

Communication Protocol and Primitive Execution

V2.5
April 28, 2000

1 Revision History

V0.1, January 13, 2000 (dpf)

- ◆ First incomplete draft.

V0.2, January 14, 2000 (dpf)

- ◆ Introduction added.
- ◆ section on message structures, message packets and primitives added
- ◆ Updated definition of messages as per group discussions.

V0.3, January 17, 2000 (dpf)

- ◆ Fixed a bug and cleaned up host-MasterDSP handshaking.
- ◆ Began including error conditions in host-MasterDSP handshaking.

V0.4, January 18, 2000 (dpf)

- ◆ Simplified host-MasterDSP handshaking as per group discussions.

V1.0, January 21, 2000 (dpf)

- ◆ ListLength included in MessagePacket as per group discussions.
- ◆ Added more operational detail regarding writing PacketLists through the InputFIFO to a final destination in the MasterDsp's memory space.

V1.1, February 8, 2000 (dpf)

- ◆ Extensive elaboration of the host to MasterDsp message passing protocol.

V1.2, February 9, 2000 (dpf)

- ◆ Comments on version 1.1 taken into account.
- ◆ Added numberOfPrimitives to 1st MessagePacket of a PacketList.

V1.3, February 10, 2000 (dpf)

- ◆ Spliced in most of primitive list document from D. Jared and J. Hill. Still need to go through it and make sure it is consistent with current thinking.

V1.4, February 14, 2000 (jch)

- ◆ J.C.Hill. Tidied up a few mistakes in transposing the primitive list into this document. Also a new additions and clarifications to the registers.

V2.0, February 18, 2000 (dpf)

- ◆ A couple of deletions from the list of primitives. A lot of work is required there.

- ◆ Reworked many sections to reflect the removal of the input and output FIFOs from the board. Still need to update the message passing description.

V2.1, leap day (dpf)

- ◆ Elaboration of primitive list. Also included legal values of primitive attributes.
- ◆ Rewrote host to MasterDsp message passing to reflect the removal of the input and output FIFOs from the ROD. Checksum and listlength errors dealt with using the circular error buffer. The new description also includes the return data case.
- ◆ Included new diagrams from T. Meyer.
- ◆ Added a couple of suggestions for how to read the circular buffers.

V2.2, March 2, 2000 (dpf)

- ◆ Included comments from March 2 ROD software phone meeting.

V2.3, March 18, 2000 (dpf)

- ◆ Added structure definition of error, information and diagnostic buffers. (see also dspBuff.h)
- ◆ Incorporated many written comments from T. Meyer, J. Hill and L. Tomasek.
- ◆ Incorporated remainder of ROD controller address map into the general read/write primitive.
- ◆ Added more primitives for ROD debugging.

V2.4, April 13, 2000 (jch)

- ◆ Update version number to V2.4. Not sure the content changes justify it, but I hope significant changes will follow. Remove S-link primitive, update S-link definitions in registers.
- ◆ April 18, 2000 (jch): Rewrite MasterDSP primitives 4) and 5) as per latest ideas.
- ◆ April 19, 2000 (jch): More changes to primitive 4) to include pixels.

V2.5, April 20, 2000 (dpf)

- ◆ Made a few minor changes agreed upon during meetings between Lukas, Tom and Damon. Some handshake bits added to the rodStatusRegister, slight change to primitive execution protocol.
- ◆ April 28, 2000 (jch): tried to clarify checksumWC primitives.

To do

- ◆ Fill in section on communication between master DSP and slave DSPs.
- ◆ Figure out how to include references for figures.
- ◆ Resolve red "ink"

2 Introduction

This document is intended to describe the lowest level software required to pass messages between the host processor and the MasterDsp and between the MasterDsp and the SlaveDsp. The communication protocols at these two interfaces should be similar.

First is a description of the overall message passing scheme. Important concepts such as primitives and primitive lists are introduced.

Then the host-MasterDsp protocol is described in detail. Described are the primitive list passing protocol and message system related error conditions.

Finally the MasterDsp-SlaveDsp protocol is described in detail.

3 Primitives and PrimitiveLists

3.1 Primitive

A **Primitive** is a message containing a command instructing a DSP to execute a single action. Primitives may be nested. This means the command contained in a primitive may be to retrieve from memory a (commonly executed) list of primitives which has been previously assembled and stored locally on the ROD and execute the actions in that list.

A primitive consists of a PrimitiveHeader and a PrimitiveBody.

- ◆ The **PrimitiveHeader** consists of three words.
 - **PrimitiveLength** is the number of words in the primitive.
 - **PrimitiveIndex** counts primitives which are sent to the DSP. This number is unique within a PrimitiveList (defined below). It used to communicate back to the sender the status of processing the list.
 - **PrimitiveID** identifies what the primitive is, e.g. read BOC register.
- ◆ The **PrimitiveBody** includes any parameters or data which must be transmitted with the primitive.

3.2 PrimitiveList

A **PrimitiveList** is a list of ≥ 1 primitives which the requestor, VME host/MasterDsp, requires the executor, MasterDsp/SlaveDsp, to process and execute without further communication. That is, the requestor only needs to be informed that the execution of the last primitive of a PrimitiveList has been completed.

Though there is no communication *required* during the execution of a primitive list, the executor does provide the requestor with information about the execution of the primitives, e.g. what was the

most recently executed primitive, via the **RodStatusRegister** for host:MasterDsp communication or via a SlaveDsp mailbox for MasterDsp:SlaveDsp communication.

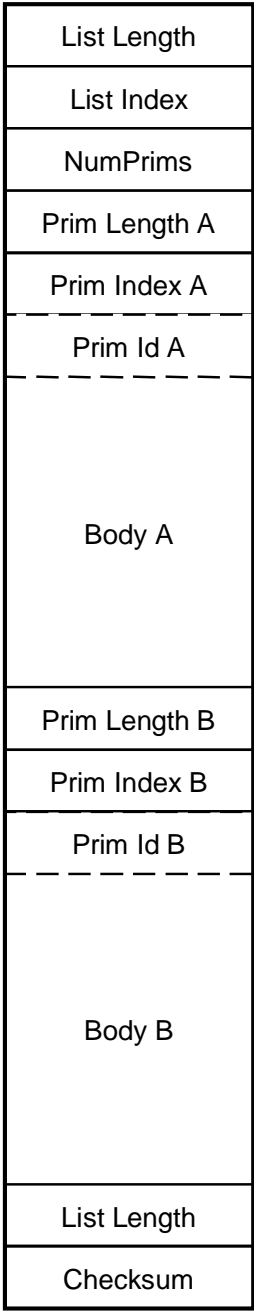
The primitives in a PrimitiveList are executed in the order in which they appear in the PrimitiveList. The execution of a primitive does not begin until the previous primitive has been completed. In particular, if there is return data associated with primitive **n**, the executor will not execute primitive **n+1** until the data associated with primitive **n** has been received and placed at the requested location.

A PrimitiveList consists of a ListHeader, a ListBody and a ListTrailer.

- ◆ **ListHeader** consists of three words
 - **ListLength** is the number of words in the PrimitiveList.
 - **ListIndex** is an accounting ID for the list.
 - **NumberOfPrimitives** is the number of primitives in the list.
- ◆ **ListBody** is the list of primitives in the order in which they are to be executed.
- ◆ **ListTrailer** consists of two words
 - **ListLength** is a repeat of ListLength from the ListHeader.
 - **ListChecksum** is included as a check of the integrity of the transmitted data. Each bit in this 32 bit word is a cumulative XOR of the corresponding bit from a subset of the words in the PrimitiveList. Initially the subset will probably be the entire list up to ListChecksum. At a later date it could be the 1st N words of the list. (Calculating ListChecksum could turn out to be a CPU hog for both CPUs involved in message passing.)

The structure of a PrimitiveList is depicted in figure "Message Format"

Message Format



Version 2.2a
23 Feb 2000

PrimitiveList with 2 primitives.

4 Message System Error Conditions

NOTE: It is planned that most error conditions will be communicated via the message system. However, it may be necessary to communicate message system errors via interrupts.

