raw pixels from exposure *m* of FEP number *n* from science run *s*, in FITS format.	Overclock values are appended to each image line. Frame-average overclocks are written to the FITS header (see Table 18).
`name.s.time.txt`	Exposure time tag files.	See §5.5.
name`.`*s*`.`*n*`.TMP.fits`	A temporary file name used for raw data.	***psci*** will rename the file as soon as it determines the exposure number.

Since out-of-limits field-values are not considered to be sufficient reason for halting the program, ***psci*** writes these messages and continues processing. The messages themselves can be suppressed by invoking ***psci*** with the `"–q"` option.

5.2 Packet Logging

When the `-l` option is used, ***psci*** writes packet-header information to the log files listed in Table 11. The format of these files is derived from that of data structures in the C language, *e.g.* Table 12, which identifies itself as the first `dataTeFaintBias` packet, the 16th packet in the stream. Note that fields whose values are enumerated in *acis_h/interface.h* will be followed by `"#"` and the enumeration. Unsigned values larger than 32767 are shown in hexadecimal base, preceded by `"0x"`. The values of arrays of fields with dimension > 9 are not shown—merely their dimension. When a packet contains one or more command blocks, e.g. `dumpedTeBlock`, which contains a science parameter block (either `loadTeBlock` or `loadCcBlock`) and an optional window block (either `load1dBlock` or `load2dBlock`), the individual sub-fields are logged, shifted to the right by 2 columns.

As shown in Table 11, the *`"name"`* supplied with the `-l` option of ***psci*** is used as a common prefix to the names of all output files. When a `dumpedCcBlock` or `dumpedTeBlock` packet is received, a comment is written into all open log files, and the science run number is incremented. Any opened science and bias data files are automatically closed, and a warning message is written to *stderr* since they should have been closed: science files by the receipt of a previous `scienceReport` packet, and bias files when complete.

TABLE 12. Example of formatted packet logs

A timed-exposure faint-mode event packet

```
dataTeBiasMap[12,4] = {
  telemetryLength              = 779
  formatTag                    = 14 # TTAG_SCI_TE_BIAS
  sequenceNumber               = 9
  biasStartTime                = 0x9e73a7e5
  biasParameterId              = 4011
  ccdId                        = 6 # CCD_S2
  fepId                        = 2 # FEP_2
  dataPacketNumber             = 4
  initialOverclocks            =   180   184   181   184
  pixelsPerRow                 = 1023
  rowsPerBias                  = 1023
  ccdRow                       = 1015
  ccdRowCount                  = 1
  compressionTableSlotIndex    = 255
  compressionTableIdentifier   = 0xffffffff
  pixelCount                   = 2048
  data                         = [768]
}
```

TABLE 12. Example of formatted packet logs (Continued)

A timed-exposure bias data packet

```
dataTeFaint[15,0] = {
  telemetryLength              = 1021
  formatTag                    = 21 # TTAG_SCI_TE_DAT_FAINT
  sequenceNumber               = 11
  ccdId                        = 6 # CCD_S2
  fepId                        = 2 # FEP_2
  dataPacketNumber             = 0
  ccdRow                       = 1
  events                       = [138]
}
```

5.3 Monitor Output

When ***psci*** is invoked with the `-m` flag, monitor records are written to *stdout*. The example in Table 13 shows the start of a science run, from the BEP's restart message (`bepStartupMessage`) through the commands (`commandEcho`) used to configure the DEA and BEP, the dump of the parameter blocks (`dumpedTeBlock`), and the beginning of interleaved event (`dataTeFaintBias`) and bias (`dataTeBiasMap`) records. These records should be piped into ***monitorScience***. They are not intended to be read in this form by humans!

To save bandwidth, and lighten the load on ***monitorScience***, ***psci*** does not write all packet fields to the monitor stream—the excluded items include all arrays and fields identified by the `nomonitor` directive in the *tlm.aux* file used to build the ***psci*** executable (see §5.12).

A different level of monitoring is achieved when *psci* is invoked with the `-s` flag. In this case, all telemetry packets will be converted to ASCII and written to the standard output stream, *stdout*, in a format identical to that used for the various log files when the `-l` flag is used.

Finally, the `-u` flag causes *psci* to print all user-type pseudo-packets containing ASCII messages (*i.e.* those of `type = 0`) to the standard error stream, *stderr*.

5.4 Science Event Modes

psci saves the science parameter block and (optional) window block that ACIS reports at the start of a science run. It also saves event data blocks until an exposure packet is received for the corresponding FEP, at which time it writes 36-byte[3] binary event records to the appropriate *"name.s.n.m.*`erv.dat`*"* file as shown in Table 14. During this process, ***psci*** checks numerous fields in each FEP's event data, exposure, and parameter block packets, as detailed in §5.13. This process should be sufficient to detect any missing, mislabeled, or out-of-order packet, but the events themselves—their row and column indices and pixel values—are not examined. Discrepancies are reported to *stderr*, in one of two formats:

3. For efficiency, two null bytes are appended to each 34 bytes of event data. Each record contains 9 32-byte words.

TABLE 13. Packet monitor stream written to *stdout*

```
bepStartupMessage sequenceNumber=0 bepTickCounter=567 version=0xffffffff
    lastFatalCode=0 lastFatalValue=0 watchdogFlag=0 patchValidFlag=1
    configFlag=1 parametersFlag=1

commandEcho sequenceNumber=1 arrival=0xa5997d1e result=1
    commandIdentifier=0 commandOpcode=9

commandEcho sequenceNumber=2 arrival=0xa5997de8 result=1
    commandIdentifier=0 commandOpcode=11

commandEcho sequenceNumber=3 arrival=0xa5997e2c result=1
    commandIdentifier=0 commandOpcode=14

dumpedTeBlock sequenceNumber=4

dataTeBiasMap sequenceNumber=5 ccdId=6 fepId=2 dataPacketNumber=0
    pixelsPerRow=1023 rowsPerBias=1023 ccdRow=1023 ccdRowCount=1
    pixelCount=2048

dataTeBiasMap sequenceNumber=6 ccdId=6 fepId=2 dataPacketNumber=1
    pixelsPerRow=1023 rowsPerBias=1023 ccdRow=1021 ccdRowCount=1
    pixelCount=2048

dataTeBiasMap sequenceNumber=7 ccdId=6 fepId=2 dataPacketNumber=2
    pixelsPerRow=1023 rowsPerBias=1023 ccdRow=1019 ccdRowCount=1
    pixelCount=2048

dataTeBiasMap sequenceNumber=8 ccdId=6 fepId=2 dataPacketNumber=3
    pixelsPerRow=1023 rowsPerBias=1023 ccdRow=1017 ccdRowCount=1
    pixelCount=2048

dataTeBiasMap sequenceNumber=9 ccdId=6 fepId=2 dataPacketNumber=4
    pixelsPerRow=1023 rowsPerBias=1023 ccdRow=1015 ccdRowCount=1
    pixelCount=2048

dataTeBiasMap sequenceNumber=10 ccdId=6 fepId=2 dataPacketNumber=5
    pixelsPerRow=1023 rowsPerBias=1023 ccdRow=1013 ccdRowCount=1
    pixelCount=2048

dataTeFaintBias sequenceNumber=11 ccdId=6 fepId=2 dataPacketNumber=0
    ccdRow=1

dataTeBiasMap sequenceNumber=12 ccdId=6 fepId=2 dataPacketNumber=6
    pixelsPerRow=1023 rowsPerBias=1023 ccdRow=1011 ccdRowCount=1
    pixelCount=2048
```

file: *packet1*[*tot1*,*cnt1*].*field1*=*val1* != *val2*
file: *packet1*[*tot1*,*cnt1*].*field1*=*val1* != *packet2*[*tot2*,*cnt2*].*field2*=*val2*

where *val1* is a (possibly signed) decimal integer, and *val2* is an integer or an enumeration from *acis_h/interface.h*. As with the out-of-range messages described in §5.1, these are merely warnings, and may be suppressed by the `-q` flag. ***psci*** also examines the `irigB` and

TABLE 14. Extended Vanderspek (ERV) record format

```
typedef struct {
  unsigned short expnum;            /* exposure number */
  unsigned short exposure;          /* exposure time (msec) */
  unsigned long  irigtime;          /* IRIG timestamp */
  unsigned short nodenum;           /* output node index */
  unsigned short col;               /* column index */
  unsigned short row;               /* row index */
  unsigned short data[9];           /* event data values */
  short          doclk;             /* delta overclock */
} RvRec;
```

`bepSciTime` fields of all `scienceFramePseudo` packets. If these are accurate, it will calculate approximate `irigtime` fields for the ERV records, but the algorithm makes three assumptions—(a) that the BEP timer runs at precisely 100 kHz, (b) that all exposures in timed exposure mode have the same duration, and (c) that until ***psci*** has received a pair of `fepTimestamp` fields, it will approximate the exposure repetition interval by the `primaryExposure` field of `loadTeBlock`

All event-finding modes generate ERV files in the same format, but the use of the 9 fields in the `data` array differs according to mode. In CC and TE graded modes, the event amplitude is recorded in `data[0]` and the grade code in `data[1]`; in graded TE mode, `data[2]` contains the mean of the 4 corner pixel amplitudes; the remaining graded-mode `data` fields contain zeroes. In faint modes, the central pixel value is reported in `data[4]`, with the neighboring pixels from the same CCD row in `data[3]` and `data[5]`. In 1x3 faint mode, the remaining 6 fields contain zeroes; in 3x3 faint modes, `data[0]` through `data[2]` contain the pixel values from the previous row, and `data[6]` through `data[8]` those from the following row. CCD column and row numbers are defined from the lower left corner of the CCD.

TABLE 15. A sample ERV event file in ASCII format

n	msec	irig	nq	col	row	pixels									Δoclk
0	2800	80856262	2	566	1	191	1057	178	217	1406	168	174	180	176	0
0	2800	80856262	0	192	2	197	181	201	175	1300	661	188	760	385	0
0	2800	80856262	1	352	2	187	213	217	161	1453	1143	178	170	177	0
0	2800	80856262	2	549	2	182	387	164	173	2094	177	169	155	165	0
0	2800	80856262	2	650	2	175	176	174	177	1599	891	181	177	181	0
0	2800	80856262	2	743	2	172	177	185	177	1292	185	160	1179	217	0
0	2800	80856262	3	1017	2	227	602	619	211	1014	936	213	211	209	0
0	2800	80856262	0	102	3	183	196	185	179	2549	184	180	180	184	0
0	2800	80856262	0	224	3	185	175	175	167	2517	192	181	173	187	0
0	2800	80856262	3	949	3	215	223	205	204	2431	205	205	217	214	0
...															

In most event modes, *psci* will report the raw pixel values, but in Faint-with-Bias mode, the corresponding bias values are available. Even then, the subtraction will be made only if the `-B` flag is specified on the *psci* command line, in which case the appropriate `"biasValues+deltaOverclocks"` is subtracted from each valid pixel value. Invalid pixels, *i.e.* those with bias values exceeding 4093, will be assigned a value of -32768.

If the `-a` flag is specified, ***psci*** writes ERV files in ASCII format to *"name.s.n.m.erv.txt"*, as shown in Table 15. A tally is kept of all exposures, events, and bias parity errors for checking against the contents of the `scienceReport` packet that should terminate the science run. Additional informatory messages will be written to *stderr* if ***psci*** is invoked with the `-v` flag, *e.g.*

```
# psci -a -l test1 -v dat.1
dat.1: start TE EV3x3 FAINTBIAS bep 0x52ccbf71 irig 935:72262 exptime 3.207
test1.2.fits: bias file written, 2100032 bytes
test1.2.erv: written 13 exposures 25127 events
dat.1: scienceReport[909,0] irig 935:72301 exp 13 fep ok ccd ok dea 0 bep 0
test1.packet.log: 456 bytes written
test1.command.log: 1570 bytes written
test1.science.log: 80351 bytes written
test1.bias.log: 371820 bytes written
psci: 930 packets read from dat.1
```

5.5 Event Frame Timestamp Files

When the `-T` flag is specified on the *psci* command line, the times of each event-mode exposure frame present in the input stream are written to the ASCII file *"name.s.time.txt"*. This consists of the quantities shown in Table 17. The timestamp file is only generated during event-mode science runs, i.e. not during raw or histogram mode. The `BepTime` value should be constant within a given file. Frames will only be listed if at least one FEP processed that `exposureNumber`. Since the external (IRIG) time will drift relative to the ~100 kHz pixel clock reported in `BepTime` and `FepTime`, the drift can be estimated by comparing BEP and IRIG timestamps within science header pseudopackets. This drift is reported in the `dIrig0` field.

TABLE 16. Contents of an event frame timestamp file

Field Name	Description
Exp	ACIS event-mode exposure frame number
BepTime	BEP start-of-run timestamp
FepTime	FEP timestamp for this frame
dFEP	FEP timestamp increment since the last frame
dFrame	Length of a frame in pixel clock units
irigTime	IRIG time (UTC) of this exposure
dIrig	Frame-to-frame time in seconds
dIrig0	Drift between IRIG and ACIS clock in seconds

5.6 Histogram Files

Although ACIS writes histograms for each CCD output node at a time, ***psci*** saves the data packets until all expected nodes have been received, and writes a binary file in FITS format (see the left hand column of Table 18) named "*name.s.n.i-j*.hist.fits" for each contributing FEP. If the -a flag is specified, this will be an ASCII file named "*name.s.n.i-j*.hist.txt", consisting of 6 header lines followed by 4096 lines of 5 columns each containing, respectively, the pixel value and one pixel count for each output node A through D (see Table 17). If the CCDs are run with restricted output nodes (i.e. QUAD_AC or QUAD_BD), the unused columns are filled with zeroes.

TABLE 17. Contents of an ASCII histogram file

```
! histogram of fep n for n exposures n to n
!
! minimumOverclock        valA   valB   valC   valD
! maximumOverclock        valA   valB   valC   valD
! meanOverclock           valA   valB   valC   valD
! varianceOverclockLow    valA   valB   valC   valD
! varianceOverclockHigh   valA   valB   valC   valD
!
000000    histA   histB   histC   HistD
000001    histA   histB   histC   HistD
...
```

5.7 Raw Mode

The contents of raw data packets are written to disk files in FITS format as soon as they are received. Since the data packets always precede the exposure packet that describes them, the output file is first named "*name.s.n.*TMP.fits", and renamed "*name.s.n.m.raw.*fits" as soon as the exposure number "*m*" is known. For greater efficiency, ***psci*** uses memory mapping (the *mmap*(2) system call) to write raw-mode and bias files. As a consequence, the "ls -l" command will indicate that these files contain at least 2 Mbytes (1 Mbyte in continuous clocking mode), but the actual disk allocation, *e.g.* from the "du -a" command, will gradually increase as the data packets are received. Care must be taken to allow for this extra disk space when ***psci*** is receiving raw data and bias maps.

The headers of raw-mode FITS files will contain the fields shown in the center column of Table 18. Overclocks are appended to each line of the image array. The data area is always 1024×1024 pixels in timed-exposure mode and 1024×512 pixels in continuous clocking mode. ***psci*** will examine the dumpedTeBlock or dumpedCcBlock packet to see whether the CCD was run in a pixel-summing mode, or in sub-array readout mode, and will replicate and shift pixels accordingly to recreate a FITS file that reconstructs the original CCD geometry.

TABLE 18. Examples of FITS file headers

Binary Histograms	Raw Mode with Overclocks	Bias Images
SIMPLE = T BITPIX = 32 NAXIS = 2 NAXIS1 = 4096 NAXIS2 = 4 NFEP = 3 NCCD = 5 CCDROW1 = 1 CCDNROWS= 1024 CCDNODES= 4 QUADMODE= 'QUAD_FULL' DEAGAIN = 125 SUM2X2 = 'NO' EXPOTIM1= 30 EXPOTIM2= 0 DUTYCYCL= 0 FIRSTEXP= 71 LASTEXP = 75 NEXP = 5 FILENAME= '...' DATETIME= '...' OCLKMINA= 181 OCLKMAXA= 186 OCLKMEAA= 184 OCLKVARA= 8 OCLKVAHA= 0 ... END	SIMPLE = T BITPIX = 16 NAXIS = 2 NAXIS1 = 1088 NAXIS2 = 1024 NFEP = 2 NCCD = 5 CCDROW1 = 1 CCDNROWS= 1024 CCDNCOLS= 1024 CCDOCLKS= 64 CCDNODES= 4 QUADMODE= 'QUAD_FULL' DEAGAIN = 123 SUM2X2 = 'NO' EXPOTIM1= 28 EXPOTIM2= 0 DUTYCYCL= 0 EXPOSURE= 21 FILENAME= '...' DATETIME= '...' END	SIMPLE = T BITPIX = 1 NAXIS = 2 NAXIS1 = 1024 NAXIS2 = 1024 NFEP = 2 NCCD = 5 CCDROW1 = 1 CCDNROWS= 1 CCDNODES= 4 QUADMODE= 'QUAD_FULL' ACISMODE= 'TE' SUM2X2 = 'NO' DEAGAIN = 75 BIASALGO= 1 BIASARG0= 0 BIASARG1= 10 BIASARG2= 0 BIASARG3= 100 BIASARG4= 70 FILENAME= '...' DATETIME= '...' INITOCLA= 180 INITOCLB= 184 INITOCLC= 181 INITOCLD= 184 END

5.8 Bias Files

These are created in the same manner as raw pixel images, i.e. they are memory-mapped into pre-allocated files of fixed length. The FITS file header format is shown in the right-hand column of Table 18. Timed exposure bias files always contain 2100032 bytes (2880 header bytes followed by 1024×1024 2-byte image pixels). As in raw mode, ***psci*** replicates and shifts the pixels to fill the entire 1024×1024 pixel array. Continuously clocked bias maps contain 1051456 bytes—the single row reported by the BEP is replicated 512 times.

5.9 Memory Readout

When the `-l` flag is specified, ***psci*** copies the contents of all memory readout packets to disk files named *"name.pkt.n.dat"*, where *"name"* is the prefix specified on the command line, *"pkt"* is the type of memory readout, and *"n"* is an index that increments whenever a packet, or group[4] of packets of this type is encountered. They are written in the native byte order of the host

machine. When the –a flag is also present, the files are written in hexadecimal notation similar to the output of the "od -X" command (or "od -x" for the 16-bit SRAM and PRAM dumps), and thy will be named "*name.pkt.n*.dat".

TABLE 19. ASCII dump of a Huffman block containing multiple tables

```
huffmanTable[0] = {
    tabid = 0xffffffff
   lowlim = 3837
  tabsize = 512
    trunc = 07 3e0000 1111100
  badbias = 21 184df8 111111011001000011000
   badpix = 21 004df8 111111011001000000000
     -256 = 16 43bf00 1111110111000010
     -255 = 17 87ef80 11111011111100001
     -254 = 18 8897c0 111110100100010001
     -253 = 19 d737e0 1111110110011101011
...
      252 = 19 0937e0 1111110110010010000
      253 = 18 f6efc0 111110111101101111
      254 = 20 a89bf0 11111101100100010101
      255 = 19 e937e0 1111110110010010111
}
huffmanTable[1] = {
    tabid = 0xfffffffe
   lowlim = 3837
  tabsize = 512
    trunc = 06 280000 010100
  badbias = 21 497af8 111110101111010010010
   badpix = 18 c557c0 111110101010100011
     -256 = 20 eaf5f0 11111010111101010111
     -255 = 19 bc77a0 10111101111000111101
     -254 = 19 9c77a0 10111101111000111001
     -253 = 21 c17af8 111110101111010000011
...
}
```

5.10 Huffman Tables

The compression tables used by the BEP to compress the contents of raw science and bias data packets are treated by ***psci*** as special cases of memory readout. Since the Huffman table block is too long to dump in a single packet, ***psci*** assembles it from several `dumpedHuffman` packets and, when it is complete, saves it to decompress any subsequent raw or bias packets. If the "–l" option is used, the Huffman block will also be written to a disk file. Since Huffman blocks will not always be part of the packet stream, ***psci*** can be told, via the "–h" option, to pre-load a previously

4. A series of packets generated from a single memory readout command when the requested length exceeded the maximum permitted telemetry packet size. Such packet groups share the same `commandId` field value.

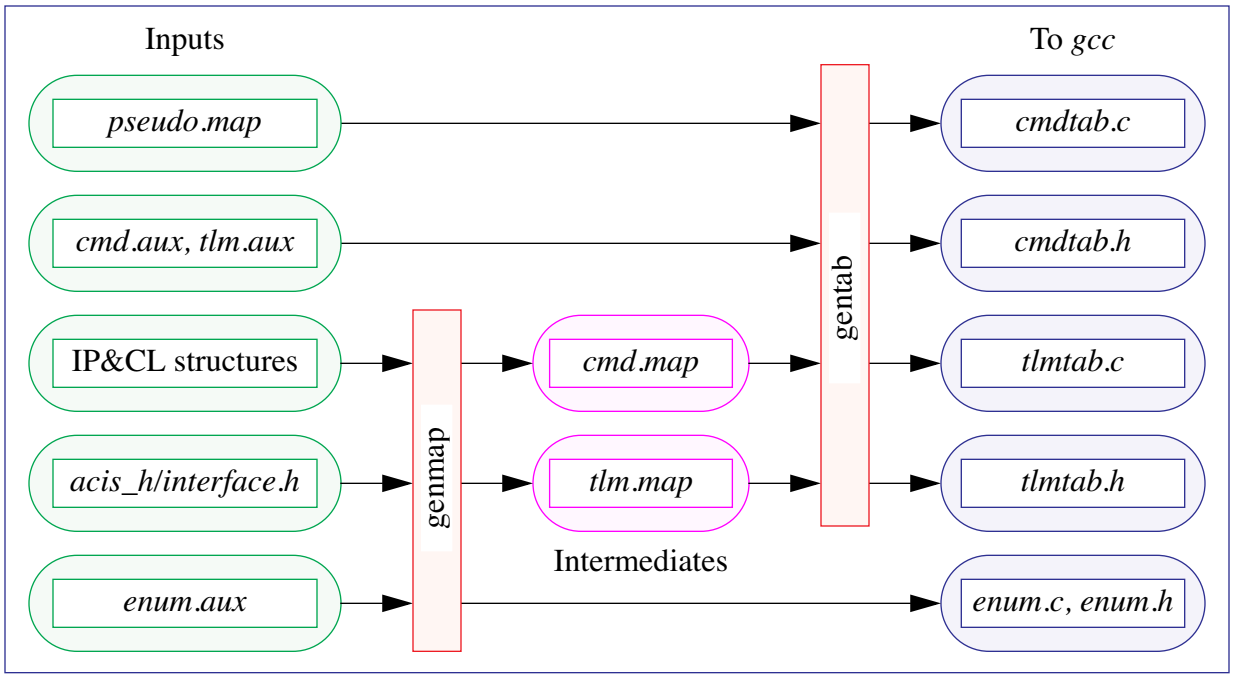
saved binary Huffman block, but this will always be supplanted by a complete Huffman block in the packet stream.

When the `"–a"` flag is used, Huffman blocks will be written to disk in ASCII format. These cannot be loaded by the `"–h"` option, but they are more readable, *e.g.* the example in Table 19.

5.11 Pseudopackets

If the `"–p"` flag is specified, these are written to `"name.pseudo.log"`. BEP & IRIG time fields are saved from `scienceFramePseudo` packets, as outlined in §5.4. above. Also, when ***psci*** is invoked with the `"-u"` flag, `userPseudo` packets with `"type"` field values of zero are interpreted as diagnostic messages from the flight software and the remainder of the packets, assumed to contain null-terminated character strings, are written to *stderr*. Otherwise, pseudopackets are ignored.

FIGURE 3. *psci* build architecture



5.12 Architecture

With the exception of the external Huffman tables discussed in §5.10, ***psci*** is self-contained—all packet and field names and formats are compiled into the executable module. The code is generated in three stages In the first, a Perl script named *genmap* reads the IP&CL structures file and creates intermediate *cmd.map* and *tlm.map* files which define the field formats and minimum and maximum values. *genmap* also extracts the enumeration values from *acis_h/interface.h* and writes them to *enum.c* as character string arrays.Then *gentab*, a second Perl script, converts *cmd.map* and *tlm.map* into C structures in *cmdtab.c* and *cmdtab.h*. Finally, *gcc* compiles the structures with the remaining source code. While the process is automatic, it can be guided by

directives in the three files *enum.aux*, *cmd.aux*, and *tlm.aux*. An example of *enum.aux* is shown in Table 20.—lines of the form `"command id name"` assign `"name"` to command packets with that `"id"`, and `"packet id name"` performs the same function for telemetry packets. This causes ***psci*** to use the new names within messages and in log and monitor files, as well as giving symbolic names to pseudopackets, since these were never described by the IP&CL.

Examples of the *cmd.aux* and *tlm.aux* files are shown in Table 21 and Table 22. They share a common syntax. The `"enumerate"` command assigns a character string array in *enum.c* to a particular command or telemetry packet field. ***psci*** will display the enumerated value in its log files and error messages. The `"nolog"` directive prevents ***psci*** from logging a particular type of command, or packet, or field. The `"nomonitor"` command, which is only used in *tlm.aux*, performs the same function—of eliminating fields from monitor packets. The more restrictive `"nomonitornull"` command prevents fields from being written to the monitor stream if they have zero value.

TABLE 20. Sample *enum.aux* file

```
#
# resolve duplicated command packet names
#
command CMDOP_START_TE         startTe
command CMDOP_BIAS_TE          startTeBias
command CMDOP_START_CC         startCc
command CMDOP_BIAS_CC          startCcBias
command CMDOP_ADD_BAD_TE_COL   addBadTeColumn
command CMDOP_RESET_BAD_TE_COL resetBadTeColumnMap
command CMDOP_ADD_BAD_CC_COL   addBadCcColumn
command CMDOP_RESET_BAD_CC_COL resetBadCcColumnMap
command CMDOP_DUMP_TE_SLOTS    dumpTeSlots
command CMDOP_DUMP_CC_SLOTS    dumpCcSlots
command CMDOP_DUMP_2D_SLOTS    dump2dSlots
command CMDOP_DUMP_1D_SLOTS    dump1dSlots
command CMDOP_DUMP_DEA_SLOTS   dumpDeaSlots
#
```

TABLE 20. Sample *enum.aux* file (Continued)

```
# resolve duplicated telemetry packet names
#
packet TTAG_DUMP_SYS_CONFIG    dumpedSysConfig
packet TTAG_DUMP_BAD_PIXEL     dumpedBadPix
packet TTAG_DUMP_BAD_TE_COL    dumpedBadTeCol
packet TTAG_DUMP_BAD_CC_COL    dumpedBadCcCol
packet TTAG_DUMP_PATCHES       dumpedPatches
packet TTAG_DUMP_HUFFMAN       dumpedHuffman
packet TTAG_DUMP_TE_SLOTS      dumpedTeSlots
packet TTAG_DUMP_CC_SLOTS      dumpedCcSlots
packet TTAG_DUMP_2D_SLOTS      dumped2dSlots
packet TTAG_DUMP_1D_SLOTS      dumped1dSlots
packet TTAG_DUMP_DEA_SLOTS     dumpedDeaSlots
packet TTAG_SCI_CC_REC_FAINT   exposureCcFaint
packet TTAG_SCI_CC_REC_GRADED  exposureCcGraded
packet TTAG_SCI_TE_REC_FAINT   exposureTeFaint
packet TTAG_SCI_TE_REC_GRADED  exposureTeGraded
#
# name the pseudopackets
#
packet 61 engineeringPseudo
packet 62 scienceFramePseudo
packet 63 userPseudo
```

TABLE 21. An example of *tlm.aux*

```
#
#  Use the following enumerations in log files
#
enumerate       formatTag          TlmFormatTagStr
enumerate       ccdId              CcdIdStr
enumerate       fepId              FepIdStr
enumerate       lastFatalCode      FatalCodeStr
enumerate       commandOpcode      CmdOpcodeStr
enumerate       result             CmdResultStr
enumerate       fepErrorCodes      FepIoErrorsStr
enumerate       query              DeaQueryCntlIdStr
enumerate       queryId            DeaQueryCcdIdStr
enumerate       fatalCode          FatalCodeStr
#
# Omit the following fields from log records
#
nolog           commandEcho.commandBody
nolog           scienceReport.ccdError
nolog           dumpedTeBlock.parameterBlockData
```

TABLE 21. An example of *tlm.aux* (Continued)

```
#
# Omit the following fields from monitor records
#
nomonitor       userPseudo
nomonitor       engineeringPseudo
nomonitor       scienceFramePseudo
nomonitor       telemetryLength
nomonitor       formatTag
nomonitor       dataTeBiasMap.compressionTableSlotIndex
nomonitor       dataTeBiasMap.compressionTableIdentifier
nomonitor       dataTeBiasMap.biasStartTime
nomonitor       dataTeBiasMap.biasParameterId
nomonitor       commandEcho.commandLength
#
# Omit the following when null
#
nomonitornull   biasParityErrors
```

TABLE 22. An example of *cmd.aux*

```
#
#  Use the following enumerations in log files
#
enumerate    commandOpcode               CmdOpcodeStr
enumerate    ccdId                       CcdIdStr
enumerate    itemId                      SystemSettingsStr
enumerate    fepId                       FepIdStr
enumerate    fepCcdSelect                CcdIdStr
enumerate    queryId                     DeaQueryCcdIdStr
enumerate    outputRegisterMode          QuadModeStr
#
# Enumerations specific to particular command blocks
#
enumerate    loadCcBlock.fepMode         CcFepModeStr
enumerate    loadTeBlock.fepMode         TeFepModeStr
enumerate    loadCcBlock.bepPackingMode  CcBepModeStr
enumerate    loadTeBlock.bepPackingMode  TeBepModeStr
#
# Remove redundant fields from log output
#
nolog        loadTeBlock.gradeSelectValue
nolog        loadTeBlock.videoResponse
nolog        loadCcBlock.gradeSelectValue
nolog        loadCcBlock.videoResponse
```

5.13 Tests applied to packet fields

Table 23 details the tests applied within the particular packet-handling modules. These are in addition to the range tests described in §5.1, which themselves verify the following fields:

- In all command blocks: `commandLength, commandOpcode`
- In all telemetry packets: `synch, telemetryLength, formatTag`

A number of other consistency checks are made in the process of creating the output data files, and may cause warning messages to appear on *stderr*. For instance, the `ccdRow` and `ccdRowCount` fields in `dataTeBiasMap` packets are verified against the sub-array fields in `loadTeBlock`. The warnings are usually self-explanatory; the user should consult the source code for further details.

