ELAN Communications Processor

Reference Manual

meiko

Meiko's full address in the US is:

Meiko Scientific
Reservoir Place
1601 Trapelo Road
Waltham MA 02154

Tel: 617 890 7676
Fax: 617 890 5042

Meiko's full address in the UK is:

Meiko Limited
650 Aztec West
Bristol
BS12 4SD

Tel: 0454 616171
Fax: 0454 618188

Issue status:

Draft	☐
Preliminary	☐
Release	☐
Obsolete	☐

Circulation control:

OVERVIEW

1.1 *Introduction*

The Meiko ELAN communications processor is the interface between processing nodes and the Meiko packet switched network. This network is constructed from switch components that are capable of switching eight bidirectional 55MBytes/s communications channels. Basic packets are between 40 and 320 bytes long, and are routed according to headers that are attached by the communications processor. The switching network supports hardware ACK/NACK for each packet, and implements a lower level protocol for flow control at the byte level. The network allows anywhere to anywhere connectivity, as well as broadcast across selected processor ranges.

1.2 *Major Objectives*

The major objectives for the communications processor are:

- To process in-coming packets from the network as quickly as possible.
- To ensure reliable message delivery.
- To minimise the number of interrupts to the main processor.
- To reduce latency during communication start-up.
- To remove the need for multiple lightweight processes on the main processor.
- To utilise main store bandwidth efficiently,
- To maintain security between unrelated operations.
- To filter out erroneous transmissions before they are transferred onto the network.

1.3 *Processing Units*

The ELAN consists of six individual processing units. In the initial implementation these are all implemented on the same microcoded engine, but future implementations might use different strategies.

The six units are:

- *Input processors* — two of these handle incoming packets from the network.
- *Thread processor* — executes short code fragments to handle checking and outputting of protocols.
- *DMA processor* — executes store-to-store DMAs, and store-to-remote DMAs.
- *Reply processor* — returns results of remote reads.
- *Command processor* — executes commands initiated from the main processor.

Three of the six processing units are capable of outputting packets into the network; these are the thread, output, and reply processors. The reply and dma processors construct packets that are appropriate to their function; that is, dma packets consist of block read or write transactions, and reply packets consist of word write and event transactions. The thread processor constructs packets using its instruction and data stream, and is the only output processor that can construct arbitrary packets.

All three processors must share a single resource, called the outputter, for transmitting their packets onto the network.

All of these processors are capable of virtual operation and can execute in any of 64K contexts. All processors can be considered as executing from queues, and in some cases can cause work to be added to other queues; an input processor, for example, may cause a reply to be added to the reply processors queue.

1.3.1 *Input Processor*

The input processor is defined by the network protocols, as described in B. The characteristic of the packet protocol is that each packet contains all the state information that is needed to process them. Communications are essentially by remote write operations, unlike transputer communications where there is assumed to be a cooperating process at the receiving end. The input processor is therefore stateless, and takes its context from the incoming packet stream.

1.3.2 *Thread Processor*

The thread processor is a small RISC processor with lightweight scheduling capabilities similar to the transputer. A number of special instructions give access to the output port and onto the network. Although the output processor is capable of running complied C programs (at a performance of about 3 to 4 MIPS), it should not be used to offload computation from the main processor; its primary aim is to handle the protocols of the input and output messages, thus ensuring that the main processor is only interrupted when it is really necessary.

1.3.3 *DMA Processor*

The DMA processor performs DMAs either locally (store-to-store), or across the network. This processor converts the data into packet format, and ensures that processes are woken-up when the DMA is complete.

1.3.4 *Reply Processor*

One very important function is the remote reading or read-modify-writing of memory for program and lock control. These are one or two word transactions which require an inputing process to generate an output packet. To help achieve some decoupling between the inputter and outputer for these operations a queue of replies is used. The reply processor works on this queue and generates correct output packets.

1.3.5 *Command Processor*

The command processor executes commands that are issued by the main processor. The interface between the command processor and the main processor has been designed so that communications can be initiated by a simple series of writes from the main processor.

1.4 *Elan Implementation*

The initial implementation of ELAN is a single CMOS sea of gates ASIC with:

- Single microcoded engine.
- 72 words x 64 bits dual ported memory, 640Mbyte/s bandwidth.
- Communications MMU.
- Two Meiko byte wide channels permitting redundant networks.
- Full level 2 Mbus interface.

- Approximately 45K gates, excluding ROM and RAM.
- 208 pin PGA packages.

1.5 *Using This Reference Manual*

This section provides information to help you use this manual. It includes an overview of the manual, a definition of the intended audience, a description of the fonts used and what they mean.

1.5.1 *Contents*

The chapter after this describes the virtual process model used in the ELAN architecture. The following five chapters describe each of the five types of processor on the ELAN processor itself. The final chapter explains the event mechanism and exceptions. The appendices describe in detail the registers and memory maps used by the ELAN processor.

1.5.2 *Audience*

This document is intended to be read by people requiring a detailed insight into the operation and capabilities of the ELAN communications processor. In addition the appendices are sufficiently detailed to enable ELAN device drivers to be written.

VIRTUAL OPERATION

The ELAN contains its own memory management unit (MMU). Any process or work item has a hardware context associated with it, and this is used by the MMU to interpret virtual addresses.

Within the ELAN network, each process has a virtual process number. This allows other processes to reference it by virtual process number. Communication consists of sending transactions to a virtual process, the destination process number is translated by a local process mapping table that is specific to each process. This table is called the virtual process table.

The translation of virtual to physical addresses via context dependent tables provides security and portability. Portability is facilitated because the mapping may be changed at any time. Security is provided, because a process can only communicate with those processes that are explicitly mapped in the table.

Generally the MMU and mapping functions will be under the control of the main processor. This means that only areas of store which have been specifically authorised for communications can be altered. Under kernel control, messages and remote writes can be enabled direct into a user spaces without further main CPU intervention.

2.1 *The Virtual Process Model*

The ELAN virtual process model has been developed to provide usable lightweight process communication within the familiar UNIX style programming model. The model that is used by ELAN incorporates the protection and security of UNIX, with much reduced process switch times.

2.1.1 *Lightweight Processes*

The lightweight processes are provided by hardware assisted scheduling. Many threads of execution may reside within a virtual process. Within a process, threads are not protected from one another, although processes *are* protected from one another, so that one cannot damage the execution (memory map) of any other. This is analogous to UNIX, where processes are protected from each other, although the reliability of any one process is entirely dependent on the programmer that created it.

Processes communicate by reading and writing between each other's memory spaces using network transactions or DMAs. Higher level signalling and locking are provided by a the virtual event mechanism.

