MULTIBEAM BLOCK CONTROL COMPUTER

ATNF Electronics Group

1. Overview

This document describes the computer system used in the Multibeam correlator. The Multibeam Block Control Computer (MBCC) provides control for the Correlator Blocks. Each block consists of a Block Backplane and up to 8 Correlator Module boards. Up to three blocks can be controlled by a single MBCC.

The computer system uses an Intel 80486 computer and runs the pSOS operating system from Integrated Systems. It receives commands from and responds to the host Correlator Control Computer (CCC). This communication is via a RS-232 serial line or ethernet using the TCP/IP protocol.

When communicating with the MBCC system the serial port can only be used to control block 0. When the network communication is used commands for Correlator Blocks 0, 1 and 2 are accepted on network ports 4000_{10}, 4001_{10}, and 4002_{10}, respectively. When correlation data is being retrieved over the network connection correlation data for Correlator Blocks 0, 1 and 2 is retrieved on network ports 4003_{10} 4004_{10} and 4005_{10}, respectively.

Command Format

Commands are passed from the CCC to the MBCC system as ASCII text. A command consists of a line of text beginning with a period (".") character followed by a two letter command and any command arguments. The command line is terminated by a carriage return character (ASCII 13), a new-line character (ASCII 10), or both. Some commands require an input data block containing extra information required by the command. This input data block immediately follows the command line. In addition, some commands return an output data block containing information obtained by the command.

Lines returned by the system to the CCC are terminated by a carriage return character (ASCII 13) followed by a new-line (ASCII 10). After any command has been executed (whether or not any input or output data blocks were involved) the system returns a line containing a single hexadecimal number which is an error code. If the command was successfully completed this error code will be zero. A non-zero error code indicates that the command could not be completed and the value indicates the nature of the error (see Return Error Codes below).

The majority of this document is used to describe the individual commands. In the descriptions the first line gives the command name and its basic function. The second line gives the command and its arguments. Each argument is listed in italic and optional arguments are enclosed in square brackets ([...]). The bullet (•) character indicates a space or tab character. The next lines contain a description of the "type" of each argument. A list of bold characters enclosed in parenthesis means that argument may assume any of the emboldened characters. Two numbers separated by two periods (..) means that argument may accept any numerical value between or including the two values. Numerical values separated by commas means that the argument may assume any of the numerical values listed. All numerical values given are in hexadecimal. (The one exception to this rule is the ETD specification given in the .EE and .LT commands.) Normal type is used for characters passed from the CCC to the system, bold type for the characters passed from the system to the CCC.

Input Data Blocks

Some commands require extra input data. This is supplied in an *input data block*. An input data block is one or more lines of ASCII text with each line containing a single value. Each line is terminated by a carriage return character (ASCII 13), a new-line character (ASCII 10), or both. The block is terminated by a line containing only the tilde character (" ~", ASCII 126). The command error code is returned after the end of block marker is received by the system. The one exception is the .DX command where data is sent in binary format for efficiency reasons.

The system will read as much data as it requires from an input data block ignoring any excess. If too much or not enough information is in the input data block then an error is returned.

Output Data Blocks

For the system to send data to CCC an *output data block* is used. The one exception is the .GC command which returns data in binary format for efficiency reasons. An output data block consists first of a line containing only the percent sign (“%”, ASCII 37), followed by one or more lines of text, and finally terminated by a line containing only the tilde character (" ~", ASCII 126). The error code for the command is returned after the block has been sent. The end of block line will always be returned even when an error occurs and no data is sent. If an error occurs while the data for the output data block is being generated then the output data block is shortened by sending the end of block line and error code immediately.

The CCC software should read all required information from an output data block then discard the rest of the data up until the end of block line. This convention allows backward compatible upgrades to the system.

For example the .GT command is used to obtain the current atomic time. The following is an example of the use of this command and illustrates the use of an output data block.

```
.GT
%
F04D16EF33EF0
~
0
```

Return Error Codes

At the end of execution of any command the system returns an error code to the host computer. Error codes consist of 16-bit unsigned numbers (four hexadecimal digits). The possible error codes are given below. Note some of these error codes relate to commands not supported on the Multibeam Correlator.