Considerate la vostra semenza:
fatti non foste a viver come bruti,
ma per seguir virtute e canoscenza.

TABLE 23. Tests applied to individual ACIS packet fields

Packet	Field	Checked against	Module
bepReadReply dumped*Slots	commandId	last packet of same type within multi-packet group	*memory.c*
	readAddress		
	requestedAddress		
	requestedWordCount		
dataCcBiasMap	biasParameterId	loadCcBlock.parameterBlockId	*bias.c*
	ccdId	loadCcBlock.fepCcdSelect	
	fepId		
dataCcFaint dataCcGraded	ccdId	exposureCcFaint.ccdId	*event.c*
		loadCcBlock.fepCcdSelect	
	dataPacketNumber	running packet count	
	fepId	exposureCcFaint.fepId	
dataCcRaw	ccdId	last packet of same type from this FEP	*raw.c*
		loadCcBlock.fepCcdSelect	
	compressionTableIdentifier	last packet of same type from this FEP	
	compressionTableSlotIndex	loadCcBlockk.rawCompression-SlotIndex	
		last packet of same type from this FEP	
	dataPacketNumber	running packet count	
	fepId	last packet of same type from this FEP	
	pixelCount	loadCcBlock fields and pixel count	
dataTeBiasMap	biasParameterId	last packet of same type from this FEP	*bias.c*
		loadTeBlock.parameterBlockId	
	biasStartTime	last packet of same type from this FEP	
	ccdId	loadTeBlock.fepCcdSelect	
		last packet of same type from this FEP	
	ccdRow	compatibility with loadTeBlock fields	
	ccdRowCount		
	compressionTableIdentifier	last packet of same type from this FEP	
	compressionTableSlotIndex	loadTeBlock.rawCompression-SlotIndex	
		last packet of same type from this FEP	
	dataPacketNumber	running packet count	
	fepId	last packet of same type from this FEP	
		loadTeBlock.fepCcdSelect	
	pixelsPerRow	pixel count	
		loadTeBlock.onChip2x2Summing	
dataTeBiasMap (contd.)	rowsPerBias	loadTeBlock.subarrayRowCount	*bias.c*

TABLE 23. Tests applied to individual ACIS packet fields (Continued)

Packet	Field	Checked against	Module
dataTeFaint dataTeGraded	ccdId	exposureTeFaint.ccdId	*event.c*
	dataPacketNumber	packet count	
	fepId	exposureTeFaint.fepId	
dataTeFaintBias	ccdId	exposureTeFaintBias.ccdId	*event.c*
		loadTeBlock.fepCcdSelect	
	dataPacketNumber	packet count	
	fepId	exposureTeFaintBias.fepId	
dataTeHist	ccdId	exposureTeHistogram.ccdId	*hist.c*
		loadTeBlock.fepCcdSelect	
	dataPacketNumber	packet count	
	fepId	exposureTeHistogram.fepId	
	outputNodeId	exposureTeHistogram.output-NodeId	
	startingBin	packet count	
dataTeRaw	ccdId	last packet of same type from this FEP	*raw.c*
		loadTeBlock.fepCcdSelect	
	compressionTableIdentifier	last packet of same type from this FEP	
	compressionTableSlotIndex	loadTeBlock.rawCompression-SlotIndex	
		last packet of same type from this FEP	
	dataPacketNumber	packet count	
	fepId	last packet of same type from this FEP	
exposureCcFaint	-	loadCcBlock.bepPackingMode	*event.c*
		loadCcBlock.commandOpcode	
		loadCcBlock.fepMode	
	biasParameterId	last packet of same type from this FEP	
	biasStartTime	last packet of same type from this FEP	
		scienceReport.runStartTime	
	ccdId	last packet of same type from this FEP	
		dataCcFaint.ccdId	
		dataCcGraded.ccdId	
		loadCcBlock.fepCcdSelect	
	eventsSent	event count	
	exposureNumber	> last packet of same type from this FEP	
	fepId	last packet of same type from this FEP	
		dataCcFaint.fepId	

TABLE 23. Tests applied to individual ACIS packet fields (Continued)

Packet	Field	Checked against	Module
exposureCcFaint (contd.)	parameterBlockId	last packet of same type from this FEP	*event.c*
		loadCcBlock.parameterBlockId	
	runStartTime	last packet of same type from this FEP	
		scienceReport.runStartTime	
	windowBlockId	last packet of same type from this FEP	
		load1dBlock.windowBlockId	
exposureCcRaw	-	loadCcBlock.commandOpcode	*raw.c*
		loadCcBlock.fepMode	
	ccdId	last packet of same type from this FEP	
		loadCcBlock.fepCcdSelect	
	exposureNumber	> last packet of same type from this FEP	
	fepId	last packet of same type from this FEP	
	parameterBlockId	last packet of same type from this FEP	
		scienceReport.parameter-BlockId	
	runStartTime	last packet of same type from this FEP	
		scienceReport.runStartTime	
	windowBlockId	last packet of same type from this FEP	
		scienceReport.windowBlockId	
exposureTeFaint exposureTeFaintBias	-	loadTeBlock.bepPackingMode	*event.c*
		loadTeBlock.commandOpcode	
		loadTeBlock.fepMode	
		scienceReport.exposuresSent	
	biasParameterId	last packet of same type from this FEP	
	biasStartTime	last packet of same type from this FEP	
		scienceReport.biasStartTime	
	ccdId	last packet of same type from this FEP	
		dataTeFaint.ccdId	
		dataTeGraded.ccdId	
		loadTeBlock.fepCcdSelect	
	eventsSent	event count	
	exposureNumber	> last packet of same type from this FEP	

TABLE 23. Tests applied to individual ACIS packet fields (Continued)

Packet	Field	Checked against	Module
exposureTeFaint exposureTeFaintBias (contd.)	fepId	last packet of same type from this FEP	*event.c*
		dataTeFaint.fepId	
		dataTeGraded.fepId	
	parameterBlockId	last packet of same type from this FEP	
		loadTeBlock.parameterBlockId	
		scienceReport.parameter-BlockId	
	runStartTime	last packet of same type from this FEP	
		scienceReport.runStartTime	
	windowBlockId	last packet of same type from this FEP	
		load2dBlock.windowBlockId	
		scienceReport.windowBlockId	
exposureTeHistogram	-	loadTeBlock.commandOpcode	*hist.c*
		loadTeBlock.fepMode	
	ccdId	last packet of same type from this FEP	
		dataTeHist.ccdId	
		loadTeBlock.fepCcdSelect	
	endExposureNumber	last packet of same type from this FEP	
	exposureCount	scienceReport.exposuresSent	
	fepId	last packet of same type from this FEP	
		dataTeHist.fepId	
	outputNodeId	dataTeHist.outputNodeId	
	parameterBlockId	last packet of same type from this FEP	
		loadTeBlock.parameterBlockId	
		scienceReport.parameter-BlockId	
	runStartTime	last packet of same type from this FEP	
		scienceReport.runStartTime	
	startExposureNumber	last packet of same type from this FEP	
		> last packet of same type from this FEP	

TABLE 23. Tests applied to individual ACIS packet fields (Continued)

Packet	Field	Checked against	Module
exposureTeRaw	–	loadTeBlock.commandOpcode	raw.c
		loadTeBlock.fepMode	
		scienceReport.exposuresSent	
	ccdId	last packet of same type from this FEP	
		loadTeBlock.fepCcdSelect	
	exposureNumber	> last packet of same type from this FEP	
	fepId	last packet of same type from this FEP	
	parameterBlockId	last packet of same type from this FEP	
		scienceReport.parameter-BlockId	
	runStartTime	last packet of same type from this FEP	
		scienceReport.runStartTime	
	windowBlockId	last packet of same type from this FEP	
		scienceReport.windowBlockId	
fepReadReply	commandId	last packet of same type within multi-packet group from this FEP	memory.c
	fepId		
	requestedAddress		
	requestedWordCount		
load1dBlock	commandOpcode	CMDOP_LOAD_1D	bias.c event.c raw.c window.c
	windowBlockId	exposureCcFaint.window-BlockId	event.c
		exposureCcRaw.windowBlockId	raw.c
load2dBlock	commandOpcode	CMDOP_LOAD_2D	bias.c event.c raw.c window.c
	windowBlockId	exposureTeFaint.window-BlockId	event.c
		exposureTeFaintBias.window-BlockId	
		exposureTeRaw.windowBlockId	raw.c
loadCcBlock	bepPackingMode	BEP_CC_MODE_FAINT	event.c
		BEP_CC_MODE_GRADED	

TABLE 23. Tests applied to individual ACIS packet fields (Continued)

Packet	Field	Checked against	Module
loadCcBlock (contd.)	commandOpcode	CMDOP_LOAD_CC	*bias.c* *event.c* *raw.c*
	fepMode	FEP_CC_MODE_EV1x3	*event.c*
		FEP_CC_MODE_RAW	*raw.c*
	parameterBlockId	dataCcBiasMap.biasParameterId	*bias.c*
		exposureCcFaint.parameterBlockId	*event.c*
loadTeBlock	bepPackingMode	BEP_TE_MODE_FAINT	*event.c*
		BEP_TE_MODE_FAINTBIAS	
		BEP_TE_MODE_GRADED	
	commandOpcode	CMDOP_LOAD_TE	*bias.c* *event.c* *hist.c* *raw.c*
	fepMode	FEP_TE_MODE_EV3x3	*event.c*
		FEP_TE_MODE_HIST	*hist.c*
		FEP_TE_MODE_RAW	*raw.c*
	parameterBlockId	dataTeBiasMap.biasParameterId	*bias.c*
		exposureTeFaint.parameterBlockId	*event.c*
		exposureTeFaintBias.parameterBlockId	
		exposureTeHistogram.parameterBlockId	*hist.c*
	subarrayRowCount	dataTeBiasMap.rowsPerBias	*bias.c*
pramReadReply sramReadReply	ccdId	last packet of same type within multi-packet group from this DEA	*memory.c*
	commandId		
	requestedIndex		
	requestedWordCount		
scienceReport	biasErrorCount	bias error count	*science.c*
	biasParameterId	value in previous exposure record(s)	
	biasStartTime		
	exposuresProduced	exposure counters	
	exposuresSent		
	parameterBlockId	value in previous exposure record(s)	
	runStartTime		
	windowBlockId		

6.0 Simulated ACIS Telemetry

This chapter describes how to generate a simulated ACIS timed-exposure telemetry stream. It proceeds in two steps. First, the FEP simulator reads commands and input data in the form of 16-bit FITS images, creating one or more “ring-buffer” files and bias maps. In the second step, these files are used to produce the telemetry stream.

6.1 fepCtlTest—simulate the ACIS front-end processor

The simulator can read a command script from its standard input stream, *stdin*, or behave like a command shell, as in the following example of a timed-exposure science run. The first group of `“set”` commands specifies the location of the input FITS image files, and the location of pixels and overclocks within those images. The second group uses `“param”` commands to define FEP-to-BEP run-time parameters, followed by an `“exec”` command to store them in the FEP, another to compute the bias map, and a `“dumpbias”` command to copy the bias map to a disk file. The third group uses the `“output”` command to direct subsequent output to a disk file, and then executes the simulated timed-exposure run.

```
#! /acis/h3/tools/bin/fepCtlTest

# Test images use CCD image frames with fe55 and co60

set input = /cdrom/ccid17-38-3/fe55co60/fe55+Co60_nowin_120.%04d.fits
set rows = 2,1025
set pixels = 4,259,344,599,684,939,1024,1279
set overclocks = 260,336,600,676,940,1016,1280,1356

# fepTimedBias() full-frame bias, 4 nodes

param type       = FEP_TIMED_PARM_3x3
param nrows      = 1024
param ncols      = 256
param quadcode   = FEP_QUAD_ABCD
param noclk      = 16
param nhist      = 0
param btype      = FEP_BIAS_1
param thresh[0]  = 100
param thresh[1]  = 100
param thresh[2]  = 100
param thresh[3]  = 100
param bparm[0]   = 5
param bparm[1]   = 10
param bparm[2]   = 0
param bparm[3]   = 100
param bparm[4]   = 70
param nskip      = 0
param initskip   = 2

exec BEP_FEP_CMD_PARAM
exec BEP_FEP_CMD_BIAS
```

```
dumpbias feco.bias.te.fframe

# fepSciTimed() in 3x3 event mode
set output = feco.ring.te.3x3
exec BEP_FEP_CMD_TIMED
```

This script causes two files to be written—`"feco.bias.te.fframe"` to contain the bias map and `"feco.ring.te.3x3"` to contain the events. The former can be viewed by any FITS image reader, the latter by the ***dumpring*** command (see Section 6.2).

TABLE 24. fepCtlTest Command Syntax

Command	Description
`dumpbias` *file*	write an already-computed bias map to *file* in FITS format
`exec` *cmd*	execute a BEP-to-FEP command—one of the following `FEP_BEP_CMD_BIAS` start bias calibration `BEP_BEP_FEP_CMD_TIMED` start timed exposure `BEP_FEP_CMD_CCLK` start continuous clocking `BEP_FEP_CMD_STOP` stop the run `BEP_FEP_CMD_PARAM` read parameters `BEP_FEP_CMD_SUSPEND` temporarily halt `BEP_FEP_CMD_RESUME` resume after suspension `BEP_FEP_CMD_STATUS` return FEP status `BEP_FEP_CMD_FIDPIX` define fiducial pixels[1]
`fidpix =` *row col...*	define one or more fiducial pixels[1]

`param` *name* `=` *val*	set a BEP-to-FEP parameter—one of the following[2]
	`type` type of parameter block:
	`FEP_NO_PARM`
	`FEP_TIMED_PARM_RAW`
	`FEP_TIMED_PARM_HIST`
	`FEP_TIMED_PARM_3x3`
	`FEP_TIMED_PARM_5x5`
	`FEP_CCLK_PARM_RAW`
	`FEP_CCLK_PARM_1x3`
	`nrows` number of rows in a CCD frame
	`ncols` number of pixels per row per node
	`quadcode` node clocking code:
	`FEP_QUAD_ABCD`
	`FEP_QUAD_AC`
	`FEP_QUAD_BD`
	`noclk` number of overclocks per row per node
	`nhist` number of frames per histogram
	`btype` type of bias calibration to perform
	`FEP_NO_BIAS`
	`FEP_BIAS_1`
	`FEP_BIAS_2`
	`thresh[4]` event detection thresholds for each node
	`bparm[5]` bias calibration parameters
	`nskip` timed exposure skip factor

TABLE 24. fepCtlTest Command Syntax (Continued)

Command	Description
`reset ctr`	reset an internal simulator counter. Currently, the only counter recognized is `"wakeup"`.
`set buf[row,col] = val`	set an element of the 2-dimensional `buf` array. `buf` is either `"bias"` or `"biasparity"`
`set input = file`	subsequent bias calibrations or science runs will read pixel data from `file`, a set of FITS images whose name should contain a numeric `"printf"` format string, e.g. `"%04d"` which will be replaced by the integers `1,2,3,...` when the FEP simulator opens successive input FITS files.
`set output = file`	subsequent "ring buffer" output will be written to `file.` See Section 6.2 for details of ring-buffer formats
`set overclocks = p1,p2,p3,p4...`	specify the ranges of overclock pixels in each raster of the input file. Node A's overclocks will span columns `p1` through `p2` (indexing from 0), Node B's will span `p3` through `p4`, etc.
`set pixels = p1,p2,p3,p4,...`	specify the ranges of data pixels in each raster of the input file. Node A's pixels will span columns `p1` through `p2` (indexing from 0), Node B's will span `p3` through `p4`, etc.
`set rows = r1,r2`	specify the range of input image rows, indexed from 0. If omitted, the range will begin with the first row and will continue for the number of rows specified by the `"param nrows"` command.
`stuff param name = val`	used in unit and coverage testing to force a BEP parameter to a particular value, bypassing the normal range checking applied to a BEP_FEP_CMD_PARAM command.
`xor buf[row,col] = val`	perform an exclusive OR of `val` with an element of the 2-dimensional `buf` array, storing the result back into the array. `buf` is either `"bias"` or `"biasparity"`

1. this feature is not currently supported in the ACIS BEP detailed design, but has been included in the FEP software.

2. see Table 47 and Table 48 of the ACIS Flight Software Detailed Design Specification (36-53200 Rev. 01++) for more details of bias parameters

fepCtlTest assumes that the input FITS files contain 16-bit pixels in *big-endian* format (most-significant byte first). Only the least significant 12 bits of each pixel will be used. The bias maps will also be written in 16-bit FITS format, but in *little-endian* format (least-significant byte first), irrespective of whether ***fepCtlTest*** is compiled for little- or big-endian machines. This is not the case with the ring-buffer output files—their byte ordering will depend on the architecture of the machine running ***fepCtlTest***.

6.2 dumpring—display ring-buffer records

This is a ***perl*** script that converts the contents of a binary ring-buffer file to ASCII text. It recognizes only a single command-line argument, the name of the input file. If omitted, it

reads from *stdin*. The formats of the various ring-buffer records are defined in Section 4.10 of the ACIS Flight Software Detailed Design Specification (36-53200 Rev. 01++). ***dumpring*** inspects the first few records to determine the byte-order, little- or big-endian. It writes a formatted ASCII listing on *stdout*, as in the following example:

```
FEPexpRec[1] = {
   expnum      = 1
   timestamp   = 0x05234672
   bias0       = 180 184 181 184
   dOclk       = 0 0 0 0
}
FEPeventRec3x3[1,1] = {
   row         = 2
   col         = 1017
   p,b         = {  227  602  619 }  {  210  210  210 }
               = {  211 1014  936 }  {  210  210  204 }
               = {  213  211  209 }  {  209  209  203 }
}
FEPeventRec3x3[1,2] = {
   row         = 4
   col         = 448
   p,b         = {  180  166  174 }  {  166  164  164 }
               = {  190 1981  166 }  {  166  722  160 }
               = {  171  658  168 }  {  165  168  168 }
}
.
.
FEPexpEndRec[1] = {
   expnum      = 1
   thresholds  = 3765
   parityerrs  = 0
}
FEPexpRec[2] = {
   expnum      = 2
   timestamp   = 0x0540f48b
   bias0       = 180 184 181 184
   dOclk       = 0 0 0 0
}
.
.
```

dumpring numbers the records consecutively, beginning with 1[1]. Thus, in the above example, the start-of-exposure record `FEPexpRec[1]` is matched by the end-of-exposure record `FEPexpEndRec[1]`. In between, the event records are `FEPeventRec3x3[1,`*n*`]`, where the event index *n* also starts at 1.

1. note that the BEP numbers its exposures beginning with 0, i.e. it subtracts 1 from the exposure numbers reported by the FEPs, which use exposure number 0 to indicate their state before the first exposure has been received.

6.3 tlmsim—create simulated telemetry packets

This command combines a pair of files created by ***fepCtlTest***, a ring-buffer and a bias map, into a stream of ACIS test packets. In addition to these files, ***tlmsim*** must be provided with suitable parameter and window blocks to include in its output. It is described in detail in Section 10.31.

The output stream contains two sorts of packets—ACIS science packets and science frame pseudo-packets (SFPP). They share a common format, as described in Table 3 on page 6, but serve different functions. ***Remember that all ACIS telemetry fields, including those in pseudo-packets, are recorded in little-endian order, with less significant bytes preceding more significant.***

The science packets simulate the ACIS serial telemetry stream that would be written to the "science data" portion of each Format-2 AXAF telemetry frame, or into the next-in-line field in Format 1. Their format is given by Table 3 on page 6, and their contents are described in the ACIS IP&CL documents and in `"http://acis.mit.edu/acis/ipcl".` Inter-packet padding has been removed. However, the real ACIS instrument also inserts 4-byte time tags into its serial telemetry whenever it receives a science frame pulse. Since these tags could appear at any offset within a packet, it is inappropriate to include them in ***tlmsim***'s science packets. If users wishes to simulate them, they must specify the **–p** option, causing ***tlmsim*** to write them as pseudo-packets.

6.3.1 Bias Map

The size of the bias map is taken from the FITS header variables `NAXIS1` and `NAXIS2`. It is usually 1024 rows and 1024 columns. ***tlmsim*** formats it into `dataTeBiasMap` packets, two rows per packet. This version of ***tlmsim*** is unable to compress the 12-bit data, so it reports the `compressionTableSlotIndex` as 255 (no compression).

The `initialOverClocks` fields in the `dataTeBiasMap` packets are copied from the ring-buffer file, so it is important that the ring-buffer and bias map files should have been generated from the same ***fepCtlTest*** input data

6.3.2 Timing

The order of the science packets is as follows:

1	`bepStartupMessage`	simulating an ACIS commanded reset
1	`commandEcho`	executing a `loadTeBlock` command
1	`commandEcho`	executing a `startScience` command
1	`dumpedTeBlock`	reporting the current `teBlock` and `window2d` block
many	`dataTeFaintBias`	
many	`exposureTeFaintBias`	bias map packets interleaved with event and exposure packets
many	`dataTeBiasMap`	
1	`commandEcho`	executing a `stopScience` command

1	`scienceReport`	reporting the statistics of the run
-	`dataTeBiasMap`	remaining bias packets until the entire map is reported

If the **–p** option is specified, a SFPP will be written inserted once per science frame, i.e. every 2.05 seconds of "real" instrument time, determined by counting the length of the science packets and assuming that they are filling the available telemetry bandwidth (512 *bps* in Format-1 and 24 *kbps* in Format-2). ***tlmsim*** normally simulates Format-2. Use the **–f** flag to create (48 times as many!) Format-1 pseudo-packets.

An AXAF/ACIS telemetry stream contains 5 types of timing information, each of which is simulated independently

1. `fepTimestamp`—the 25-bit value of the FEP's megahertz counter which would be latched whenever the start of a new frame were received (`VSYNC` from PRAM). The simulated value is taken from the ring-buffer records created by ***fepCtlTest***, which currently uses the value of the UNIX microsecond timer for this field.
2. `bepTimestamp`—the 32-bit value of the BEP's megahertz counter which would be latched (a) whenever a S/C science pulse is received or (b) whenever the BEP sends a "start processors" command to the DEA(s). The former would be inserted synchronously into the telemetry, and is simulated by ***tlmsim*** as the `bepSciTime` field in the SFPP; the latter would be reported as `biasStartTime` and `runStartTime` in science packets. They are simulated as the instantaneous value of the UNIX microsecond timer, except that the `biasStartTime` is then arbitrarily decreased by 120 seconds.
3. `bepTickCounter`—the BEP also contains a 10 Hz counter whose value would be reported as the `bepTickCounter` field in the `bepStartupMessage` packet and the `arrival` field of `commandEcho` packets. ***tlmsim*** assigns an initial value of 500 to this counter, and increments it within subsequent packets according to the time elapsed as read from the UNIX microsecond timer.
4. `majorFrameId` and `minorFrameId`—these SFPP fields contain the VCDU fields that would occur at the start of each AXAF telemetry minor frame. They are simulated by keeping count of the telemetry mode (the **–f** flag) and of the data volume being written. `minorFrameId` is incremented for every 750 bytes of ACIS data (8 bytes in Format-1), and `majorFrameId` is incremented every 128 minor frames. Both are initialized to zero at the start of the simulation.
5. IRIG-B—the day and second count in the SFPPs are initialized from the UNIX system clock, and the milli- and micro-second timers are initialized to zero. They are incremented by 0.25625 seconds for each minor frame (750 bytes of ACIS data in Format-2, 8 bytes in Format-1).

6.3.3 Miscellaneous

The ***tlmsim*** output always assigns correct lengths to telemetry packets and to encapsulated commands and parameter blocks. It numbers the science packets sequentially from zero (in the `sequenceNumber` fields), and computes correct checksums. All commands are

executed successfully (`CMDRESULT_OK` in the result field of `commandEcho` packets), and all events in the ring-buffer file are copied to telemetry packets—none is removed by (simulated) BEP filters. The only errors that are simulated are instances of bias map corruption within the FEP. They are created by including `"xor parity"` or `"xor parityplane"` commands within the ***fepCtlTest*** script, and they result in errors being written to the ring-buffer file and thence to `dataBiasErr` packets in the ***tlmsim*** output stream.

6.4 Examples

In the simplest case, the output from ***tlmsim*** may be piped directly into ***ltlm***, e.g.

```
tlmsim -c te.1 -p -w win.1 ring.1 bias.1 | ltlm | more
```

If you only want to inspect packets of a particular type, e.g. `exposureTeFaintBias` whose `formatTag` value is 22, invoke ***ltlm*** with the ***-p 22*** option. If you also want to see `dataTeFaintBias` packets (`formatTag` 23), use ***-p22 -p23*** together. If you forget a `formatTag` value, invoke ***ltlm -lt*** with no other arguments to list them all.

```
ltlm -v -p22 -p2 tlm.out11 | more
dataTeFaintBias[0] = {
  synch                 = 0x736f4166
  telemetryLength       = 1021
  formatTag             = TTAG_SCI_TE_DAT_FAINTB (23)
  sequenceNumber        = 11
  ccdId                 = 6
  fepId                 = 2
  dataPacketNumber      = 0
  events                = [138]
}
.
.
.
dataTeFaintBias[36] = {
  synch                 = 0x736f4166
  telemetryLength       = 675
  formatTag             = TTAG_SCI_TE_DAT_FAINTB (23)
  sequenceNumber        = 237
  ccdId                 = 6
  fepId                 = 2
  dataPacketNumber      = 10
  events                = [91]
}
exposureTeFaintBias[2] = {
  synch                 = 0x736f4166
  telemetryLength       = 20
  formatTag             = TTAG_SCI_TE_REC_FAINTB (22)
  sequenceNumber        = 238
  runStartTime          = 0x619a9cf4
  parameterBlockId      = 0x00000fab
  windowBlockId         = 0x00000baf
```

```
  biasStartTime         = 0x5a75d31b
  biasParameterId       = 0x00000fab
  ccdId                 = 6
  fepId                 = 2
  fepTimestamp          = 0x015e661d
  exposureNumber        = 2
  eventsSent            = 1471
  thresholdPixels       = 3672
  discardEventAmplitude = 0
  discardWindow         = 0
  discardGrade          = 0
  deltaOverclocks       =    1    0    0    0
  biasParityErrors      = 0
  initialOverclocks     = 180 184 181 184
}
```

Note that, rather than listing the contents of a large array such as events, ***ltlm*** merely lists the number of elements within square brackets. If you wish to see the array values themselves, invoke ***ltlm*** with the ***-v*** flag, *viz.*

```
ltlm -v -p22 -p23 tlm.out11 | more
dataTeFaintBias[0] = {
  synch                 = 0x736f4166
  telemetryLength       = 1021
  formatTag             = TTAG_SCI_TE_DAT_FAINTB (23)
  sequenceNumber        = 11
  ccdId                 = 6
  fepId                 = 2
  dataPacketNumber      = 0
  events[0] = {
    ccdRow              = 2
    ccdColumn           = 1017
    pulseHeights        =  227  602  619  211 1014  936  213  211  209
    biasValues          =  210  210  210  210  210  204  209  209  203
  }
  events[1] = {
    ccdRow              = 4
    ccdColumn           = 448
    pulseHeights        =  180  166  174  190 1981  166  171  658  168
    biasValues          =  166  164  164  166  722  160  165  168  168
  }
  events[2] = {
    ccdRow              = 6
    ccdColumn           = 646
    pulseHeights        =  177  169  178  175 2525  172  176  177  171
    biasValues          =  170  167  167  169  165  165  173  168  168
```

7.0 ACIS Timing Algorithms

This section provides a guide for producing algorithms which map science run times to observatory time and hence to UTC and TDB. The BEP and FEP counters are driven by the BEP's internal clock which will drift relative to Spacecraft Event Time (SCET), *i.e.*, any counter that is driven by the spacecraft's ultra-stable 1024000 Hz oscillator, and this drift must be modeled in order to assign accurate SCETs to events within ACIS. In this section, ACIS telemetry packet fields will appear in `monotype`.

When ACIS receives a Science Frame Pulse from the spacecraft, it preserves the 32-bit value of its internal 100 kHz timer and *immediately* writes it to the serial telemetry stream, inserting it into any telemetry packet that it might be writing at the time. The first 4 bytes of ACIS science telemetry following a Science Frame Pulse will always contain this timestamp, called `refTime` in this section. Since Science Frame Pulses occur every 2.05 seconds, and can be related to UTC via the VCDU counters in the telemetry minor frame headers, it is possible to relate `refTime` itself, and hence any arbitrary value of the BEP 100 kHz clock, to UTC.

Section 7.1 describes how to determine the UTC timeline when ACIS is operating either in Continuous Clocking mode (i.e. via a `CMDOP_START_CC` command) or in Timed Exposure Mode (`CMDOP_START_TE`) with a single exposure time (`dutyCycle=0` in the current `teBlock`). When `dutyCycle` is non-zero, a single exposure of length `primaryExposureTime` will be followed by `dutyCycle` exposures lasting `secondaryExposureTime`. This situation is covered in Section 7.2.

7.1 The Timeline of Single Exposure Time Modes

Since it takes a finite time to move the image out of each CCD, the X-Ray integration time of a timed exposure is less than the time from one exposure to the next. The following describes the overall steps in determining the start time of a timed exposure (or of a particular image frame in continuous-clocking mode.) The method first computes the time in units of the BEP clock, and then relates that to observatory time through the synchronous science header timestamps.

1. Determine `runStartTime`, the starting time of the science run in BEP timer units. It is reported in the `scienceReport` packet that terminates the run, as well as in each individual `exposure` packet.
2. Add the DEA startup time, `startupTicks`.

   This is the initial delay, in BEP clock units, between the commanded start-of-run time and the start of the first CCD exposure period. It is a function based on the clocking parameters and method. This function will be provided as part of the AS-BUILT ACIS Software Detailed Design Specification, MIT 36-53200.
3. Inspect any pair of consecutive `fepTimestamp` values in `exposure` packets.