2.1.2 *Programming Model - Events*

The Elan model is principally based upon the communicating process model of parallelism. Several paradigms exist here, the most notable being Communicating Sequential Processes (CSP) or, in its more developed form, OCCAM. The CSP concept of synchronisation turns out to be too restrictive for many applications; the ELAN model therefore uses a much looser but more powerful mechanism known as *events*.

An event is a memory location with specific contents defining the state of the event. Events act like user defined interrupts. Processes can non-busily wait on an event, poll events or cause events.

2.1.2.1 Queued Events

Queued events are allowed where several processes may wait on an event queue. As events occur a waiting process will be woken off the top of the queue. This facilitates the writing of secure and efficient resource sharing code.

When queued events are used, only the access to the dual-word event location is treated as an event access; queue handling must be done with local permissions. In this way protected event vectors can provide a signal mechanism across the network, with similar security to a standard trap interface.

2.1.3 *Protection*

The virtual processes are protected from each other by hardware contexts in a MMU. Threads within processes do not require such protection; a thread within its process has the same security as the process itself. A virtual process will normally be acting on behalf of a larger more weighty applications process. This may have a number of co-operating virtual processes providing it with different services or functions. The applications process will not share the same hardware context as a virtual process but instead will have areas of overlapping memory spaces pertinent to the co-operation at hand.

One major problem with a networked system is security; security can usually be provided but at substantial overhead in communication startup. The ELAN communications processor overcomes this by incorporating a paged MMU for all memory accesses. This MMU is appropriate for implementing standard UNIX kernels and has additional access types to provide protection to remote and event accesses. This allows different permissions to be set up for areas of memory accessed by other processes.

Programmers information for the MMU is given in Appendix H.

2.2 *Network Protocol*

The network protocol is accessed from a virtual process by sending packets to other processes. These are groups of transactions which alter the memory and scheduling state of the destination process. A packet is sent by executing an `OPEN` instruction and by specifying the recipient's virtual process number. A number of transactions are then sent by use of the `SEND_TRANS` instruction, and the packet is terminated by use of the `CLOSE` instruction.

2.3 *Virtual Process Table*

To allow control over the virtual processes with which a process can communicate, a table (called the Virtual Process table) is used. These tables reside in physically mapped store and aren't accessible directly by a virtual process. Each virtual process has a different table. The address of the virtual table base is held in a context control block which is loaded as required for each change of context. In addition to the 'permission' aspect of the table a translation is made of virtual process number to physical processor number and hardware context.

The structures that are used to define the virtual process table are defined in Appendix A.

2.4 *Route Table*

After converting a virtual process identifier to a physical processor number, a transmission route for the communication is obtained from the route table. The base of the route table is defined by a register within the communications processor.

2.4.1 *Routing Strategy*

The routing strategy can be altered through the contents of the routing table. This table contains four routes for each destination processor one of which is selected randomly for each packet. Each route is defined by a 16 byte entry, any of which may be the same. The random selection of one-from-four routes is intended to provide some degree of protection from congestion within the network. Random selection has better worst case performance than other more active forms of communications load balancing while being simple to implement.

2.4.2 *Broadcast Communications*

The ELAN allows broadcast communications to a contiguous array of ELITE links. To allow one or more of the destination processes to be omitted from this contiguous array, the MMU is programmed to ignore the incoming messages.

2.4.3 *Route Table Modification*

To stop processors using a route the destination processor has to be removed from the virtual process tables. If a route has to be removed quickly it can be done so by giving NULL entries by zeroing the first of each route in the route table. An attempt to use a NULL route causes the outputting processor to trap.

The table can be placed at the same physical location within node's store to enable easy modification of all tables in the network, via a broadcast command.

INPUT PROCESSOR

The input processor receives packets from the network. Its objective is to remove them and process them as quickly as possible to avoid network congestion.

3.1 Network Packet Protocol

The network byte order is big endian, although communication processors accessing the network may have little endian memory systems provided that they put data onto the network in the network byte order.

A packet is composed of route information, a start-of-packet (SOP) delimiter, a number of transactions, and an end-of-packet (EOP) delimiter. Route data is added to the head of the packet by the sender, and is stripped off as it is used in the network. An inputter removes any extra route information that has not been been stripped by the network up to the SOP.

```
packet = route bytes, SOP, transaction, <transactions,> EOP.
```

3.1.1 Transaction Format

Each transaction within the packet consists of one or more 64-bit words, followed by a 16-bit cyclic redundancy check (CRC). The first double word contains a transaction type, a context, and an address. Transactions can always be handled immediately they are received, and contain all the information that is necessary to execute them. In particular, a transaction can always be handled without doing any further network operations; this avoids cyclic dependencies.

The transaction executed are by the ELAN input registers are detailed in Appendix B "Input processor".

3.1.2 *Packet Acknowledge and Negative Acknowledge*

Where a packet consists of a number of transactions, the transactions are executed in the order they arrive. Each packet will be either acknowledged (ACK) or negative acknowledged (NACK). The point at which the ACK is sent in a packet is determined by the AckNow bit (Appendix B). If a transaction has its AckNow bit set, and has a valid CRC, then an ACK will be sent.

If the receiver returns an ACK then all transactions up to and *including* the transaction that generated the ACK will have been received and processed. If the sender receives a NACK, this means the receiving input was unable to send an ACK. The reasons for the input being unable to send an ACK could be one of the following.

a) an input was busy
b) a CRC error was detected
c) a network conditional failed

Where an input processing device has multiple channels, Elan 1.0 has two, then transactions after an ACK on one channel will be processed before any further transactions on the other channel. This ensures that the order in which ACKs are sent corresponds to the order in which transactions are processed. Devices outputing onto the network should not send another packet until they have received an ACK or NACK for the current one so as to ensure that packets arrive in the correct order, irrespective of the route they take. A packet must contain at least one transaction where the ackNow bit is set. NACK may be sent at any time on a multiple transaction packet unless an ACK has been sent. NACK means that at least one transaction of the multiple transactions in a packet has not occurred. When a circumstance arises where the inputter would have sent a NACK if an ACK had not already been sent, (in particular a CRC error in a transaction after ackNow), that transaction, and all the remaining transactions in the packet, have to be trapped to an error buffer.