These errors will be placed in the InterruptID field of the RODStatusRegister

1. Timeout
2. Bad Checksum
3. Input buffer overflow
4. Output buffer overflow

5 Primitives

5.1 Common primitives, used by both the MasterDsp and the SlaveDsp

1) Place command list in DSP

Attributes:

- a) Name of list
- b) Data(command list)

2) Play command list in DSP *(We need to work out how to communicate variable parameters into a stored primitive list.)*

Attributes:

- a) Name of list

3) Read stored command lists, returns the number of stored lists and the list of list names.

4) Read details of a stored command list

Attributes:

- a) List name

5) Reset buffer

Attributes:

- a) Buffer name
 - 1) -1 ALL
 - 2) 1 INPUT
 - 3) 2 OUTPUT
 - 4) 3 CONFIGURATION
 - 5) 4 ERROR
 - 6) 5 INFORMATION
 - 7) 6 DIAGNOSTIC

6) Set Buffer Read Mode

Attributes:

- a) Buffer name
 - 1) 4 ERROR
 - 2) 5 INFORMATION
 - 3) 6 DIAGNOSTIC
- b) Buffer read mode
 - 1) 0 RINGBUFF(set readPtr to writePtr after read)
 - 2) 1 LINBUFF (set readPtr and writePtr to BASE after read)

7) Set Buffer Overflow Mode

Attributes:

a) Buffer name

- 1) 4 ERROR
- 2) 5 INFORMATION
- 3) 6 DIAGNOSTIC

b) Buffer write mode

- 1) 0 NOOVERWRITE (throw out new data on overflow)
- 2) 1 OVERWRITE (overwrite old data on overflow)

8) Set input upstream list checksumWC (incoming primitive list)

As discussed in section 3.2, it will be possible to use only the first N words of the primitive and reply lists for the checksum calculation. This primitive, primitive 9) following, and primitives 10) and 11) for the MasterDsp are provided to set how many words are to be used. Note that the sender and receiver must agree on the number of words to be used.

Attributes:

a) ChecksumWC

- 1) 0 checksum checking is turned off
- 2) 0xFFFFFFFF checksum checking is on for all words in the list
- 3) N use the first N words for checksum checking

9) Set output upstream list checksumWC (outgoing reply list)

Attributes:

a) ChecksumWC

- 1) 0 checksum checking is turned off
- 2) 0xFFFFFFFF checksum checking is on for all words in the list
- 3) N use the first N words for checksum checking

5.2 MasterDsp primitives

1) Set ROD mode

Attributes:

a) Mode

- 1) 1 COMMAND
- 2) 2 RUN

On power up or hard reset, the ROD will initialize itself and go into COMMAND mode.

2) Write or read a register or memory location(s)

(The destination addresses were taken from the current version of the RodControllerMemoryMap, <http://design.lbl.gov/~mlnagel/CurrentHWDocs/index.html> There are some known errors there which have not been corrected here, e.g. an M byte wide register at (byte) address n is often followed by another register at (byte) address n+1 rather than n+M. The documents will remain in synch, i.e. as RodControllerMemoryMap is updated this document will be updated. It is also possible that this document will replace RodControllerMemoryMap.)

If a destination is read only, from the point of view of the MasterDsp, this is indicated by a **RO** superscript.

Where there is a range in the destination address field a symbol is often used. The symbols are in bold face. Their meanings and legal ranges are:

F = formatter number

L = data link number within formatter

M = condensed mode mask number for 16 SCT data links
NN = data link number
PP = data link number
QQ = control link number

DETECTOR	modules	F	L	NN	M	PP	QQ
SCT	48	0-7	0-b	0-2f, 40-6f	0-5	0-5f	60-8f
B-layer	6	0-1	0-2	0-2, 40-42	x	0-5	60-65
pxl brl	26	0-6	0-3	0-c, 40-4c	x	0-19	60-79
pxl fwd	28	0-6	0-3	0-d, 40-4d	x	0-1b	60-7b

Where the size of the data field differs for the SCT and pixels, the size in bits is given as SCT-size/pixel-size.

Attributes:

a) mode

- 1) 1 READ
- 2) 2 WRITE

b) destination

	name	address	size	object
1)	ERROR_FLAGS[F _{n...0} , L _{m...0}] ^{RO}	0x00FL0	15b	formatter
2)	CONFIGURATION[F _{n...0} , L _{m...0}]	0x00FL1	4b/2b	formatter
(1)	b0: 0/1 = static mask: link on/off			
(2)	b1: 0/1 = format data/xmit raw data			
(3)	b2: 0/1 = condensed/uncondensed format (SCT only)			
(4)	b3: 0/1 = edge mode (SCT only)			
3)	HEADER_TRAILER_LIMIT[F _{n...0}]	0x00Ff0	8b/9b	formatter
4)	ROD_BUSY_LIMIT[F _{n...0}]	0x00Ff1	5b/6b	formatter
5)	READOUT_TIMEOUT_REG[F _{n...0}]	0x00Ff2	7b	formatter
6)	DATA_OVERFLOW_REG[F _{n...0}]	0x00Ff3	8b/10b	formatter
7)	ERROR_MASK	0x008NN	16b	EFB
8)	FORMAT_VRSN_LSB	0x00888	16b	EFB
9)	FORMAT_VRSN_MSB	0x00889	16b	EFB
10)	SOURCE_ID_LSB	0x0088a	16b	EFB
11)	SOURCE_ID_MSB	0x0088b	16b	EFB
12)	CONDENSED_MODE	0x0090M	16b/0b	router
13)	SLINK_EVENT_NSELECT	0x00910	16b	router
14)	DSP0_EVENT_SELECT	0x00920	16b	router
15)	DSP0_EVENT_SELECT_MOD	0x00921	16b	router
16)	DSP0_TRANSER_SIZE	0x00922	16b	router
17)	DSP0_FIFO_WORD_COUNT ^{RO}	0x00923	16b	router
18)	DSP1_EVENT_SELECT	0x00930	16b	router
19)	DSP1_EVENT_SELECT_MOD	0x00931	16b	router
20)	DSP1_TRANSER_SIZE	0x00932	16b	router
21)	DSP1_FIFO_WORD_COUNT ^{RO}	0x00933	16b	router
22)	DSP2_EVENT_SELECT	0x00940	16b	router
23)	DSP2_EVENT_SELECT_MOD	0x00941	16b	router
24)	DSP2_TRANSER_SIZE	0x00942	16b	router
25)	DSP2_FIFO_WORD_COUNT ^{RO}	0x00943	16b	router
26)	DSP3_EVENT_SELECT	0x00950	16b	router
27)	DSP3_EVENT_SELECT_MOD	0x00951	16b	router
28)	DSP3_TRANSER_SIZE	0x00952	16b	router
29)	DSP3_FIFO_WORD_COUNT ^{RO}	0x00953	16b	router
30)	PENDING_EVENT_COUNT	0x011PP	4b	RR FPGA
31)	DATA_LINK_TIMING	0x020PP	16b	BOC
32)	CTRL_LINK_TIMING	0x020QQ	16b	BOC
33)	COARSE_TIMING	0x02090	16b	BOC
34)	INPUT_MEM_A_LSB	0x01800	48b	frmtr in mem