   If $\texttt{fepTimestamp}_{i-1}$ is less than $\texttt{fepTimestamp}_{i}$, exposure `n` starts at

$$\texttt{exposureStartTime}_{n} = \texttt{runStartTime} + \texttt{startupTicks} + \texttt{n} * (\texttt{fepTimestamp}_{i} - \texttt{fepTimestamp}_{i-1})$$

expressed in BEP timer units. Otherwise, if the FEP time-stamp of exposure `i` is less than that of exposure `i–1`, the counter has wrapped, and the start time of exposure `n` is

$$exposureStartTime_n = runStartTime + startupTicks + n * (2^{25} + fepTimestamp_i - fepTimestamp_{i-1})$$

4. Now examine a pair of AXAF telemetry science frames generated during the run.

The first 4 bytes of ACIS science data in frame number `i` contain the science header timestamp ($refTime_i$). If $refTime_{i+1}$ is greater than $refTime_i$, the number of BEP clock ticks per 2.05 second science frame is

$$ticksPerFrame = refTime_{i+1} - refTime_i$$

Otherwise, if $refTime_{i+1}$ is less than $refTime_i$,

$$ticksPerFrame = 2^{32} + refTime_{i+1} - refTime_i$$

5. Locate the science frame time-stamp nearest to $exposureStartTime_n$.

This should occur at the science frame numbered `nf`, where

$$nf = i + integer((exposureStartTime_n - refTime_i)/ticksPerFrame)$$

The frame `i` should be chosen close to the start of the run. If $refTime_i$ is less than `runStartTime`, use instead

$$nf = i + integer((exposureStartTime_n - refTime_i - 2^{32})/ticksPerFrame)$$

Drift between the BEP clock and the observatory clock may be enough to cause an error in calculating `nf`, so a search should be made through nearby science frames for the one with the `refTime` value closest to $exposureStartTime_n$.

6. Extract or compute the Universal Time ($frameUT_{nf}$) corresponding to the start of the frame.

This operation is determined by the contents of the spacecraft science frame. It was originally planned to store the estimated UTC values directly into the header fields of each science telemetry frame. It is now proposed to store only the telemetry frame sequence number, and derive UTC during ground processing. The actual method chosen to relate the start of the telemetry frame to UTC is beyond the scope of this document.

7. Determine the precise observatory time of the start of exposure `n`:

$$exposureUT_n = frameUT_{nf} + 2.05 * (exposureStartTime_n - refTime_{nf})/ticksPerFrame$$

where 2.05 represents the time in seconds between successive science frame pulses (corresponding to 8 minor frames of 1025 bytes each at a rate of 32,000 bits per second; actual telemetry rates may differ.)

7.2 The Timeline of Alternating Exposure Time Modes

The following describes the overall steps in determining the start time of a particular exposure in Timed-Exposure Mode when two exposure times are used:

1. Compute `runStartTime + startupTicks`, as above.
2. Inspect several consecutive `exposure` records and compute three repetition intervals.

   $int1 = fepTimestamp_{i} - fepTimestamp_{i-1}$

   $int2 = fepTimestamp_{i+1} - fepTimestamp_{i}$

   $int3 = fepTimestamp_{i+2} - fepTimestamp_{i+1}$

   where exposure number `i` is evenly divisible by (`dutyCycle+1`). If any of these rates is negative, increment it by 2^{25}.
3. Compute the total number of primary exposures and exposure cycles.

   ```
   primaryCount = integer((dutyCycle + n) / (dutyCycle + 1))
   cycleCount = integer(n / (dutyCycle + 1))
   ```
4. Compute the exposure starting time in BEP timer units.

   $$exposureStartTime_{n} = runStartTime + startupTicks + (n - primaryCount - cycleCount) * int1 + primaryCount * int2 + cycleCount * int3 + (cycleCount - primaryCount) * (E_2 - E_1)$$

   where E_1 and E_2 are, respectively, the commanded `primaryExposureTime` and `secondaryExposureTime` from the timed exposure parameter block, converted to BEP clock units.
5. Follow steps Section 4.–Section 7. of the preceding section to translate $exposureStartTime_{n}$ to UTC.

8.0 Frame Buffer Specification

M. Doucette

27 October 1995

Revised 7 December 1995

8.1 Significant Changes in this Version

- The terms *frame* and *image* no longer refer to the same thing. The term *file* has been added.
- The “Repeat Pixel” function will not be implemented unless it is deemed necessary and if time permits. Repeat telemetering a single pixel can be done by repeat telemetering a segment of length one.
- The “parameter file frame changing” function will not be implemented.
- 6 DEA-to-FEP outputs will be implemented.
- The repeat frame directive must appear as the first word in the downloaded frame.
- Multiple images within a frame is valid as long as there is only one “Last Pixel Flag”.

8.2 Terms

- *Buffer*

  The term *Buffer* will refer to the *Frame Buffer*, an in-house designed and fabricated box used to supply test images to a Front End Processor (FEP).
- *Image*

  The term *image* will mean a set of pixels as defined in a DPA/DEA Interface Control Document”.
- *Directive*

  A special pixel having a Pixel Code (unused by the FEP) used exclusively by the Frame Buffer for function control.
- *File*

  The term *file* will refer to that set of data words downloaded from the DECstation to the Buffer box.
- *Frame*

  The term *frame* will refer to the set of data stored in the Buffer's memory. Frames include both image and directive pixels.

8.3 Initial Requirements/Specifications

The following requirements/specifications for a “Frame Buffer” box were proposed:

- Autonomously generate and output a repeating ramp image.

- Receive a single downloaded file from a DECstation via a Direct Memory Access (DMA) interface.
- Accept full or partial frames.
- Output the downloaded frame once, a specified number of times, or repeat the frame continuously.
- Inject a deterministic delay between telemetered frames.
- Telemeter a segment of pixel(s) “n” times.
- Generate a base frame which repeats and in which pixel data increments according to downloaded parameters. (Refer to the Status section.)
- Provide 6 DEA-to-FEP output circuits having indentical data and signal timing.

8.4 Basic Design Concept

The Buffer is designed around a Motorola 56001 Digital Signal Processor (DSP). All code will be in ROM. A DMA interface is incorporated for receiving downloaded files from the DECstation (and possibly DMA equipped SPARCstations). An internal memory stores a full 16 Mbit frame plus up to 128K miscellaneous pixels (e.g., Over Clocks, Ignores, directives, etc.). The memory is organized in 9 131,072 x 16-bit pages. Electrical and data format of signals interfacing to flight hardware will be reproduced as defined in the “DPA/DEA Interface Control Document”. Integrated circuits interfacing to flight hardware will be commercial versions of flight components. Signal timing will be done with hardware circuitry versus software control. The Pixel Clock frequency is derived from an “RC Coupled CMOS Gate Mulitvibrator” circuit running at approximately 6.4 MHz. Since the frequency is determined by a resistor and capacitor, its stability will not be very good.

The Frame Buffer will not supply FEP power. Buffer internal power will be over-voltage protected.

Buffer requirements such as telemetering a segment of pixels “n” times, or a frame “n” times, are accomplished by using special “directive” pixels inserted in the appropriate locations in the downloaded file. These directives use five (TBR) Pixel Codes not used by the FEP. In some cases the 12 data bits in the directive or the 12 data bits in a second pixel are an argument to the directive. Directive pixels must be inserted by the user in the file to be downloaded.

8.5 Operating Modes

There are two modes of operation selected by a front panel switch: Ramp and Normal Mode.

8.5.1 Ramp Mode

In Ramp Mode the Buffer continuously telemeters pixels which are generated in real time—this is not a stored file. The ramp image begins with 4 Ignore pixels followed by 4 Vertical Syncs followed by 1024 rows of 1024 columns of Valid pixels. Column 0 of each row contains the row number (0 thru 3FF hex). Columns 1 thru 1023 contain the column

number (1 thru 3FF hex). Each row ends with 32 Over Clocks having an incrementing data field (0 thru 1F hex) followed by 4 Horizontal Syncs.

8.5.2 Normal Mode

When the Buffer is in Normal Mode and is powered on, a file must be downloaded to the Buffer memory. Once downloaded, the Buffer will telemeter the image to the FEP in accordance with directives within the frame data. A reset will cause the Buffer to monitor the DMA interface in preparation for receiving data from the DECstation. A reset will not cause the loss of a currently stored frame, and telemetering may be initiated by the DECstation issuing a "Go" command. All pixels in the downloaded file are stored in memory. The Pixel Code of each pixel is decoded as it is received, and transfers will be terminated only upon receiving a "Last Pixel Flag" directive (see below). After the download is complete, the Buffer will then telemeter the image to the FEP. Directives are not telemetered to the FEP.

8.6 Directive Functions

In the following paragraphs, unused bits in hexadecimal fields are marked with the letter 'X'.

8.6.1 "EXXX" Last Pixel Flag (LPF)

The LPF marks the last pixel in the downloaded file. The data bits in this pixel are ignored. Attempting to download pixels after a LPF will have an indeterminate effect on the Buffer operation. A frame may contain multiple images but only one LPF directive.

8.6.2 "Annn" Repeat Segment "nnn" times (RS)
"Xnnn" Segment Length argument (SL)

This function requires two arguments. The first argument is the 12 data bits in the directive which defines the number of times (up to FFF hex) the segment is to be written to the FEP. The second argument is in the following pixel and defines the number of pixels (up to FFF hex) in the segment. The pixel following the second argument is the first pixel of the segment.

8.6.3 "7nnn" Repeat Frame "nnn" times (RF)

Repeat this frame "nnn" times. If the downloaded file does not begin with this directive, or if its argument "nnn" is "000 hex", the frame will repeat continuously. This directive must appear as the first word in the downloaded file—it is only tested once, and will be ignored if it occurs later in the file.

8.6.4 "6000" Go (TBR)

This single directive (or, more appropriately, command) downloaded from the DECstation causes the Buffer to begin telemetering a currently stored frame. This directive is ignored unless a frame has been stored in the buffer.

8.7 Current Status

Two circuit boards have been wired and a third is in progress. (Hardware revision levels below are for initial use only.)

- S/N 01 hardware is at Rev. 01. The DSP code installed in this unit is configured to generate ramp images only—image data files cannot be downloaded to this unit at this time.
- S/N 02 hardware is at Rev. 04. This unit has been used to develop DSP code to receive downloaded files, decode directives, and output image data in accordance with those directives. Code to generate a ramp image is not installed in this unit as yet.

The current version of the DSP code processes directives correctly. Timing has been a concern because, within the 2.5 μsec pixel period, each pixel is read, decoded and processed before it is telemetered to the FEP. "Repeat Segments" within the frame are processed in about 1.5 μsec. Continuous frames are also not a problem, taking about 0.75 μsec to process. The additional code required to process a "Repeat Frame" proved to be a problem. Periodically it has taken more than 2.5 μsec to set up for the next frame. As a result, the last pixel of the previous frame would be re-telemetered before the first pixel of the coming frame could be latched into the output circuits. To avoid this problem, I have the DSP running with minimum wait states. To gain some margin, the current 24 MHz DSPs will be replaced with 33 MHz devices. The download time for short (1k word) files has been measured at approximately 3.2 μsec/word. The requirement in which the Buffer would generate a "base" frame which would continuously telemeter pixels which change data from one image to the next according to downloaded parameters will not be implemented.

8.8 Proposed Additional Features

8.8.1 Front Panel Status LEDs

- Power On LED — Self explanatory.
- DMA Request LED — Indicates that the DECstation is waiting to download a file. It remains lit during the download and is cleared by the Buffer receiving a LPF or a RESET.
- Telemetry In Progress — Lit while the Buffer is telemetering Pixels to the FEP.

8.8.2 Error LED(s)

- Pixel Re-write LED — Lit when a timing error occurs and the Buffer re-telemeters a Pixel. This LED would be cleared by a RESET.

9.0 ACIS Data Analysis and Database

To: ACIS Team
From: Jonathan W. Woo
Subject: ACIS Data Analysis and Database
Date: 4 May 1995

This memo describes the following subjects related to the MIT ACIS lab calibrations: data formats, standard preliminary analysis procedures, utility software, and the database structure of events collected from flight devices.

9.1 Data Format

At present, there are four different data formats used in the initial data reduction processes. (1) Raw data read from ACIS CCDs are first recorded in the FITS files of full frame images. (2,3) From the raw image FITS files, events are then collected and stored in one of two event list formats—Ftool science file (FSF) format and ACIS RV (ARV) format, or in both. (4) Some event lists are also stored in IDL table files.

9.2 Raw Image Format

The raw data read out from ACIS CCDs by current test electronics are initially recorded in 2D FITS image files. Each value of readout pixels is registered as a 16 bits short integer record. In the beginning of each raw FITS file, extended FITS Header keywords can be found.

The following first group of keywords are there to conform to the FITS standard:

```
SIMPLE =            T   / file does conform to FITS standard
BITPIX =           16   / number of bits per data pixel
NAXIS  =            2   / number of data axes
NAXIS1 =         1120   / length of data axis 1
NAXIS2 =          256   / length of data axis 2
EXTEND =            T   / FITS dataset may contain extensions
```

These keywords specify that this FITS file contains a 2D short integer (16 bits) image array (1120x256).

The next six keywords are used to map a subframe array captured/saved from the full frame array (when the entire CCD is read out). The full frame array includes all active image pixels and “expected” overclock pixels (OCs) that should be consistent with OCs in the subframe array.

```
RASTER1 =        1120   / Original column dimension of entire CCD
RASTER2 =        1030   / Original row dimension of entire CCD
CRPIX1  =           1   / starting column number of sub image array
CRPIX2  =           1   / starting row number of sub image array
CRVAL1  =           1   / starting column number of remapping array
```

```
CRVAL2  =         200   / starting row number of remapping array
```

Thus RASTER1 and RASTER2 define the “expected” dimension of the full frame array. These keywords are useful when a subframe array of a given device is compared with a different (in locationwise) subframe array of the same device. The values of CRPIX1 and CRPX2 is usually 1. CRVAL1 and CRVAL2 should be the coordinate values of the first pixel from the subframe array relative to the full frame array (see Figure 4).

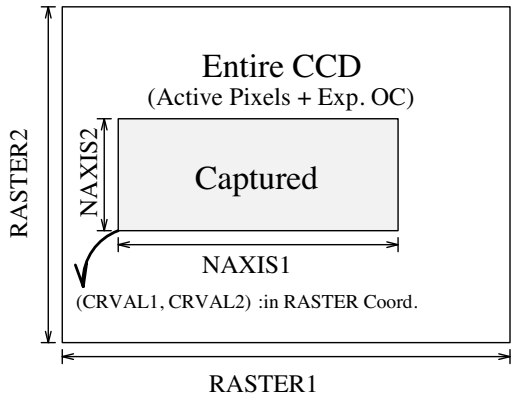


FIGURE 4. Relative coordinate system of a subframe array with respect to the full frame array.

The following keywords are there for record keeping purposes:

```
MISSION = 'AXAF    '    / Mission name
TELESCOP= 'MIT_CCD '    / Telescope (facility) name
INSTRUME= 'ACIS    '    / Instrument name
OBS_MODE= 'GROUND  '    / Observation mode
OBS_ID  = 'PTB Whitelight' / Observation purpose
DATAMODE= 'SPECIAL '    / data mode
CREATOR = '/usr/acis/s56dsp/hostvid'  / creator of data
                        / acquisition and control software
DETNAM  = 'w34c3   '    / CCD ID
AEID    = 'acis3   '    / Electronics ID
ORIGIN  = 'MIT     '    / SAO/MIT
SRC_ID  = 'BESSY   '    / Name of X-ray source device
OBJECT  = 'PTB     '    / Name of X-ray source anode
OBSERVER= 'sek'         / User ID of calibration operator
HOSTID  = '0xefffff284' / host machine CPU ID
HOSTNAME= 'otto'        / Host machine name
DATE    = '27/04/95'    / FITS file creation date (dd/mm/yy)
```

These keywords are self-explanatory.

The next set of keywords specify the locations of the active readout image pixels in the readout array:

```
IAMINCOL=          5   / Minimum column number of Output Node A
```

```
IAMAXCOL=         260   / Maximum column number of Output Node A
IAMINROW=           1   / Minimum row number of Output Node A
IAMAXROW=         256   / Maximum row number of Output Node A
IBMINCOL=         285   / Minimum column number of Output Node B
IBMAXCOL=         540   / Maximum column number of Output Node B
IBMINROW=           1   / Minimum row number of Output Node B
IBMAXROW=         256   / Maximum row number of Output Node B
ICMINCOL=         565   / Minimum column number of Output Node C
ICMAXCOL=         820   / Maximum column number of Output Node C
ICMINROW=           1   / Minimum row number of Output Node C
ICMAXROW=         256   / Maximum row number of Output Node C
IDMINCOL=         845   / Minimum column number of Output Node D
IDMAXCOL=        1100   / Maximum column number of Output Node D
IDMINROW=           1   / Minimum row number of Output Node D
IDMAXROW=         256   / Maximum row number of Output Node D
```

These are the row and column boundaries of the four read out nodes which contains active image pixels which exclude overclock pixels, underclock pixels, and undefined parallel transfer clock pixels from the readout array.

The frame acquisition time (actually for the previous frame) and a rough exposure time (difference between two UNIX time calls at the two consecutive data synchronization marks) are recorded in the following keywords:

```
DATE-OBS=  '27/04/95'   / Date(dd/mm/yy) of exposure start (EDT)
TIME-OBS=  '05:08:39'   / Time(hh:mm:ss) of exposure start (EDT)
ONTIME  =   1.528E+00   / Exposure time (sec between two sync)
```

The following keywords are initially set by data acquisition programs:

```
FILENAME= 'i24ey001.1170.fits'
FILEPATH= '/otto/d3/w34c3/27Apr0357/'
OVERFILL=       24704   / over fill
VERBOSE =           1   / verbose (0=off;1=on)
TRIGGER =        F0F0   /
TRCOUNT =          16   /
```

Voltage, current, and temperature readings of various clocks and corresponding ADU values are recorded in the following keywords:

```
V_IA_H  =      +4.490   / Voltage of CCD Image Clock High (V)
A_IA_H  =       44262   / ADU of CCD Image Clock High (ADU)
V_IA_L  =      -5.033   / Voltage of CCD Image Clock Low (V)
A_IA_L  =       19884   / ADU of CCD Image Clock Low (ADU)
V_FS_H  =      +6.956   / Voltage of CCD Frame Storage Clock High (V)
A_FS_H  =       50576   / ADU of CCD Frame Storage Clock High (ADU)
V_FS_L  =      -1.888   / Voltage of CCD Frame Storage Clock Low (V)
A_FS_L  =       27935   / ADU of CCD Frame Storage Clock Low (ADU)
V_OR_H  =         0.0   / Voltage of CCD Output Register Clock High (V)
A_OR_H  =           0   / ADU of CCD Output Register Clock High (ADU)
```

```
V_OR_L  =         0.0   / Voltage of CCD Output Register Clock Low (V)
A_OR_L  =           0   / ADU of CCD Output Register Clock Low (ADU)
V_RG_H  =      +7.522   / Voltage of CCD Reset Clock High (V)
A_RG_H  =       52025   / ADU of CCD Reset Clock High (ADU)
V_RG_LP =      +4.018   / Voltage of CCD Reset Clock Low+ (V)
A_RG_LP =       43054   / ADU of CCD Reset Clock Low+ (ADU)
V_SCP   =      +7.027   / Voltage of CCD Scupper (V)
A_SCP   =       50758   / ADU of CCD Scupper (ADU)
V_OG_P  =      +1.020   / Voltage of CCD Output Gate+ (V)
A_OG_P  =       35378   / ADU of CCD Output Gate+ (ADU)
V_RDA   =      +7.021   / Voltage of CCD Reset Drain (V)
A_RDA   =       50742   / ADU of CCD Reset Drain (ADU)
V_HTR_C =      +0.014   / Amp of Heater Current
A_HTR_C =       32803   / ADU of Heater Current
V_HTR_V =      +0.014   / Heater Voltage (V)
A_HTR_V =       32804   / ADU of Heater Voltage
V_RG_LM =         0.0   / Voltage of CCD Reset Clock Low - (V)
A_RG_LM =           0   / ADU of CCD Reset Clock Low - (ADU)
V_VDD_A =     +19.170   / Voltage of CCD Vdd_A (V)
A_VDD_A =       49126   / ADU of CCD Vdd_A (ADU)
V_VDD_B =     +19.173   / Voltage of CCD Vdd_B (V)
A_VDD_B =       49129   / ADU of CCD Vdd_B (ADU)
V_VDD_C =     +19.105   / Voltage of CCD Vdd_C (V)
A_VDD_C =       49071   / ADU of CCD Vdd_C (ADU)
V_VDD_D =     +19.218   / Voltage of CCD Vdd_D (V)
A_VDD_D =       49167   / ADU of CCD Vdd_D (ADU)
V_BJ    =         0.0   / Voltage of CCD Back Junction (V)
A_BJ    =           0   / ADU of CCD Back Junction (ADU)
V_EB_A  =   +5800.000   / Voltage of node A Electronics Bias (V)
A_EB_A  =        5800   / ADU of node A Electronics Bias (ADU)
V_EB_B  =   +5500.000   / Voltage of node B Electronics Bias (V)
A_EB_B  =        5500   / ADU of node B Electronics Bias (ADU)
V_EB_C  =   +5700.000   / Voltage of node C Electronics Bias (V)
A_EB_C  =        5700   / ADU of node C Electronics Bias (ADU)
V_EB_D  =   +5900.000   / Voltage of node D Electronics Bias (V)
A_EB_D  =        5900   / ADU of node D Electronics Bias (ADU)
CAM_TEMP=      +0.013   / Camera Temperature (C)
CCD_TEMP=     -242.00   / CCD Temperature (C)
```

The DE binary bit settings for the “gse” data acquisition program are recorded in the following keywords:

```
DE_MODE0=           0   / CHKCR: MODE0 bit (0=off;1=on)
DE_MODE1=           0   / CHKCR: MODE1 bit (0=off;1=on)
DE_HSEQ =           1   / CHKCR: HSEQ bit(0=off;1=on) the hard sequencer
DE_TSAP =           0   / CHKCR/TESTAP asyn loopback test (global hk pipe)
DE_BOARD=           1   / CHKCR: BOARD bit (0=off;1=on)
DE_RASTR=           1   / CHKCR: RASTER bit (0=off;1=on)
```

```
DE_RIGHT=           0   / VHKCR: RIGHT bit (0=off;1=on)
DE_SHFT0=           0   / VHKCR: SHIFT0 bit (0=off;1=on)
DE_SHFT1=           0   / VHKCR: SHIFT1 bit (0=off;1=on)
DE_SHFT2=           0   / VHKCR: SHIFT2 bit (0=off;1=on)
DE_EN12 =           0   / VCRX: EN12 bit (0=off;1=on)
DE_TSDVP=           0   / VCRX: TESTDVP bit (0=off;1=on)
DE_HEATR=           0   / DHKCR: Turns the heater (0=off;1=on)
DE_SWAP =           0   / DHKCR: Swaps output shift reg phs 2 with 3
```

9.2.1 FSF Format

In the standard ACIS analysis procedure, an event (3x3 pixel island whose center pixel value is the local maximum above event threshold) detected by the event finding algorithm (adopted from **rv_findevents**) is stored in the FSF event files. The FSF event files are in a binary FITS table extension format similar to the ASCA science event file format. Each FSF file has at least the following three components: a dummy primary image extension with zero dimension, a binary table extension containing events, and a binary table extension of good time intervals (GTIs). Since most of the FITS Header keywords in FSF are adopted from ASCA convention, a detailed description of the keywords is not presented in this memo.

In the binary table extension, each event is defined by 10 fields:

```
TIME, READX, READY, CHIPX, CHIPY, LABX, LABY, FRAME, PHAS, CCDNODE.
```

The TIME field is the absolute time (seconds after 1994.0 in UNIX time) of the frame exposure when the event was detected. The center pixel position of a detected event (above a specified event threshold) is recorded in readout, physical CCD, and lab reference coordinate systems, respectively as READX & READY, CHIPX & CHIPY, and LABX & LABY columns. The FRAME field contains the integer numbers of the frames where the detected events extracted. The frame numbers are counted from the TSTART time when the first data frame is exposed.

Therefore, the frame number has only a relative meaning for a single FSF data set. In a mathematical expression, nframe = (t-TSTART)/exposure. The PHAS field contains 9 short integer pulse height values of a 3x3 pixel island (see Figure 5). Finally, the CCDNODE field specifies the read-out node.

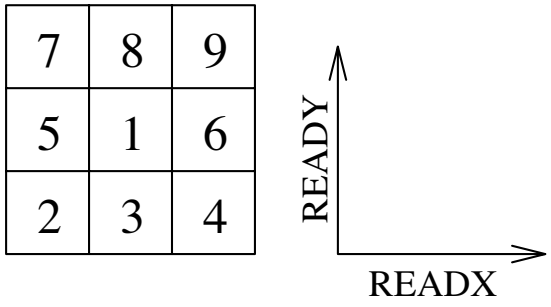


FIGURE 5. Sequence of pixel values from 3x3 pixel island stored in the event record of an FSF.

9.2.2 ARV Format

In order to use the available software developed for the previous laboratory X-ray CCD calibration efforts, all events recorded in FSFs are converted to a binary record format called "ARV" (ACIS RV) which is a modified version of "RV" format. The ARV format is a binary format of individual events identified as a 3x3 pixel island distribution. Each data structure is a 32 Byte binary record consisting of frame number, exposure time (in msec), acistime (sec from 1994.0), nodenum, x & y position of the center pixel, and the 9 pixel values of the 3x3 island in the order shown in Figure 6. While acistime is 4 Byte long integer, the rest of the variables are all in 2 Byte short integers.

```
struct ds_data_str
  {
    short nframe;
    short exposure;
    long acistime;
    short nodenum,x,y,data[9];
  };
```

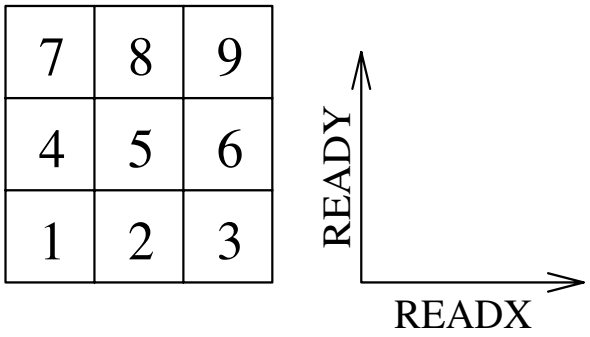


FIGURE 6. Sequence of pixel values from 3x3 pixel island stored in ARV format.

9.2.3 IDL Format

Some events are recorded in IDL binary (32 Byte record) table files similar to ARV format (ask sjones about details).

9.3 Analysis Procedure

For a typical run of data acquisition, a couple hundred consecutive raw image frames and several (~10) bias frames are taken. **ACISANAL1** (a tcsh script) is first run to collect events after correcting for the bias level for each pixel. An improved data acquisition time of each event is recalculated using the refined exposure time which is a constant value throughout a given data set. Once an event list is extracted by **ACISANAL1**, **ACISANAL2** (a tcsh script) is run to generate a number density image of events, pulse-height-distributions (PHDs) of two grade combinations (g0 and g0234), light curves, pulse-height histograms of each grade, and pulse-height histograms of neighboring pixels of the center pixels in the 3x3 event islands.

9.3.1 ACISANAL1

This script first produces a mean clip bias frame from all the available bias frames in a data set, by executing the **meanbiasclip** program:

```
ls ${bias_heading}.????.fits | meanbiasclip ${bias}
```

The mean bias frame prepared in this step will be used in the **acis_mit_findevents** routine to estimate the bias level for each pixel.

After a FITS file containing the mean bias frame is created, the script then prepares to run the main routine, **acis_mit_findevents** to extract events. As a first step for the preparation, the script executes the following line to collect all data file names into a temporary file:

```
ls ${data_heading}.????.fits > tmp_data.list
```

And then, the script calculates the exposure time of the data set (which is a constant value for a single data set). It uses the first 20 files to estimate a rough exposure time by executing the **get_exp** program which makes a simple average of values stored in the FITS header keyword ONTIME from the first 20 raw image frames. The script calls **get_exp** again to make a fine adjustment of the exposure time by averaging all ONTIME values which are close to the initial rough exposure time within 50% level. With these two iterations, the calculated exposure time should be good to 1/1000 s.

The following two lines in the **ACISANAL1** script are executed to calculate the exposure time:

```
set init_exp = `get_exp -1.0 20 < tmp_data.list | lines 1-1`
set exp = `get_exp ${init_exp} 0 < tmp_data.list | lines 1-1`
```

Once an accurate exposure time and an average bias frame are obtained, the **ACISANAL1** script runs the **acis_mit_findevents** program to extract events. **acis_mit_findevents** first reads in the mean bias frame and subtracts it off from each data image frame before extracting events. For each quadrant, it adds back the mean value of the bias frame subtracted image over-clock region to each image pixel for a fine temporal bias level adjustment. Then, the program goes through each pixel of the bias subtracted image array to find events where each event's central pixel value is above the low event threshold and is also the local maximum in the 3x3 island. Events are then recorded in a FSF event binary table FITS file. Each event record is identified with the UNIX time (absolute time is good to within a sec and relative time among events in a single data set is good to within ~1 ms), the position (readout and device coordinates) of the center pixel, quadrant id, and the 9 pixel values of the 3x3 island. This step is executed by:

```
acis_mit_findevents -el ${low_ev_th} -i stdin -bs ${bias} -o
${acis_evts_fits} -h ${acis_hk_fits} -exp ${exp} < tmp_data.list
```

To do further analysis using available RV tools, the FSF event list is converted into ARV format by running the **acis_fits_to_rv** program.

```
acis_fits_to_rv ${acis_evts_fits} stdout > ${acis_evts}
```

Since it is convenient to calculate readout noise at this stage, the **acis_readoutnoise** tcsh script is called to calculate readout noise values for 4 image and overclock regions using the first two bias files.

```
acis_readoutnoise ${bias_heading}.0001.fits ${bias_heading}.0002.fits
```

9.3.2 ACISANAL2

ACISANAL2 is a tcsh script which creates secondary products from an ARV event file: pulse-height distributions (histograms and XSPEC input file—PHA), event number density map, light curves, and event corner pixel pulse-height distributions. A number density map is created as a 2D FITS image file, PHA files are in the XSPEC binary format, and the rest of the products are in QDP file formats.