Asserting AckNow on the first of a group of transactions allows an overlapping of acknowledges so as to preserve network bandwidth where network latency is of the order of a transaction time.

```
Input     SOP     T1     T2     T3     T4     EOP     SOP     T1
Output              ACK                                      ACK
```

3.2 *Network Conditionals*

Network conditionals provide a way of testing the value of an object across a network. If the condition is false a NACK is sent and the rest of the packet is discarded. If the condition is true the next transaction will be executed. This permits conditional writing of blocks of data, where a flag value can be tested, and a number of speculative writes in a packet will be performed only if the flag is set. Multiple conditional transactions are permitted in one packet, the effect being of a sequentially evaluated AND of the conditions. Only one ACK or NACK per packet may be sent.

Note that as NACK is also used to signal that a packet has been rejected for some reason, (bad CRC etc), receiving a NACK from a network conditional does not mean that the logical complement is true. ACK means that the conditional was performed and was true, NACK means the converse (not performed OR not true).

Network conditionals are of particular value when used with broadcast, to synchronise large numbers of processors. For example barrier synchronisation can be achieved by doing a broadcast tr_EQ, to check that all processors in a group have set a flag.

3.3 *Context Filter*

The inputter has a context filter register, which if it matches the input packet context will NACK the packet *automatically*. This is intended to be used to filter out communications to a particular hardware context. Its use is detailed in Appendix B on the Input processor.

3.4 *Broadcast*

The network is capable of broadcasting to any contiguous range of processors. Any packet may be broadcast. The ACK and NACK signals from the various processors are recombined in the network. All components must respond. The packet transmitter sees an ACK when all components have sent an ACK, or a NACK when all components have responded and at least one has responded with a NACK.

Receiving a NACK does not mean a packet has not occurred on any of the inputters. The only statement that can be made is that an ACK means that the transaction has been successfully received on all components of the broadcast. When conditional execution of transactions is employed it should be remembered that each inputter sees its own ACK or NACK value, not the value returned to the outputter.

3.5 *Exceptions*

Exceptions that can be generated by an input processor are:

```
DATA_ACCESS_EXCEPTION
EVENT_QUEUE_OVERFLOW
EVENT_INTERRUPT
QUEUE_OVERFLOW
UNIMPLEMENTED
MEM_ADDRESS_NOT_ALIGNED
```

Input specific exceptions:

```
IPROC_NACK_AFTER_ACK
```

If the inputter tries to send a NACK when it has already set an ACK was sent then an IPROC_NACK_AFTER_ACK exception is taken.

THREAD PROCESSOR

The thread processor executes virtual processes and threads, which can be generated from compiled C source code. The thread processor has two basic types of instruction: local instructions execute as normal instructions on a register based machine; network instructions are used to create network transactions that normally execute on a remote input processor.

Processes that execute within the thread processor are virtual processes, and these are described in detail in chapter 2

4.1 *Registers*

The thread processor has two types of registers associated with it; working registers (*r registers*) and control/status registers. Working registers are used for normal operations and control/status registers keep track of and control the state of the processor.

4.1.1 *r registers*

All *r* registers are 32 bits wide. They are divided into a zero register and 8 *in* registers and 8 *out* registers.

register number	Name
r[24] to r[31]	Ins
r[16] to r[23]	Unimplemented
r[8] to r[15]	Outs
r[1] to r[7]	Unimplemented
r[0]	Zero

4.2 *Network Instructions*

Network instructions are used to generate packets containing transactions.

4.2.1 OPEN, SENDTRANS, *and* CLOSE

The remote instructions OPEN and CLOSE are used to delimit a network packet; OPEN defines the virtual processor to which the packet is destined, and CLOSE sends the EndOfPacket signal after an acknowledge has been received. Between these delimiters each remote instruction translates into a single network transaction. Local instructions may execute between OPEN and CLOSE, but remote instructions executed outside of the delimiters will cause a trap.

The OPEN instruction does the translation of the destination virtual processor number into a physical processor number and context. It also fetches the route bytes for that destination processor and assembles them into a StartOfPacket. This will begin to make its way to the destination processor. The OPEN instruction may fail if the process cannot be translated, in which case it will trap.

Following an OPEN instruction, the SENDTRANS command is used to send transactions.

The CLOSE instruction ensures that an acknowledge has occurred, and *waits* if it has not. When an acknowledge has occurred an end of packet signal (EOP) is sent. The received acknowledge is examined, and a value of 0 is returned if a NACK was received, and a value of 1 if it was acknowledged by an ACK.

4.2.2 *Time Out*

The period between an OPEN and a CLOSE has to be restricted by a timeout mechanism. During the execution of an OPEN instruction a bit is set in the status register, and a TIMEOUT trap is generated if this is not cleared within the time-out period. The TIMEOUT trap is synchronous, and occurs during the execution of the CLOSE instruction.

4.3 *Local Instructions*

The local instructions are optimised for communications with additions for the local event control, locked memory operations, scheduling, and DMAs. They execute on a windowed register model.

The local instructions are executed on a windowed register model. Only registers i7–i0, o7–o0 and g0 and the icc are implemented.

Upon de-scheduling a process, registers i0–i7, o0–o5, o7, and Iptr are saved. The integer condition codes are saved in the status register until another thread process begins to execute. The context and SP are implied in a process. The stack frame being stored relative to SP.

The SP can only point to 32 byte boundaries so that the SP can only be adjusted in 8 word increments. This enables the de-scheduling, SAVE and RESTORE operations to be done with block read/writes.

4.4 *Scheduling*

The thread processor has a simple run queue scheduling model. Processes which are run are placed at the back of the run queue. Processes are stored with the context in which they are executing. Processes are taken off the run queue and run until they de-schedule themselves or are forced to by a trap. Four instructions are used to control scheduling. The run queue is held as memory based FIFO.

A process is specified by its Stack Pointer (SP or *o6*) and context. A de-scheduled process has all its state stored in its stack frame. After quoting a processes wptr to another process, the other process may try and run it while it is still executing. The SP should therefore *never* be changed and the process should SUSPEND as soon as possible to avoid confusion.

DMA PROCESSOR

The DMA processor can be used for local (store-to-store) transfers, or for network (store-to-remote-store) transfers. For network transfers, the DMA processor constructs packets with arbitrary numbers of read/write block transactions. Unlike conventional DMAs, the amount of data transferred by a transaction, and the number of transactions within the packet, are variable and dependent on network loading. At times of high load the packet size may be reduced (between 4 to 16 transactions), and the transfer may even be suspended. The maximum amount of data transferred within a single transaction is 32 bytes.

5.1 *DMA Descriptor*

To request a DMA the main processor constructs a DMA descriptor and writes the address of this descriptor to the ELAN command port. The ELAN's command processor extracts these descriptors from the port and adds them onto the DMA queue.