35)	INPUT_MEM_A_MSB	0x01804	48b	frmtr in mem
36)	INPUT_MEM_B_LSB	0x01808	48b	frmtr in mem
37)	INPUT_MEM_B_MSB	0x0180c	48b	frmtr in mem
38)	INPUT_MEM_A_CTRL0	0x01810	32b	frmtr in mem
39)	INPUT_MEM_A_CTRL1	0x01814	32b	frmtr in mem
40)	INPUT_MEM_B_CTRL0	0x01818	32b	frmtr in mem
41)	INPUT_MEM_B_CTRL1	0x0181c	32b	frmtr in mem
42)	DEBUG_MEM_A_LSB	0x01820	43b	frmtr out mem
43)	DEBUG_MEM_A_MSB	0x01824	43b	frmtr out mem
44)	DEBUG_MEM_B_LSB	0x01828	43b	frmtr out mem
45)	DEBUG_MEM_B_MSB	0x0182c	43b	frmtr out mem
46)	DEBUG_MEM_A_CTRL0	0x01830	32b	frmtr out mem
47)	DEBUG_MEM_A_CTRL1	0x01834	32b	frmtr out mem
48)	DEBUG_MEM_B_CTRL0	0x01838	32b	frmtr out mem
49)	DEBUG_MEM_B_CTRL1	0x0183c	32b	frmtr out mem
50)	EVENT_MEM_A_LSB	0x01840	46b	EFB out mem
51)	EVENT_MEM_A_MSB	0x01844	46b	EFB out mem
52)	EVENT_MEM_B_LSB	0x01848	46b	EFB out mem
53)	EVENT_MEM_B_MSB	0x0184c	46b	EFB out mem
54)	EVENT_MEM_A_CTRL0	0x01850	32b	EFB out mem
55)	EVENT_MEM_A_CTRL1	0x01854	32b	EFB out mem
56)	EVENT_MEM_B_CTRL0	0x01858	32b	EFB out mem
57)	EVENT_MEM_B_CTRL1	0x0185c	32b	EFB out mem
58)	DSP0_HPIC	0x80000	16b	DSP0 HPIC
59)	DSP0_HPIA	0x88000	16b	DSP0 HPIA
60)	DSP0_HPID_I	0x90000	16b	DSP0 HPID, inc
61)	DSP0_HPID	0x98000	16b	DSP0 HPID
62)	DSP1_HPIC	0xa0000	16b	DSP1 HPIC
63)	DSP1_HPIA	0xa8000	16b	DSP1 HPIA
64)	DSP1_HPID_I	0xb0000	16b	DSP1 HPID, inc
65)	DSP1_HPID	0xb8000	16b	DSP1 HPID
66)	DSP2_HPIC	0xc0000	16b	DSP2 HPIC
67)	DSP2_HPIA	0xc8000	16b	DSP2 HPIA
68)	DSP2_HPID_I	0xd0000	16b	DSP2 HPID, inc
69)	DSP2_HPID	0xd8000	16b	DSP2 HPID
70)	DSP3_HPIC	0xe0000	16b	DSP3 HPIC
71)	DSP3_HPIA	0xe8000	16b	DSP3 HPIA
72)	DSP3_HPID_I	0xf0000	16b	DSP3 HPID, inc
73)	DSP3_HPID	0xf8000	16b	DSP3 HPID

Note in the above 16 destinations, address lines 15 and 16 of the Master DSP's EMIF, EA[16:15], map to the HCNTRL lines of the Slave DSP's HPI port, HCNTRL[1:0]. In addition, Master DSP EA[2] controls Slave DSP HWIL.

74)	ROD_STATUS_REG_0	0x01000	32b	RR FPGA
75)	ROD_STATUS_REG_1	0x01004	32b	RR FPGA
76)	ROD_STATUS_REG_2	0x01008	32b	RR FPGA
77)	VME_CMND_REG_0 ^{RO}	0x01020	32b	RR FPGA
78)	VME_CMND_REG_1 ^{RO}	0x01024	32b	RR FPGA
79)	RESERVED_REG_0	0x0100c	32b	RR FPGA
80)	RESERVED_REG_1	0x01010	32b	RR FPGA
81)	RESERVED_REG_2	0x01028	32b	RR FPGA
82)	RESERVED_REG_3	0x0102c	32b	RR FPGA
83)	RESERVED_REG_4	0x01030	32b	RR FPGA
84)	STATUS_LED	0x01070	8b	RR FPGA
85)	FE_CMND_MASK_0_LSB	0x01180	32b	RR FPGA
86)	FE_CMND_MASK_0_MSB	0x01184	32b	RR FPGA
87)	FE_CMND_MASK_1_LSB	0x01188	32b	RR FPGA

88)	FE_CMND_MASK_1_MSB	0x0118c	32b	RR_FPGA
89)	FE_CMND_CONFIG	0x01190	16b	RR_FPGA
90)	READOUT_MODE_A	0x01300	16b	formatter
91)	READOUT_MODE_CONTROL_A	0x01302	16b	formatter
92)	READOUT_MODE_STATUS_A	0x01304	16b	formatter
93)	READOUT_MODE_B	0x01306	16b	formatter
94)	READOUT_MODE_CONTROL_B	0x01308	16b	formatter
95)	READOUT_MODE_STATUS_B	0x0130a	16b	formatter
96)	L1_DELAY	0x0130c	16b	RR_FPGA
(1)	Constraint: L1_DELAY > 27 (length of CAL command)			
97)	DYNAMIC_MASK	0x01400	16b	EFB
98)	DYNAMIC_MASK_CONTROL	0x01402	16b	EFB
99)	DYNAMIC_MASK_STATUS	0x01404	16b	EFB
100)	EVENT_HEADER_FIFO_CONTROL	0x01406	16b	EFB
101)	EVENT_HEADER_FIFO_STATUS	0x01408	16b	EFB

c) addressIncrement

When dataLength is greater than 1, this is the number of bytes by which the address will be incremented between the reads or writes.

d) dataStore

For read operations this indicates where the data should be placed. The DSP will build something like a primitive with the return data and place it at the requested location.

1) 0xffffffff

Append the return data primitive to the most recent return data primitive written to the output message buffer of the SDRAM. The 1st return data packet of any primitive list starts at the 1st location in the output message buffer. When all return data "primitives" are available fields in the return message analogous to list length, list index, number of primitives, and checksum are filled in.

2) 0XXXXXXXX

Place the return data primitive at the specified address.

e) dataLength

This is the number of data words to read or write. The address is incremented between successive reads and writes according to addressIncrement.

f) data

The data to be written; write operations only. This can be a multiword field depending on dataLength.

3) Write to or read from detector mounted ICs (ABC, pixel FE, MCC)

The bitstream provided in the data field below will be transmitted directly to the SPI. It is assumed that any required configuration will have been performed beforehand via other primitives. This includes, for example, setting the FE_CMND_MSK registers and, in the case of reads, setting up the ROD data path to deal with the return data.

In the case of the pixel FE configuration data writes, the 1st part of the command is transmitted at 40 MHz (this is the part which sets up the MCC) and the 2nd part is transmitted at 5 MHz (this is the actual bit stream to be transmitted to the FE chip registers). It will be the responsibility of the host processor to pad these fields.

(This needs some work. ABC reads are done by setting the FE chip configuration register and issuing a trigger - if memory serves, John?? Do we want the VME host to be responsible for setting the fast command mask etc, or do we want the VME host to be able to

issue a primitive which says send me the register contents of module N and have the Master DSP do everything? For pixels, there is a lot of setting up to do in order to read any given register. Do we want the VME host to have to do all of this setting up or do we want the host to be able to say send me the contents of the ??? register of module M or FE chip C,M?? dpf)

Attributes:

a) DataLength

This is the length in 40 Mbps bits of the data to be written.

b) Data

4) Write to or read from the configuration database (pixel and SCT)

This primitive allows writing or reading specific elements from the configuration database on the ROD. On a write, the data in attribute c below must be arranged in the same byte order as it appears in the database. For reads, the return data will be organized in the return message buffer by the DSP in the same byte order as it appears in the database. In attribute b, module number 0xFF means all modules and chip number 0xFF means all chips. The details of the items accessed are different for pixels and SCT. Setting the value of element = 14 (ie. 0xE) or 15 (0xF) in attribute b allows access to a complete part of the data structure, which occupies contiguous areas of SDRAM, and hence can be accessed more efficiently by via direct access to the memory blocks.

I've changed things in the SCT part so that the chip number occupies 8 bits. This caters for the pixels and avoids too many differences. In principle we could then incorporate the "select" bit in the SCT chip number, but personally I don't like this idea. jch.

Attributes:

a) mode

1) 1 Read

2) 2 Write

b) Destination

(module)<<12) + (chip<<4) + element

Element: SCT

1) 1 Mask Register

2) 2 Configuration Register

3) 3 Threshold/Calibration Register

4) 4 Bias Register

5) 5 Strobe Register

6) 6 Trim DACs

7) 8 Select (chip number ignored)

8) 14 Access trim DACs for one chip (oneChipDacConfig).

If chip = 0xFF, access structure for one module (oneModuleDacConfig).

If module = 0xFF, access complete ROD structure (oneRodConfig).

9) 15 Access registers for one chip (oneChipRegConfig).

If chip = 0xFF, access whole module structure (oneModuleRegConfig).