The **ACISANAL2** script first creates a number density map of detected events whose pixel grade distributions are either singles or splits (i.e., grades 0, 2, 3, or 4).

```
rv_gflt ${ev_th} ${sp_th} -p 0 4095 -g 0 2 3 4 < ${acis_evts} | rv_ef
${x_img} ${y_img} > ${heading}_sp${sp_th}_g0234_img.fits
```

For each of the four read-out nodes, two pulse-height histograms (g0 & g0234) are created. In the script, the following two lines in the *foreach* loop are executed:

```
rv_streamfilt -c ${ro_node} < ${acis_evts} | rv_gflt ${ev_th}
${sp_th} -p 0 4095 -g 0 | grad_ph ${sp_th} | awk '{print $4}' | histo
1 0.5 4095.5 4096 > ${heading}_c${ro_node}_sp${sp_th}_g0.qdp
rv_streamfilt -c ${ro_node} < ${acis_evts} | rv_gflt ${ev_th}
${sp_th} -p 0 4095 -g 0 2 3 4 | grad_ph ${sp_th} | awk '{print $4}' |
histo 1 0.5 4095.5 4096 > ${heading}_c${ro_node}_sp${sp_th}_g0234.qdp
```

Next, the script calculates the variation in count rate (counts/frame) for each read-out node as a function of frame number counted from the first raw image frame of a given set (~200 frames).

```
rv_streamfilt -c ${ro_node} < ${acis_evts} | rv_gflt ${ev_th}
${sp_th} -p 0 4095 -g 0 2 3 4 | grad_ph ${sp_th} | awk '{print $7}' |
lightcv > ${heading}_c${ro_node}_sp${sp_th}_g0234_lightcv.qdp
```

And, pulse-height histograms of each grade (0 to 7) are produced in QDP by the following tcsh command:

```
rv_streamfilt -c ${ro_node} < ${acis_evts} | rv_gflt ${ev_th}
${sp_th} -p 0 4095 | rv_ev2pcf ${ev_th} ${sp_th} /dev/null >
${heading}_c${ro_node}_pcf.qdp
```

Neighbor histograms (corner pixel distributions and top, bottom, left, and right pixel distributions of detected events) are produced by the following command:

```
rv_streamfilt -c ${ro_node} < ${acis_evts} | neighborhist >
${heading}_neighborhist_c${ro_node}.qdp
```

In QDP files, as a product of this step, “readout fraction” (total pixel values divided by the total number of pixels for each bin) is plotted against ADU.

To use XSPEC for the spectral analysis (with convolution of response matrix), two PHA files are produced (g0 & g0234):

```
rv_streamfilt -c ${ro_node} < ${acis_evts} | rv_gflt ${ev_th}
${sp_th} -p 0 4095 -g 0 2 3 4 | rv_ev2pha ${ev_th} ${sp_th} 1 >
${heading}_c${ro_node}g0234.pha
rv_streamfilt -c ${ro_node} < ${acis_evts} | rv_gflt ${ev_th}
${sp_th} -p 0 4095 -g 0 | rv_ev2pha ${ev_th} ${sp_th} 1 >
${heading}_c${ro_node}g0.pha
```

9.3.3 Data Products

A summary table and sample plots of products generated by ACISANAL2 are presented in this section. A sample plot of light curve is shown in Figure 7, a sample plot of PH-histogram is in Figure 8, a sample plot of primary calibration file is in Figure 9, and a sample plot of read-out noise distribution is in Figure 10.

TABLE 25. Summary of products from **ACISANAL2**

Type	Sample_filename	Format	Purpose
Evts Num Density	test_sp20_g0234_img.fits	FITS	Event Flux
PH histogram	test_c0_sp20_g0.qdp	QDP	Spectral Resolution
PH histogram	test_c0_sp20_g0234.qdp	QDP	Spectral Resolution
PH spectrum	test_c0_sp20_g0.pha	XSPEC	Spectral response/QE meas.
PH spectrum	test_c0_sp20_g0234.pha	XSPEC	Spectral response/QE meas.
Primary Cal File	test_c0_pcf.qdp	QDP	Grade branching ratio meas.
Light Curve	test_c0_sp20_g0234_lightcv.qdp	QDP	Check source/detector stability
Noise Summary File	readoutnoise.out	ASCII	Read-out Noise σ (ADU)
Noise Model	readoutnoise00im.model	ASCII	1 Gaussian fit
Noise spectrum	readoutnoise00im.qdp	QDP	Read-out Noise meas.

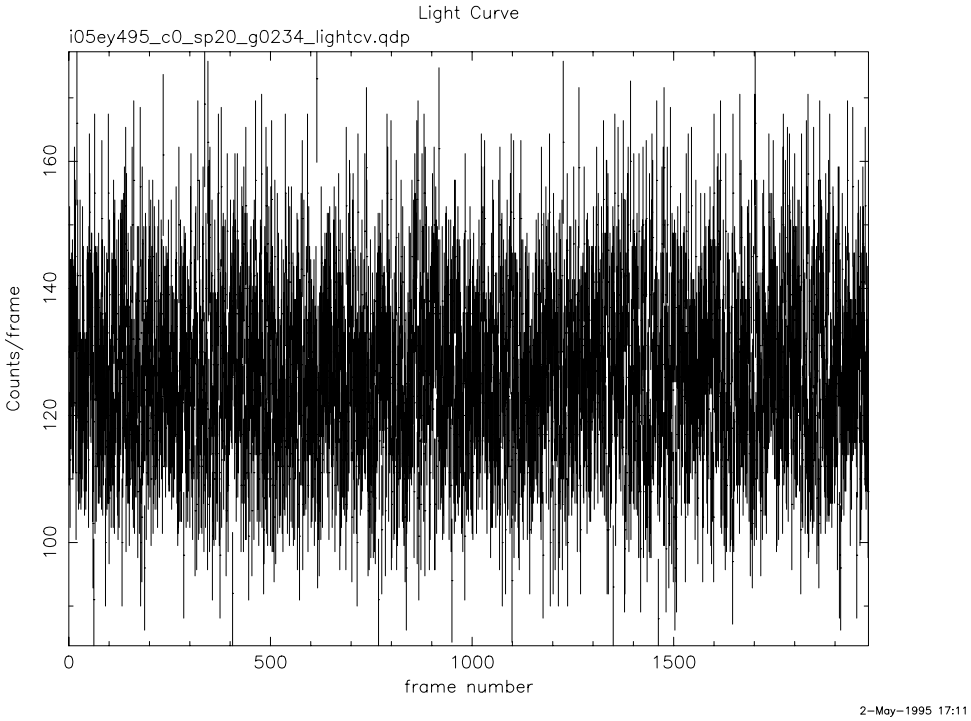


FIGURE 7. A sample light curve

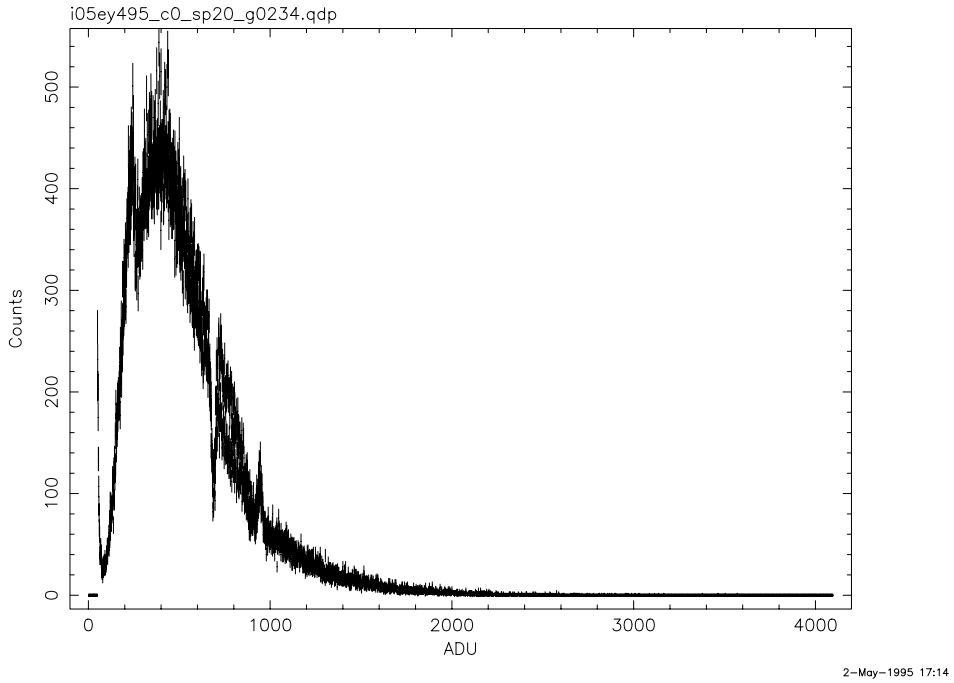


FIGURE 8. A sample PH histogram/spectrum.

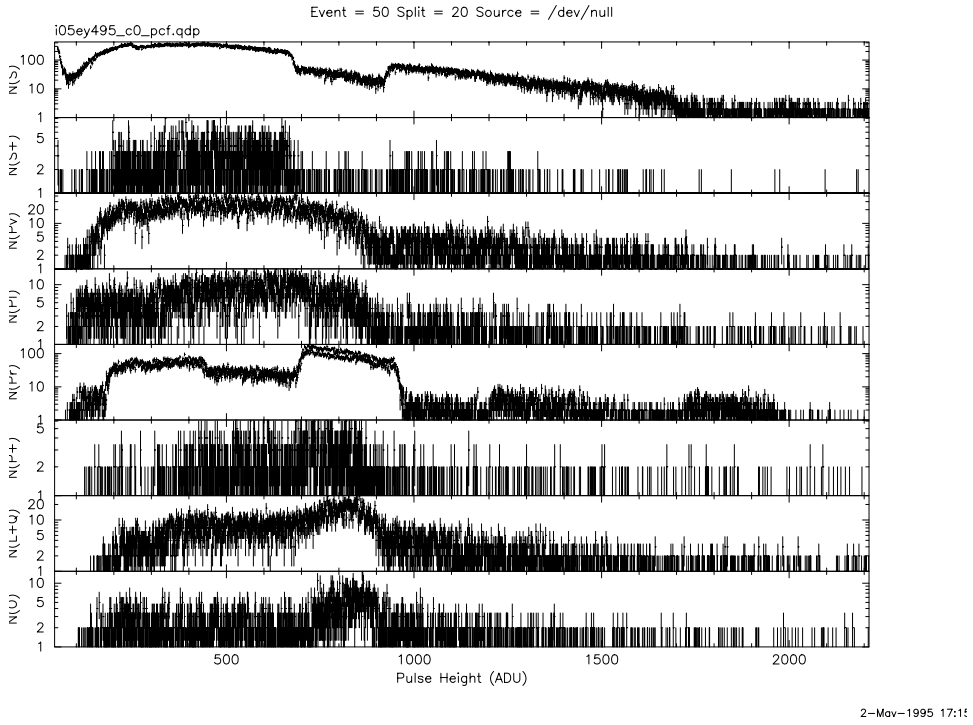


FIGURE 9. A sample primary calibration file

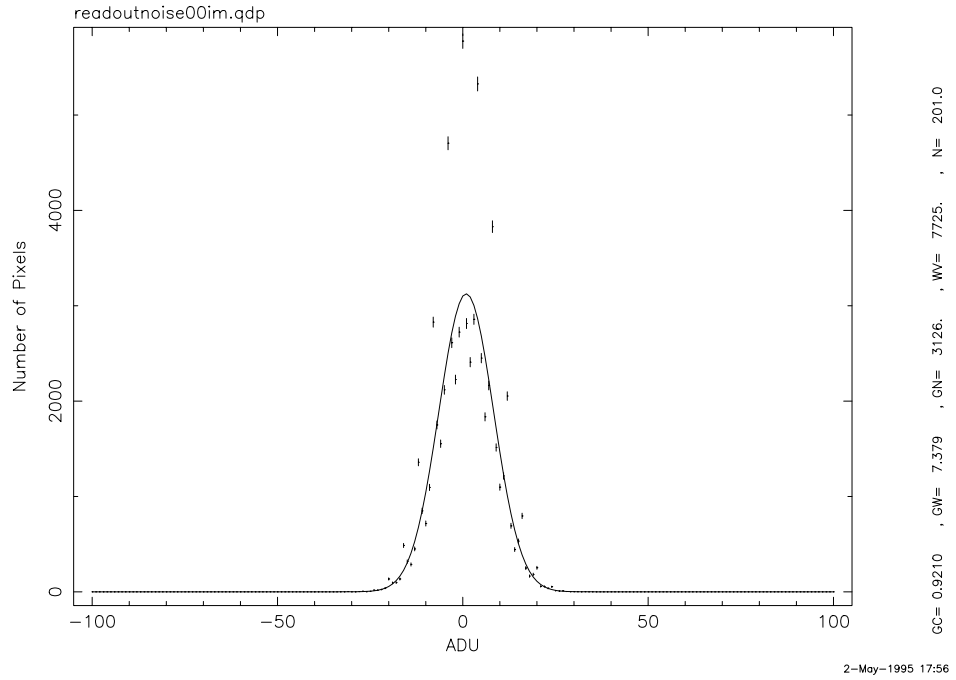


FIGURE 10. A sample readout noise file

9.4 Utility Software

There are a few utility modules in the "/usr/acis/bin" directory. **ana2gau** calculates the FWHM value of 2–Gaussian model determined by a fit to the PH histogram. The ARV event list can be displayed in ascii format with **acisrvtoascii**.

9.5 Database

The ACIS flight device calibration database will be constructed as subdirectory trees. On the top level, there will be one directory for each device and the directory name will be named after the device id (e.g., c17–103–1). For a given device directory, there will be subdirectories to identify lab experiment setups (e.g., IFM, HIREFS, HE–A, HE–B, SES, BESSY). As a next lower level subtree, the date of a particular data acquisition will be used (e.g., 01may95). A FSF event FITS file, a house keeping FITS file, an ARV file, two bias FITS files, and the averaged bias FITS file will be stored in the next lower level which specifies the source energy range (e.g., carbon_k, oxygen_k). Any ASCII note regarding to the particular data set will be placed at the same directory.

As an example,

```
p4 jww@ohno> pard
/ohno/d1/c17-103-1/bessy/09mar95/t1957_i05ey495
p4 jww@ohno> ls
c17-103-1_bessy_09mar95_1957_i05ey495.evts
c17-103-1_bessy_09mar95_1957_i05ey495_bias.0001.fits
c17-103-1_bessy_09mar95_1957_i05ey495_bias.0002.fits
c17-103-1_bessy_09mar95_1957_i05ey495_bias_avg.fits
c17-103-1_bessy_09mar95_1957_i05ey495_evts.fits
c17-103-1_bessy_09mar95_1957_i05ey495_hk.fits
products/
temperature.dat
```

The secondary products from **ACISANAL2** will be also stored in a subdirectory call "products":

```
p4 jww@ohno> pwd
/ohno/d1/c17-103-1/bessy/09mar95/t1957_i05ey495/products
p4 jww@ohno> ls
i05ey495_c0_pcf.qdp                    i05ey495_neighborhist_c0.qdp
i05ey495_c0_sp20_g0.qdp                i05ey495_neighborhist_c1.qdp
i05ey495_c0_sp20_g0234.qdp             i05ey495_neighborhist_c2.qdp
i05ey495_c0_sp20_g0234_lightcv.qdp     i05ey495_neighborhist_c3.qdp
i05ey495_c0_sp20_g0.pha                i05ey495_sp20_g0234_img.fits
i05ey495_c0_sp20_g0234.pha             readoutnoise.mesg
i05ey495_c1_pcf.qdp                    readoutnoise.out
i05ey495_c1_sp20_g0.qdp                readoutnoise00im.model
i05ey495_c1_sp20_g0234.qdp             readoutnoise00im.qdp
i05ey495_c1_sp20_g0234_lightcv.qdp     readoutnoise00oc.model
```

```
i05ey495_c1_sp20_g0.pha                 readoutnoise00oc.qdp
i05ey495_c1_sp20_g0234.pha              readoutnoise01im.model
i05ey495_c2_pcf.qdp                     readoutnoise01im.qdp
i05ey495_c2_sp20_g0.qdp                 readoutnoise01oc.model
i05ey495_c2_sp20_g0234.qdp              readoutnoise01oc.qdp
i05ey495_c2_sp20_g0234_lightcv.qdp      readoutnoise02im.model
i05ey495_c2_sp20_g0.pha                 readoutnoise02im.qdp
i05ey495_c2_sp20_g0234.pha              readoutnoise02oc.model
i05ey495_c3_pcf.qdp                     readoutnoise02oc.qdp
i05ey495_c3_sp20_g0.qdp                 readoutnoise03im.model
i05ey495_c3_sp20_g0234.qdp              readoutnoise03im.qdp
i05ey495_c3_sp20_g0234_lightcv.qdp      readoutnoise03oc.model
i05ey495_c3_sp20_g0.pha                 readoutnoise03oc.qdp
i05ey495_c3_sp20_g0234.pha
```

A log summary file of these directory trees, called “data.log” can be found in the “/usr/acis/database/” directory. This file can be used as a guide to locate events and products. The content of this file should be:

```
CDID         EXP    DATE(HH:MM)     SRCID      DIR    USER  MOD-DATE
c17-103-1  BESSY  09mar95(19:57)  i05ey495     *     jww   05/02/95

* = /ohno/d1/c17-103-1/bessy/09mar95/t1957_i05ey495
```

10.0 UNIX Commands

10.1 ACISshell

NAME

ACISshell – run a shell from which the CTUE can get ACIS commands

SYNOPSIS

`ACISshell`

DESCRIPTION

ACISshell creates a TCP/IP socket at the port that the CTUE will connect to when accepting commands from the ACIS EGSE. Once the connection is established, the program executes a standard Bourne shell, which it indicates by displaying the *ACISshell%* command prompt.

This shell and all of the programs that it launches inherit the open socket descriptor that connects to the CTUE. (This socket descriptor is available in environment variable CTUE_CMD_SD.) To send commands to the CTUE use:

```
ACISshell% buildCmds < ACIS command script file | sendCmds | writeCCB
```

or:

```
ACISshell% cat < CTUE command block file > &$CTUE_CMD_SD
```

ACISshell tries to open port 541 for CTUE commands. This port number is in the range that only root can open. If **ACISshell** is not running with root priviledges, it will try to open port 7541 for CTUE commands. The appropriate port number must be entered in the CTUE's `c:\windows\ethernet.cfg` file. The IP address of the host where **ACISshell** will run must also be entered in this file. The CTUE will connect only to the IP address and port indicated.

To connect to the CTUE, the user should first execute **ACISshell** and then use the CTUE's GUI to select ACIS commands from the Ethernet control section. **ACISshell**, like the CTUE, handles only a single connection. When that connection terminates, so does the program and so does the connection to the CTUE. However, commands can be sent repeatedly while the *ACISshell%* prompt is displayed.

AUTHOR

Demitrios Athens, MIT CSR

Ann M. Davis, MIT CSR for the network code.

SEE ALSO

writeCCB(1)

CTUE User's Manual

10.2 acisBepUnix

NAME

acisBepUnix – Simulate the ACIS Back End Processor

SYNOPSIS

acisBepUnix

DESCRIPTION

This program simulates the Back End Processor software on an Ultrix workstation. The program launches a command server program, *cserver,* to acquire command packets, and writes its binary telemetry to *stdout*. The *cserver* program is launched to establish a command socket on the local host machine. The used socket number is *7000*.

Command packets may be built and sent to the running simulation using *buildCmds* and *cclient*. Telemetry can be examined using *psci* or *ltlm*. (Note: Typically, a telemetry server, such as *filterServer* is setup to read *stdout* of the program and then distribute the telemetry to one or more clients. The clients then decode the received telemetry packets).

FEP hardware and software can be simulated using *acisFepUnix*. Each instance of a simulated FEP is a distinct invocation of *acisFepUnix*. The BEP simulator communicates with the simulated FEPs through shared memory segments.

OPTIONS

None

RESTRICTIONS

cserver must be in the user's search path.

This program is currently only supported for the DECstations running Ultrix.

FILES

None

SEE ALSO

cserver(1), cclient(1), acisFepUnix(1), buildCmds(1), psci(1), ipcs(1), ipcrm(1)

DIAGNOSTICS

Fatal Messages:

None

Warning Messages:

None

Informatory Messages:

Discrete Software Telemetry codes (LED codes) appear on *stderr* as they are written to the "hardware" by the flight software.

The message "WarmFuzzy" is written to *stderr* once about every 5 seconds.

Commands to the DEA are decoded and written to *stderr*.

10.3 acisFepUnix

NAME

acisFepUnix – Simulate the ACIS Front End Processor

SYNOPSIS

acisFepUnix *fepId*

DESCRIPTION

This program simulates a single Front End Processor software on an Ultrix workstation. The FEP is indicated by *fepId*. The FEP simulation program communicates with the BEP simulation program, *acisBepUnix,* using shared memory segments and signals. Images are loaded into the FEP simulation using *fepImage2,* which also uses shared memory and signals to communicate with the FEP process. The BEP program simulates a reset to the FEP simulation using the SIGUSR2 (31) signal. The FEP image loader program simulates FEP interrupts using the SIGUSR1 (30) signal.

OPTIONS

None

RESTRICTIONS

Only one *acisFepUnix* program with a given *fepId* may be active on a single machine at a time.

This program is currently only supported for the DECstations running Ultrix.

FILES

None

SEE ALSO

acisBepUnix(1), fepImage2(1), ipcs(1), ipcrm(1)

DIAGNOSTICS

Fatal Messages:

If the program fails to obtain shared memory segments, it will issue an error message to *stderr* and exit with a -1.

Warning Messages:

None

Informatory Messages:

During initialization, the program prints information about its shared memory segments to *stderr*.

When the FEP program is "reset" it prints a "Calling fepCtl()" message to *stdout*.

10.4 acispkts

NAME

acispkts – extract ACIS packets and pseudopackets from LRCTU output

SYNTAX

acispkts [–ep] [–np] [–nt]

DESCRIPTION

acispkts reads the *stdout* stream from the LRCTU handler, *rctu2b*, identifies individual ACIS telemetry packets, and writes them to the standard output stream, *stdout*.

Unless the **–np** option is specified, *acispkts* will generate one science frame pseudopacket and one engineering pseudopacket for every simulated LRCTU science pulse, at 2.05 second intervals. The science frame pseudopacket will contain the 32-bit BEP science pulse timestamp and a time-of-day clock value in IRIG-B format derived from the *gettimeofday*() system call. The origin of the IRIG day number is 0h, January 0 of the current year. CCSD major and minor frame counts start at zero. Engineering pseudopackets contain the 8 LED hardware status bits, recorded in two 16-bit data fields, one bit per nibble, e.g. status bits of '00110110' will be reported as 0x0011, followed by 0x0110.

OPTIONS

–ep generate a user pseudoppacket (a formatId of TTAG_USER) whenever LRCTU or ACIS packet synchronization is lost. If this option is omitted, the information will be written to *stderr*, (see Diagnostics, below).

–np suppress the writing of Science Header and Engineering pseudo-packets to *stdout*. The default behavior is to generate one of each for every equivalent science frame simulated by the LRCTU.

–nt suppresses the printing of ACIS timestamp and LED information to *stderr*.

AUTHOR

Ann Davis <amd@space.mit.edu> and Peter Ford, <pgf@space.mit.edu>, MIT CSR

SEE ALSO

rctu2b(1), filterServer(1)

DIAGNOSTICS

- **Acis Time:** *xxxxxxxx* **Bilevels:** *bbbbbbbb*
  this message is generated at 2.05 second intervals unless the **–nt option is specified.** *xxxxxxxx* is the BEP timestamp at the most recent science pulse time, and *bbbbbbbb* are the current LED values.

- **Fill error: got** *'string'*
  the character received from the LRCTU doesn't match the ACIS synchronization string, "*fAos*". If the **–ep** option is selected, this message will be written to *stdout* in a user pseudopacket (TTAG_USER). Otherwise, it will be written to *stderr*.

- **Invalid ACIS packet length**
  the length of the current packet is less than 2 32-bit words or greater than 1024. If the **–ep** option is selected, this message will be written to *stdout* in a user pseudopacket (TTAG_USER). Otherwise, it will be written to *stderr*.

- **Lost LRCTU frame synch,** *n* **bytes to recover**
  the expected LRCTU synch code (0x55aa) was not received. *n* bytes were skipped until the next synch was

found. If the **–ep** option is selected, this message will be written to *stdout* in a user pseudopacket (TTAG_USER). Otherwise, it will be written to *stderr*.

- **Questionable frame length** *n*
  the LRCTU frame length is neither 768 nor 744. If the **–ep** option is selected, this message will be written to *stdout* in a user pseudopacket (TTAG_USER). Otherwise, it will be written to *stderr*.

10.5 bcmd

NAME

bcmd – translate ACIS command script to binary

SYNOPSIS

bcmd [**–V**] [**–r**] [*file*]

DESCRIPTION

This Perl script reads a script of ACIS commands, checks syntax and field values, and writes the binary output to *stdout*, prefixing each command with a 4-byte header specifying the command type and RCTU channel code.

OPTIONS

–V (show Version) write to *stderr* the RCS revision number of this command and of the IP&CL used to generate it.

–r (raw mode) omit the 4-byte prefix to each command, which are all assumed to be software serial command. The presence of any hardware serial or pulse command will be considered illegal and *bcmd* will terminate abnormally.

SYNTAX

The *bcmd* command scripts obey the following rules: a "#" sign begins a comment and causes the remainder of the line to be ignored. Blank lines will also be ignored. Each command must appear on a single line, unless it ends with a "{" sign, in which case it continues on as many lines as necessary to define the block, each containing a single "keyword = value" statement, and terminating with a line containing a single "}". The keyword names are identical to those in the IP&CL structures tables.

In the following command lists, the construction "< *x* | *y* >" means "necessarily, either *x* or *y*".

Software Serial Commands

```
add n cc badcolumn <{|file>
add n te badcolumn <{|file>
add n badpixel < {|file>
add n patch d d <{|file>
change n systemconfig <{|file>
continue n uplink <{|file>
dump n badpixel
dump n cc
dump n cc badcolumn
dump n dea
dump n huffman
dump n patchlist
dump n systemconfig
dump n te
dump n te badcolumn
dump n window1d
dump n window2d
exec n n
exec n d <{|file>
exec n fep d n
exec n fep d d <{|file>
```

```
load n cc d <{|file>
load n dea d <{|file>
load n te d <{|file>
load n window1d d <{|file>
load n window2d d <{|file>
read n d n
read n fep d d n
read n pram d d n
read n sram d d n
remove n patch <{|file>
reset n badpixel
reset n cc badcolumn
reset n te badcolumn
start n cc n
start n cc bias n
start n dea n
start n te n
start n te bias n
start n uplink d d d <{|file>
stop n dea
stop n science
wait n
write n d <{|file>
write n fep d d <{|file>
write n pram d d <{|file>
write n sram d d <{|file>
```

Hardware Serial Commands

```
halt bep
program eeprom n n
run bep
select bep <a|b>
select eeprom <programming|telemetry>
set <warmboot|bootmodifier> <off|on>
set radiationmonitor <high|low>
verify eeprom n
```

Hardware Pulse Commands

```
<open|close> <door|vent|relief> <a|b>
<openabort|closeabort> <door|vent> <a|b>
<enable|disable> <dabake|daheater|dea|door|dpa|pressure|relief|vent>
      <a|b>
<poweron|poweroff> <dabake|daheater|dea|dpa> <a|b>
```

Italicized arguments represent user-supplied values; those in normal type must appear unaltered, except that upper and lower case letters may be freely interchanged, except in file names. All numeric arguments may be entered as signed decimal integers or, if preceded by "0x", as unsigned hexadecimal integers. If preceded by "0" alone, they will be interpreted as unsigned octal integers.

Within parameter blocks, i.e. between the "{" and "}" characters, the following fields are determined from the command line and from other keywords, so they do *not* need to be specified:

```
ccBlockSlotIndex
checksum
```

```
commandLength
commandIdentifier
commandOpcode
deaBlockSlotIndex
teBlockSlotIndex
windowSlotIndex (in window1d and window2d blocks, only)
```

When a block contains fixed-length arrays, e.g. *fepCcdSelect* in *teBlock*, the values, 6 in this case, must appear on the same line, i.e.

```
fepCcdSelect = 0 1 2 3 10 10
```

When a block contains varying-length arrays of structures, e.g. *windows* in *window2d*, the structures must be defined in the following manner:

```
load 2 window2d 1 {
  windowBlockId           = 0x00000abc
  windows = {
    ccdId                 = 0
    ccdRow                = 50
    ccdColumn             = 150
    width                 = 99
    height                = 99
    sampleCycle           = 1
    lowerEventAmplitude   = 0
    eventAmplitudeRange   = 65535
  }
  windows = {
    ccdId                 = 0
    ccdRow                = 250
    ccdColumn             = 350
    width                 = 99
    height                = 99
    sampleCycle           = 1
    lowerEventAmplitude   = 0
    eventAmplitudeRange   = 65535
  }
}
```

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

buildCmds(1), lcmd(1)

DIAGNOSTICS

- *command*: **bad wait time**
  the waiting time (in seconds) must be a positive integer.
- *command*: **command not implemented**
  some hardware serial commands, e.g. "set EEPROM", are unimplemented in this version of *bcmd*.
- *command*: **command packet too long:** *n* **bytes**
  no commands can contain more than 256 words (512 bytes). Longer packets, e.g. large patches, bad pixel or column table loads, must be split up into smaller data sets and resubmitted.

- *command*: **illegal in raw mode**
  hardware serial and pulse commands are illegal when the **–r** flag is used.
- *command*: **unknown command**
  bcmd does not recognize the command, either because of its leading command word or because it is followed by incorrect keywords.
- *command*: **unrecognized keyword:** *word*
  the named *word* is not legal for this particular command.
- *name*: **bad field offset:** *bits*
  the starting offset of a table or data array must be an exact multiple of 16 bits. This error should not occur in a fully debugged version of *bcmd*.
- *name*: **field missing from block definition**
  the input script contained no value for the *name* field.
- *name*: **illegal field value:** *n*
  the value *n* supplied for field *name* was outside the limits defined for this field in the IP&CL structures table.
- *name*: **missing data array**
  no array or data file was specified for a command that expects one.
- *name*: **wrong number of field values:** *n*, **not** *m*
  the number of fields supplied for *name* did not match that in the IP&CL structures table.

10.6 buildCmds

NAME

buildCmds – generate an ACIS binary command stream

SYNTAX

buildCmds [**–a**] [**–i**] [**–l** *item*] [**–m**] [**–p**] [**–?**]

DESCRIPTION

buildCmds creates a binary ACIS command stream from a script read from *stdin*.

OPTIONS

–a output command packets in hexadecimal to *stdout*. default: output binary to *stdout*.

–i print the release number of the IP&CL used to generate commands.

–l *item*
if *item* is "blockList", print a list of valid parameter block keywords. otherwise, print a list of valid parameter keywords used to define the parameter block named *item*.