The DMA processor extracts descriptors from the queue when it is ready to process them. The DMA processor constructs the necessary packets and encapsulates the data within the transactions.

5.2 *Typing of DMAs*

The DMA engine can handle typed data. The default DMA type is byte. If the store ordering of both sender and receiver is the same then typing does not matter. If the store ordering is different then typed transfers ensure that the correct movement of data occurs. Data must be aligned to the required type size. Local byte block moves can be used to move data to any alignment.

5.3 *DMA Failures*

Within a large network of ELAN processors the majority of traffic being transferred will be in the form of DMAs. Errors will occur on the links connecting these processors, and are detected by the network. Bit errors on data bytes will cause a CRC failure and the packet will be NACKed.

To provide some insulation from these errors the DMA descriptor contains a fail count. This is the number of non-corrupting failures that can be tolerated in a particular DMA. If a failure occurs and the fail count is non zero the DMA will be re-queued such that the failing part of the DMA is re-attempted, and the fail count is decremented.

If the DMA failure count is zero when a failure occurs then the `DMA_FAILED_COUNT_ERROR` exception occurs.

5.4 *Exceptions*

Exceptions that can be generated by the DMA processor are:

```
DATA_ACCESS_EXCEPTION
UNIMPLEMENTED
OUTPUT_INVALID_PROCESS
OUTPUT_INVALID_ROUTE
OUTPUT_ALREADY_OPEN
EVENT_INTERRUPT
QUEUE_OVERFLOW
```

DMA specific exceptions:

```
DMA_FAILED_COUNT_ERROR
```

5.5 *Normal and Secure Transfers*

DMAs are defined with two modes of operation: normal and secure. In normal mode the objective is to move data at as high a bandwidth as possible. In secure mode the correct writing of data is assured.

normal and secure transfers are really only different for remote DMAs.

Normal DMA opens a packet and outputs a DMA transaction with the SENDACK bit set. It continues outputing DMA transactions without the SENDACK bit set until either:

a) the ACK or NACK is received.

b) 16 transactions have been sent.

c) the DMA time slice period is exceeded.

When any of the above are satisfied, the DMA processor prepares the END_OF_PACKET to be sent. If the DMA time slice period has been exceeded or a NACK has been received the DMA descriptor is updated and placed on the back of the queue. Otherwise the process is repeated.

Remote DMA opens a packet and outputs 16 transactions provided that DMA time slice period isn't exceeded. These are sent without SENDACK set, and an additional NULL transaction is sent at the end with SENDACK set. This ensures that errors occurring at the input are observed by the DMA processor.

5.6 *DMA Timeslice Period*

The DMA timeslice period is 50 micro seconds. This value was decided upon such that if only the DMA process is running and the network isn't congested then the overhead of time slicing would be less than 5%. For Elan 1.0 this is approximately 50 micro seconds or the time to send 2K bytes.

REPLY PROCESSOR

It is a desirable requirement that input transactions are dealt with as quickly possible. To help achieve this some decoupling is required between the inputter and outputer for transactions that require a reply, such as `tr_readword`. This decoupling is achieved by using the reply processor.

The reply processor, like the DMA processor, removes work from a queue of tasks, each task being defined by a descriptor. Because replies are queued until the outputter is free to send, a reply generating input need only arbitrate for the memory system, and not the outputter as well.

Replies are formulated at the input and put on a reply queue. The reply processor is responsible for taking replies off the queue and turning them into packets and inserting them onto the network. A reply takes the form of a number of read results, up to three, and a `tr_remotereply`. Each read specifies a return write address, and the `tr_remotereply` specifies the processor and context in which those writes will occur and an event to be set.

6.1 *Reply Descriptor*

The reply queue has a fixed entry size of 8 words, or 32 bytes, and takes the form:

word no	Contents	Description
0	Process, Context	Destination process
1	Address	Event to set after writes
2	Data	Data write for address 1
3	Address	address 1 Word
4	Data	Data write for address 2
5	Address	address 2 Word
6	Data	Data write for address 3
7	Address	address 3 Word

The reply processor will produce a packet with three `tr_writewords` and a `tr_setevent`. If any of the write addresses are NULL then the reply processor ends the packet and sends the `tr_setevent`. The address of the event may also be NULL, and this will be interpreted by the inputter as a NULL operation.

The descriptor gives the context on which the reply is to be formulated, and the process number to which it is to be output. The context and process number need to be translated by the reply processor; note that this is done at the same time as the reply is output. Placing the translated values of hardware context and processor number in the queues would not allow a context to be disabled rapidly at the translation tables. The context is held in the low half of the word, the destination processor number is held in the upper half.

6.2 *Exceptions*

The reply processor may generate any of the following exceptions:

```
OUTPUT_INVALID_ROUTE
OUTPUT_INVALID_PROCESS
OUTPUT_TIMEOUT
```

Reply processor specific exceptions:

```
RPROC_NACKED
```

A `RPROC_NACKED` trap is generated if a reply packet generates a Nack.

COMMAND PROCESSOR

7.1 Command Processor Specification

The command processors purpose is to provide a checked calling interface between a UNIX system and the ELAN processor. This allows ELAN processes to be initiated using an area of paged memory as the protection mechanism. In this area different pages initiate different commands, access to these pages can be controlled by the kernal executing on the UNIX system.

The command processor serves an additional purpose in the initial implementation, turning physical interrupts, into EVENT signals.

The command processor is activated by either a physical interrupt becoming ready or a command port slot being set full. The first thing the command processor does is determine which type of event caused it to wake up.

7.2 Command Source

Do the command in the command port and when done place the parameters in the data register and clear the full bit.

Commands available on the comms processor are detailed in appendix E "Command Processor Instructions".

7.3 Registers

The following internal and external registers are used to control the command processor:

Command Context Table
For commands requiring a context, a field of 2 bits in the command value are used to determine which context from a vector of 16 bit contexts is to be used. The value is looked up by the command processor. The encoding of the context selection in the command value allows contexts to be guarded using page table entries.

The context translation vector consists of a single double length register, which is sufficient to store 4 contexts. (More can be supported)

Interrupt Base Register
This points to the vector of event locations that are to be used for internal interrupts. The events are double word locations so for example interrupt 5 has a vector entry of this base register plus ten.

CommsProcIntMaskReg
This is an interrupt mask register, and is ANDed with the Interrupt register to determine whether an interrupt is to be taken by the ELAN. This register is used in conjunction with the main processor interrupt mask register in the slave memory map to control the taking of interrupts.

7.4 *Command Port*

The command port is a memory mapped object in the Elan processors slave address space through which the main processor can queue up commands with a single read/write word swap operation.