If module = 0xFF, access complete ROD structure (oneRodConfig).

Element: pixel

1) 1 TDAC0

2) 2 TDAC1

3) 3 TDAC2

4) 4 readoutmask

5) 8 CSR (chip number ignored)

- 6) 9 CAL (chip number ignored)
- 7) 10 FEEN (chip number ignored)
- 8) 15 Access structure for 1 chip (oneChipConfig)
If chip = 0xFF, access structure for one module (oneModuleConfig).
If module = 0xFF, access complete ROD structure (oneRodConfig).

c) data (writes only)

5) Reconfigure a FE module from the local database in the SDRAM

It may be a good idea that the DSP is responsible to verify the checksum of the data it reads from the SDRAM data structure. This requires that we keep some checksum information in the database. We can discuss what we want the granularity of this information to be (per module, per chip, ...), determined by the flexibility we need to provide for reconfiguration. It is probably a good idea in the beginning we allow reconfiguration of all modules of a ROD or of any single module via this primitive, i.e. we do not allow specification of any combination of scattered registers in scattered modules. (There may be a runtime feature by which the MasterDsp regularly refreshes some subset of the FE registers, but this would not be under direct control of primitives. Rather it would be set up at the start of run.) dpf

Attributes:

- a) (module<<8) + chip, where module = 0xFF means all modules, chip = 0xFF means all chips.

6) Set CAL and L1A sequence parameters

This will be the method for setting parameters of the L1A and CAL commands for straightforward calibration and noise runs. A sequence is either CAL-L1A, CAL-L1A-L1A, L1A, or L1A-L1A. During the execution of these sequences, initiated by the "Issue CAL and L1A sequences" primitive below, the interval between the sequences and the interval between the individual commands of a sequence are fixed by the following attributes.

Attributes:

- a) number of sequences
- b) interval between sequences (End of sequence n to start of sequence n+1)
- c) interval between CAL pulse and 1st L1A. (0 = no CAL pulse)
- d) interval between 1st L1A and 2nd L1A (0 = no 2nd L1A)
- e) event type (to trap data in proper DSP)

7) Issue CAL and L1A sequences

This primitive causes the number and type of sequences described by the last "Set CAL and L1A sequence parameters" to be issued. It is assumed that the CAL and L1A sequence parameters, the data path, FE_CMND_MASK, the SlaveDSPs, and the detector mounted electronics have been set up prior to issuing this primitive.

8) Send primitive list to a SlaveDsp

- a) SlaveDSP number
- b) primitive list

This is a primitive list which will be transmitted to the input message buffer of a SlaveDSP. The primitive list has the same format as a primitive list for the MasterDSP. This allows the MasterDSP to transmit primitives to a SlaveDSP at specific times during the execution of its own primitive list.

The entire SlaveDsp PrimitiveList is embedded in the MasterDsp PrimitiveList as attribute b. The MasterDsp knows how many words to send to the slave by reading the first word of the slave's

primitive list before writing it. In the initial version of the software, the MasterDsp will not proceed with it's own primitive list until it receives a DspAck from the SlaveDsp.

9) Read SlaveDsp data

The requested data will be placed in the output message buffer of the MasterDSP. (Alternatively, the SlaveDSP SDRAM can be directly accessed via primitive 1.)

Attributes:

- a) DSP number
- b) Data class *(This field must be worked on. It should provide the MasterDSP with enough information to know where the data is sitting in the memory space of the slave DSP, and how much data is there. The following fields are place holders.)*

- 1) Histogram
- 2) Trapped event
- 3) Fitted data
- 4) Error counter

10) Set output downstream list checksumWC (primitive list to slave)

See the discussion under common primitive 8) (section 5.1).

Attributes:

- a) ChecksumWC
 - 1) 0 checksum checking is turned off
 - 2) 0xFFFFFFFF checksum checking is on for all words in the list
 - 3) N first N words of the list are used for checking
- b) slave DSP number

11) Set input downstream list checksumWC (reply list from slave)

Attributes:

- a) ChecksumWC
 - 1) 0 checksum checking is turned off
 - 2) 0xFFFFFFFF checksum checking is on for all words in the list
 - 3) N first N words of the list are used for checking
- b) slave DSP number

12) CheckOutput

The MasterDsp responds with a known message to test the integrity of the return data path. The message can originate in the MasterDSP, in the SDRAM or in the RRFPGA and should check as exhaustively as possible all pin to pin connections.

Attributes:

- a) Destination
 - 1) 0 Master DSP internal memory
 - 2) 2 RRFPGA *(add a 0xdeadbeef register??)*

13) Echo

Write an arbitrary data word to the MasterDsp which echoes it back to the host. The word can be bounced from the MasterDSP, the SDRAM or the RRFPGA.

Attributes:

- a) Data
- b) Destination
 - 1) 0 Master DSP internal memory
 - 2) 1 SDRAM
 - 3) 2 RRFPGA *(add a test R/W register??)*

14) Memory test

This primitive causes repeated writes/readback sequences of all locations in the MasterDSP's SDRAM.

5.3 SlaveDsp primitives

- 1) **Set mode** *(or we may just have 2 versions of slaveDsp code)*
 - a) Mode (a bit field)
 - 1) 1 calibration
 - 2) 2 error count and event synchronization correction
 - 3) 4 event trapping (monitoring events)
 - 4) 8 collect occupancy histograms
- 2) **Setup for calibration**

Attributes:

 - a) fitting function
 - 1) 1 no fit, keep raw histograms
 - 2) 2 S curve
 - 3) 3-9 reserved
 - 4) 10+n nth order polynomial
 - b) X axis source *(for pixels address mapping needs to be worked out)*
 - 1) element number
 - 2) element number + TOT (pixels only)
 - 3) element number + control variable (e.g. DAC step)
 - c) Y axis maximum
 - 1) 1 1 byte (256 counts)
 - 2) 2 2 bytes (65K counts)
- 3) **Setup for error counter and event synchronization correction**

Attributes:

 - a) resynchronization mode
 - 1) 1 off (do not perform resynchronization)
 - 2) 2 on
 - b) alarm threshold *(may want to define by error type)*
- 4) **Setup for event trapping (monitoring events)** *(Can be deleted if there are no attributes.)*
- 5) **Setup for occupancy plots**

Attributes:

 - a) Y axis maximum
 - 1) 1 1 byte (256 counts)
 - 2) 2 2 bytes (65K counts)
 - b) number of events (-1 = accumulate same number as in the reference occupancy histograms)
 - c) alarm tolerance
 - 1) -1 send raw hists (do not compare with reference)
 - 2) s alarm at s sigma deviation from reference
- 6) **Accumulate reference occupancy histogram**

Attributes:

 - a) Y axis maximum
 - 1) 1 1 byte (256 counts)
 - 2) 2 2 bytes (65K counts)
 - b) number of events
 - c) date
- 7) **Read parameters (Y axis maximum, number of events, date) of current reference occupancy histograms**
- 8) **Process calibration histograms**

Attributes:

 - a) fitting function
 - 1) 2 S curve
 - 2) 3-9 reserved
 - 3) 10+n nth order polynomial

- b) scan parameters *(This should include things like the number of pulses per point, the axis values of the points, etc. From the host side, it may be easier if these parameters are sent down when the scan is being set up rather than when it is finished. People involved in that end of things should comment.)*
- 9) **Clear histogram(s)**
Attributes:
a) histogram number (-1 to clear all)
 - 10) **Clear event counter(s)**
Attributes:
a) event counter (-1 to clear all)
 - 11) **SlaveCheckOutput**
The SlaveDsp responds with a known message. Tests the integrity of the return data path through the RodResourcesFPGA. The message can originate in the SlaveDSP or in its SDRAM and should check as exhaustively as possible all pin to pin connections.
Attributes:
a) Destination
 - 1) 0 Slave DSP internal memory
 - 2) 1 SDRAM
 - 12) **SlaveEcho**
Write an arbitrary data word to a SlaveDsp which echoes it back to the MasterDSP. The word can be bounced from the Slave DSP or its SDRAM.
Attributes:
a) Data
b) Destination
 - 1) 0 Slave DSP internal memory
 - 2) 1 SDRAM
 - 13) **Memory test**
This primitive causes repeated writes/readback sequences of all locations in the SlaveDSP's SDRAM.