–m format a single parameter block. Default: *buildCmds* accepts a series of commands, some of which may contain parameter blocks

–p omit the 4–byte command type and channel headers. Default: prefix each command block with type and channel identifiers.

–? print a description of *buildCmds* options.

COMMAND SCRIPTS

The input script consists of one or more lines of ASCII text. Blank lines, and all text following the first instance of a '#' character within a line, will be treated as comments and ignored. All other lines must contain one or more tokens, separated by whitespace (blanks or tabs). The first token must be a recognized command as listed below. The second token must be a numeric *cmdId*. All tokens are case insensitive, file names must be strings in small letters. The underlined tokens indicate values introduced by user, the not underline tokens are keywords.

The full syntax is as follows:

command *cmdId* **[** *arg_1* **...** *arg_n* **]**

command
the action to be executed by the ACIS instrument or by its EGSE.

cmdId
a decimal integer in the range 0-65535, identifying the specific command. It is used to identify the command reply in the telemetry packets.

The remaining arguments arg_1 ... arg_n depend on the particular command. They are of the following types:

address
a decimal or hexadecimal ('0x' prefix) number identifying a memory location.

length
a decimal or hexadecimal ('0x' prefix) number specifying the length of a memory segment.

memory
a mnemonic identifying the memory segment to which the action is applied, e.g: bep, fep, pram, or sram.

memoryId
a mnemonic identifying a subsection of memory by name, e.g: slotid.

param_block
the contents of an ACIS parameter block. Parameter blocks are represented by brace-enclosed command line lists of the form

```
{
  parameterBlockName = nameId
  keyword1 = value 1
  keyword2 = value 2
  .
  keywordN = value N
}
```

The value of the parameterBlockName keyword identifies the type of parameter block. Valid values are:

- `arguments`
- `badcolumn`
- `badpixel`
- `ccblock`
- `configsetting`
- `deablock`
- `patches`
- `teblock`
- `window1d`
- `window2d`

The `parameterBlockName` keyword is required, and must be the first keyword in the block definition. Keyword values are expressed as decimal or hexadecimal (‘0x’ prefix) integers. Any number of newlines may appear within the enclosing braces. Some parameter blocks, namely `window1D`, `window2D`, and, `deaBlock`, contain keywords followed by arrays of structures, each containing several keywords. In these cases, the special keyword `arrayDim` must be added immediately before the array to indicate its dimension.

Within the parameter block, and within each enclosed sub-structure, the order of keywords must follow the order in the IP&CL tables, i.e. ascending bit offset from the start of the block. Omitted keywords will be assigned default values—their most recently specified value, or zero if they have not yet been used in the block.

Each ACIS command must therefore appear on a separate input line. The only exception is that in-line param_blocks may contain newline characters within the enclosing braces.

SERIAL SOFTWARE COMMANDS

The following commands will be prefixed by a 4-byte header that sends them to the software serial command port of the DPA.

ADD

This command reads data to add in BEP memory and outputs a binary stream to the standard output, *stdout*, which contains the commands in the format requested by the ACIS software.

add *cmdId* **badPixel** *paramBlock*
add set of pixels to Timed Exposure Bad Pixel Map

add *cmdId* **cc badColumn** *paramBlock*
add set of columns to Continuous Clocking Bad Column Pixel Map.

add *cmdId* **patch** *patchId address file*
add a patch to the specified *address* corresponding to *patchId* slot.

add *cmdId* **te badColumn** *paramBlock*
add set of columns to Timed Exposure Bad Column Pixel Map.

When the entire block cannot be stick in one command, the buildCmds subdivides the parameter block into sections and outputs several commands.

A list of the parameter keywords contained in the parameter block is given in the example section.

Examples:

```
add 12 cc badColumn {
  paramBlockName = badColumn
  ccdId = 2
  ccdColumn = 34
}

add 23 te badColumn {
  paramBlockName = badColumn
  ccdId = 2
  ccdColumn = 34
  ccdId = 4
  ccdColumn = 5
}

add 15 badPixel {
  paramBlockName = badPixel
  ccdId = 2
  ccdRow = 4
  ccdColumn = 50
  ccdId = 5
  ccdRow = 4
  ccdColumn = 60
}
```

The two sets: `ccdId`, `ccdColumn`, and `ccdId`, `ccdRow`, `ccdColumn`, can be repeated until a maximum of TBD times. Note that the order of the keywords is significant—`ccdId` must precede `ccdRow`, which must precede the `ccdColumn` to which it refers.

CHANGE

This command reads the list of parameters to change in the System Configuration parameter block and outputs a binary stream, *stdout*, which contains the command in the format requested by the ACIS software.

change *cmdId* **systemConfig** *paramBlock*
request to overwrite the existing system configuration block as indicated by the entries in the command packet

A list of the parameter keywords contained in the parameter block is given in the example section.

Example:

```
change 22 systemConfig {
  paramBlockName = configSetting
  itemId = 2
  itemValue = 8
}
```

The `itemId` and `itemValue`, can be repeated until a maximum of TBD times.

CONTINUE

This command reads a binary file containing the data to uplink and outputs a binary stream to the standard output, *stdout*, which contains the command and the data to uplink in the format requested by the ACIS software. The maximum size of the binary file is 249 16-bit words corresponding to one command packet.

continue *cmdId* **uplink** *file*
request to continue the uplink boot using the data contained in the packet

file is the name of a binary file containing the data to uplink.

Example:

```
continue 35 uplink file
```

DUMP

This command reads the identification of the memory section to dump and outputs a binary stream to the standard output, *stdout*, which contains the command in the format requested by the ACIS software.

dump *cmdId* **badPixel**
request to dump the Bad Pixel Map

dump *cmdId* **cc**
request to dump every Continuous Clocking parameter block stored in memory

dump *cmdId* **cc badColumn**
request to dump the Continuous Clocking Bad Column Pixel Map

dump *cmdId* **dea**
request to dump every DEA parameter block stored in memory

dump *cmdId* **huffman**
request to dump huffman tables

dump *cmdId* **patchList**
request to dump the patchList

dump *cmdId* **systemConfig**
request to dump the System Configuration parameter block

dump *cmdId* **te**
request to dump every Timed Exposure parameter block stored in memory

dump *cmdId* **te badColumn**
request to dump the Timed Exposure Bad Column Pixel Map

dump *cmdId* **window1D**
request to dump every window 1D blocks stored in memory

dump *cmdId* **window2D**
request to dump every window 2D blocks stored in memory

Example:

```
dump 35 badPixel
dump 36 cc badColumn
dump 37 te
```

EXEC

This command reads memory bank, address, and optional arguments of the function to execute and outputs a binary stream to the standard output, *stdout*, which contains the command in the format requested by the ACIS software.

exec *cmdId address*
request to execute the function located in bep memory at the indicated address

exec *cmdId address argumentBlock*
request to execute the function located in bep memory at the indicated address with the specified arguments

exec *cmdId* **fep** *fepId address*
request to execute the function located in fep memory at the indicated address

exec *cmdId* **fep** *fepId address argumentBlock*
request to execute the function located in fep memory at the indicated address with the specified arguments

fepId specifies the FEP memory where the function is located. The value is a decimal integer in the range 0-5. If the **fep** and *fepId* keywords are missing, the default memory is BEP.

argumentBlock contains the list of arguments used by the function.

Example:

exec 2 0x1111111
execute the function located in the BEP at address 0x11111111

exec 56 fep 5 0x12345678
execute the function located in the FEP 5 at address 0x12345678

```
exec 44 0x22222222 {
  paramBlockName = arguments
  functionArguments = 1111
  functionArguments = 3
}
```

execute the function located in the BEP at the address 0x22222222 with two arguments, 1111 and 3.

```
exec 24 fep 3 0x1234 {
  paramBlockName = arguments
  functionArguments = 2222
  functionArguments = 5
}
```

execute the function located in the FEP 3 at the address 0x1234 with two arguments, 2222 and 5.

LOAD

The command reads the parameter block information and outputs a binary stream to the standard output, *stdout*, which contains the command in the format requested by the ACIS software.

load *cmdId* **cc** *slotId paramBlock*
request to load the Continuous Clocking parameter block in the specified slot

load *cmdId* **dea** *slotId paramBlock*
request to load the dea parameter block in the specified slot

load *cmdId* **te** *slotId paramBlock*
request to load the Timed Exposure parameter block in the specified slot

load *cmdId* **window1D** *slotId paramBlock*
request to load the window 1-D parameter block in the specified slot

load *cmdId* **window2D** *slotId paramBlock*
request to load the window 2-D parameter block in the specified slot

cc, te, dea, window1D, and window2D are keywords identifying the parameter block to which the action is applied. In particular, cc specifies the Continuous Clocking parameter block, te the Timed Exposure parameter block, dea the DEA housekeeping parameter block, window1D and window2D respectively the Window 1-D and Window 2-D parameter block.

The parameterBlocks "window1D" and "window2D" contain an additional parameter name "arrayDim" which indicate the number of windows contained in the paramBlock. A list of the parameter keywords contained in the parameter block is given in the example section. As explained above, keyword order is significant.

Examples:

```
load 2 cc 3 {
  paramBlockName           = ccBlock
  parameterBlockId         = 2030
  fepCcdSelect             = 0 1 2 3 4 5
  fepMode                  = 1
  bepPackingMode           = 2
  ignoreBadColumnMap       = 1
  recomputeBias            = 1
  trickleBias              = 0
  rowSum                   = 8
  columnSum                = 6
  overclockPairsPerNode    = 8
  outputRegisterMode       = 2
  ccdVideoResponse         = 1 1 0 1 1 0
  fep0EventThreshold       = 100 100 100 100
  fep1EventThreshold       = 100 100 100 100
  fep2EventThreshold       = 100 100 100 100
  fep3EventThreshold       = 100 100 100 100
  fep4EventThreshold       = 100 100 100 100
  fep5EventThreshold       = 100 100 100 100
  fep0SplitThreshold       = 0   0   0   0
  fep1SplitThreshold       = 0   0   0   0
  fep2SplitThreshold       = 0   0   0   0
  fep3SplitThreshold       = 0   0   0   0
  fep4SplitThreshold       = 0   0   0   0
  fep5SplitThreshold       = 0   0   0   0
  lowerEventAmplitude      = 0
  eventAmplitudeRange      = 40
  gradeSelections          = 0
  windowSlotIndex          = 1
  rawCompressionSlotIndex  = 255
  ignoreInitialFrames      = 2
  biasAlgorithmId          = 2 2 2 2 2 2
  biasRejection            = 1 2 4 6 5 6
  fep0VideoOffset          = 1000 1000 1000 1000
  fep1VideoOffset          = 1000 1000 1000 1000
  fep2VideoOffset          = 1000 1000 1000 1000
  fep3VideoOffset          = 1000 1000 1000 1000
  fep4VideoOffset          = 1000 1000 1000 1000
  fep5VideoOffset          = 1000 1000 1000 1000
  deaLoadOverride          = 0
  fepLoadOverride          = 0
}

load 3 te 2 {
  parameterBlockName       = teBlock
  parameterBlockId         = 0x2345
```

```
  fepCcdSelect              = 0 1 2 3 4 5
  fepMode                   = 2
  bepPackingMode            = 1
  onChip2x2Summing          = 0
  ignoreBadPixelMap         = 0
  ignoreBadColumnMap        = 0
  recomputeBias             = 1
  trickleBias               = 1
  subarrayStartRow          = 0
  subarrayRowCount          = 1023
  overclockPairsPerNode     = 16
  outputRegisterMode        = 0
  ccdVideoResponse          = 0 0 0 0 0 0
  primaryExposure           = 1
  secondaryExposure         = 10
  dutyCycle                 = 0
  fep0EventThreshold        = 100 100 100 100
  fep1EventThreshold        = 100 100 100 100
  fep2EventThreshold        = 100 100 100 100
  fep3EventThreshold        = 100 100 100 100
  fep4EventThreshold        = 100 100 100 100
  fep5EventThreshold        = 100 100 100 100
  fep0SplitThreshold        = 0   0   0   0
  fep1SplitThreshold        = 0   0   0   0
  fep2SplitThreshold        = 0   0   0   0
  fep3SplitThreshold        = 0   0   0   0
  fep4SplitThreshold        = 0   0   0   0
  fep5SplitThreshold        = 0   0   0   0
  lowerEventAmplitude       = 0
  eventAmplitudeRange       = 65535
  gradeSelections           = 0
  windowSlotIndex           = 0
  histogramCount            = 0
  biasCompressionSlotIndex = 255 255 255 255 255 255
  rawCompressionSlotIndex   = 0
  ignoreInitialFrames       = 2
  biasAlgorithmId           = 2  2  2  2  2  2
  biasArg0                  = 10 10 10 10 10 10
  biasArg1                  = 0  0  0  0  0  0
  biasArg2                  = 1  1  1  1  1  1
  biasArg3                  = 0  0  0  0  0  0
  biasArg4                  = 0  0  0  0  0  0
  fep0VideoOffset           = 1000 1000 1000 1000
  fep1VideoOffset           = 1000 1000 1000 1000
  fep2VideoOffset           = 1000 1000 1000 1000
  fep3VideoOffset           = 1000 1000 1000 1000
  fep4VideoOffset           = 1000 1000 1000 1000
  fep5VideoOffset           = 1000 1000 1000 1000
  deaLoadOverride           = 0
  fepLoadOverride           = 0
}
```

```
load 4 dea 4 {
  paramBlockName           = deaBlock
  deaBlockId               = 123456
  sampleRate               = 4
  arrayDim                 = 2
  ccdId                    = 5
  queryId                  = 2
  ccdId                    = 5
  queryId                  = 2
}

load 4 window1D 4 {
  parameterBlockName       = window1D
  windowBlockId            = 45
  arrayDim                 = 3

  ccdId                    = 1
  ccdColumn                = 2
  width                    = 4
  sampleCycle              = 6
  lowerEventAmplitude      = 7
  eventAmplitudeRange      = 8

  ccdId                    = 1
  ccdColumn                = 2
  width                    = 4
  sampleCycle              = 6
  lowerEventAmplitude      = 7
  eventAmplitudeRange      = 8

  ccdId                    = 1
  ccdColumn                = 2
  width                    = 4
  sampleCycle              = 6
  lowerEventAmplitude      = 7
  eventAmplitudeRange      = 8
}

load 2 window2D 3 {
  paramBlockName           = window2D
  windowBlockId            = 50
  arrayDim                 = 3

  ccdId                    = 0
  ccdRow                   = 490
  ccdColumn                = 490
  width                    = 20
  height                   = 20
  sampleCycle              = 0
  lowerEventAmplitude      = 0
  eventAmplitudeRange      = 64535

  ccdId                    = 0
  ccdRow                   = 500
  ccdColumn                = 500
  width                    = 100
```

```
  height                    = 100
  sampleCycle               = 1
  lowerEventAmplitude       = 0
  eventAmplitudeRange       = 64535

  ccdId                     = 0
  ccdRow                    = 0
  ccdColumn                 = 0
  width                     = 1023
  height                    = 1023
  sampleCycle               = 0
  lowerEventAmplitude       = 0
  eventAmplitudeRange       = 0
}
```

READ

The command reads memory bank, address, and number of words to read and outputs a binary stream to the standard output, *stdout*, which contains the command in the format requested by the ACIS software.

read *cmdId address length*
request to read a chunk of bep memory of the indicated length at the specified address

read *cmdId* **fep** *fepId address length*
request to read a chunk of fep memory of the indicated length at the specified address

read *cmdId* **pram** *ccdId address length*
request to read a chunk of pram memory of the indicated length at the specified address

read *cmdId* **sram** *ccdId address length*
request to read a chunk of sram memory of the indicated length at the specified address

The memory bank is identified by the keywords: fep *fepId*, sram *ccdId*, and pram *ccdId*. A missing memory indicates the bep memory.

address is an hexadecimal or decimal integer indicating the memory address where to start the reading. This value is in the range 0-0xffffffff for fep and bep memory reading, and in the range 0-0xffff for sram and pram reading.

length is an hexadecimal or decimal integer indicating the number of words to read. This value is in the range 0-0xffffffff for fep and bep memory reading, in the range 0-0xffff for pram and sram reading.

Example:

read 4 0x103d87a0 40
request to read 40 32-bit words in the bep starting from address 0x103d87a0

read 1 fep 2 0x10003400 2
request to read 2 32-bit words in the fep 2 starting from address 0x10003400

read 2 pram 2 0x2340 5
request to read 5 16-bit words in the pram 2 starting from address 0x2340

read 2 sram 4 0x1fff 5
request to read 5 16-bit words in the sram 4 starting from address 0x1fff

REMOVE

This command reads the identifiers of the patches to remove and outputs a binary stream to the standard output, *stdout*, which contains the command in the format requested by the ACIS software.

remove *cmdId* **patch** *patchBlock*
request to remove the patch list contained in the *patchBlock*

Example

```
remove 1 patch {
  paramName = patches
  patchId = 1
  patchId = 2
  patchId = 8
}
```

remove three patches: 1,2,and 8 from the patch list

RESET

This command reads the name of the memory block to reset and outputs a binary stream to the standard output, *stdout*, which contains the command in the format requested by the ACIS software.

reset *cmdId* **badPixel**
request to reset the bad pixel map

reset *cmdId* **cc badColumn**
request to reset the Continuous Clocking bad column map

reset *cmdId* **te badColumn**
request to reset the Timed Exposure bad column map

cc, te, badPixel, and badColumn are keywords identifying the memory to reset. In particular, cc badColumn specifies the Continuous Clocking badColumn map, te badColumn the Timed Exposure parameter badColumn map.

Examples:

reset 4 badPixel
reset the badPixel Map

reset 6 te badColumn
reset the Timed Exposure badColumn map

reset 10 cc badColumn
reset the Continous Clocking badColumn map

START

This command reads the *slotId* containing the parameter of the science run to start and outputs a binary stream to the standard output, *stdout*, which contains the command in the format requested by the ACIS software. This command is also used to start DEA housekeeping monitor and uplink boot.

start *cmdId* **cc** *slotId*
request to start the Continuous Clocking science run whose parameters are stored in the specified slotId

start *cmdId* **cc bias** *slotId*
request to start the Continuous Clocking bias calculation whose parameters are stored in the specified slotId

start *cmdId* **dea** *slotId*
request to start the DEA Housekeeping monitor whose parameters are stored in the specified slotId

start *cmdId* **te** *slotId*
request to start the Timed Exposure science run whose parameters are stored in the specified slotId

start *cmdId* **te bias** *slotId*
request to start the Timed Exposure bias calculation whose parameters are stored in the specified slotId

start *cmdId* **uplink** *file*
request to start the uplink boot using the data stored in the indicated file

cc, te, dea, and bias are keywords identifying the parameter block to which the action is applied. In particular, cc specifies the Continuous Clocking parameter block, te the Timed Exposure parameter block, dea the DEA housekeeping parameter block.

uplink is a keyword identifying the boot mode.

Examples:

start 22 te 4
: start Timed Exposure science run with the parameters stored in slotId 4

start 33 cc 3
: start Continous Clocking science run with the parameters stored in slotId 3

start 44 cc bias 0
: start Timed Exposure bias calculation with the parameters stored in slotId 0

start 55 te bias 1
: start Continous Clocking bias calculation with the parameters stored in slotId 1

start 66 dea 2
: start Dea Housekeeping monitor with the parameters stored in slotId 2

start 77 uplink fileName
: start the uplink boot with the data contained in the fle *fileName*

STOP

This command stops science run or DEA Housekeeping monitor. It outputs a binary stream to the standard output, *stdout*, which contains the command in the format requested by the ACIS software.

stop *cmdId* **dea**
: request to stop the currently running DEA Housekeeping monitor

stop *cmdId* **science**
: request to stop the currently running science exposure

science, and dea are keywords identifying the task to stop.

Examples:

```
stop 22 science

stop 33 dea
```

WRITE

This command read a binary file, identifies the memory bank where to write the file content, and outputs a binary stream to the standard output, *stdout*, stdout , which contains the command in the format requested by the ACIS software.

write *cmdId address file*
: request to write the binary file content in the bep memory at the specified address

write *cmdId* **fep** *fepId address file*
: request to write the binary file content in the fep memory at the specified address

write *cmdId* **pram** *ccdId address file*
: request to write the binary file content in the pram memory at the specified address

write *cmdId* **sram** *ccdId address file*
: request to write the binary file content in the sram memory at the specified address

The memory bank to write is identified by the keywords fep *fepId*, pram *ccdId*, and sram *ccdId*. A missing memory indicates the bep memory.

address is an hexadecimal or decimal integer indicating the memory address where to start the reading. This value is in the range 0-0xffffffff for fep and bep memory reading, and in the range 0-0xffff for sram and pram reading.

file is the name of the file containing the data to write in memory. The file size is limited by the memory bank where the data should be stored. If the data contained in the file cannot be loaded in one command, their content will be divided into sections and loaded by multiple commands.

Examples:

write 4 0x00002345 asm
write the content of the "asm" file in the bep at the address 0x00002345

write 5 fep 4 0x00000056 bsm
write the content of the "bsm" file in the fep 4 at the address 0x00000056

write 6 pram 4 0x0024 csm
write the content of the "csm" file in the pram 4 at the address 0x0024

write 7 sram 5 0x0034 dsm
write the content of the "dsm" file in the sram 5 at the address 0x0034

SERIAL HARDWARE COMMANDS

The following commands will be prefixed by a 4-byte header that sends them to the hardware serial command port of the DPA.

HALT

This hardware serial command outputs a binary stream to the standard output *stdout*, containing the command in the format requested by the ACIS hardware.

halt bep
request to reset the BEP processor.

RUN

This hardware serial command outputs a binary stream to the standard output *stdout*, containing the command in the format requested by the ACIS hardware.

run bep
request to run the BEP processor.

SELECT

This hardware serial command outputs a binary stream to the standard output *stdout*, containing the command in the format requested by the ACIS hardware.

select EEPROM programming
request to select EEPROM programmer readout

select EEPROM telemetry
request to select software bi-level telemetry.

select bep *bepId*
request to select one of the two BEP processor (0 select BEP A, 1 select BEP B).

SET

This hardware serial command outputs a binary stream to the standard output *stdout*, containing the command in the format requested by the ACIS hardware.

set bootModifier off
request to clear the bootModifier flag

set bootModifier on
request to set the bootModifier flag

set radiationMonitor high
request to set the radiation flag

set radiationMonitor low
request to clear the radiation flag

set warmBoot off
request to clear the warm boot flag

set warmBoot on
request to set the warm boot flag

HARDWARE PULSE COMMANDS

The following commands will be sent to the command port of the Power Supply Mechanism Control (PSMC), as specified in the "*CTUE Command Format*". Each pulse command has two versions, depending on whether it is to be sent to the A-side or B-side of the redundant PSMC hardware. These are identified by the *id* parameter in the following table, where *id* must be either "0" for the A-side, or "1" for the B-side.

CLOSE

close door *id*
close instrument door *id*

close vent *id*
close vent subsystem valve *id*

close relief *id*
close vent subsystem small valve *id*

CLOSEABORT

closeabort door *id*
abort the closing of instrument door *id*

closeabort vent *id*
abort the closing of vent subsystem valve *id*

DISABLE

disable daBake *id*
disable commands to bakeout heater *id*

disable daHeater *id*
disable commands to housing heater *id*

disable dea *id*
disable commands to DEA power supply *id*

disable door *id*
disable commands to door mechanism *id*

disable dpa *id*
disable commands to DPA power supply *id*

disable pressure *id*
disable commands to pressure sensor *id*

disable relief *id*
disable commands to vent subsystem small valve *id*

disable vent *id*
disable commands to vent subsystem valve *id*

ENABLE

enable daBake *id*
enable commands to bakeout heater *id*

enable daHeater *id*
enable commands to housing heater *id*

enable dea *id*
enable commands to DEA power supply *id*

enable door *id*
enable commands to door mechanism *id*

enable dpa *id*
enable commands to DPA power supply *id*

enable pressure *id*
enable commands to pressure sensor *id*

enable relief *id*
enable commands to vent subsystem small valve *id*

enable vent *id*
enable commands to vent subsystem valve *id*

OPEN

open door *id*
open instrument door *id*

open vent *id*
open vent subsystem valve *id*

open relief *id*
open vent subsystem small valve *id*

OPENABORT

openabort door *id*
abort the opening of instrument door *id*

openabort vent *id*
abort the opening of vent subsystem valve *id*

POWEROFF

poweroff dabake *id*
power off bakeout heater *id*

poweroff daheater *id*
power off housing heater *id*

poweroff dea *id*
power off DEA power supply *id*

poweroff dpa *id*
power off DPA power supply *id*

POWERON

poweron dabake *id*
power on bakeout heater *id*

poweron daheater *id*
power on housing heater *id*

poweron dea *id*
power on DEA power supply *id*

poweron dpa *id*
power on DPA power supply *id*

TURNOFF

turnoff dabake
power off and disable both bakeout heaters

turnoff dabake *id*
power off and disable bakeout heater *id*

turnoff daheater
power off and disable both housing heaters

turnoff daheater *id*
power off and disable housing heater *id*

turnoff dea
power off and disable both DEAs

turnoff dea *id*
power off and disable DEA *id*

turnoff dpa
power off and disable both DPAs

turnoff dpa *id*
power off and disable DPA *id*

TURNON

turnon dabake *id*
enable and power on bakeout heater *id*

turnon daheater *id*
enable and power on housing heater *id*

turnon dea *id*
enable and power on DEA *id*

turnon dpa *id*
enable and power on DPA *id*

AUTHOR

Rita Somigliana, MIT CSR <rita@space.mit.edu>

10.7 cclient

NAME

cclient – send commands to a command server process on a remote host

SYNOPSIS

cclient *hostname socket_number*

DESCRIPTION

cclient sends commands to a command server process, typically *cserver*, on a remote host.

EXAMPLES

The following UNIX pipe uses *cclient* as part of commanding ACIS:

```
buildCmds < myCmdScript | sendCmds | writeCCB | cclient poplar 8000
```

AUTHOR

Ann M. Davis, MIT CSR

STATUS

The current version is working according to specification.

10.8 cserver

NAME

cserver – Read data from socket and write it to stdout

SYNOPSIS

cserver *socketNumber*

DESCRIPTION

This program waits for a connection to its socket, *socketNumber,* reads data from the connection and writes the read data to *stdout*. It continues to read and write data until the connection is broken by the client, and then waits for a new connect.

OPTIONS

None

RESTRICTIONS

This program can handle only one connection at a time.

FILES

None

SEE ALSO

cclient(1), socket(2), accept(2)

DIAGNOSTICS

Fatal Messages:

Various socket and connection error messages to *stderr*.

Warning Messages:

None

Informatory Messages:

When waiting for a connection, the program writes "Waiting for Connection" to *stderr*. Once a connection is attempted, the program writes a message indicating from which machine the attempt is being made, to *stderr*.

10.9 diff6

NAME

diff6 – compare multiple ACIS binary event files against each other

SYNOPSIS

diff6 [**–v**] *file1 file2* [*file3...file6*]

DESCRIPTION

This program reads the binary ERV event files generated by *psci*, and compares them record-by-record. If it finds a mis-match, it attempts to re-synchronize.

OPTIONS

–v run *diff6* in verbose mode, writing informatory messages to *stderr*. If omitted, only one single error message will be written at the conclusion of the program.

EXAMPLE

The following pipe uses *filterClient* to read ACIS telemetry packets from *emily*, runs them through *psci* to decommutate them, generating both data files (the **–l** flag) and monitor records (the **–m** flag). The latter are passed through *sciglue* to scramble them, and on into *monitorScience* to display (hopefully) useful statistics on an X-window display.

```
filterClient -h emily | psci -v -l run123 -m | sciglue | monitorScience
```

While *monitorScience* is displaying fancy stuff on the screen, *psci* is doing the hard work of unpacking the bias maps and event files. The latter will be written to a series of disk files with names "*run123*.*nrun*.*nfep*.erv.dat", where *run123* is a prefix inherited from the *–l* option of *psci*, *nrun* is a "science run index" that is incremented at the start of each science run, and *nfep* is the index of the particular FEP that generated the events. After *psci* has generated an end-of-run message, e.g.

```
stdin: scienceReport[54321,0] run 1 irig 12345:678 exp 2 fep ok ccd ok
      dea 0 bep 1
```

indicating that the event files have been written successfully, they may be compared by

```
diff6 -v run123.1.*.erv.dat
```

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

psci(1)

DIAGNOSTICS

- *count* **mis-matches reported**
  diff6 has found this number of mis-matches during the
- *count* **records processed from each of** *count* **files. No problemo!**
  The message we hope for, but don't always get.
- *file*[*rec,exp,row,col*] **!=** *file*[*rec,exp,row,col*]
  a mis-match has occurred: the message includes the file names, the record numbers (starting at 0), and the exposure number, row, and column recorded in each.

- *file*: **no data**
  the input file was empty.
- *file*: **premature EOF**
  One (or more) of the input files ended before at least one of the others.
- **too many input files**
  6 is the limit.

10.10 dumpring

NAME

dumpring – translate ACIS FEP ring buffer records to ASCII

SYNOPSIS

dumpring [*file*]

DESCRIPTION

This Perl script reads a FEP ring buffer file and lists its contents in ASCII on the standard output stream, *stdout*. If *file* is omitted, the input is taken from the standard input stream, *stdin*. Records are indexed by their exposure number and by the count of records of the same type generated by that exposure. All indices, including exposure numbers, start at 1. The record and field names are taken from *fepBep.h*.

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

fepBep.h

DIAGNOSTICS

unknown code *id* **for record** *nexp,nrec*
the type code in record *nrec* of exposure number *nexp* is invalid.

BUGS

• the contents of raw image rows and histograms are not reported.

10.11 fepCtlTest

NAME

fepCtlTest – simulate ACIS front end processor

SYNOPSIS

fepCtlTest [*file*]

DESCRIPTION

This command subjects the high-level ACIS front end processor software to a series of tests, directed by a script read from *file*, or, if omitted, from the standard input stream , *stdin*. All input lines are echoed to *stderr*, with the following additional output, detailing the interaction between the FEP and the (simulated) BEP:

BEP COMMAND *name* **[** *parm* **...]**
a command has been received from the simulated BEP mailbox

FEP REPLY TYPE=*type* **CODE=***retc*
the FEP is replying to a BEP *type* command with a return code *retc*.

FEP REPLY TYPE=*type* **MODE=***mode* **BIAS=***flag* **PARITY=***addr* **OCLK=(***n,n,n,n***)**
the FEP is replying to a BEP_FEP_CMD_STATUS command.