Error	**Description**
0000	Ok. The command completed successfully.
7001	IllegalCommand. A line starting with a period (“.”) has been received but the next two characters were not recognised as a legal command.
7002	MissingArgument. A command line argument which is required by the given command was not present in the command line.
7003	IllegalArgument. A command line argument either contained an illegal character and could not be converted, or contained an illegal value.
7004	IllegalMode. The command line contained an illegal mode.

7005	MemoryAllocError. An error occurred when the system attempted to allocate a memory block.
7006	DataBlockValueError. An input data block element either contained illegal characters and could not be converted, or contained an illegal value.
7007	MissingDataBlockElement. There were not enough lines in the input data block following the command.
7008	TooManyDataBlockElement. There were too many lines in the input data block following the command.
7009	ExceededRecursiveLimit. Calls to the system shell were nested too deeply. This can only occur when using the .EX command.
700A	FileNotFound. The given disk file name was not found. This can only occur when using the .EX command.
700B	ProgFileNotFound. The given memory file name was not found. This can only occur when using the .PA, .PD, .PE, and .PX commands.
700C	ProgFileCRCError. A CRC error occurred when reading the given memory file. This can only occur when using the .PA, .PD, .PE, and .PX commands.
700D	NonExistentMemory. The interface status register indicated an attempt was made to access a memory location which does not exist.
700E	MemoryAddressOverflow. The interface status register indicated that the address register overflowed while accessing a memory location.
700F	UnknownETD. The Event Timing Description for the given index has not been defined with .LT command. This can only occur when using the .EE command.
7010	BATOffLine. The event generator indicates the BAT clock signal is not present or broken.
7011	EGFIFOError. An event generator FIFO memory fault occurred.
7012	EGPreambleError. The system failed to find the preamble to the event generator version/serial number string.
7013	EGSerNumTooLong. The event generator version/serial number string was longer than expected.
7014	ETDTooLong. The given Event Timing Description was too long. This can only occur when using the .LT command.
7015	BadETDDesc. An error occurred while the .LT command was parsing an Event Timing Description. This can only occur when using the .LT command.
7016	NoEventGen. No event generator is present in the system.
7018	NonexistentModule. The addressed Correlator Module is not present.
7019	IllegalModuleCombination. Only one Correlator Module can be selected for the command.

701A	DelayUnitOwned. A delay unit in the given delay unit block is already assigned. This can only occur when using the .DB command.
701B	UnknownDelayTable. The given delay unit block index has not be defined with the .DB command.
701C	OutOfDelayBulkTables. All available delay unit blocks have been used. This can only occur when using the .DB command.
701D	ETDQueueOverflow. The message queue where ETD processing requests are placed is full. This usually indicates previous ETD processing has not been completed. This can only occur when using the .EE command.
701E	CorrDataNoSend. The requested correlator data could not be sent.
701F	DoneNotHigh. The Done pin did not go high indicating the Xilinx chip failed to program.
7020	XilinxCFG. An error occured while reading the Xilinx configuration data.
7021	DoneNotLow. The done pin did not go low indicating the Xilinx chip failed to reset.

2. Correlator Block Commands

The following is a description of the commands which are available on the Multibeam Block Control Computer (BCC) system. They are used to control and monitor the Correlator Block.

.PM .PM

Reads from or writes to the Multibeam Correlator registers.

.PM • *address* • [*data0*] • [*data1*] • [*data2*] •........[*data31*]
address: Base address of Correlator Module
data0 - dataN: Register address and data to load into register.

This command reads from or writes to the registers of the designated Correlator Module. To Write the command must supply the address and at least 1 register/address value. The data is made up of 2 fields: bits 0 - 15 contain data, bits 16 - 20 contain the register number and bits 21 - 31 are 0; for example, the data c3456, puts the data 3456 into register c (12).

To read supply only the Correlator Module base address. An output data block is returned with 33 entries. Same data encoding is used as with the write. the 33rd entry is the serial number and it takes the register number 3F.