6 Communication between the host and the MasterDsp

6.1 General Scheme

The ROD implements a register-like interface to the RCC. That is, there are just a few addresses on each ROD to which the RCC has access. A description of these addresses as well as a comprehensive set of ROD implementation diagrams showing where the addressed object resides can be found at <http://www-wisconsin.cern.ch/~atlas/off-detector/ROD/ROD.html> under Current hardware documents. Communication between the VME host and the MasterDSP is coordinated through the RodStatusRegisters and VmeCommandRegisters.

- 1) **RodStatusRegister0, RodStatusRegister1 and RodStatusRegister2**
These are 32 bit registers to which the ROD has R/W access and the VME host has R access. ROD status bits and fields are defined in these registers. See bit definitions below.
- 2) **VmeCommandRegister0 and VmeCommandRegister1**
These are 32 bit registers to which the ROD has R access and the VME host has R/W access. VME command bits as well as handshaking bits

for PrimitiveList transmission are defined in these registers. See bit definitions below.

6.2 Fields in RodStatusRegister and VMECommandRegister

The following lists and describes the bits in the RodStatusRegister and the VmeCommandRegister. For the bit assignments in these registers, refer to <http://www-wisconsin.cern.ch/~atsiod/registers.h>

RodStatusRegister (R/W from the ROD, R from VME)

1. **outListReady** (handshake)
2. **dspAck** (handshake)
3. **busy** (status, primitive list transmission and primitive execution)
4. **executing** (status, primitive execution)
5. **listIndex[3:0]** (status)
6. **primIndex[19:0]** (execution status), this is updated each time a primitive is executed. It allows the host to monitor the execution of the primitive list and identify where an error has occurred.
7. **nestIndex[7:0]** (execution status) In the case of nested primitive lists, this index updates each time a primitive in the called-up list is completed. *(The sizes of the ListIndex, PrimitiveIndex and NestIndex fields are suggestions only. For ListIndex, 1 bit which the host toggles each time a list is transmitted would probably suffice; we only want to make sure we don't execute the same list twice. We do reserve a 32 bit word in the ListHeader for ListIndex.)*
8. **errBuffNotEmpty**
9. **infBuffNotEmpty**
10. **diagBuffNotEmpty**
11. **primListAborted**
12. **rodReset**
13. **readoutReset**
14. **S-LinkInitialized**
15. **rodBusy** (ROD status)
16. **interruptIssued** (cleared on read) This interrupt could be the result of data being ready in the output FIFO, ROD is busy, command sequence in the input FIFO has completed, etc. The bit can be used to poll the ROD in place of interrupts if the user desires. The exceptions must be queued (whether the mechanism to check the conditions is interrupt- or polling-based), so that a 2nd one cannot be lost while the first is being handled. This is necessary as the interrupts may be from multiple asynchronous sources. A bit in the command register will be written when the existing exception has been dealt with.
17. **interruptID[7:0]** (defines the meaning of the interrupt)
The contents of this field may also be used as part of the interrupt vector to allow the RCC to know the source of the interrupt without having to read this register.
18. **rodMode** 0/1 indicates the ROD is in command/data taking mode
19. **S-LinkXOFF** (ROD status) - indicates whether the S-link has sent an XOFF (ie. LFF# is set).
20. **S-LinkDown** The S-link has gone down (ie. LDOWN# is set).
21. **S-LinkOnOff** The S-link is enabled/disabled (ie. UDOWN# is set or cleared)

22. **S-LinkTest** The S-link is in test mode (ie. UTEST# is set).
23. **efbStopOutput** indicates the Event Fragment Builder is inhibiting formatter output.
24. **routerStopOutput** indicates the Router is inhibiting Event Fragment Builder output.
25. **commandIgnored** (refers to an "Illegal Command" -i.e. a command that is inappropriate under these conditions. It is probably only useful if the bit is cleared after read. It may be of limited use, e.g. if list of commands are being processed.
26. **rodError** (indicates a ROD error has occurred since the last read of this bit, cleared on read)
27. **interruptsEnabled**

VMECommandRegister (R from the ROD, R/W from VME)

1. **inListReady** (handshake)
2. **abortListExecution** (command)
3. **errorBufferReadRequest**
4. **infoBufferReadRequest**
5. **diagnosticBufferReadRequest**
6. **resetRod**
7. **resetReadout** (clears all readout pointers, error counters, ...)
8. **initializesS-Link** (ie. generate URESET#)
9. **enable/disable S-Link** (ie. set or clear UDOWN#)
10. **test S-link** (ie. set UTEST#)
11. **enableInterrupts**
12. **clearException** (interrupt or polling)

6.3 PrimitiveList transmission and PrimitiveList execution

There are many details on the host side which are not described here. Details can be found at <http://www-wisconsin.cern.ch/~atsiod/shared.html>. Click on the test stand software document.

The following describes the procedure for passing a PrimitiveList from the host processor to the MasterDsp and for the extraction and execution of the PrimitiveList.

"polling" means periodically checking a condition before proceeding on this thread; other activities can be executed between checks.

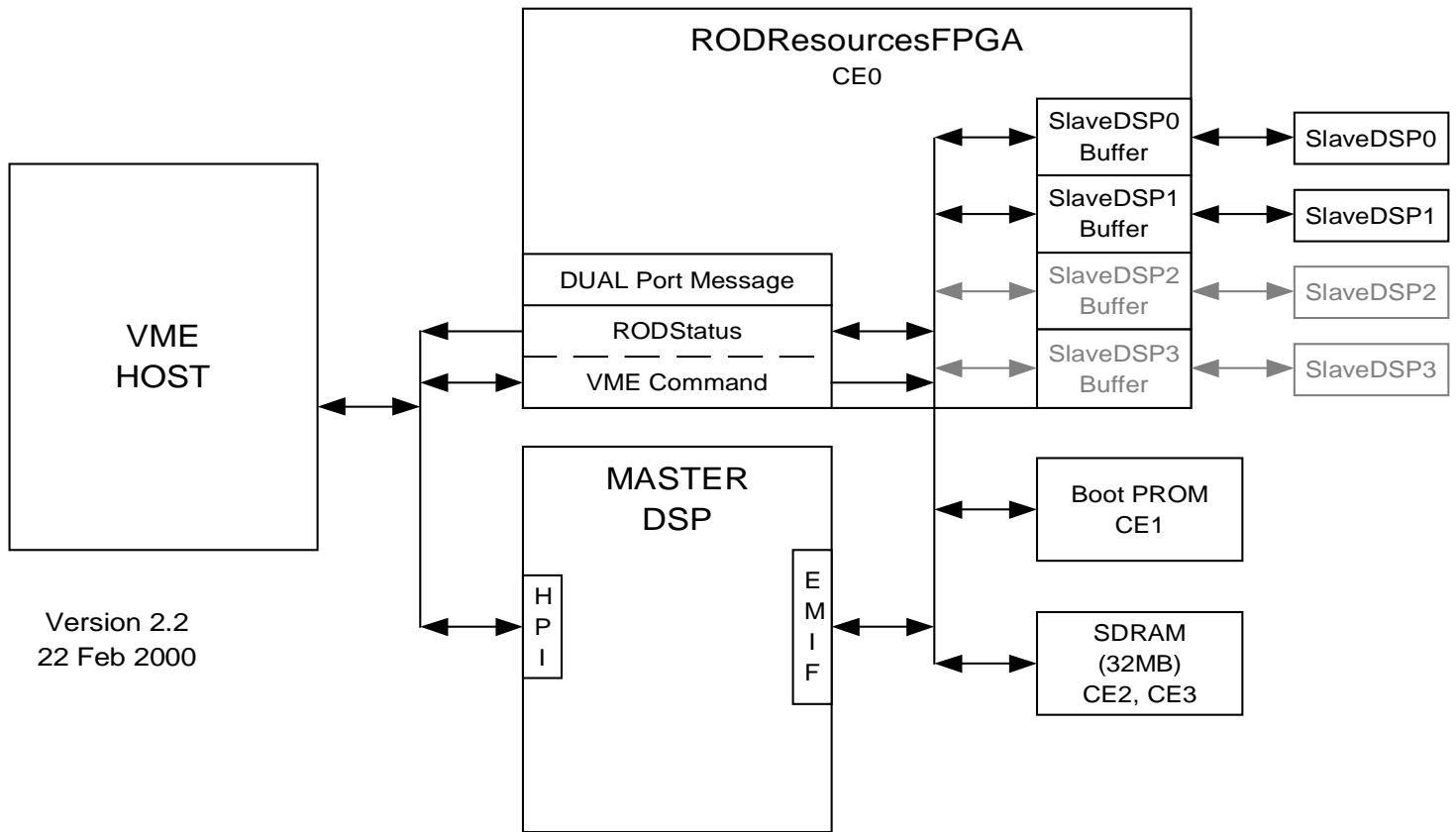
- 1) **host** builds the PrimitiveList to be transmitted to the MasterDsp.
- 2) **host** writes PRIM_BUFF_BASE to HPIA register of MasterDSP.
- 3) **host** writes the PrimitiveList to HPID with autoincrement on.
- 4) **host** polls for **dspAck** == 0.
- 5) **host** sets **inListReady** = 1.
- 6) **host** polls for **dspAck** == 1.