*file***:** *usec* **user** *ssec* **sys**
an image frame has been processed, using *usec* seconds of user CPU time and *ssec* seconds of system CPU time.

*file***: bias0 = {** *n, n, n, n* **}**
A bias map has been calculated. The initial overclock averages are shown.

EOF on stdin
The input script has been processed.

COMMAND LANGUAGE

dumpbias *file*
write a bias map to *file* in FITS format.

exec *cmd* execute a BEP command. The names are as defined in the IP&CL nodes, e.g. BEP_FEP_CMD_STATUS, BEP_FEP_CMD_PARAM, etc.

fidpix = *row col ...*
define one or more fiducial pixels.

fidpix = bad
issue a null-length fiducial-pixel command.

param *name* **=** *val*
set an internal parameter field. *name* is a member of the FEPparmBlock structure as defined in *fepBep.h*, i.e. *type*, *nrows*, etc.

reset *ctr*
reset an internal counter. Currently, only *wakeup* is implemented.

set *buf***[***row,col***] =** *val*
set the *row* and *col* element of *buf* (either "bias" or "biasparity") to *val*.

set input = *file*
read data from the FITS image. *file* should contain "%d" to be replaced with the exposure number.

set maxfile = *n*
set limit to input files

set output = *file*
write ring buffer to file

set overclocks = *p1,p2, ...*
specify ranges of overclock pixels

set pixels = *p1,p2, ...*
specify ranges of data pixels

set rows = *r1,r2*
specify range of input rows

stuff param *name* **=** *val*
set FEP parameter field

xor *buf***[***row,col***] =** *val*
XOR value into a buffer **Notes:** The "param" command sets fields within the parameter block that will be sent to the FEP by "exec BEP_FEP_CMD_PARAM", whereas the "stuff" command updates the field directly in the FEP's FEPparmBlock parameter block, which tests the ability of the software to detect damaged parameters after they have been loaded.

The only "ctr" is "wakeup" which causes *FIOgetNextCmd*() to simulate BEP commands arriving while a FEP mode is running.

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

dumpring(1), tlmsim(1)

DIAGNOSTICS

Bad xor command: *text*
the command must specify a legitimate row and column address

Unknown call: *name*
the entry point name is unrecognized

Unknown command: *text*
the command is unrecognized

*file***: bad bias checksum:** *n*
the bias map checksum is invalid

*file***: bad rows value:** *n* **<** *n***-1**
the row count requested is incompatible with the FITS image size

*file***: file count exceeded**
no more image files will be read in the current science run

*file***: too few rows:** *n* **<** *n*
the row count requested is incompatible with the FITS image size

10.12 fepImage2

NAME

fepImage2 – Load an image into a Unix FEP Simulation

SYNOPSIS

fepImage2 *fepId* sequencer

DESCRIPTION

This program reads a DEA to FEP pixel stream from *stdin* and feeds the image to a running Unix FEP simulation process, which is simulating the FEP indicated by *fepId*. A Unix FEP simulation process (see *acisFepUnix(1)*) must be running with the same *fepId* prior to using this program. The input for this program can be produced using several programs, including *genObjectImage* and *loadFitsImage*.

OPTIONS

None

RESTRICTIONS

This program is currently only supported for the DECstations running Ultrix.

FILES

None

SEE ALSO

acisFepUnix(1), genObjectImage(1), loadFitsImage(1), ipcs(1), ipcrm(1)

DIAGNOSTICS

Fatal Messages:

If the system attempts to write a pixel beyond the end the image memory (due to some internal error, or to a badly formed input image), the program write print the contents of some key FEP register values (from shared memory) to *stderr* and exit with a -1 exit code.

Warning Messages:

None

Informatory Messages:

If the program would be blocked by the FEP when trying to write an image, rather than discard the image, the program prints a "Dummy Image VSYNC - Sleep 3" message to *stderr* and then sleeps for 3 seconds. It will then re-attempt to load the image. This will continue indefinitely until either the FEP accepts the image, or until the program is aborted (usually via a Ctrl-C).

As the program loads the image into the FEP, it prints the current image start row, CCD start row and Row Count values used for the load to *stdout*.

10.13 filterClient

NAME

filterClient – receive ACIS telemetry from filterServer socket

SYNOPSIS

filterClient [**–D**] [**–h** *host*] [**–m** *mode*] [**–p** *port*]

DESCRIPTION

This Perl script makes a TCP connection to the specified *filterServer* port, and copies packets of type *mode* to the standard output stream, *stdout*.

OPTIONS

–D run *filterClient* in debug mode.

–h *host*
specifies the name of the host running *filterServer*. If omitted, this is assumed to be the local machine

–m *mode*
selects the type of telemetry packets to be extracted from *filterServer*. *mode* is a string of between one and four letters, as follows:

```
s  science telemetry
e  engineering pseudo-packets
p  science frame pseudo-packets
h  DEA housekeeping packets
```

If omitted, the default is **seph**, i.e. extract all types of packet.

–p *port*
specifies the port number on the *filterServer* host. If omitted, the default is 7002.

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

processScience(1)

10.14 filterServer

NAME

filterServer – Distribute ACIS telemetry to TCP clients

SYNOPSIS

filterServer [**–D**] [**–S**] [**–b** *size*] [**–l** *file*] [**–n** *num*] [**–p** *port*] [**–v**]

DESCRIPTION

This Perl script reads ACIS telemetry from its standard input and accepts connections from *filterClient* processes. It writes the buffered telemetry to each connected client, until that client disconnects. If an end-of-file occurs on its standard input, it disconnects all clients immediately.

OPTIONS

–D run *filterServer* in debug mode, writing all I/O activity to the log file (if **–l** was specified), or to *stderr* if it wasn't. **–D** implies **–v**.

–S sleep for 0.1 second after reading each input packet. This is intended to aid in debugging the *filterClient* process.

–b *size*
specifies the maximum size of each client's telemetry buffer. If a client's buffer grows larger than this amount, the socket to the client will be broken. The default is 1 MByte.

–l *file* write a list of socket activity (opens and closes) to the log file (if **–l** was specified), or to *stderr* if it wasn't. **–l** implies **–v**.

–n *num*
specify the maximum number of concurrent connections. The default is 8.

–p *port*
specify the number of the TCP port on which *filterServer* listens for connections. The default is 7002.

–v write a list of socket activity (opens and closes) to *stderr*.

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

filterClient(1)

10.15 genObjectImage

NAME

genObjectImage – generate an ACIS pixel image from an ASCII script

SYNOPSIS

genObjectImage [**–D**] [**–c** *cols*] [**–e**] [**–m** *mode*] [**–n** *count*] [**–o** *oclks*] [**–r** *rows*] [**–s** *iseed*] [**–u**] [**–v**] [*infile* [*outfile*]]

DESCRIPTION

genObjectImage reads commands from *infile* (or, if omitted, from *stdin*), and generates a stream of pixels to *outfile* (or, if omitted, to *stdout*), in a format suitable for passing to the ACIS *Image Loader*.

OPTIONS

–D for debugging purposes, writes the output in the form of a FITS image file, in the native byte ordering of the CPU, with the rows running top to bottom and the nodes, columns and overclocks from left to right, beginning with the columns and ending with the overclocks. For example, for 4 nodes (mode = ABCD), the four columns will be followed by the 4 sets of overclocks. Each set will be separated by a null pixel; two rows of null pixels will separate the columns from the overclocks. Pixel ordering within each column follows the CCD mapping order, i.e. nodes B and D are not reversed. Pixel OP codes (bits 0–3) are set to zero and VSYNC, HSYNC, and NOOP pixels are suppressed.

–c *cols*
specifies the number of columns per output node, overriding any "columns =" statement in the input script. If omitted entirely, the number depends on the "mode", 256 for ABCD, 512 otherwise.

–e instructs *genObjectImage* to write a Last Pixel Flag (LPF) code at the end of the output stream, telling the frame buffer to start loading its data into the instrument. This flag must always be specified on the last, or only invocation of *genObjectImage* to load the frame buffer.

–m *mode*
selects the CCD quadrant readout mode that is to be simulated, overriding any "mode =" statement in the input script. *mode* must be either "ABCD", "AC", or "BD" (or their lower case equivalents). If omitted entirely, the default is "ABCD".

–n *count*
commands the frame buffer to write the following data to the instrument a total of *count* times. If omitted, the frame buffer loops continuously until reset. This option can only be specified on the first of a series of invocations of *genObjectImage* when loading the frame buffer.

–o *noclks*
specifies the number of overclocks to be used per quadrant, overriding any "overclocks =" statement in the input script. If omitted entirely, the default is 4.

–r *rows*
specifies the number of rows in the output image, overriding any "rows =" statement in the input script. If omitted entirely, the default is 1024.

–s *seed*
specifies a seed integer to be used to start the random number generator, overriding any "seed =" statement in the input script. If omitted entirely, the default is the value returned by a call to the *time*() function.

–u omits the special image-loader codes. No compression is possible; *stdout* contains only valid FEP pixel codes.

–v runs *genObjectImage* in verbose mode, writing a description of each node and display object to *stderr*, and summarizing the number of pixels in the output image.

INPUT SCRIPTS

A script consists of a series of ASCII statements of the form

```
keyword = value . . .
```

followed by zero or more object "call-outs" of the form

```
object row column
```

which will be described below. The script statements are case-insensitive—upper and lower case characters may be freely mixed. Each statement must appear on a separate line, but may be surrounded by any number of blanks and tabs. Any number of spaces or tabs may separate statement arguments.

Null lines, lines containing only whitespace, and lines whose first non-whitespace character is '#', are interpreted as comments, and ignored.

The statements are grouped logically into five classes:

• Global Parameter Assignments

These statements affect the entire output image. All except **wedge** and **noop** may be overridden by options on the *genObjectImage* command line, as described above.

columns = *ival*
The number of pixels to write for each output node. This may be overridden by the **–c** option. The default value is determined by the **mode**: **256** if **ABCD**, otherwise **512**.

mode = *string*
The CCD output shift register mode to be simulated. Possible *string* values are **ABCD**, **AC**, and **BD**. This may be overridden by the **–m** option. The default value is **ABCD**.

noop = *ival where what*
The number of NOOP pixels to write between various parts of the output pixel stream. *where* may be either **before** or **after**. **what** may be either **vsync**, **hsync**, or **oclks**. That number of NOOPs will be written (for each of the 4 output nodes) before or after VSYNC or HSYNC OP codes, or before or after writing each row's overclock section.

overclocks = *ival*
The number of overclock pixels to write for each output node. This may be overridden by the **–o** option. The default value is **4**.

rows = *ival*
The number of rows of pixels to write. This may be overridden by the **–r** option. The default value is **1024**.

seed = *ival*
An integer to initialize the random number generator used when **dbias** or **doverclock** is non-zero.

wedge = *drow dcol drowcol*
Adjust all pixels in this image by adding

```
drow * irow + dcol *  icol + drowcol * irow *  icol
```

where the row index, *irow*, varies from 0 for the first row to *rows*–1 for the last, and the column index, *icol*, varies from 0 for the first pixel of node A to (*nodes* * *columns* – 1) for the first pixel of node D. This statement allows *genObjectImage* output to mimic various CCD behaviors, *e.g.*, light leaks.

• Node Blocks

A node block, a group of statements delimited by "begin node" and "end node" statements, should be specified for each CCD output node that is to be simulated. It is good practice to define all 4, A, B, C, and D, even when the **mode** is **AC** or **BD**, since you may want to override it on the command line. The

contents of the node block generates the background pixel and overclock values, on which the other display object are superimposed.

begin node = *char*

bias = *ival*
Initialize all data pixels (but not overclocks) in this node to the integer value, *ival*. This serves as a base on which to apply various adjustments.

dbias = *fval*
Adjust all pixels in this node by adding a zero-mean Gaussian random variable of variance *fval*. This mimics the real variability of CCD pixel values.

overclock = *ival*
Initialize all overclocks in this node to the integer value, *ival*. This serves as a base on which to apply various adjustments.

doverclock = *fval*
Adjust all overclocks in this node by adding a zero-mean Gaussian random variable of variance *fval*.

end node = *char*

- **Event Blocks**

Each event block, a group of statements delimited by "begin event" and "end event" statements, defines an array of integers to be added to the CCD pixel array. Each block is named, and these names are used later in the script (see "*Object Call-outs*", below) to perform this task, perhaps many times at different image locations.

begin event = *name*

columns = *ival*
The width of the **values** array.

rows = *ival*
The height of the **values** array.

values = *ival ival...ival*
A series of *columns* × rows integer values to be added to the CCD pixel image. The values are supplied in row-major order, i.e. the first row values are followed by the second row, and so on. Within a row, the values are in ascending column order.

end event = *name*

- **Blob Blocks**

A blob block, a group of statements delimited by "begin blob" and "end blob" statements, specifies an elliptical artifact to add to the CCD image. It is defined by its width, height, orientation angle, and maximum (additional) pixel value. The blob is modeled as a double Gaussian in width and height. Each block is named, and these names are used later in the script (see "*Object Call-outs*", below) to perform this task, perhaps many times at different image locations.

begin blob = *name*

angle = *fval*
The rotation angle of the blob, in degrees counter-clockwise.

height = *fval*
The Gaussian width of the blob in the column direction, before rotating through **angle**.

value = *fval*
The additional pixel value of the center of the blob.

width = *fval*
The Gaussian width of the blob in the row direction, before rotating through **angle**.

end blob = *name*

- **Object Call-outs**

  Zero or more statements of the form

  ```
  name row column
  ```

  where *name* represents a previously defined object (event or blob) which is to be added to the CCD image, centered at *row* and *column*. These call-outs are necessary for adding events and blobs—merely defining them in "begin" blocks won't affect the image because their centers are undefined.

SCRIPT EXAMPLE

```
#
# General Image Parameters
#
Rows            = 1024
Columns         = 256
Mode            = ABCD
Overclocks      = 4
Seed            = 12345678
#
# Simulated light leak
#
Wedge           = 0.001 -0.0025 0.000001
#
# Add NOOPs before and after SYNC codes
#
Noop            = 4 before VSYNC
Noop            = 4 after  VSYNC
Noop            = 2 before Oclks
Noop            = 2 before HSYNC
Noop            = 2 after  HSYNC
#
# Describe the CCD output nodes
#
Begin Node      = A
  Bias          = 230
  dBias         = 5
  OverClock     = 190
  dOverClock    = 2
End Node        = A

Begin Node      = B
  Bias          = 230
  dBias         = 5
  OverClock     = 190
  dOverClock    = 2
End Node        = B

Begin Node      = C
  Bias          = 230
  dBias         = 5
  OverClock     = 190
  dOverClock    = 2
End Node        = C
```

```
Begin Node      = D
  Bias          = 230
  dBias         = 5
  OverClock     = 190
  dOverClock    = 2
  End Node      = D
#
# Define an X-ray event
#
Begin Event     = event_1
  Rows          = 3
  Columns       = 3
  Value         = 0 0 0 0 1500 0 0 0 0
End Event       = event_1
#
# Define a split X-ray event
#
Begin Event     = event_2
  Rows          = 3
  Columns       = 3
  Value         = 0 0 0 0 1000 500 0 0 0
End Event       = event_2
#
# Define a "bloom" event
#
Begin Blob      = bloom
  Width         = 8
  Height        = 16
  Angle         = 30
  Value         = 1000
End Blob        = bloom
#
# Insert some events into the image
#
event_1         100200
event_1         475891
bloom           25129
event_2         100131
# end of script
```

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

Frame Buffer Specification, ACIS Memo, M. Doucette, MIT, October 27, 1995.

The Coordinate System of ACIS FITS Files, ACIS Memo, J. W. Woo and S. E. Kissel, MIT, December 7, 1994.

DIAGNOSTICS

- **bad blob statement:** *statement*
  unrecognized statement within a "blob" block.

- **bad blob name:** *name*
  the name in an "end blob" statement must match that in the preceding "begin blob" statement. "begin" blocks may not be nested.
- **bad event statement:** *statement*
  unrecognized statement within an "event" block.
- **bad event name:** *name*
  the name in an "end event" statement must match that in the preceding "begin event" statement. "begin" blocks may not be nested.
- **bad keyword:** *name*
  the "noop" value must be followed by "before" or "after", which must be followed in turn by either "hsync", "vsync", or "oclks".
- **bad node statement:** *statement*
  this statement is illegal within "begin node" and "end node" statements.
- **bad node:** *char*
  the node designator must be a single character in the range ’A’ through ’D’ (or equivalently ’a’ through ’d’). The node name in an "end node" statement must match that in the preceding "begin node". "begin" blocks may not be nested.
- **bad value[*n*]:** *value*
  the *n*’th item in the "values =" list of an "event" block is invalid.
- **duplicate blob:** *statement*
  object names (events and blobs) must be unique.
- **duplicate event:** *statement*
  object names (events and blobs) must be unique.
- **misplaced statement:** *statement*
  "begin" blocks may not be nested.
- **missing columns**
  the "event" block must contain a "columns =" statement.
- **missing height**
  the "blob" block must contain a "height =" statement.
- **missing rows**
  the "event" block must contain a "rows =" statement.
- **missing value**
  the "blob" block must contain a "value =" statement.
- **missing values**
  the "event" block must contain a "values =" statement.
- **missing width**
  the "blob" block must contain a "width =" statement.
- **negative value:** *statement*
  variances (*dbias* and *dover*) must be non-negative.
- **unknown statement:** *statement*
  the statement is unrecognized.
- **unknown mode:** *statement*
  the mode is unrecognized. Valid values are "ABCD", "AC", "BD", and their lowercase equivalents.

10.16 genPixelImages

NAME

genPixelImages – generate an ACIS image from a command script

SYNTAX

genPixelImages [–n] [–?]

DESCRIPTION

genPixelImages reads input commands from *stdin* and writes images to *stdout* in a format suitable for transferring to an "Image Loader". This format consists of 16 bit-words containing frame-buffer directives, FEP synchronization codes, and pixel and overclock values. Each image begins with four VSYNC codes and may contain from 1 to 1024 "rows", each beginning with four HSYNC codes. Each row may contain between 4 and 1024 "columns", divided into "nodes" (four in "ABCD" mode, two in either "AC" or "BD" mode), and followed by 0 to 15 pairs of overclocks per node. Note that the fourth, diagnostic, clocking mode generates no pixel values. It is therefore simulated by "ABCD" mode and no separate *genPixelImages* option is required.

Before generating any output, *genPixelImages* verifies (1) that the dimension of the images (number of rows, columns, and overclocks) are within ACIS constraints; (2) that the number and position of pixels and overclocks correspond to the image requested, i.e. each row contains the required number of pixels, followed by the required number of overclocks; (3) that each *repeatRowBlock* command operates on one or more whole rows.

OPTIONS

–n do not reorder the image bytes.

–? print option list

COMMAND SCRIPTS

The *stdin* commands consist of lines of ASCII text obeying a formal grammar. Each output image is described by a statement of the form

row *n* **col** *m* **overclock** *o mode pixelDefinition*

where *n*, *m*, and *o* are integers (*m* and *o* must be even) and mode specifies the CCD readout node configuration-ABCD, AC, or BD. These are followed on the same line by *pixelDefinition*- a series of commands that define the pixels and overclocks that will comprise the image.

The basic commands are

p *v*

which defines a single pixel with value *v* (an integer between 0 and 4095), and

c *v*

which defines a single overclock with value *v* (also an integer between 0 and 4095). A pixel or overclock value may be repeated by preceding the *p* or *c* command by *r*, e.g.

r *n* **p** *v*

generates a sequence of *n* pixels, each of value *v*. *r*, *p*, and *c* commands may be grouped by enclosing them in parentheses and repeated by prefixing the group with a **repeatSec** command, e.g.

repeatSec *i* **(** **r** *n* **p** *v* **r** *m* **c** *v* **)**

repeats the commands within the parentheses a total of *i* times. This process continues until one or more full rows have been defined. A group of rows may be repeated with the **repeatRowBlock** command, e.g.

```
repeatRowBlock n [
  repeatSec n ( r n p v r n p v )
  repeatSec n ( r n p v r n c v )
]
```

Note that the *repeatSec* commands are grouped by enclosing them in square brackets. Similarly, *repeatRowBlock* commands may be grouped by enclosing them within braces, e.g.

```
row 200 col 1024 overclock 4 ac {
  repeatRowBlock 50 [
    repeatSec 2 (
      r 100 p 1 r 100 p 2 r 100 p 3 r 100 p 4 r 100 p 5
      r 100 p 6 r 100 p 7 r 100 p 8 r 100 p 9 r 100 p 10
      r 24 p 11 c 30 c 31 c 32 c 33
    )
  ]
  repeatRowBlock 50 [
    repeatSec 2 ( r 100 p 20 r 100 p 30 r 824 p 40 r 4 c 34 )
  ]
}
```

The last image must be followed by an **end** command. Blank lines, and all text between a "#" character and the end of that line, will be treated as comments and ignored.

genPixelImages locates each pixel in the 2-dimensional coordinate system of *n* rows and *m* columns defined by the physical CCD image store. Since the pixels are simultaneously sampled by 2 or 4 output nodes, *genPixelImages* must rearrange the pixel order according to the output node configuration. It also outputs VSYNC codes before each image, HSYNC codes before each row, and, only in AC or BD readout mode, a "null" pixel between each data pixel. It can also be instructed to add "null" pixels before or after synchronization codes, e.g.

```
row 8 col 8 overclock 4 delay vsync before 3 abcd [
  repeatSec 2 ( r 8 p 1 r 4 c 10 )
  repeatSec 3 ( r 4 p 2 r 4 p 8 r 4 c 30 )
  repeatSec 3 ( r 8 p 3 r 4 c 20 )
]
```

If *genPixelImages* is invoked with the single command **go**, it instructs the Frame Buffer to keep sending to the FEPs the image or images currently stored in its buffer, until commanded to halt. Alternatively, the image definitions may be prefixed with **repeatFile** *n* to repeat the images exactly *n* times. If this is omitted, the images will be repeated until the Frame Buffer is commanded to halt,

```
( genPixelImages <<EOF
repeatFile 1 row 8 col 8 ac {
  repeatRowBlock 8 [
    repeatSec 1 ( p 1 p 2 p 3 p 4 p 5 p 6 p 7 p 8 )
  ]
}
end
EOF
) | putImages
```

DETAILED EXAMPLES

The reported examples show the correspondence between input commands and pixel image represented by them, and illustrate the available command to control the transfer of the image to the "Frame Buffer".

Pixel Pattern

This example shows the use of pixel and overclock patterns with and without repetitions to define pixel image.

```
row 3 col 4 overclock 4 abcd (
  p 100 p 200 p 250 p 200      r 4 c 120
  r 4 p 240                    r 4 c 130
  r 4 p 150                    c 100 c 110 c 120 c 130
)
```

Image size: 3x4 + 4 overclocks. Image values row by row:

```
100  200  250  200    120 120 120 120
240  240  240  240    130 130 130 130
150  150  150  150    100 110 120 130
```

repeatSec

This example shows the use of *repeatSec* to define rows.

```
row 9 col 8 overclock 4 abcd [
  repeatSec 3 (r 8 p 40 r 4 c 15)
  repeatSec 2 (r 1 p 100 r 1 p 150 r 1 p 100 r 1 p 10)
  repeatSec 1 (r 4 c 8)
  repeatSec 2 (r 1 p 120 r 1 p 250 r 1 p 120 r 1 p 10)
  repeatSec 1 (r 4 c 8)
  repeatSec 2 (r 1 p 120 r 1 p 250 r 1 p 120 r 1 p 10)
  repeatSec 1 (r 4 c 8)
  repeatSec 3 (r 8 p 40 r 4 c 6)]
end
```

Image size: 9x8 + 4 overclocks per row.

Image values row by row:

```
  40  40  40  40  40  40  40  40   6  6  6  6
  40  40  40  40  40  40  40  40   6  6  6  6
  40  40  40  40  40  40  40  40   6  6  6  6
 100 150 100  10 100 150 100  10   8  8  8  8
 120 250 120  10 120 250 120  10   8  8  8  8
 100 150 100  10 100 150 100  10   8  8  8  8
  40  40  40  40  40  40  40  40   6  6  6  6
  40  40  40  40  40  40  40  40   6  6  6  6
  40  40  40  40  40  40  40  40   6  6  6  6
```

repeatRowBlock

This example shows the use of *repeatRowBlock* to generate images containing multiple equal blocks of rows

```
row 14 col 8 overclock 2 ac {
repeatRowBlock 2 [
  repeatSec 2 (r 8 p 10 r 2 c 8)
  repeatSec 2 (r 1 p 100 r 1 p 150 r 1 p 100 r 1 p 10)
  repeatSec 1 (r 2 c 10)
  repeatSec 2 (r 1 p 120 r 1 p 200 r 1 p 120 r 1 p 10)
  repeatSec 1 (r 2 c 12)
```

```
  repeatSec 2 (r 1 p 100 r 1 p 150 r 1 p 100 r 1 p 10)
  repeatSec 1 (r 2 c 10)
  repeatSec 2 (r 8 p 10 r 2 c 8) ]
repeatRowBlock 2 [
  repeatSec 1 (r 8 p 10 r 2 c 8)
  repeatSec 2 (r 1 p 100 r 1 p 150 r 1 p 100 r 1 p 10)
  repeatSec 1 (r 2 c 10)
  repeatSec 2 (r 1 p 120 r 1 p 200 r 1 p 120 r 1 p 10)
  repeatSec 1 (r 2 c 10)
  repeatSec 2 (r 1 p 100 r 1 p 150 r 1 p 100 r 1 p 10)
  repeatSec 1 (r 2 c 10)
  repeatSec 2 (r 8 p 10 r 2 c 8) ]
}
end
```

The image size: 10x8 + 2 overclocks

The image values row by row:

```
  8   8   8   8   8   8   8   8    8  8
  8   8   8   8   8   8   8   8    8  8
100 150 100  10 100 150 100  10   10 10
120 200 120  10 120 200 120  10   10 12
100 150 100  10 100 150 100  10   12 10
  8   8   8   8   8   8   8   8    8  8
  8   8   8   8   8   8   8   8    8  8
  8   8   8   8   8   8   8   8    8  8
  8   8   8   8   8   8   8   8    8  8
100 150 100  10 100 150 100  10   10 10
120 200 120  10 120 200 120  10   10 12
100 150 100  10 100 150 100  10   12 10
  8   8   8   8   8   8   8   8    8  8
  8   8   8   8   8   8   8   8    8  8
```

Delay

This example show the use of *delay* to introduce time delays before beginning to image transfer.

```
row 8 col 8 overclock 4 delay vsync before 3 after 2 abcd [
  repeatSec 2 (r 8 p 100           r 4 c 10)
  repeatSec 3 (r 4 p 200 r 4 p 100 r 4 c 15)
  repeatSec 3 (r 8 p 150           r 4 c 120) ]
end
```

The image loading is delayed by the insertion of three NULL pixels before and two after the VSYNC code. No delay is introduced between rows.

repeatFile

This example shows the use of *repeatFile* to send the same file multiple times to the "Frame Buffer".

```
repeatFile 2 row 22 col 8 overclock 4 abcd {
repeatRowBlock 2 [
  repeatSec 2 (r 8 p 1 r 4 c 10)
  repeatSec 3 (r 4 p 2 r 4 p 3 r 4 c 30)
  repeatSec 3 (r 8 p 4 r 4 c 20) ]
repeatRowBlock 3 [
  repeatSec 2 (r 8 p 5 r 4 c 33)]
```

```
}
end
```

The output image is preceded by a keyword requesting the "Frame Buffer" to repeat the loading of the same image twice.

AUTHOR

Rita Somigliana, MIT CSR <rita@space.mit.edu>

SEE ALSO

genObjectImage(1), loadFitsImage(1), putImages(1)

10.17 getPackets

NAME

getPackets - read AXAF-I telemetry frames and extract ACIS-related information

SYNOPSIS

`getPackets`

DESCRIPTION

getPackets reads a stream of AXAF-I telemetry frames from its standard input and writes a stream of ACIS telemetry packets and pseudo-packets to its standard output. *getPackets* typically receives AXAF-I telemetry frames from *SHIM*. It extracts ACIS-related information and passes it to its client, which is typically *filterServer*. *getPackets* processes only AXAF-I telemetry formats 1 and 2. It accepts but ignores all other formats. It must first identify the current telemetry format — either 1 or 2. In the former, ACIS science data is being generated at 512 bits/sec; in the latter at 24 Kbits/sec. In both cases, *getPackets* assembles the serial telemetry from ACIS, locates the individual packets by their synch words and lengths, and writes them to *filterServer* as a stream of separate logical records. *getPackets* also sends *filterServer* two types of "pseudo-packet", i.e. records whose format mimics genuine ACIS telemetry packets but whose Format Tag codes are distinguished from those used by the instrument itself.

One type, the Science Frame Pseudo-Packet, contains telemetry-generated and ACIS-generated timestamps.

The other, the Engineering Pseudo-Packet, contains ACIS and other AXAF-I engineering data that resides in the non-science areas of the telemetry frames. *getPackets* determines the content of the engineering pseudo-packet by reading a file at run time. This file specifies the locations in the AXAF-I telemetry frames of the telemetry mnemonics that *getPackets* will extract and write as engineering pseudo-packets. The **FILES** section below explains how to specify which file *getPackets* will use.

The interface between getPackets and its various clients (via the *filterServer/filterClient* combination) may be described as a single stream of binary bytes, with no timing constraints. *getPackets* writes a stream of telemetry packets and pseudo-packets to standard output. These are passed through *filterServer* to each *filterClient,* which writes a user-selected sub-set to its standard output. The four packet types that *getPackets* writes are: "ACIS Science Packets", "ACIS Analog Housekeeping Packets", "Science-Frame Pseudo-Packets", and "Engineering Pseudo-Packets". Each packet consists of a telemetry header followed by application data, as defined in the IP&CL. The header formats and contents are defined in Table 26.

TABLE 26. Header Format and Content

Packet Header Fields	Field Length (bits)	Science or Housekeeping Packet	Science Frame Pseudo-Packet	Engineering Pseudo-Packet
Synch	32	0x736f4166	0x736f4166	0x736f4166
Length	10	Varying[1]	7	Varying[1]
Format Tag	6	Varying	62	61
Sequence Number	16	incremented by 1 for each packet in the telemetry stream	0[2]	0[2]

1. The packet length (number of 32 bit words) varies with the contents.
2. The sequence number of a pseudo-packet is always zero.