.MI .MI

Initialise multibeam total power acquisition task.

.MI • *nsamplers* • [*address* • *sampler*] $_{nsamplers}$
nsamplers: 1..F
address: module address

sampler: 0 1

This command initialises the total power task and enables collection of the total power counts from the samplers listed in the command parameters. nsamplers is the total number of samplers to be read. Each multibeam correlator module is capable of accessing two samplers. The collected total power counts are retrieved with the **.gp** command.

.GP .GP

Get multibeam correlator total power counts.

.GP

This command retrieves the multibeam correlator total power counts read during the last integration. The data is returned in a standard data block. The data block consists of nsamplers $\times$ 2 $\times$ number of times the multibeam sampler SAM_DATA_READY signal switches during an integration, four-digit hex numbers. There is a maximim of 12 values per line with each SAM_DATA_READY period also terminated by a new line. For example, with 8 samplers and two SAM_DATA_READY periods per integration the data block would have the format;

```
%
1d23 1d73 a547 a532 c228 c111 2344 2311 9665 9767 2343 2123
1232 1254 6547 6587
1234 1325 ef11 ed23 3647 3254 8342 8453 5534 5433 6289 6211
a234 a792 b232 b211
~
```

The sampler to which each pair of numbers refers is the same as specified in the **.mi** command, which must have been issued previously. The number of lines of data returned is equal to the
.

.GC .GC

Retrieve multibeam correlator correlation data by ethernet.

.GC • *address* • *chip*
address: module address
chip: 0 1

This command retrieves the multibeam correlation data over the ethernet. The data is returned in binary format for efficiency reasons. 4100 bytes of data will be sent in response to the **.gc** command - (1024 lags + sample count) 32 bit words. The zeroth lag is sent first and in increasing order thereafter. The 1025th word is the total sample count.

.RX .RX

Reset multibeam correlator Xilinx chip.

.RX • *address* • *chip*
address: module address
chip: [0 = DMA Interface, 1 = Data Controller]

This command resets the specified Xilinx chip on the multibeam correlator module. This command is issued prior to programming a Xilinx chip with the **.dx** command.

.DX .DX

Program multibeam correlator Xilinx chip.

.DX • *address* • *chip* • *bytes*
address: module address
chip: [0 = DMA Interface, 1 = Data Controller]
bytes: number of bytes of programming data

This command programs the specified Xilinx chip on the multibeam correlator module. Immediately after this command is issued bytes of binary programming data, read from the appropriate .xil file, should be sent.

.CD .CD

Program clock divider and clock switch.

.CD • divider • clksrc
divider: [0..7] divider setting
clksrc: [0 = 128 MHz, 1 = 32 MHz]

This command is only used when the correlator is being driven by the LBA-DAS. The command sets the divider ratio for an external clock divider which feeds the correlator and selects the input source to the clock divider from the two clocks supplied by the DAS. The divided clock frequency is given by

$$Output\_clk\_freq = Input\_clk\_freq / 2^{divider} .$$

3. Event Generator Commands

The event generator is a custom hardware device designed and built at the ATNF. It can generate up to 16 timing signals and a clocking signal. It includes a frame grabber used to obtain information from the Binary Atomic Time (BAT) clock, aka the Hunt clock. The following is a description of the commands which are available on both the Block Control Computer (BCC) and Delay Unit Control Computer (DUCC) systems when equipped with this device.

To control the event generator's timing signals an **Event Timing Description** (ETD) is used. An ETD is a recursive data structure which defines a stream of events for the event generator. An ETD acts like a program which is followed to generate the data (time and events) to send to the event generator. It consists of the ETD commands listed below. Of special importance is the `Sequence` command which allows an ETD to be recursive. It is used to create loops within an ETD.

The ETD processor contains 8 event and 8 time registers which can be manipulated by the ETD. Each event register holds a 16-bit unsigned number, and each time register hold a 64-bit unsigned number. The registers are accessed through the processing of the ETD. Event register 0 has special meaning in the ETD commands and is referred to as the accumulator. The accumulator can also be accessed by specifying $0 in the same way as the other registers. There is also a carry register which is used in processing the *Increment* command. It can be cleared using the *ClearCarry* command.