- 7) **MasterDsp**, when it sees **inListReady** sets **busy** bit. (**busy** is not part of the message passing protocol; it is a list processing status bit.)
- 8) **MasterDsp** reads **listLength**, **listIndex** and **numPrims** from the list header and **listLength** and **checksum** from the list trailer.
- 9) **MasterDsp** reads the **primitiveList**, calculates the checksum and verifies that it matches the transmitted checksum. The last word, **ListChecksum** is stored locally in **hostChecksum**. Also verifies that the **listLength** is within bounds and that the **listLength** from the header matches the **listLength** from the trailer. Writes to **errBuff** and returns with an error if necessary. In case of error, proceed to step where **dspAck** is set to 1.
- 10) **MasterDsp** writes **listIndex** to the **RodStatusRegister**. (**listIndex** is not part of the message passing protocol; it is a list processing status bit. It allows the host to check that the correct list is being processed.)
- 11) **MasterDsp** sets **executing** bit. (**executing** is not part of the message passing protocol; it is a list processing status bit only.)
- 12) **MasterDsp** begins processing and executing the primitives in **PrimitiveList**. After the execution of each primitive has been completed, the **MasterDsp** writes its **PrimitiveIndex** to the **RodStatusRegister**. Control is returned to main polling loop between primitives so that **abort** flag and other polled items, e.g. slave mailbox bits, can be checked. The message handler (described here) does not timeout on primitives, it simply waits for control to be returned. If a primitive might timeout this must be handled by the primitive routine itself and a timeout error flag returned to the message handler.
- 13) **MasterDsp**, if there is output data in the reply buffer, calculates the checksum for the output data and writes the wrapper around the output message.
- 14) **MasterDsp** sets **executing** = 0 after all primitives of the **PrimitiveList** have been executed.
- 15) If there is(is-not) return data available in the output message buffer **MasterDsp** sets **outListReady** = 1(0) in the **RodStatusRegister**.
- 16) **MasterDsp** sets **dspAck** = 1 in **RodStatusRegister**.
- 17) **MasterDsp** polls for **inListReady** == 0
- 18) **host** records value of **OutListReady**, from the read of **RodStatusRegister** on which it found **dspAck** == 1.
- 19) **host** sets **InListReady** = 0 in **VmeCommandRegister**.
- 20) **MasterDsp** sets **dspAck** = 0 and **busy** = 0 in **RodStatusRegister**.
- 21) If **OutListReady** was == 1
 - a) **host** sets **HPIA** to **OUTPUT_BUFFER_BASE**

- b) **host** reads 1st word from the output message buffer. This is the length of the reply message in the output message buffer.
- c) **host** reads the remaining data from the output message buffer.

The interface between the host processor and MasterDsp is depicted in figure "Host-Master Interface". This procedure is depicted in figure "Host Writes/Reads with MasterDSP"

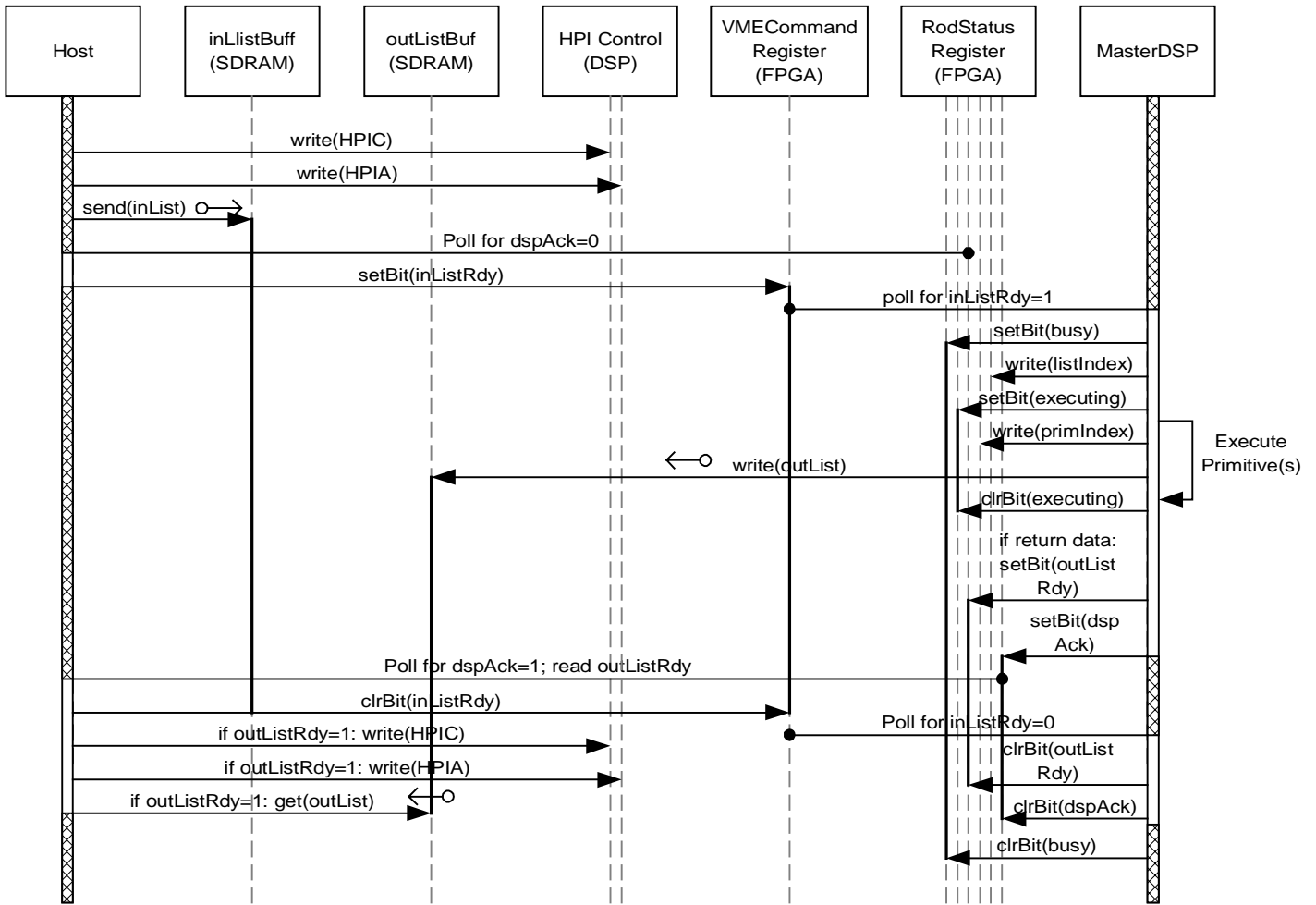
HOST-MASTER INTERFACE



Version 2.2
22 Feb 2000

Sketch of the interface between the VME host processor and the MasterDsp.