All fields are written in "little-endian" format, e.g. the packet synch word, 0x736f4166, is written as 4 bytes, 0x66, 0x41, 0x6f, and finally 0x73. The contents of all packets originating within ACIS are defined in the

IP&CL. The data portion of the Science Frame pseudo-packet is described in Table 27 and that of the Engineering pseudo-packet in Table 28. .

TABLE 27. Science Frame Pseudo-Packet Format and Content

Field Name	Source			*filterClient* Output Format	Description
	Location	Start	Length		
`format`	Virtual Channel ID	bit 10	3 bits	unsigned int	Frame format identifier, either 1 (signifying 512 bps) or 2 (24 kbps), *i.e.* the AXAF *tlm* code + 1.
`major-FrameId`	CCSDS Header	bit 16	17 bits	unsigned int	Virtual Channel Data Unit Major Frame Count (0 to 131071)
`minor-FrameId`	CCSDS Header	bit 33	7 bits	unsigned short	Virtual Channel Data Unit Major Frame Count (0 to 127)
`irigb`	Science Header	byte 32	6 bytes	unsigned short [3]	Time (msec) from the IRIG-B interface
`bepSciTime`	Science Data	byte 56	4 bytes	unsigned int	Latched version of the BEP science pulse 1 MHz timestamp
	Next-in-line Data	TBD			

TABLE 28. Engineering Pseudo-Packet Format and Content

Field Name	Source			*filterClient* Output Format	Description
	Location	Start	Length		
`format`	Virtual Channel ID	bit 10	3 bits	unsigned int	Frame format identifier, either 1 (signifying 512 bps) or 2 (24 kbps), *i.e.* the AXAF *tlm* code + 1.
`majorFrameId`	CCSDS Header	bit 16	17 bits	unsigned int	Major Frame Counter (0 to 131071)
followed by an array of one or more elements, each consisting of the following fields					
`data`	Variable location within major frame	var	8 bits	unsigned char	Engineering data
`minorFrameId`	CCSDS Header	bit 33	7 bits	unsigned char	Virtual Channel Data Unit Frame Counter (0 to 127)
`minorFrame-Byte`				unsigned short	Byte number in the minor frame (0 to 1024)

The 6-byte IRIG-B timestamp in the AXAF-I minor frame is the result of packing 4 separate bit fields into a 48-bit string. However, *getPackets* treats the 6 bytes of the IRIG-B timestamp as 3 unsigned 16-bit integers. It copies them into the Engineering Pseudo-Packet and converts them to little-endian format, which is the ACIS standard. Table 29 describes how to decipher the Engineering Pseudo-Packet's *irigb* field.

Each packet will be written to the *getPackets* standard output stream as soon as the last data byte that contributes to it is read from *SHIM*. A Science Frame Pseudo-Packet will be written after the last byte of each

TABLE 29. IRIG-B Field Format and Contents

Field Name	Bit Length	Byte	Word
Julian Day	11	0,1	0
Seconds	17	1,2,3	0,1
Milliseconds	10	3,4	1,2
Microseconds (always zero)	10	4,5	2

complete minor frame that contains a science data area header has been read. An Engineering Pseudo-Packet will be written after the last byte of each complete major frame is read.

EXAMPLES

The following UNIX pipe uses *getPackets* as part of commanding the ACIS instrument:

```
... | buildCmds | sendCmds | shim lrctu | getPackets | filterServer ...
```

FILES

$ACISTOOLSDIR The path name of the ACIS Test Tools directory. If this variable does not exist in the user's environment, *getPackets* will exit.

$ACISTTMFILE The name of the file that specifies the locations in the AXAF-I telemetry frames of the telemetry mnemonics that *getPackets* will extract and write as engineering pseudo-packets. If this variable exists in the user's environment, *getPackets* will attempt to open **$ACISTOOLSDIR/lib/$ACISTTMFILE**. If it doesn't, *getPackets* will attempt to open **$ACISTOOLSDIR/lib/acisEng.ttm** instead.

AUTHOR

Demitrios Athens, MIT CSR

STATUS

The current version should work with either the CTUE or the LRCTU (if its telemetry output is passed through *ltp2mnf*).

The current version is only capable of processing Format 2 telemetry. Its behavior if fed any other format is unknown but is liable to be objectionable.

The current version is implemented by having *getPackets* fork into 2 processes. One process, the original one as it turns out, reads AXAF-I telemetry and produces pseudo-packets. It also pipes the ACIS telemetry packet stream to the other process, which assembles and writes the ACIS packets. Both processes change their *argv*[0] environment value to either "axafFrames" or "acisPackets" as appropriate. This fools ps(1) into reporting the new name of each process. However, top(1) reports both as "getPackets."

10.18 lcmd

NAME

lcmd – list contents of ACIS command stream

SYNOPSIS

lcmd [**–B**indent] [**–V**] [**–lc**] [**–lh**] [**–lp**] [**–r**] [**–v**] [**–w** *cols*] [*file ...*]

DESCRIPTION

This Perl script reads one or more ACIS command files and writes a formatted listing to the standard output stream, *stdout*. If no input file is specified, *lcmd* reads the standard input stream, *stdin*.

The output consists of a series of "*keyword = value*" lines, in which the keywords are derived from the IP&CL field names, translating to uppercase all characters that immediately follow an embedded blank, and then removing the blank, e.g. "command length" is written as "commandLength". The "*value*" fields are rendered in decimal, with the following exceptions • 32-bit addresses and block IDs are in hexadecimal, prefixed with "0x"; • command tags are displayed as enumerated names, e.g. "CMDOP_LOAD_1D", followed by their decimal value in parentheses; • the timed-exposure "gradeSelection" value is written as a series of 8-character hexadecimal fields.

Each command begins with a header line of the form "*name[n] = {*", and ends with a matching right brace. Command blocks are separately numbered, starting at zero.

OPTIONS

–B *n* inserts *n* blank characters at the start of each output line—used by *ltlm* when invoking *lcmd* internally. The default is to insert no blanks.

–V write two lines to *stderr* reporting the RCS update level of this command and of the IP&CL structure description to which it refers.

–lc does no processing; merely lists the names of ACIS software serial command blocks, their "commandIdentifier" codes, and decimal equivalents.

–lh does no processing; merely lists the names of ACIS hardware serial commands, their IP&CL mnemonics, and hexadecimal equivalents.

–lp does no processing; merely lists the names of ACIS high-level pulse commands, their IP&CL mnemonics, and their decimal and hexadecimal RCTU channel values.

–r *raw* mode—the commands are not assumed to be prefixed by 4-byte headers, two 16-bit binary integers containing the command type and channel number, respectively.

–v becomes *very* verbose, listing the values of all large arrays. Otherwise *lcmd* will only list the number of elements in each array.

–w *cols*
sets the column width—array values that exceed this width will be continued on the next output line. The default value is 73.

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

The Web page "*http://acis.mit.edu/ipcl*", which tabulates ACIS command and telemetry packet formats.

DIAGNOSTICS

bad commandOpcode: *code* **in packet** *n*
the "commandOpcode" in command *n* is illegal. *lcmd* lists only the command header.

10.19 lerv

NAME

lerv – display contents of binary ERV event data files

SYNOPSIS

lerv [**–N**] [**–V**] [*file*]

DESCRIPTION

This Perl script reads an ERV event file, e.g. one generated by *psci*, and writes the contents of each record in ASCII to *stdout*. The ERV record data fields are as follows:

```
typedef struct {
   unsigned short expnum;     /* exposure number [5] */
   unsigned short exposure;   /* exposure time (msec) [5] */
   unsigned long  irigtime;   /* IRIG timestamp [8] */
   unsigned short nodenum;    /* output node index [1] */
   unsigned short col;        /* column index [4] */
   unsigned short row;        /* row index [4] */
   unsigned short data[9];    /* event data values [4] */
   short          doclk;      /* delta overclock [2] */
} RvRec;
```

They are written out as decimal numbers, right-justified with blank fill and separated by single blanks. The numbers in square brackets in the above definitions represent the *minimum* field widths. If the ASCII value of a particular field exceeds this, the remaining fields will be shifted right in the output line to accommodate it. Thus, although most output lines will contain 80 characters followed by a newline, some could be longer.

OPTIONS

–N interpret the input field as big-endian (MSB first) integers. Use this flag to read a MSB event file, e.g. one created on a Sun, on a LSB machine, e.g. a DecStation.

–V interpret the input field as little-endian (LSB first) integers. Use this flag to read a LSB event file, e.g. one created on a DecStation, on a MSB machine, e.g. a Sun.

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

psci(1)

10.20 lhuff

NAME

lhuff – display contents of binary Huffman table block

SYNOPSIS

lhuff [**–N**] [**–V**] [*file*]

DESCRIPTION

This Perl script reads a binary Huffman table block, e.g. as created by *psci*, and writes the contents in ASCII to *stdout*.

OPTIONS

–N interpret the input field as big-endian (MSB first) integers. Use this flag to read a MSB Huffman file, e.g. one created on a Sun, on a LSB machine, e.g. a DecStation.

–V interpret the input field as little-endian (LSB first) integers. Use this flag to read a LSB Huffman file, e.g. one created on a DecStation, on a MSB machine, e.g. a Sun.

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

psci(1)

DIAGNOSTICS

- *file*: **bad table header**
  one of the 4-byte offsets at the beginning of *file* points beyond the end of the file.
- *file*: **bad table length:** *n*
  the length of a single table exceeds that of the remainder of the file.

10.21 loadFitsImage

NAME

loadFitsImage – convert FITS file to FEP image load format

SYNOPSIS

loadFitsImage [**–a**] [**–m** *mode*] [**–n** *noclks*] [**–o** *col1,...,col8*] [**–p** *col1,...,col8*] [**–r** *nrows*] [**–s** *row1*] [**–v**] [*file ...*]

DESCRIPTION

This command loads one or more FITS images and reformats them for an ACIS Image Loader, rearranging the pixels and overclocks according to the simulated CCD readout mode specified by the **–m** option. If the list of files is omitted, a single FITS image will be read from the standard input stream, *stdin*.

OPTIONS

–a determine the image dimensions and the location of the valid pixels and overclocks from the FITS header keywords alone. When the **–v** flag is specified, the values of the **–n**, **–o**, **–p**, **–r**, and **–s** options are copied to *stderr*.

–m *mode*
selects the CCD quardant readout mode that is to be simulated. *mode* must be either *abcd*, or *ac*, or *bd*.

–n *noclks*
specifies the number of overclocks to be used per quadrant. At least this number must be available in each of the input FITS files.

–o *col1,col2,col3,col4,col5,col6,col7,col8*
selects the column limits of overclocked pixels in the 4 quadrants of the input FITS files. The pixels from quadrant A are located in columns *col1* through *col2* of each line, those of quadrant B in columns *col3* through *col4*, etc. It is assumed that the input pixel order is as in the ACIS Memo cited below.

–p *col1,col2,col3,col4,col5,col6,col7,col8*
selects the column limits of good pixels in the 4 quadrants of the input FITS files. The first column is indexed 0. The pixels from quadrant A are located in columns *col1* through *col2* of each line, those of quadrant B in columns *col3* through *col4*, etc. It is assumed that the input pixel order is as in the ACIS Memo cited below.

–r *nrows*
specifies the number of output rows to write, each of which will be followed by a HSYNC code.

–s *row1*
specifies the index of the first row in the input image (counting from 0) to write to the output.

–v puts *loadFitsImage* into verbose mode, writing statistics to *stderr*, detailing the number of rows and columns of each output image.

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

The Coordinate System of ACIS FITS Files, ACIS Memo, J. W. Woo and S. E. Kissel, December 7, 1994.

DIAGNOSTICS

EOF while reading FITS header
the input probably isn't in FITS format.

I*c*MAXCOL: header keyword missing
the **–a** option has been specified, but the FITS header doesn't contain sufficient information to specify the pixel locations.

I*c*MINCOL: header keyword missing
the **–a** option has been specified, but the FITS header doesn't contain sufficient information to specify the pixel locations.

bad pixel range in quadrant *val*
the number of pixels in each quadrant is not the same.

bad BITPIX value: *val*
the value of the FITS header keyword BITPIX must be 16.

bad NAXIS1 value: *val*
the value of the FITS header keyword NAXIS1 must be not less than 1024+4×*noclks*, where *noclks* is as specified in the **–n** option.

bad NAXIS2 value: *val*
the value of the FITS header keyword NAXIS2 must be not less than *nrows*+*row1*, where *nrows* is as specified by the **–r** option, and *row1* by the *–s* option.

bad overclock offsets: *list*
the overclock column *list* must contain exactly 8 unsigned integers, separated by commas, without any other whitespace or punctuation.

bad pixel offsets: *list*
the pixel column *list* must contain exactly 8 unsigned integers, separated by commas, without any other whitespace or punctuation.

unable to allocate input buffer
the value of the FITS header keyword NAXIS1 is too large and *loadFitsImage* is unable to allocate a buffer to hold the input record.

unexpected EOF in record *val*
an end-of-file was encountered while reading record *val* of the input file, indicating that the file is truncated, or that one or more FITS header keywords are incorrect.

*name***:** *n* **lines, each** *n* **pixels +** *n* **oclks**
generated by the **–v** verbose flag, this indicates that the FITS file *name* generated a number of output lines, each containing the specified number of pixels and overclocks.

BUGS

On-chip summation is not currently implemented. It is assumed that all output nodes generate either 256 pixels (ABCD mode) or 512 pixels (AC or BD mode).

10.22 logGet

NAME

logGet – get CTUE command abort log messages

SYNOPSIS

logGet

DESCRIPTION

logGet listens on the TCP/IP port number that the CTUE will try to connect to for sending command abort log messages. It prints any messages it receives on its standard output.

If *logGet* is run with root priviledges, it will listen on port 542 for a connection from the CTUE. If it is run with regular user priviledges, *logGet* will listen on port 7542. The appropriate port number must be entered in the CTUE's **c:\windows\ethernet.cfg** file. *logGet* adds its own time stamp to each message.

EXAMPLES

The following UNIX command uses *logGet* as part of commanding ACIS:

```
acisEgseHost% logGet
```

AUTHOR

Ann M. Davis, MIT CSR

Demitrios Athens, MIT CSR

STATUS

The current version is working according to specification.

10.23 ltlm

NAME

ltlm – list contents of ACIS telemetry stream

SYNOPSIS

ltlm [**–E**] [**–V**] [**–e** *type*] [**–f** *from*] [**–lc**] [**–lt**] [**–p** *type*] [**–s**] [**–t** *to*] [**–v**] [**–w** *cols*] [*file ...*]

DESCRIPTION

This Perl script reads one or more ACIS telemetry files and writes a formatted listing to the standard output stream, *stdout*. If no input file is specified, *ltlm* reads the standard input stream, *stdin*.

The output consists of a series of "*keyword = value*" lines, in which the keywords are derived from the IP&CL field names, translating to uppercase all characters that immediately follow an embedded blank, and then removing the blank, e.g. "command length" is written as "commandLength". The "*value*" fields are rendered in decimal, with the following exceptions • synch codes, 32-bit addresses, timestamps and block IDs are in hexadecimal, prefixed with "0x"; • command and packet tags are displayed as enumerated names, e.g. "TTAG_CMD_ECHO", followed by their decimal value in parentheses; • the timed-exposure "gradeSelection" value is written as a series of 8-character hexadecimal fields.

Each telemetry packet and embedded command block begins with a header line of the form "*name[n] = {*", and ends with a matching right brace. Each type of command block and telemetry packet is separately numbered, starting at zero. Embedded arrays of data structures are also numbered from zero, but the index is reset at the start of each array instance.

OPTIONS

–E write two lines to perform extended error checking on ACIS packets. Currently, the only test is to verify that the packet sequence numbers are in ascending order. Missing or out-of-sequence packets are noted by messages to *stderr*.

–V write two lines to *stderr* reporting the RCS update level of this command and of the IP&CL structure description to which it refers.

–e *type*
excludes output of those packets whose decimal "formatTag" value is *type*. Allowed values are tabulated by invoking *ltlm* with the **–lt** flag. Multiple **–e** options may be specified to exclude more than one type of packet.

–f *from*
don't list packets with "sequenceNumber" values less than *from*.

–lc does no processing; merely list the names of ACIS command blocks, their "commandIdentifier" codes, and decimal equivalents.

–lt does no processing; merely list the names of ACIS telemetry packets, their "formatTag" codes, and decimal equivalents.

–p *type*
restricts output to those packets whose decimal "formatTag" value is *type*. Allowed values are tabulated by invoking *ltlm* with the **–lt** flag. Multiple **–p** options may be specified to list more than one type of packet.

–s print only a tally of packet statistics: the number of packets, the packet type, and its TTAG format code.

–t *to* don't list packets with "sequenceNumber" values greater than *to*.

–v becomes *very* verbose, listing the values of all large arrays within the telemetry packets. Otherwise *ltlm* will only list the number of elements in each array.

–w *cols*
sets the column width—array values that exceed this width will be continued on the next output line. The default value is 73.

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

The Web page "*http://acis.mit.edu/ipcl*", which tabulates ACIS command and telemetry packet formats.

DIAGNOSTICS

bad commandOpcode: *code* **in packet** *n*
the "commandOpcode" in packet *n* is illegal. *ltlm* lists only the command header.

bad formatTag: *tag* **in packet** *n*
the format tag in the header of packet *n* is illegal. *ltlm* ignores this packet and reads the next.

decompression unimplemented in packet *n*
this version of *ltlm* cannot list the values of compressed pixels in "dataCcRaw", "dataTeBiasMap", or "dataTeRaw" packets.

10.24 monitorScience

NAME

monitorScience – display ACIS run-time information on an X11 server

SYNTAX

`monitorScience`

DESCRIPTION

monitorScience creates a multi-window display, reads the *stdout* stream from *processScience*, and writes it into one of the windows.

monitorScience uses colors to signal various conditions: fatal anomalies are shown in red, warnings in blue. The initiation and termination of science or monitor requests are displayed in green. Colors are also used to distinguish decimal numbers (black) from hexadecimal numbers (brown).

Black and brown colors are also used to facilitate the reading of parameter names. In science data windows, they distinguish telemetry packet names (black) from parameter names (brown). In the Dea Housekeeping window, packet header fields are colored brown, and data fields black (see below). In Sw Housekeeping, packet header fields are brown, data fields are black, red, or blue, depending on the type of information displayed.

Normally, window backgrounds are colored "linen", but background of the most recently updated window is colored "white".

WINDOW LIST

Echo Command Packets
This window displays the command name, arrival time, execution time, and command ID of all commands echoed from the ACIS instrument. The list shows what commands were sent to the BEP, and the order in which they were executed.

Science General Parameters
This window displays information about any BepStartupMessage","Dump Parameter", and "Science Report" telemetry packets. It identifies the most recent science parameter block, and the result of the most recent science run.

Science Data
This window displays information about the most recent science and bias telemetry packets received from ACIS. The window is divided vertically into three regions -- two for science data and one for bias data -- and horizontally into six columns, one for each FEP. The first science data packet is displayed in the top window, the next in the middle window, the third in the top window again, and so on.

Fatal Messages
This window displays the time at which a fatal error was detected by the BEP and the corresponding error code and value.

Read and Write
This window displays information about the execution of write-to-memory commands, i.e. "Read Bad Columns" (TE and CC), "Read Bad Pixels", "Read System Configuration", "Read Patches", "Read Huffman Tables", and "Read Parameter Slots" (TE, CC, 2D window, 1D window, and DEA).

Sw Housekeeping
This window displays information about "Software Housekeeping" telemetry packets in the form of a table containing rows of two types. One contains the start and end times of the housekeeping collection period, the other contains the name, repetition times, and value of each monitored parameter.

Dea Housekeeping

This window displays information about "Dea Housekeeping" telemetry packets in the form of a table containing rows of two types. One contains the "Dea Block id" and "Command Id" of the most recent "Start Dea Housekeeping" command and the BEP time corresponding to the most recent housekeeping packet. The other contains monitor information:; the "Board Id", "Query Id" name and value.

ENVIRONMENT VARIABLES

ACISTOOLSDIR

must point to the top-level test directory within which the *monitorScience* modules are located, e.g. "*~acis/toolstest*".

TCL_LIBRARY

must point to the TCL run-time library, e.g. "*$ACISTOOLSDIR/src/Other/tcl7.4/lib/tcl7.4*".

TK_LIBRARY

must point to the TK run-time library, e.g. "*$ACISTOOLSDIR/src/Other/tk4.0/lib/tk4.0*".

AUTHOR

Rita Somigliana, MIT CSR <rita@space.mit.edu>

10.25 processDEAhkp

NAME

processDEAhkp – extract DEA query response data from a DEA housekeeping ACIS telemetry packet

SYNOPSIS

processDEAhkp -f outFileName [*filename...*]

DESCRIPTION

processDEAhkp reads DEA housekeeping ACIS telemetry packets either from the standard input or from one or more files specified as command line arguments. It rewrites the **outFileName** whenever it reads a DEA housekeeping ACIS telemetry packet from its input.

For each query response that the DEA housekeeping packet contains, *processDEAhkp* writes to the **outFileName** the *ccdId*, *queryId*, and *value*. The output is in ASCII and is the decimal value of the item in the packet. A blank delimits the end of each of the first two items and a newline delimits the end of the last item. *processDEAhkp* expects its input to consist of DEA housekeeping ACIS telemetry packets. It silently ignores any other packets in its input. It also silently ignores the other elements of the DEA housekeeping ACIS telemetry packet: *deaBlockId*, *commandId*, and *bepTickCounter*. *processDEAhkp* always removes any existing contents of **outFileName** when starting.

EXAMPLES

The following UNIX pipe uses *processDEAhkp* as part of monitoring DEA housekeeping information:

```
filterClient -h | processDEAhkp -f DEAhkp | monitorDEAhkp -f DEAhkp ...
```

AUTHOR

Demitrios Athens, MIT CSR <da@space.mit.edu>

STATUS

The current version has been been tested but not used extensively.

The current version does not ensure that its first argument is "-f" but simply assumes that its second argument, if present, is the *outFileName*. If the "-f" option is missing and *processDEAhkp* is to read its input from a file, then the contents of the first input file will be lost.

10.26 psci

NAME

psci – parse an ACIS serial telemetry stream

SYNOPSIS

psci [**–DVacmpqsuv**] [**–h** *file*] [**–l** *prefix*] [*file*]

DESCRIPTION

This program reads a stream of ACIS packets, verifies their format and internal consistency, and optionally, sorts, reformats, and writes them to a series of data streams and disk files. Packets are read from the input *file*, or, if omitted, from the standard input stream, *stdin*. They are subjected to a variety of tests. If the **–l** option is specified, their headers are translated into ASCII and written to log files. If **–m** is specified, a one-line description is written to *stdout*, suitable for display by *monitorScience*(1). Most packets are then discarded, and *psci* goes on to read the next one, but some are retained, as follows:

- the most recent exposure header packet from each FEP,
- all event data packets, until a corresponding exposure header packet is encountered,
- multi-packet memory read-out packets originating from a single BEP command,
- the most recent *dumped*Block* and *dumpedHuffman* packets.

psci has been compiled with tables derived directly from the IP&CL Structures database. Packets with unrecognized TTAG codes (as defined in the "*acis_h/interface.h*" file) cause warning messages to be written to *stderr*, and are ignored. All fields in recognized packets and pseudopackets are then checked against their IP&CL limits—bit fields are expanded to "unsigned long int" values unless their minimum permissible values (column 15 in the IP&CL structure tables) are negative, in which case, *psci* treats them as twos-complement signed integers and expands them to "long int". If a field is discovered to be out of range, *psci* writes a message to *stderr*:

```
file: packet[ntotal,ncount].field[index] above maximum (val > maxval)
file: packet[ntotal,ncount].field[index] below minimum (val < maxval)
```

All *stderr* messages begin with a "*file:*" argument; for errors and warnings, this is the name of the input file (or *stdin*); for informatory messages, it is usually the name of an output file. Packets are designated by their IP&CL names, e.g., *exposureTeRaw*, followed by *ntotal*, the sequence number of the packet within the input stream, and *ncount*, its sequence number among packets of its particular type. Both counts start at zero, so the first packet is [0,0]. Multi-dimensional fields within packets are followed by an array index, which also starts at [0]. The value of the field is displayed as a decimal integer. Since out-of-limits field-values are not considered to be sufficient reason for halting the program, *psci* writes these messages and continues processing. The messages themselves can be suppressed by invoking *psci* with the **–q** option.

When the **–l** option is used, *psci* writes packet-header information to log files. The format of these files is derived from that of data structures in the C language. Fields whose values are enumerated in "*acis_h/interface.h*" will be followed by "#" and the enumeration. Unsigned values larger than 32767 are shown in hexadecimal base, preceded by "0x". The values of arrays of fields with dimension > 9 are not shown—merely their dimension. When a packet contains one or more command blocks, e.g. *dumpedTeBlock*, which contains a science parameter block (either *loadTeBlock* or *loadCcBlock*) and an optional window block (either *load1dBlock* or *load2dBlock*), the individual fields are also logged, shifted to the right by 2 columns.

When a *dumpedCcBlock* or *dumpedTeBlock* packet is received, a comment is written into all open log files, and the science run number is incremented. Any opened science and bias data files are automatically closed, and a warning message is written to *stderr* since they should have been closed: science files by the receipt of a previous *scienceReport* packet, and bias files when complete.

When *psci* is invoked with the **–m** flag, monitor records are written to *stdout*.

OPTIONS

–D run *psci* in debug mode, appending the source file name and line number to all error and warning messages. If the program terminates abnormally, the failing packet header will also be logged to *stderr*.

–V print the RCS revision ID of the *psci* program and of the IP&CL structure tables used in its compilation.

–a write log and data files in ASCII mode. If omitted, these files will consist of 32-bit binary fields in the host machine's byte order.

–c concatenate log files, i.e. append to any existing files of the same name. If omitted, existing log files will be overwritten.

–h *file*
initialize the Huffman compression tables from *file*, which must consist of a binary Huffman block. The format is that of a binary Huffman file written by a previous invocation of *psci*.

–l *prefix*
generate log and data file, prefixing their names with the *prefix* string. If omitted, no files will be written.

–m write monitor information to *stdout*, one line per input packet, in the format accepted by *monitorScience*.

–p write the contents of pseudo-packets to "*prefix*.pseudo.log", where *prefix* is specified by the **–l** option. If omitted, pseudopackets will be ignored, except for those requested by the **–u** flag.

–q execute *psci* in quiet mode, suppressing all warning messages.

–s write all log files to *stdout*, rather than to disk files. This and the **–m** flag are mutually exclusive.

–u write text messages in user pseudo-packets to *stderr*. If omitted, user pseudopackets will be ignored. These messages are null terminated strings beginning at word 4 of those packets whose 3rd word is zero.

–v execute *psci* in verbose mode, writing informatory messages to *stderr*, detailing the names and contents of log and data files, the status of *scienceReport* packets, etc.

FILES

Files used to build *psci*

`acis_h/interface.h`	`enumerations`
`cmd.aux`	`command definition overrides`
`enum.aux`	`more enumerations`
`ipcl-struct*.tsv`	`IP&CL structures`
`pseudo.map`	`pseudopacket definitions`
`tlm.aux`	`telemetry definition overrides`

Files generated by *psci* (ASCII listings of packet contents)

`prefix.s``.bias.log`	`bias packet headers`
`prefix``.command.log`	`BEP commands`
`prefix``.deahk.log`	`analog housekeeping`
`prefix.s``.science.log`	`science mode packets`
`prefix``.packet.log`	`miscellaneous packets`
`prefix``.pseudo.log`	`pseudopackets (–p only)`
`prefix``.swhk.log`	`S/W housekeeping packets`

Memory Dumps (ASCII if **–a** specified, otherwise binary)

prefix.bepReadReply.*n*.dat	BEP memory readout (binary)
prefix.bepReadReply.*n*.txt	BEP memory readout (ASCII)
prefix.dumpedSysConfig.*n*.dat	Configuration tables (binary)
prefix.dumpedSysConfig.*n*.txt	Configuration tables (ASCII)
prefix.dumpedBadPix.*n*.dat	Bad pixel lists (binary)
prefix.dumpedBadPix.*n*.txt	Bad pixel lists (ASCII)
prefix.dumpedBadTeCol.*n*.dat	Bad Te column lists (binary)
prefix.dumpedBadTeCol.*n*.txt	Bad Te column lists (ASCII)
prefix.dumpedBadCcCol.*n*.dat	Bad Cc column lists (binary)
prefix.dumpedBadCcCol.*n*.txt	Bad Cc column lists (ASCII)
prefix.dumpedPatches.*n*.dat	Patch list (binary)
prefix.dumpedPatches.*n*.txt	Patch list (ASCII)
prefix.dumpedHuffman.*n*.dat	Huffman tables (binary)
prefix.dumpedHuffman.*n*.txt	Huffman tables (ASCII)
prefix.dumpedTeSlots.*n*.dat	Te parameter blocks (binary)
prefix.dumpedTeSlots.*n*.txt	Te parameter blocks (ASCII)
prefix.dumpedCcSlots.*n*.dat	Cc parameter blocks (binary)
prefix.dumpedCcSlots.*n*.txt	Cc parameter blocks (ASCII)
prefix.dumped2dSlots.*n*.dat	2D window blocks (binary)
prefix.dumped2dSlots.*n*.txt	2D window blocks (ASCII)
prefix.dumped1dSlots.*n*.dat	1D window blocks (binary)
prefix.dumped1dSlots.*n*.txt	1D window blocks (ASCII)
prefix.dumpedDeaSlots.*n*.dat	DEA H/K list (binary)
prefix.dumpedDeaSlots.*n*.txt	DEA H/K list (ASCII)
prefix.fepReadReply.*n*.dat	FEP memory readout (binary)
prefix.fepReadReply.*n*.txt	FEP memory readout (ASCII)
prefix.pramReadReply.*n*.dat	PRAM memory readout (binary)
prefix.pramReadReply.*n*.txt	PRAM memory readout (ASCII)
prefix.sramReadReply.*n*.dat	SRAM memory readout (binary)
prefix.sramReadReply.*n*.txt	SRAM memory readout (ASCII)

Data Files (produced by FEP *n* from science run *s*.)

prefix.s.n.erv.dat	event data (binary ERV format)
prefix.s.n.erv.txt	event data (ASCII ERV format)
prefix.s.n.i–j.hist.fits	pixel histograms (FITS format)
prefix.s.n.i–j.hist.txt	pixel histograms (ASCII)
prefix.s.n.bias.fits	bias FITS files
prefix.s.n.m.raw.fits	raw pixel FITS files
prefix.s.n.TMP.fits	temporary raw pixel files

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

"IP&CL Structure Definitions", MIT CSR

DIAGNOSTICS

Fatal Messages:

The following messages will always be written to *stderr*, after which *psci* will exit with non-zero system return code. If **–D** is specified, the header of the last packet will be logged to *stderr* before exiting.