A basic command register consists of a 32 bit data word, and a 32 bit command word. The command register has the following assignments:

```
Bit 31 FullnotEmpty
Bit 30 Finished
Bit 29 Error
Bit 28 to 15 Read as Zero
Bit 14 to 9 Command type
Bit 8 to 0 Immediate
```

A command port may have multiple command-data register pairs.

Commands may be issued by writing to the register pair and ensuring that the full bit is set. The command processor signals completion by clearing the full bit. Commands may return values in the 32 bit data register. Scheduling a command using this mechanism would require polling the command register to check that it is empty, then writing to the command data pair, requiring multiple read and write operations.

In order to minimise the overhead of issuing a command, a second image of the command register is provided. This memory image is guarded so that a single word swap operation is sufficient to issue a command.

Within the command issue image, reading and writing have the following effect. Any read will return the number of the next command slot that is free. If all the command slots are full a negative number is returned. If a command slot is free then a word write causes data to be written to the next free data slot, and the command to be set to full with the lower 15 bits being taken from bits 17 to 3 of the address. A write when there are no free slots still acknowledges to the memory interface, however no state is modified.

Note that although this is a special use of read and write, data within the command processor is still only changed on write memory cycles.

From the user's perspective to issue a command C with immediate address I and data D, he performs a RmW operation to a word vector indexed by C,I. This returns a small positive integer (the data register which any read data will be returned in) if successful, or a negative value if not. Note that for correct operation when using RmWs the value of Full must be sampled at the start of the transaction so that the same value is used for both the read and the write.

Using the command issue image it is possible for the user image to be consistent over different implementations with different depths of command queue. So that user code is truly command queue length independent, only one outstanding read operation should ever be permitted.

Commands are executed in a round robin order of command registers, ensuring that commands are executed in the order they are submitted.

7.4.1 *Command Port Address Map*

The address map of a command port is chosen to map onto a 4k byte paged MMU. A command port physical address is aligned to a 256KByte boundary. The 4KByte page of this is used for direct access to the command port registers, which are allocated in pairs, starting from the first port address. Each successive physical page after this maps a different command. (Note that this means that command 0 is invalid as it cannot be issued). In particular a command port that uses only one command can be mapped in two pages, or the entire command set can be mapped in a single 256KByte transaction. Note that mapping of individual pages permits individual commands to be mapped in and out of user space. More detail on the ELAN memory map is provided in Appendix J "Communications Processor Memory Map".

```
FFiF80000      data reg 0
               command  0
               data reg 1
               command  1
```

Multiple images of command slots to page boundaries.

```
FFiF81000        command 1 zero immediate
FFiF8100C        command 1 immediate 3


FFiF83C00        command 3 immediate  3x300

i defined by MBus ID pins
```

Commands are defined in Appendix E, "Command Processor Instructions".

7.5 *Interrupt Source*

Upon an internal interrupt becoming ready, ie the Interrupt Register ANDed with the CommsProcIntMaskReg being The action on an interrupt is to indirect into the interrupt event vector and cause a set event on that location (These are always done in context0). The communications processor interrupt mask is then updated to mask out the interrupt source and then the command processor deactivates.

7.6 *Exceptions*

Exceptions that can be generated by an command processor are:

```
DATA_ACCESS_EXCEPTION
EVENT_INTERRUPT
QUEUE_OVERFLOW
MEM_ADDRESS_NOT_ALIGNED
```

Unimplemented commands are not trapped since they can only be caused by wrongly mapped MMU pages, and as such are the responsibility of the system code to prevent them occurring.

EVENTS, EXCEPTIONS, AND INTERRUPTS

8.1 *Virtual Events*

The virtual event mechanism provides a generalised implementation of interrupt style signalling across a distributed memory machine. The mechanism is asymmetric, unlike the OCCAM channel, in that only one side commits. (The OCCAM channel style synchronisation primitive can however be built from a pair of events.) An event location is set by a thread that wishes to signal, and waited on either by suspending or polling by the thread waiting for that signal. The event mechanism may be protected so that it acts between processes as a distributed implementation of a trap mechanism, (with the important difference that the process signalling the trap does not automatically suspend itself).

The operation of remote events depends on which thread has the event in its address space. Generally speaking the most efficient implementation will be one in which the thread that is to wait has the event location in its memory space. In this case the thread suspends itself locally and is woken up by a setevent network transaction.

It is also desirable to be able to wait on an event location in a remote process's memory space. For reasons of security the actual waiting thread address cannot be exported to another process. Instead the thread exports the address of an event location local to its own memory, and then suspends itself on that local event.

A process from the main processor may also wait on an event location. In this case the value written to the wait location is distinguished by the number in word0. The interrupt is not signaled immediately, but is put on the run queue and flagged when it reaches the top of the queue.

8.1.1 *Event Locations*

The event location consists of two 32 bit words. This pair of words must be double word aligned. The two words can have the following values:

Word0	Word1	LSBS (word1)	meaning
X	0	00	clear event location
X	1	01	ready event location
proc	DWaddr	00	remote thread waiting (DWaddr = remote event address)
count	queue	10	clear queued event location (queue may be full)
0	queue	11	ready queued event (queue is empty)
-1	Iaddr	00	main processor waiting for interrupt
-2	Waddr	00	local thread waiting
-3	Daddr	00	DMA desc waiting
-4	Dummy	00	Null event waiting
<-4	Eaddr	00	Local event waiting

Note that the two LSBs of the waiting address always have the following meaning:

Bit 0 The event is ready (1) or not (0).

Bit 1 A queue of waiting processes begins at the location pointed to by waiting address. The number of entries on this queue is held in the proc number location. If this is zero the fptr must be zeroed by the next process to wait on the queue.

A waiting object is now indicated by bits [31:2] of the second event word being non-zero. The first word is not effected by set event, unless the event was a queue and contained a waiting item.

```
typedef struct EVENT
{
   union {
      int32 count;
      proc_t procid;
   } word0;
   union {
      thread_t *thread;
      queue_t *queue;
      int32 value;
   } word1;
} event_t;
```

```
#define event_count word0.count
#define event_procid word0.procid
#define event_thread word1.thread
#define event_queue word1.queue
#define event_value word1.value

#define READY (ev) (ev-> event_value & 1)
#define HASQ (ev) (ev-> event_value & 2)

typedef struct eventQ
{
   int32 eq_size;
   int32 eq_front;
   EVENT eq_Q[];
}
```

8.1.2 *Queued Event Locations*

So that multiple threads can wait for an event, event queues are provided. An event queue address must be double word aligned. The structure of an event queue is,

```
int size;                    // number of queue slots
int fptr;                    // index into queue pointing
to front.
struct qentry queue [size]; // an array of queue
entries.
```

The number of items in the queue is the value in w0 of the event location. The value of index is always reset to zero when the queue is empty.