Host Writes/Reads with MasterDSP



Processor states: Executing 
 Polling 

Version 1.1a
 23 Feb 2000

Notes:

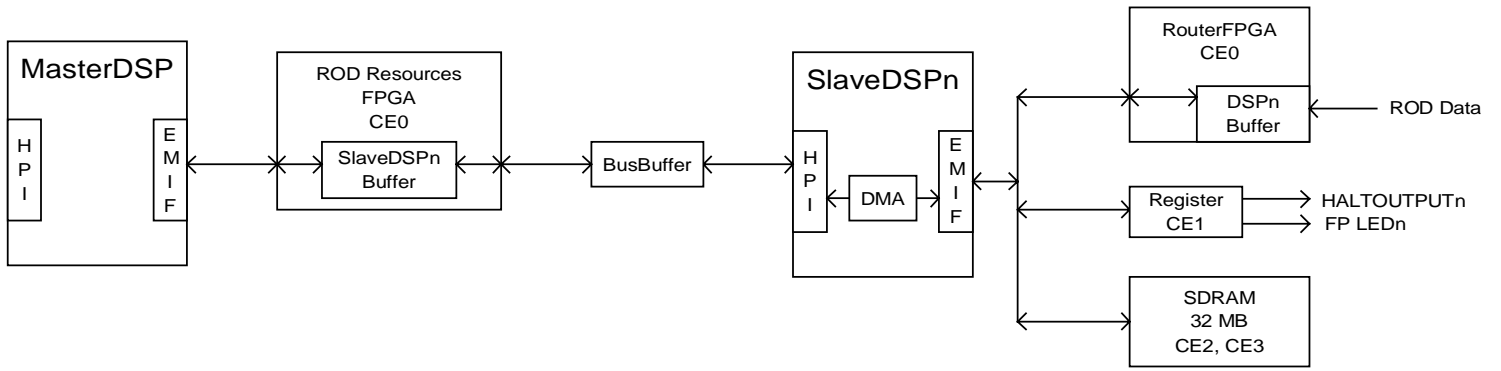
1. outListRdy must be set at or before the time when dspAck is set.
2. If there is no reply data, outListRdy remains 0 and the fetch of outList is omitted.

Sequence diagram showing the host processor passing a **PrimitiveList** to the MasterDsp, the MasterDsp executing the primitives and, if applicable, informing the host that requested data is available.

7 Communication between the MasterDsp and the SlaveDsp

Only pictures for now...

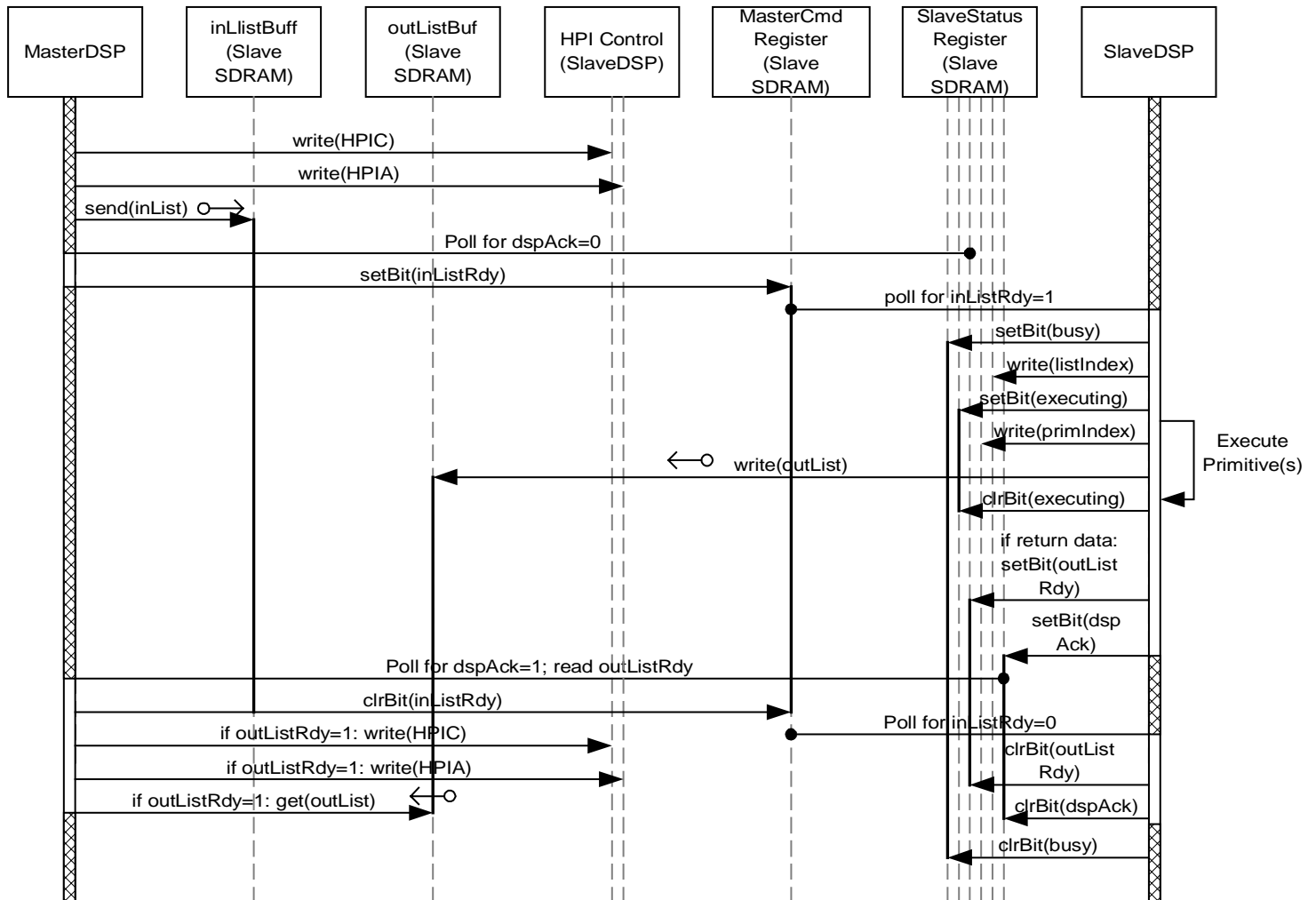
MASTER-SLAVE INTERFACE



Version 1.2
20 Jan 2000

Sketch of interface between the MasterDsp and the SlaveDsp.

Master Writes/Reads with SlaveDSP



Processor states: Executing
 Polling

Version 1.3
 23 Feb 2000

Notes:

1. outListRdy must be set at or before the time when dspAck is set.
2. If there is no reply data, outListRdy remains 0 and the fetch of outList is omitted.

Sequence diagram showing the MasterDsp passing a **PrimitiveList** to a SlaveDsp, the SlaveDsp executing the primitives and, if applicable, informing the MasterDsp that requested data is available.

8 Reading the error, information and diagnostic buffers

These buffers, resident in SDRAM, are each described by a data structure which is resident in internal DSP memory. These structures are called `errBuffer`, `infoBuffer` and `diagBuffer` for the error, information and diagnostic buffers respectively and are of type `static struct asciiBuffer` which is defined as