- *packet*[*ntot,npkt*] **Huffman table slot** *n* **is empty**
  A raw-mode or bias packet referenced a non-existent Huffman table slot.
- *packet*[*ntot,npkt*] **illegal Huffman table slot:** *n*
  A Huffman table slot header is damaged.
- *packet*[*ntot,npkt*] **truncated Huffman table slot** *n*
  A Huffman table header contains a length field that is too large for the Huffman block.
- **–l and –s are mutually exclusive**
  The **–l** option directs logging output to disk files; **–s** sends it to *stdout*.
- **–m and –s are mutually exclusive**
  The **–m** option writes monitor information to *stdout*; the **–s** option uses *stdout* for logging.
- **no more memory**
  Insufficient memory was available to initialize *psci* symbol tables.
- **unable to allocate memory for packet** *n*
  psci ran out of memory for storing packets.

Warning Messages:

These are written to *stderr* unless the **–q** flag appears on the *psci* command line.

- *packet*[*ntot,npkt*] **‘***file***’ partially filled**
- **last data packet was** *packet*[*ntot,npkt*]
  a science run was terminated while incomplete raw-mode data files were being written.
- *packet*[*ntot,npkt*] **‘***file***’ partially filled**
- *packet*[*ntot,npkt*]**.ccdRow=***n* **in last bias packet**
  a science run was terminated while incomplete bias files were being written.
- *packet*[*ntot,npkt*] **missing parameter block**
  A science packet cannot be processed because no science-mode parameter block has been received.
- *packet*[*ntot,npkt*] **no Huffman table specified**
  An initial table can be pre-allocated by using the **–h** option.
- *packet*[*ntot,npkt*] **no histogram data packets received**
  An *exposureTeHistogram* packet was received with no preceding *dataTeHist* packets.
- *packet*[*ntot,npkt*] **no parameter block was found**
  A *scienceReport* packet was received without any prior *dumpedCcBlock* or *dumpedTeBlock* packets.
- *packet*[*ntot,npkt*]**.readAddress != ***packet*[*ntot,npkt***.requestedAddress + 4***n*
  The *readAddress* is inconsistent with the position of this packet within a group of memory-readout packets.
- *packet*[*ntot,npkt*]**.readIndex != ***packet*[*ntot,npkt*]**.requestedIndex + 2***n*
  The *readIndex* is inconsistent with the position of this packet within a group of DEA memory-readout packets.
- *packet*[*ntot,npkt*] **unpacking failure in=***m*/*n* **out=***p*/*p*
  This packet has failed Huffman decompression. *n* 32-bit words (out of *m*) remain to be unpacked; *p* 12-bit values (out of an anticipated *q*) have resulted.
- *packet*[*ntot,npkt*] **unpacking length inconsistency**
  The unpacked length of this packet does not conform to that expected from its header fields.
- *packet*[*ntot,npkt*] **warning: no data packets**
  A raw-mode exposure packet was not preceded by any raw-mode data packets.
- *packet*[*ntot,npkt*]**.startExposureNumber=***n* **<=** *packet*[*ntot,npkt*]**.endExposureNumber=***n*
  The *startExposureNumber* of this histogram exposure packet is not greater than the *endExposureNumber* of the previous exposure packet.

- *packet*[*ntot,npkt*]**.ccdRow=***n* **>** *packet*[*ntot,npkt*]**.(subarrayStartRow+subarrayRowCount)=***n*
  The *ccdRow* field of the current packet is inconsistent with the *subarrayStartRow* and *subarrayRowCount* fields in the science-mode parameter block.
- *packet*[*ntot,npkt*]**.***field*[*n*] **above maximum** (*val* **>** *maxval*)
  The named field is larger than the limit defined in the IP&CL Structures table.
- *packet*[*ntot,npkt*]**.***field*[*n*] **below minimum** (*val* **<** *minval*)
  The named field is smaller than the limit defined in the IP&CL Structures table.
- *packet*[*ntot,npkt*]**.(ccdRow-ccdRowCount)=***n* **<** *packet*[*ntot,npkt*]**.subarrayStartRow=***n*
  The *ccdRow* and *ccdRowCount* fields of this packet are inconsistent with the *subarrayStartRow* field in the science-mode parameter block.
- *packet*[*ntot,npkt*]**.sequenceNumber=***n* **!=** *n*
  The packet sequence number is not one larger than that of the preceding (non-pseudo-) packet.
- **packet** *n* **ignored: unknown type** *n*
  The packet's *formatTag* is not defined in the IP&CL Structures table.
- **unrecognized** *commandOpcode n* **in** *packet*[*ntot,npkt*]
  The *commandOpcode* is not defined in the IP&CL Structures table.
- **warning: non-zero dutyCycle, ERV timetags may be inaccurate**
  psci does not attempt to generate "reasonable" *acistime* fields for timed-exposure mode with non-zero *dutyCycle*.

Informatory Messages:

These are only written to *stderr* if the **–v** flag appears on the *psci* command line.

- **written** *n* **bytes to log file '***file***'**
  a log file has been successfully closed.
- *packet*[*ntot,npkt*] **written to '***file***'**
  a memory-read packet, or groups of packets, has been written to a disk file.
- *packet[ntot,npkt] run n irig n:n exp n fep n ccd n dea n bep n*
  A science run has terminated with the specified number of exposures and FEP, DEA, and BEP completion codes.
- **Executing** *packet*[*ntot,npkt*] **run** *n*
  A science run has begun.
- **written** *n* **bytes to bias file '***file***'**
  A bias file has been successfully closed.
- **start** *packet* **run** *n fepmode bepmode* **bep** *timestamp* **irig** *days:secs* **exptime** *secs*
  A science run has started and *psci* has figured out the approximate IRIG start time and inter-exposure interval.
- **written** *n* **exposures** *n* **events to '***file***'**
  An event file has been successfully closed.
- **written histogram to '***file***'**
  A histogram file has been successfully closed.
- **written** *n* **bytes to raw image file '***file***'**
  A raw-mode FITS file has been successfully closed.

10.27 runacis

NAME

runacis – Execute ACIS FEP and BEP software simulators

SYNOPSIS

runacis [**–B** *file*] [**–D** *dir*] [**–F** *file*] [**–b** *file*] [**–c**] [**–h** *host*] [**–nc** *ccd*] [**–nf** *fep*] [**–x** *file*]

DESCRIPTION

runacis starts a simulated ACIS back-end processor (BEP) on a remote DecStation host and copies its telemetry stream to the standard output stream, *stdout*. It reads its standard input stream, *stdin*, copying the contents to the software serial port of the simulated BEP. This input stream may be constructed from the standard output of *buildCmds*, invoked with the **–p** flag to suppress channel code prefixes.

The **–b** and **–x** options are used to specify programs that will supply CCD pixel values to a simulated ACIS front-end processor (FEP) that will be started on the same remote host as was the BEP simulator. The **–b** file is intended for bias pixels, and **–x** for subsequent science-mode pixels. If both **–b** and **–x** are omitted, no FEP simulator will be started.

All error messages from the simulator(s) will be written to files named *host.type*.log in the current directory, where *host* is the name of the remote DecStation that is running the simulator(s), and *type* is **bep** for output from *acisBepUnix*, **fep** for output from *acisFepUnix*, **img** for output from the *fepImage2* loader, **cmd** for output from the *cclient* command sender, and **tlm** for output from *filterClient*, the telemetry receiver. The standard error stream, *stderr*, from *runacis* itself will only contain short informatory messages, e.g. "*acisBepUnix: starting*", etc., along with the *stderr* of commands specified in the **–b** and **–x** options.

OPTIONS

–B *file*
: specifies the name of the BEP software simulator. The default is *acisBepUnix*.

–D *dir*
: specifies the name of the directory tree within which the various executables (*acisBepUnix*, *acisFepUnix*, *fepImage2*, etc.) are to be found. If *dir* is a relative pathname, it is assumed to be located within *~acis*.

–F *file*
: specifies the name of the FEP software simulator. The default is *acisFepUnix*. It will only be started if the **–x** option is used.

–b *file*
: starts the executable *file*, passing its standard output to a copy of *fepImage2* running on the remote host. This standard output should therefore consist of a stream of DEA pixel values with legal control codes. For instance, rows should be ended with HSYNC codes and frames with VSYNC codes. Unless either this option or **–x** (*q.v.*) is specified, no FEP simulator will be started on the remote host.

–c
: clean up the software simulator processes running on the remote host and release any shared memory that remains allocated.

–h *host*
: the name of the remote DecStation host that is to run the simulators.

–nc *ccd*
: the number of the CCD/DEA to simulate. The default is CCD 0. *runacis* will only start a single simulated CCD, and then only if the **–x** option is used.

–nf *fep*

the number of the FEP to simulate. The default is FEP 0. *runacis* will only start a single simulated FEP, and then only if the **–x** option is used.

–x *file*

waits until *acisBepUnix* writes the string "*:invokeDataProcess*" to its *stderr* stream, and then starts the executable *file*, passing its standard output to a copy of *fepImage2* running on the remote host. The contents are as described for the **–b** option described above.

EXAMPLE

In the following, only the first telemetry packet, a *cmdEcho*, and a *scienceReport* packet, are shown.

```
$ buildCmds –p < te.1 | runacis –h quasar –b genbias –x genfep |
ltlm –v –e11
acisBepUnix: starting
acisFepUnix: starting
filterClient: starting
quasar: configuring
commandEcho[0] = {
  synch                    = 0x736f4166
  telemetryLength          = 8
  formatTag                = 7 (TTAG_CMD_ECHO)
  sequenceNumber           = 1
  arrival                  = 0x00000097
  result                   = 1 (CMDRESULT_OK)
  changeConfigSetting[1] = {
    commandLength          = 7
    commandIdentifier      = 0
    commandOpcode          = 32 (CMDOP_CHANGE_SYS_ENTRY)
    entries[0] = {
      itemId               = 0 (SETTING_DEA_POWER)
      itemValue            = 1
    }
    entries[1] = {
      itemId               = 1 (SETTING_FEP_POWER)
      itemValue            = 1
    }
  }
}
cclient: starting
fepImage2: starting FEP 0 sequencer for bias
genbias: bias 1 written
genbias: bias 2 written
genbias: bias 3 written
fepImage2: starting FEP 0 sequencer for science run
genfep: image 1 written
genfep: image 2 written
genfep: image 3 written
scienceReport[0] = {
  synch                    = 0x736f4166
  telemetryLength          = 12
  formatTag                = 15 (TTAG_SCI_REPORT)
  sequenceNumber           = 13
```

```
  runStartTime            = 0x09af8da0
  parameterBlockId        = 0x12345678
  windowBlockId           = 0xffffffff
  biasStartTime           = 0x04460e20
  biasParameterId         = 0x12345678
  exposuresProduced       = 2
  exposuresSent           = 2
  biasErrorCount          = 0
  fepErrorCodes           = 0 0 0 0 0 0
  ccdErrorFlags           = 0 1 1 1 1 1
  deaInterfaceErrorFlag   = 0
  terminationCode         = 1 (SMTERM_STOPCMD)
}
acisBepUnix: stopping science run
```

The user must now type CTRL–C to stop the run. *runacis* will respond "*Cleaning up*", and will continue to run until the processes are killed on the remote host and the shared memory released.

FILES

host.bep.log *stderr* from *acisBepUnix* running on *host*.
host.cmd.log *stderr* from *cclient* sending commands to *host*.
host.fep.log *stderr* from *acisFepUnix* running on *host*.
host.img.log *stderr* from *fepImage2* running on *host*.
host.tlm.log *stderr* from *filterClient* receiving telemetry from *host*.

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

acisBepUnix(1), acisFepUnix(1), buildCmds(1), cclient(1), fepImage2(1), filterClient(1), filterServer(1), ltlm(1), processScience(1)

DIAGNOSTICS

Bad –nc value
the CCD value must lie in the range 0 through 9.

Bad –nf value
the FEP value must lie in the range 0 through 5.

BUGS

runacis makes various assumptions about the time required for the various processes to start up. If the network connection to the remote host is particularly slow, it will fail.

If *acisFepUnix* isn't correctly cleaned up on the remote host, a subsequent *runacis* job may fail because the shared memory segments are still in use. The remedy is to invoke *runacis* with the **–c** flag to repeat the clean-up and then run it a second time in its usual manner.

10.28 sciglue

NAME

sciglue – convert psci monitor output for monitorScience

SYNOPSIS

sciglue [*file*]

DESCRIPTION

This Perl script reads the standard output of *psci*, invoked with the *–m* flag, and converts it into a format suitable for input to *monitorScience*.

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

monitorScience(1), psci(1)

10.29 sendCmds

NAME

sendCmds - send the output of buildCmds to shim

SYNOPSIS

`sendCmds`

DESCRIPTION

sendCmds receives a binary command stream from *buildCmds*, containing command type, command channel, command data triplets. It constructs 23-bit, formatted command strings, packages them into 24-bit strings to simplify the output interface, and sends them to stdout, which is assumed to be piped to *SHIM*. *sendCmds* expects its standard input to consist of pairs of 16-bit words (command type and command channel) followed by zero or more 16-bit words comprising the command data, as described below. All 16-bit words are assumed to start with their least significant bytes, i.e. little-endian order. The format and content of commands are contained in the AXAF IP&CL documents..

TABLE 30. Command Type and Channel Definition

Command Type		Command Channel		Description
Name	Value	Name	Value	
Serial Digital	2	Software	2	Command used to control the ACIS software
		Hardware	3	Command used to control the ACIS hardware
High Level Pulse	0	Pulse Cmd Channel Number	0-98	PS and MC commands, whose action is determined by the Command Channel value
No-Op	3	TBD	TBD	Potential RCTU/CTUE operation commands

High Level Pulse commands are completely specified by the Command Type and Command Channel pair. Therefore, no command data will follow.

Serial Digital Hardware commands will consist of a single 16-bit word that will immediately follow the Command Type and Command Channel pair.

TABLE 31. *sendCmds* Command Formats

Serial Digital Commands		High Level Pulse Commands	
Bit[1]	Contents	Bit	Contents
0	Unspecified	0	Unspecified
1-2	Command Type	1-2	Command Type
3-18	Command Data	3-14	Unspecified
19-23	Command Channel	15-23	Command Channel

1. bit 0 is the most significant bit and is transmitted/received first

Serial Digital Software commands will consist of from 3 to 256 16-bit words contained in an ACIS software command packet that will immediately follow the Command Type and Command Channel pair. All software command packets contain length fields which are extracted by *sendCmds* to determine how to read the remainder of the command packet.

When *sendCmds* reads an illegal command type or command channel from stdin, it writes an error message to stderr and terminates processing. When *sendCmds* reads an illegal software command packet from stdin, it writes an error message to stderr, but **it processes the command anyway**. No warning is given if an unknown opcode is encountered. These shortcomings will be fixed in a later version.

EXAMPLES

The following UNIX pipe uses *sendCmds* as part of commanding the ACIS instrument to start executing stored timed exposure parameter block 1:

```
echo 'start 22 te 1' | buildCmds | sendCmds | shim ...
```

AUTHOR

Demitrios Athens, MIT CSR

STATUS

The current version does not check for and remove critical commands.

The current version has been used with the LRCTU and the CTUE.

The current version produces its output in the **bit** order shown above, which is as defined in the CTUE and the Command and Data Management equipment specifications. However, to make the software work successfully with the LRCTU, the current version REVERSES THE **BYTE** ORDER OF ALL SERIAL DIGITAL COMMAND DATA that it receives. This seems to work when talking to the CTUE.

The current version has successfully handled both software and hardware Serial Digital commands and High Level Pulse commands.

The values currently used for software and hardware command channels are updated to the current IP&CL values as expressed in the beta version of the "interface.h" file.

The curent version will may be modified to emit No-Op commands that will be used to convey a command count for use in assembling CTUE command blocks.

Software commands with unrecognized opcodes are passed through. This responds to a request from the flight software folks to provide flexibility during testing.

Hardware command data is not checked for validity.

10.30 shim

NAME

shim - common interface for both CTUE and LRCTU

SYNOPSIS

shim ctue|lrctu

DESCRIPTION

shim provides a consistent interface between ACIS and all user applications that either generate ACIS commands or receive ACIS telemetry. It can communicate with ACIS using either the CTUE/RCTU combination or the Jim Littlefield RCTU emulator, the LRCTU. It always expects its command input to be in the 24-bit *sendCmds* format. It always produces its telemetry output in the AXAF-I minor frame format.

OPTIONS

ctue

communicate with ACIS using the CTUE/RCTU combination. *shim* extracts the 23-bit input command words from the 24-bit *sendCmds* format and packs them into 48-bit Ground Command format strings. It then assembles these strings into command blocks, which it sends to the CTUE/RCTU via a TCP/IP interface. *shim* receives telemetry via the same interface and passes it to its client, which is typically *getPackets*, without modification.

lrctu

communicate with ACIS using the Jim Littlefield RCTU emulator, the LRCTU

When sending commands to an LRCTU, *shim* reads the 24-bit output of *sendCmds* and passes it to the LRCTU via a serial interface. Note that High Level Pulse commands will not be executed because the LRCTU does not support them.

When receiving telemetry from an LRCTU via the same serial interface, *shim* reformats the LRCTU telemetry packets into AXAF-I, Format 2, telemetry minor frames, adding frame synchs and CCSDS headers as appropriate. *shim* always resets the LRCTU hardware (by sending an ASCII character with many zero bits to the LRCTU's reset port) and then copies the LRCTU's operating code to the device's memory.

EXAMPLE

```
... buildCmds | sendCmds | shim lrctu | getPackets ...
```

ENVIRONMENT

ACISTOOLSDIR

The name of the top-level ACIS Test Tools directory. *shim* will use this directory as its starting point when looking for executables. See the **FILES** section immediately below.

FILES

${ACISTOOLSDIR}/bin/sun4/lrctu.unix

Jim Littlefield's code that runs on UNIX and communicates with the LRCTU through the fast tty device.

${ACISTOOLSDIR}/bin/mips/lrctu.mongoose.srec

Jim Littlefield's code that runs on the Mongoose processor on the LRCTU and communicates with UNIX through the fast tty device.

/dev/ttyC?0

LRCTU command and telemetry fast tty device (? is typically 4)

/dev/ttyC?1
LRCTU hardware reset fast tty device (? is typically 4)

/dev/ct.o
fast tty device driver

/etc/rc/sts
fast tty device system boot file

/etc/sts/bin/cdmknods
fast tty device system boot file

AUTHOR

Demitrios Athens, MIT CSR

STATUS

The current version only works with the Jim Littlefield RCTU Emulator.

The current version creates AXAF-I telemetry major frames from LRCTU telemetry by piping LRCTU telemetry to *ltp2mnf*.

The current version does not put bilevels (LED's) reported in the LRCTU telemetry packets into the AXAF-I minor frames. Expect this to change soon.

The current version of the LRCTU does not produce real-time or even near-real-time timestamps when it issues science header pulses to the BEP. It merely copies the BEP timestamp, which is the latched value of a free running counter, and uses this information in its telemetry packet headers. The ability to obtain near-real-time timestamp information from the LRCTU has been proposed but has neither been made a requirement nor been implemented. *ltp2mnf* creates pseudo IRIG-B timestamps and puts them into the AXAF-I format minor frames that it creates. Times come from the host system clock, not from an actual IRIG-B interface, and are not adjusted to account for processing or transmission delays.

The ability to command BEP and FEP hardware resets through the LRCTU has been proposed but has neither been made a requirement nor been implemented.

10.31 tlmsim

NAME

tlmsim – simulate ACIS telemetry packet stream

SYNOPSIS

tlmsim [**–c** *file*] [**–f**] [**–p**] [**–t**] [**–w** *file*] *ringfile biasfile*

DESCRIPTION

This command generates a stream of ACIS telemetry packets from a timed-exposure ring-buffer file (*ringfile*) and a bias map in FITS format (*biasfile*). These files are generated by the *output* and *dumpbias* directives in *fepCtlTest* scripts.

OPTIONS

–c *file*
Load a timed-exposure parameter block from *file*, which must contain a series of "*keyword = value*" lines, as exemplified in the following section. If this option is omitted, *tlmsim* will use a default block.

–f Simulate Format 1 output, i.e. with ACIS out of the focal plane and restricted to 512 baud downlink. This will affect the ratio of telemetry packets to science-frame pseudo-packets.

–p Generate science-frame pseudo-packets in the format described in the ACIS Test Tools document. If this option is used in conjunction with the **–f** flag, a *huge* number of pseudo-packets will be generated.

–t Use the current UNIX system clock as a basis for BEP timestamps. If omitted, these fields will be zeroed out, making it easier to compare *tlmsim* output files.

–w *file*
Load a 2-dimensional window block from *file*, which must contain a series of "*keyword = value*" lines, as exemplified in the following section. If this option is omitted, *tlmsim* uses no windows.

PARAMETER BLOCK EXAMPLE

Keywords may be expressed in upper, lower, or mixed case characters. In this example, a simple 3x3 faint-with-bias science run is defined, using CCD number 6 and FEP number 2, as specified by the *fep*CcdSelect* keywords. All keyword values are unsigned integers, or arrays of unsigned integers, with the following exceptions: • *parameterBlockId*, *deaLoadOverride*, and *fepLoadOverride* are hexadecimal integers with "0x" prefixes; • *gradeSelection* is an array of 8 hexadecimal integers, *without* any "0x" prefix. With these exceptions, *tlmsim* does not check the values in any way. Blank lines, and all characters between "#" and newline, are ignored.

```
teBlockSlotIndex          = 1
parameterBlockId          = 0x00000fab
fepCcdSelect              = 10 10 6 10 10 10
fepMode                   = 2
bepPackingMode            = 1
onChip2x2Summing          = 0
ignoreBadPixelMap         = 0
ignoreBadColumnMap        = 0
recomputeBias             = 1
trickleBias               = 1
subarrayStartRow          = 0
subarrayRowCount          = 0
overclockPairsPerNode     = 15
outputRegisterMode        = 0
```

```
ccdVideoResponse            = 0 0 0 0 0 0
primaryExposure             = 28
secondaryExposure           = 0
dutyCycle                   = 0
fep0EventThreshold          =    0    0    0    0
fep1EventThreshold          =    0    0    0    0
fep2EventThreshold          =  100  100  100  100
fep3EventThreshold          =    0    0    0    0
fep4EventThreshold          =    0    0    0    0
fep5EventThreshold          =    0    0    0    0
fep0SplitThreshold          =    0    0    0    0
fep1SplitThreshold          =    0    0    0    0
fep2SplitThreshold          =    0    0    0    0
fep3SplitThreshold          =    0    0    0    0
fep4SplitThreshold          =    0    0    0    0
fep5SplitThreshold          =    0    0    0    0
lowerEventAmplitude         = 0
eventAmplitudeRange         = 65535
gradeSelection              = ffff0000 0 0 0 0 0 0 0
windowSlotIndex             = 65535
histogramCount              = 0
biasCompressionSlotIndex    =  255  255  255  255  255  255
rawCompressionSlotIndex     = 255
ignoreInitialFrames         = 20
biasAlgorithmId             =    0    0    1    0    0    0
biasArg0                    =    0    0    0    0    0    0
biasArg1                    =    0    0   10    0    0    0
biasArg2                    =    0    0    0    0    0    0
biasArg3                    =    0    0  100    0    0    0
biasArg4                    =    0    0   70    0    0    0
fep0VideoOffset             =    0    0    0    0
fep1VideoOffset             =    0    0    0    0
fep2VideoOffset             =  127  128  129  130
fep3VideoOffset             =    0    0    0    0
fep4VideoOffset             =    0    0    0    0
fep5VideoOffset             =    0    0    0    0
deaLoadOverride             = 0x00000000
fepLoadOverride             = 0x00000000
```

WINDOW BLOCK EXAMPLE

Keywords may be expressed in upper, lower, or mixed case characters. In this example, two windows are defined, one for CCD 5 and the other for CCD 6. All keyword values are unsigned integers, except for *windowBlockId*, which must be a hexadecimal integer with leading "0x". Otherwise, *tlmsim* does not check the values in any way. Blank lines, and all characters between "#" and newline, are ignored.

```
windowSlotIndex             = 1
windowBlockId               = 0x00000baf

ccdId                       = 5
ccdRow                      = 1
ccdColumn                   = 1
width                       = 1022
height                      = 1022
```

```
sampleCycle                = 0
lowerEventAmplitude        = 0
eventAmplitudeRange        = 65535

ccdId                      = 6
ccdRow                     = 1
ccdColumn                  = 1
width                      = 1022
height                     = 1022
sampleCycle                = 0
lowerEventAmplitude        = 0
eventAmplitudeRange        = 65535
```

AUTHOR

Peter G. Ford, MIT CSR

SEE ALSO

dumpring(1), fepCtlTest(1), ltlm(1)
ACIS Test Tools, ACIS Document, Version 1.0, July, 1996
ACIS IP&CL Structures, Version 1.17, July, 1996

DIAGNOSTICS

bad argument: *arg(s)*
tlmsim too many arguments have been specified on the command line.

does not scan: *line*
the statement in a parameter or window block is invalid, either because the keyword is unrecognized, or the value(s) is(are) illegal, or because there is no equals sign between them.

missing file name(s)
takes exactly two non-optional command line arguments, the name of the ring-buffer file, followed by the name of the FITS bias file.

multiple CCDs selected
the parameter block can contain only one *fep*ccdSelect* field whose value is less than 10, i.e. *tlmsim* can only simulate a single CCD at a time.

no CCD selected
the parameter block must contain at least one *fep*ccdSelect* whose value differs from 10.

no window specified
the window block contains no window-specific keywords.

out of order: *line*
one or more header keywords comes after the first window-specific keyword in a window block.

unknown flag: *flag*
the flag is unrecognized. There must be whitespace between the flag and the file name in the **–c** and **–w** options.

unsupported biasCompressionSelect[*n*] value: *val*
this version of *tlmsim* requires that all *biasCompressionSelect* fields have the value 255, i.e. no compression at all.

unsupported rawCompressionSelect value: *val*
this version of *tlmsim* requires that the *rawCompressionSelect* field has the value 255, i.e. no compression at all.

10.32 writeCCB

NAME

writeCCB – package the output of sendCmds into CTUE command blocks

SYNOPSIS

writeCCB [actu|bctu] [abus|bbus]

DESCRIPTION

writeCCB receives a series of 24-bit command strings from **sendCmds** that contain command type, command data, and command channel triplets. It constructs CTUE command blocks that contain the commands and sends them either to stdout, or, over a TCP/IP connection, to the CTUE's command port. **writeCCB** expects its standard input to consist of 3-byte groups as produced by **sendCmds**. **writeCCB** places each 24-bit command into a 48-bit CTUE format command that looks like this:

TABLE 32. CTUE command

Bit[1]	Contents
0-6	Spacecraft address
7	CTU A/B select
8	Fixed bit '1'
9-11	Command routing
12-16	RCTU address
17-39	sendCmds output
40	Fixed bit '1'
41-47	Polynomial check code

1. bit 0 is the most significant bit and is transmitted/received first

writeCCB places each 48-bit CTUE format command into a command block that looks like this:

TABLE 33. CTUE command block

Byte[1]	Contents
0	Command count
1-6	CTUE Command
7-8	Check sum

1. byte 0 is the most significant byte and is transmitted/received first

writeCCB automatically checks the environment when starting. If it discovers a CTUE_CMD_SD variable, it writes the output CTUE command blocks to the socket descriptor that the variable indicates. (**ACISshell** will set this variable when the CTUE connects to accept ACIS commands.) Output goes to *stdout* otherwise.

OPTIONS

actu | bctu

selects either the A or B side of the spacecraft's CTU. If the user does not specify a side, **writeCCB** defaults to using the A side.

abus | bbus

selects either Bus A or Bus B command routing from either the CTUE or the spacecraft CTU to the RCTU. If the user does not specify a bus, **writeCCB** defaults to using the A bus.

EXAMPLES

The following UNIX pipe uses **writeCCB** as part of commanding the ACIS instrument to start executing stored timed exposure parameter block 1. The command prompt indicates that **ACISshell** has been executed and that the CTUE has established a connection. *ACISshell%* echo 'start 22 te 1' | buildCmds | sendCmds | writeCCB

AUTHOR

Demitrios Athens, MIT CSR

SEE ALSO

sendCmds(1) ACISshell(1)

STATUS

The current version has been used with the CTUE only.

The **actu|bctu** option is probably meaningless and best left alone when using a CTUE, which has only an A side. (The B side was so good that it's being saved for the album.)

The interaction of the **abus|bbus** option and the CTUE's own GUI, which allows the operator to select a command bus, is a complete mystery. It's probably best not to specify the option unless you're looking for adventure.

Appendix A Test Tool Status

TABLE A–1. Test Tools and their Current Status

Program Name	Author	Status	Comments
GSE Transport Software			
ACISshell	Athena	Implemented	
acispkts	Davis/Ford	Implemented	Replaced by shim | getPackets
filterClient	Ford	Implemented	
filterServer	Ford	Implemented	
sendCmds	Athens	Implemented	
getPacktes	Athens	Implemented	
shim	Athens	Implemented	
writeCCB	Athens	Implemented	
GSE Test Tool Software			
analyzeData	Woo	Implemented	Function provided by existing programs
bcnd	Ford	Implemented	Alternative to buildCmds
buildCmds	Somigliana	Implemented	
diff6	Ford	Implemented	Untested
lcmd	Ford	Implemented	
lerv	Ford	Implemented	
lhuff	Ford	Implemented	
ltlm	Ford	Implemented	
monitorDEAHousekeeping	Shaff	Implemented	May require modifications
monitorEngineeringData	Athens	In progress	
monitorScience	Somigliana	Implemented	Filter input through sciglue first
psci	Ford	Implemented	Replacement for processScience
processDEAhkp	Athens	Implemented	No documentation
processEngrData	Athens	In progress	No documentation
runacis	Ford	Implemented	
DEA Image Operations			
getImages	Shaff	Implemented	
genPixelImages	Somigliana	Partially Implemented	Lacks image compression
genObjectImage	Ford	Implemented	
generatedExpectedData	-	Abandoned	
loadFitsImage	Ford	Implemented	
putImages	Athens	Implemented	