8.1.2.1 Queued Event Location Entry

A queue entry consists of a double word. The double word can contain either a local thread address, a local interrupt address, or a remote set address. A null entry on the queue, of zero, zero, is also allowed. This allows items to be simply deleted from queues, but note that it does not immediately free up queue space, also that it will cause the queue to appear to have threads waiting on it, and a set event ignores a null entry, ie it will try to move onto the next entry in the queue if one exists. In this case if no entry exists the event location will be left set.

A queue entry consists of a double word and is similar to an event location in structure. The two words can have the following values:

Word0 Word1 meaning

proc	DWaddr	remote thread waiting (DWaddr = remote event address)
-1	Iaddr	main processor waiting for interrupt
-2	Waddr	local thread waiting
-3	Daddr	DMA desc waiting
-4	X(<>0)	Null event waiting
<-4	Eaddr	Local event waiting

8.1.3 *Operations on an Event Location*

The following operations can occur on an event location.

- Local wait.
- Local set
- Remote wait.
- Local clear
- Local/Remote Test.

8.1.3.1 Local Wait

A local wait is executed by the thread processor. This tests the value of the event location for readiness. If ready the event is cleared and execution continues. If not ready the thread address is either stored in the event, or if queued in the attached queue location. If either a single event already has something waiting, or a queue exists but is full, a queue overflow exception is generated.

8.1.3.2 Set Operations

Local set is executed by the thread processor. This tests to see whether the event has anything waiting on it. If not the the ready bit is set. If something is waiting in the event, then the event location is reset to zero and the event done.

The event may either be,

remote thread Remote set event is put on the reply queue.

local thread Thread is scheduled.

event interrupt An interrupt is generated.

Sets on a queued event location proceed in a similar manner except that the data is found in the queue. Remote sets operate identically to local sets, except that access permissions are different ie RemoteEvent instead of LocalEvent.

8.1.3.3 Remote Wait

Remote waits pass a double word value consisting of a process and an address in that process that fully describes another event. The process value must be non NULL. For a valid process value if the event is ready, it is cleared, and a remote set placed on the reply queue. If the event is not ready the thread is put in the event location.

8.1.3.4 Local Clear

Clear event clears the ready flag. Any queue is left unaltered.

8.1.3.5 Local and Remote Test

The status of an event can be tested either locally or over the network. The status of any event can be checked simply from the event double-word, as a non zero value in word0, means something is waiting, and a set ready bit means the queue is ready.

8.1.4 *Atomic Access Considerations*

Queue handling must be atomic. Communication processors ensure atomic access by locking the memory bus for the whole of the access sequence. Programs running on the main processor should lock out the communications processor from memory access by using the memory access inhibit mechanism in the communications processor.

8.1.5 *Protection*

Event locations have a special level of memory protection. Local threads have complete read write access to events mapped into their memory space. Remote threads however are only allowed to do set operations, and remote waits, and tests in store areas with remote event only permission. This ensures that thread creation is not possible by outside threads, and that the only threads running in a context will be those created locally. (Remote running of threads is of course possible in an area of store with remote write and event permission, by remote writing the thread address to some location and immediately setting it). The queue areas should be in areas of store with local R/W permission only. The inputter needs to access these with the required access types for queue handling operations.

8.2 *Exceptions*

Exceptions are indicated by a non zero value in the trap type field of the status register of the processor with the exception. This generates a maskable interrupt. The exception is not set until the processor has put itself into a known state in the exception area, the processor then stops executing. The wakeup function in the status register is set to WakeupNever.

The main processor handles exceptions through the command port. The exception handler extracts the queue item which caused the exception from the internal registers of the ELAN processor. The main processor may then restart the unit, by updating the wakeup address then the wakeup function in the status register.

The process exception area the following layout for each processor.

STATUS Status register at point where exception was noted. The Memory error bit may be set, in which case the MMU error locations are valid. These are in the next two locations.

FSR MMU fault status register.

FADDR Fault address. If the exception was the result of an EVENT interrupt to the main processor this is the address value from the EVENT.

CONTEXT Value of CurrContext at time the exception was detected This may not be the value of the context for the process for which the error occurred in the case of MBus errors. ie BUS ERROR, TIME OUT ERROR and UNCORRECTABLE ERROR.

8.2.1 *Exception Sources*

`NO_TRAP`

Reset state of trap type. No trap has occurred

`DATA_ACCESS_EXCEPTION`

Error detected by MMU. In this case FSR and FADDR are valid. Meanings of FSR and FADDR are defined in appendix H "MMU User Guide".

`MEM_ADDRESS_NOT_ALIGNED`

Virtual address was mis-aligned, (eg non zero bottom two bits for word access)

OUTPUT_INVALID_PROCESS

(outputting processes only)
Attempt to output to a virtual process number not currently mapped in the virtual process table for this context.

OUTPUT_TIMEOUT

(outputting processes only)
Output device timed out. The outputing process had opened the packet for greater than 4 milli-seconds before checking the packet acknowledge.

EVENT_QUEUE_OVERFLOW

(only on threads processor, or input processor)
Attempted to wait on a queued event with insufficient space left in the queue structure.

QUEUE_OVERFLOW

Processor attempted to queue a descriptor on reply, thread or DMA queue where there is insufficient space left in the queue structure. The descriptor queue structures for these processors have 16 locations dedicated to context0 processes.

Any unit which performs set events can cause overflows on either the thread queue or the reply queue; whether it is the thread queue or the reply queue depends on whether the object in the event is local, or remote. All units apart from the reply processor perform set events.

Various explicit thread operations can cause overflows. A DMA instruction can cause a DMA queue overflow, and a RUN instruction can cause a thread queue overflow.

The command processor has a command to enqueue an item onto any of the three queues, each of which may overflow.

The input processor can cause DMA queue overflows through remote DMA operations, and reply queue overflows by remote read operations.

`UNIMPLEMENTED`

Unimplemented is a valid trap for all the processors except the reply processor. The command processor unimplemented is not currently checked, as this can be guarded by the main processors MMU.

`PACKET_SEQUENCE_ERROR`

This is generated by any thread instruction on the output device which occurs in the wrong order, (ie, SEND_TRANS, without OPEN, or multiple OPENs).

`DPROC_FAIL_COUNT_ERROR`

DMA processor decrements its fail count every time it receives a Nack. When this value is zero an exception is generated.