The input data block used to define an ETD consists of a number of text lines. Each line contains a command and any needed arguments separated by spaces or tabs in the same way as command lines.

All specifications of time in an ETD definition are of a data type called *reduced-BAT* . The integer part of a reduced-BAT time represents the least significant 48-bits of atomic time in microseconds. It can contain up to 12 hexadecimal digits. The integer part may optionally be followed by a decimal point (".") and up to 8 hexadecimal digits giving the microsecond fraction. Actual event times are truncated to the nearest microsecond but the fractional part is retained to minimize accumulated errors.

All specifications of events in an ETD definition are of a data type called *event-def*. Such a value can take two forms: It can be a literal consisting of a hexadecimal number in the range 0..FFFF, or it can be the contents of an event register. An event register is specified by a "$" character followed by a hexadecimal number in the range 0..8. For example, $6 refers to register 6. For compatibility with previous software versions literals maybe prefixed with the # character.

The time offset argument of the event definition ('e') and sequence ('s') commands, and the period argument of the sequence ('s') command can also take two forms. They can be a reduce-BAT literal, or the contents of a time register. A time register is specified by a "$" character followed by a hexadecimal number in the range 0..8. For example, $6 refers to register 6.

Examples:

This ETD will generate a 1Hz square wave with 50% duty cycle on the least significant bit of the event generator output. The square will be generated for 32767_{10} complete cycles starting at the time given in the .EE command.

```
S • 0 • 2 • FFFE • 7A120
X • 0001
E • 0 • $0
```

This ETD will generate a 10 millisecond period square wave for the first half of each second, and a 20 millisecond period wave of 8 millisecond width pulses for the second half of each second on output bit 0. Output 1 wil be low for the first half of each second and high for the second half. It will run for 65535 seconds and output bit 1 will be left high.

```
S • 0 • 2 • FFFF • F4240
S • 0 • 2 • 32 • 2710
E • 0 • 1
E • 1388 • 0
S • 7A120 • 2 • 16 • 4E20
E • 0 • 3
E • 1F40 • 2
```

ETD Commands

And:

A.• *and-value*

where:

and-value (event-def) is a number which is ANDed with the current contents of the accumulator and the result is placed back into the accumulator.

This command performs an AND function of *and-value* and the current contents of the accumulator and places the result back into the accumulator.

ClearCarry:

C

This command clears the carry register. The carry register is used in conjunction with the ETD Increment command.

EventDefinition:

E.• *time-offset.*• *event* [• *mask*]

where:

time-offset (reduced-bat) is the offset from the current base time at which this event is to occur.

event (event-def) is a value which is output by the event generator at the time of the event.

mask (event-spec) is a value which defines the output bits of the Event Generator which are to be changed by the EventDefinition.

This is the fundamental definition of an event which is generated at the time calculated by the elements of the ETD in which it is contained. If *mask* is omitted then all bits in event are used. If *mask* is present then bits in *mask* which are low are not altered on the Event Generator output, ie they remain as they were in the previous event. After the event is produced, the accumulator contains the event.

Get:

G.• *value*

where:

value (event-def) is a number which is placed into the accumulator.

This command places the *value* (either a literal or the contents of one of the event registers) into the accumulator.

Increment:

I.• *mask* [• *increment*]

where:

mask (event-def) is a bit pattern which determines the field of the accumulator which is to be accumulated. The argument must contain a single cluster of high bits.

increment (event-def) is the amount by which the *mask* field of the accumulator is to be increased. If *increment* is omitted then it defaults to one.

The value of *increment* and the carry register is added to a bit field in the accumulator determined by the value of *mask*. The result is placed back into the field of the accumulator and the carry is placed into the carry register. The carry register may be cleared using the ETD command ClearCarry.

Using the carry register, non-contiguous fields in the accumulator may be incremented by an arbitrary amount. The carry register is first cleared and then the least significant field is incremented. The rest of the fields are then incremented by zero in order from the least to the most significant.