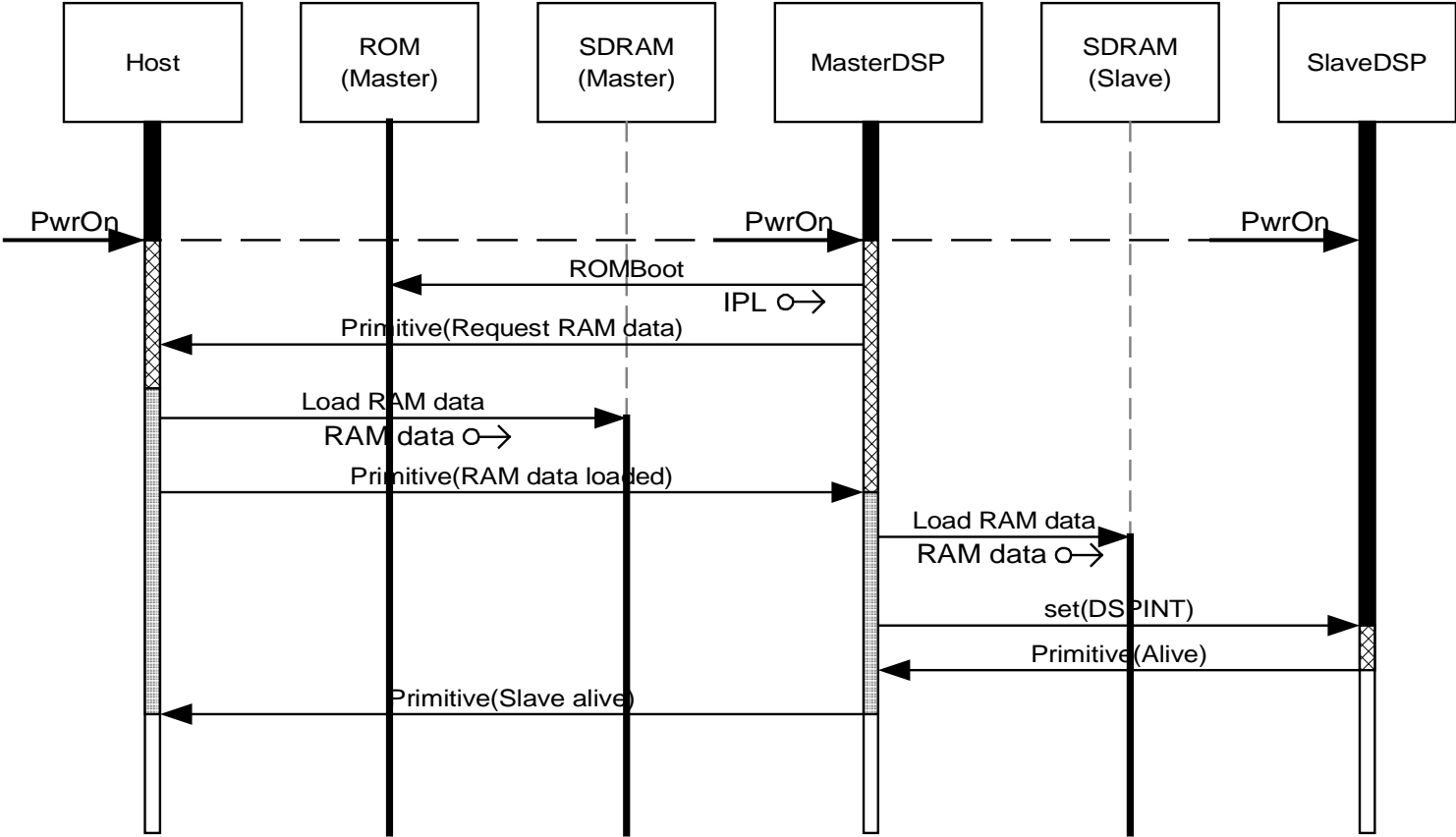
```
static struct xxxBuffer{
    unsigned int dataStart;    /* address of 1st buffer element */
    unsigned int dataEnd;     /* address of last buffer element */
    unsigned int readPtr;     /* address of next element to read */
    unsigned int writePtr;    /* address of last element written */
    int mode;                 /* how to treat pointers after read */
    int overwrite;           /* how to treat overflow condition */
    int overflow;            /* flags buffer overflow condition */
}
```

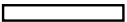



where `xxx` = `err`, `info` or `diag`.

The protocol to read from buffer `xxx` is

- 1) **MasterDsp** checks that `xxxBuffReadRequest == 0`.
- 2) **MasterDsp** sets `xxxBuffNotEmpty = 1` in `RodStatusRegister`. The **MasterDsp** can continue writing to buffer `xxx` until the host indicates that it will read the buffer.
- 3) When the **host** sees `xxxBuffNotEmpty == 1` it sets `xxxBuffReadRequest = 1` in the `VmeCommandRegister`.
- 4) The **host** polls for `xxxBuffNotEmpty == 0`.
- 5) The **MasterDsp** freezes the buffer, i.e. stops writing to it. The **MasterDsp** may finish any pending writes prior to freezing. The buffer is considered frozen after the **MasterDsp** has updated `xxxBufferReadPtr` and `xxxBufferLength` for the last write.
- 6) The **MasterDsp** sets `xxxBuffNotEmpty = 0`.
- 7) The **MasterDsp** polls for `xxxBuffReadRequest == 0`. During this time it can not use the resources which the host will need to read the buffer. (This may be handled automatically via priority.)
- 8) The **host** reads the buffer structure header.
- 9) The **host** reads the buffer contents via the HPI in the usual way. Note that this may require two HPI block reads if the buffer has wrapped around.
- 10) The **host** sets `xxxBuffReadRequest = 0`.
- 11) The **MasterDSP** sets `readPtr`, `writePtr` and `overflow` to the appropriate values.

DSP Initialization

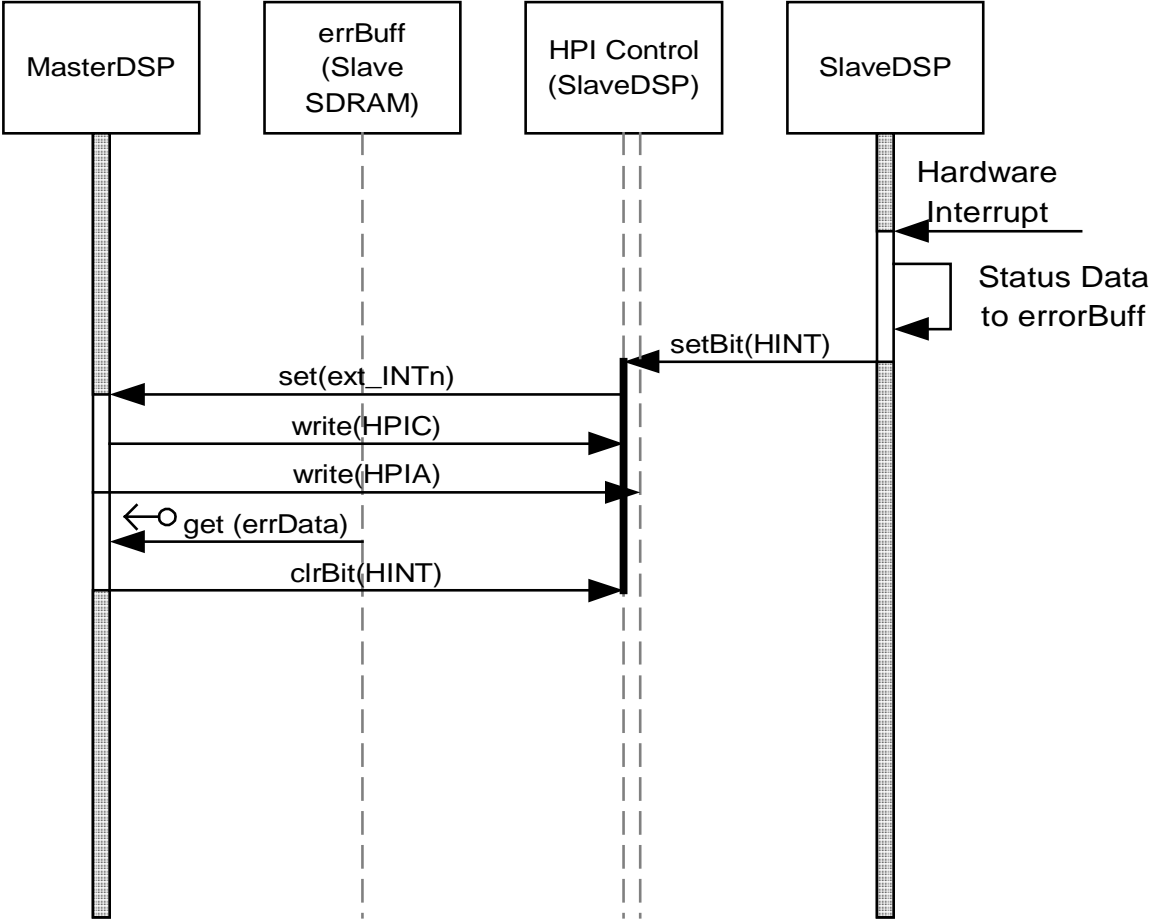


Processor states: Executing 
 Initializing self 
 Initializing others 
 reset 

Version 1.1a
 24 Feb 2000

DSP initialization sequence on ROD power up or reset.

Hardware Interrupt to SlaveDSP



Processor states: Interrupt processing
 Background

Version 1.1
 24 Feb 2000

Sequence for handling hardware interrupts to a SlaveDsp.