`RPROC_NACKED`

The reply processor has no fail count so generates an exception immediately if it is Nacked.

`IPROC_ERROR`

Caused by CRCerror, BadEOP, BadLength. Cause of error detectable from status register.

8.2.2 *Exceptions During Output*

An exception can occur on either the thread, reply, or DMA processor while it has the output device open. The microcode which generates an exception will automatically close down the devices and sends an EOP error down the line. The status register indicates whether or not a processor had the output open at the time of an exception.

8.2.3 *Exceptions During Input*

If an input exception causes a trap on one of the two inputters it may be necessary for the trap code to read the received transaction from the comms processor's internal memory. Each input process has a cyclic input buffer of 16 words. The internal ram locations 'TEMP_INPUT0' and 'TEMP_INPUT1' contain the internal ram pointers to the beginning of the transaction in this buffer dependent on the inputter number. They are held in an internal format that is decoded by ANDing the read value with 0xf and using this as an index into the respective input buffer.

ie If TEMP_INPUT1 = 0xce, then the index is 0xe. The transaction code will be found at address 0xde and the address at 0xdf. Data words for the transaction would be found at 0xd0, dependent on the size bits set in the transaction word.

The external location 'NextTransFront' contains the pointer to the place where the next transaction will be placed in the buffer. This may be used to delimit the transaction but using the size from the transaction itself will probably be quickest since this will require less slave reads.

More complex exceptions should be dealt with by turning on the context filter for the exceptional context. The remaining transactions of the current packet should be extracted in turn until the EndOfPacket. If the packet isn't going to be dealt with correctly it should additionally be NACKed if possible. Subsequent transactions can be extracted in a timely fashion by asserting the trap on transaction function.

The following microword restart addresses can be used:

`NAckAndMoveToNextTransaction`

Will NAck the packet and move to the next transaction. It increments the buffer pointers and tells the input queue logic that the transaction has been consumed. If the Ack has already been sent then a IPROC_NACK_AFTER_ACK exception will occur.

`MoveToNextTransaction`

Will move to the next transaction in the buffer or wait for one to arrive. It increments the buffer pointers and tells the input queue logic that the transaction has been consumed.

`HandleTransaction`

Restarts the inputter at the current transaction. If the transaction hasn't been ACKed or NACKed the input can be restarted at the 'HandleTransaction' microword. The reason for the exception will need to have been cleared.

8.2.4 *Exceptions on Command Processor*

Exceptions on the command processor are implemented in a slightly different way to the other processors because the command processor is used to access the internal state of the ELAN processors. The state of the command processor is held within the command ports. These are first extracted (and cleared) and exceptions are disabled from the command processor using the *CProcReset* bit in the ELAN command register. Normal command port read and write access to the internal registers can be used to clear the command processes status register. The *CProcReset* bit is then disabled. After the reason for the exception has been cleared (i.e DATA_ACCESS_EXCEPTION on unmapped descriptor for command port "thread processor place on queue" command) the commands can be placed back onto the command port, and execution recommence. Note that an internal interrupt from the command port can not be handled by the command processor.

8.2.5 *Exception on Reply Processor*

If an exception occurs on the reply processor the trap-handler may need to get the descriptor to help diagnosis, or to restart the reply. The descriptor is held in the internal registers REPLY_BUFFER_0 to REPLY_BUFFER_7.

This buffer is always valid during execution of the reply processor. The state indicating the buffer's validity is held in the external register ReplyBufferValid. This needs to be cleared by the exception handler if the descriptor is to be removed.

The reply processors status register SuspendAddr has to be reset to the default reset value, see Appendix L.

8.3 *Interrupts*

Source of interrupts are:

Exceptions Exceptions from each of the ELAN processors.

External External active low interrupt lines.

Alarm Wake up for alarm.

Hush Hush Error.

Halted Procs halted status.

The implementation of the device is an interrupt master or source. When used as an Mbus slave device a single interrupt out signal is generated, and the internal interrupts register can be read to determine the source of the interrupt. There is an 18 bit processor interrupt source register. Also an 18 bit interrupts mask value. The single interrupt out line is the OR of the 18 processor masked interrupts.

8.3.1 *Communications Processor Handled Interrupts*

The comms processor uses the same interrupts but has its own separate mask register. The comms processor handles interrupts by turning them into context 0 set events. These are located as a vector in store starting at the Internal_Interrupt_Base location, an internal register which must be set up before enabling any comms processor interrupts. After the set event has been executed, the comms interrupt mask bit for that interrupt is automatically cleared. The actual interrupt handling code, (which would be waiting on the event location) is responsible for re-enabling the interrupt when appropriate.

MMU AND VIRTUAL PROCESS TABLE STRUCTURES

A.1 *Root Context Table*

The root context table points to the MMU entries for each context. The internal register CONTEXT_PTR defines the base of the context control block tables (CCB). Each of these four word CCBs defines the level 1 page table entry (PTE) for the MMU and the virtual process table. The CCB entry is defined as,

```
struct
{
   unsigned int context_page_table_entry;
   unsigned int virtual_process_table_base;
   unsigned int virtual_process_table_size;
   unsigned int reserved_space;
} context_control_block;
```

The root context table has 65536 entries. Each entry need not be filled however since the system code will only be able to generate and allow references to contexts. This restriction of reference to contexts is achieved by translating user quoted process numbers through a virtual process table (VPT).

A.2 *Virtual Process Table*

The virtual process table is a list of the processes with which a particular process can communicate. This list is indexed by the virtual process number.

```
struct virtual_process_table_entry virtual_process_table[];
```

A list entry gives the physical processor number and context number of the destination process, as a single word entry,

```
struct
{
   unsigned context_number : 16;
   unsigned processor_number : 16;
} virtual_process_table_entry ;
```

The upper half of the word is the physical processor number and the lower half the hardware context of the process on that physical process.

-1 is an invalid entry in the table. To reduce the store requirements of the VPTs, the context control block has a maximum virtual process number (virtual_process_table_size). Virtual process numbers are checked against this before indexing in the table, processes of greater or equal number are treated as invalid entries. Invalid entries will cause an exception to be generated.

The virtual process table begins at the physical address given by shifting left virtual_process_table_base 4 bits, and zeroing the LSBs. The VPTs are therefore 32 byte aligned.