Not:

N

This command performs an NOT function of the current contents of the accumulator and places the result back into the accumulator.

Or:

O • *or-value*

where:

or-value (event-def) is a number which is ORed with the current contents of the accumulator and the result is placed back into the accumulator.

This command performs an OR function of *or-value* and the current contents of the accumulator and places the result back into the accumulator.

Put:

P • *register*

where:

register (event-def) is a register into which the contents of the accumulator are deposited. The argument must be in the register form of *event-def.*

This command places the contents of the accumulator into one of the event registers.

Sequence:

S • *time-offset* • *elements* • *repetitions* • *period*

where:

time-offset (reduced-BAT) is the offset from the current base time at which the sequence is to begin.

elements (0..FFFF) is the number of ETDs following this line which form the list to be executed.

repetitions (event-def) is the number of times the following list of ETDs is to be repeated. Note that this argument is of type *event-def*, so that the number or repetitions can be independent of the actual ETD loaded, ie it can be retrieved from a register.

period (reduced-BAT) is the time to be taken for one repetition. This must be equal to or greater than the time actually taken to execute all ETDs of the sequence once.

The Sequence command provides the control structure for the ETD. Following it should be a list of *elements* ETDs which are executed sequentially *repetitions* times.

When an ETD is found to consist of a sequence, a base time variable is created. The initial value for the base time is the current value for the inherited base time plus the value of the *time-offset* for this sequence. Each time the sequence is repeated, the value of the *period* argument is added to the base time. The base time is used as the base for the *time-offset* arguments of all the ETDs within the list. The execution of a sequence does not alter the value of the inherited base time. The inherited base time at the top level of the ETD comes from the ETD execution command.

XOr:

X • *xor-value*

where:

xor-value (event-def) is a number which is XORed with the current contents of the accumulator and the result is placed back into the accumulator.

This command performs an XOR function of *xor-value* and the current contents of the accumulator and places the result back into the accumulator.

.EE .EE

Execute an Event Timing Description (ETD).

.EE [• *etd-spec* [• *start-time*]]

etd-spec: 0..50_{10}

start-time: 0..FFFF FFFF FFFF [.0..FFFFFFFF]

The .EE command executes a Event Timing Description (ETD) which was previously loaded using the .LT command. The *start-time* argument specifies the time (in reduced BAT with optional fractional part) at which the ETD is to begin execution. If the *start-time* argument is not present then the ETD begins approximately 1 second after the command is received. The *etd-spec* argument specifies which ETD buffer is to be used. If it is not given it is assumed to be zero. An ETD which has been loaded into memory may be executed any number of times using the .EE command. The .EE command returns the error code immediately and the ETD is executed in the background.

.EI .EI

Initialise the event generator.

.EI

This command initialises the event generator. It clears the event generator outputs, stops the generation of events, and clears the software registers and all ETD buffers. This command should be executed before any commands which deal with the event generator.

.GT .GT

Get the current atomic time and accumulated leap seconds (DUTC).

.GT

This command returns the current atomic time and DUTC. The command uses the event generator to grab a frame from the clock and then decodes the time and DUTC from it. The output data block consists of a single line containing a 64 -bit unsigned hexadecimal number representing the Binary Atomic Time (BAT) in microseconds, followed by a space, followed by a hexadecimal number representing the DUTC.

.LT .LT

Load an Event Timing Description (ETD).

.LT [• *etd-spec*]
etd-spec: 0..50_{10}

This command loads an Event Timing Description (ETD). The ETD is contained in an input data block following the command. It contains the control information for the event generator outputs (see the introduction to this section). This command does not execute the ETD but simply parses it into a memory data structure. Once in memory an ETD may be executed using the .EE command which also sets the start time for the ETD. In this way an ETD may be executed any number of times using a different starting time for each run.

A number of ETDs may be present in the system at any time and the *etd-spec* argument specifies which of the ETD buffers is to hold the ETD. If it is not given it is assumed to be zero. If the given ETD buffer contains a previous definition it is overwritten.