A.3 *Route Table*

The route for a physical process is found from the route table. The comms processor internal register ROUTE_TABLE_PTR defines the top 32 bits of the physical address at which the route table begins. Each entry in the table contains four routes for each physical process one of which is randomly selected. The four routes can be set to be the same to give a single route. A route is represented by 16 bytes. The route table can be placed at the same location in each comms processors store to enable the easy removing of a particular physical processor by removing its route. Routes to a particular processor can then invalidated by broadcasting a write of the NULL entry to each of the four routes.

```
struct route_table_entry *route_table_base_register[][4];

struct
{
  char route[16];
} route_table_entry;
```

A.4 *MMU Page Tables*

The MMU page tables are referenced by context through the CCB, to the context page table entry or level 1 entry.

A.5 *Context0 Processes*

Context0 is a name that is given to high level processes that are used for booting, and error processing; these processes are analogous to the supervisor processes used in UNIX. Context0 processes are distinct hardware contexts that have unique privileges by virtue of their context number.

Context 0 processes run at priority over other processes on all queues. However lower priority queue items are not interrupted. Priority is established by front of queue scheduling. This means that queue overflow checking always leaves room (16 queue entries) for some context0 processes.

Hardware contexts 0-31 are the context0 process context values. Context0, hardware context 0x10, processes only can wait on the internal event vector.

A.6 *Translations without MMU*

Prior to the MMU being enabled on the ELAN, virtual addresses map onto physical addresses without translation, the extra bits PA[35:32] coming from the lower four bits of the context register.

When the MMU is enabled virtual addresses are translated through the page tables.

On reset the MMU is disabled.

INPUT PROCESSOR

B.1 *Format of Input Transaction*

Transactiontype	16 bits
Context	16 bits
Address	32 bits
<Data[]>	32 bytes of data for tr_blockwrite or some even number of words of parameters for all other transactions.
CRC	16 bit CCITT standard CRC

The bit fields of the transaction type are define as follows:

<15> ackNow

<14:12> Size 0, 1, 2, 4, 8, 16, 32, or 64 double words. Bit 14 is forced to zero currently so that only size 0, 1, 2, 4 are allowed.

<11> WriteBlock

<10:0>

~WriteBlock	op code
WriteBlock	<10:9> Data type
	<8:0> Part write size

B.2 *Network Transactions*

tr_writeblock

```
p0-p7 32 bytes of data
```

Write a 32 byte block of data to the address specified in the transaction header. If less than a block is to be written the part write size field of the transaction is set to the number of bytes to write, and the start address comes from the lower five bits of address. Data type is sent with the block so that if the byte order of the receivers memory system differs from the network byte order (big endian) byte swapping can be performed.

tr_DMArequest
`p0-p7 DMA descriptor`

Put the DMA descriptor described in parameters p0-p7 onto the DMA queue. Bit 15 of of p0 is set to 1 to indicate that the DMA was created remotely. The context from the transaction is inserted into the context field of the DMA descriptor.

tr_writeword
`p0 new value`

Write one word into the address specified in the transaction.

tr_readword
`p1 reply address`

Read one word from the address specified in the transaction. Data value read is put on the reply queue along with the reply address. A tr_remotereply is then required in the same packet to specify the return process.

tr_remotereply
`p0 reply process`

Specifies the reply process `p0`, for reply data form remote word reads. The transaction address specifies an event address.

Example:

Received packet

```
tr_readword (add1, , destadd1)
tr_readword (add2, , destadd2)
tr_remotereply(event, procB)
```

Reply packet sent to "procB"

```
tr_writeword (destadd1, mem[add1])
tr_writeword (destadd2, mem[add2])
tr_setevent  (event )
```

Note that the number of reads before a reply will be limited to the capacity of the reply processor (to 3 in the current implementation).

tr_readwriteword
```
p0 new value
p1 reply address
```

Perform a locked read then write operation and return the read data to the reply address. Destination processor must be specified by a tr_remotereply as for tr_readword.

tr_atomicaddword
```
p0 add value
p1 reply address
```

Perform an atomic add of the add value to the specified address. If the reply address is other than NULL, then the original value is returned as for tr_readword or tr_readwriteword. If there is no need to return the value, the reply address is set to NULL, and no tr_remotereply is necessary in the packet.

tr_testandwriteword
```
p0 write data
p1 reply address
p2 test value
```

Perform an atomic compare of the test value with the data at the specified address. If the two are equal, then the write data is written, if not equal then nothing is written. The read value is always returned as for tr_readword or tr_readwriteword. If there is no need to return the value, the reply address is set to NULL, and no tr_remotereply is necessary in the packet.

tr_setevent
```
no parameters other than address
```

Cause an event on the virtual event location specified by the address. If the event is already set no action occurs. If there is a thread waiting this is run.

tr_clearevent
```
no parameters other than address
```

Clear event specified by the transaction address. There is no handshaking on this operation. If the event is not ready, it has no effect.

tr_waitevent
```
p0 suspending process
p1 suspending event
```

If the event is not ready, it writes the address of a remote set location into the suspend location of the event specified by the transaction address. This may be either a queue or a single location. If the event is ready, the reply processor performs a set to the suspending event, and the event is cleared.

tr_eventready
tr_noteventready
```
no parameters other than address
```

Poll an event location to see whether it is ready. The value of the condition is returned via the ACK/NACK mechanism described previously. Both senses are provided since NACK may mean that the packet has been rejected rather than the condition is false.

tr_EQ
tr_NEQ
tr_GTE
tr_LT
```
p0 testvalue.
```

Perform specified comparison between the test value and the word location specified by the address field of the transaction. Return value via the ACK/NACK mechanism.

B.3 *Transaction Type Code Summary*

Basic memory transactions	size	opcode	conditional
tr_writeblock	4		no
tr_writeword	1	01000	no
tr_readword	1	01001	no
tr_remotereply	1	01010	no
tr_DMArequest	4	01011	no
Locked memory transactions			
tr_readwriteword	1	01100	no
tr_atomicaddword	1	01101	no
tr_testandwriteword	2	01110	no

Event transactions

tr_setevent	0	00000	no
tr_clearevent	0	00001	no
tr_waitevent	1	00010	no
tr_eventready	0	10000	yes
tr_noteventready	0	10001	yes

Conditional transactions

tr_EQ	1	11000	yes
tr_NEQ	1	11001	yes
tr_GTE	1	11010	yes
tr_LT	1	11011	yes

Unimplemented transactions

01111
00011
001xx
1001x
1x1xx

Note: Set Event NULL is NULL transaction; the value of the transaction is all zeros.

B.4 *Input Reply Buffer*

On receiving a network read command the input checks that it has access to a reply buffer. This is the place where it will store the reply data until it inputs the reply command. If one is not available the input will attempt to send a NACK, (and trap if an ACK has been sent). The input will read the READ data and place it in the buffer. Also in the buffer it places the address, from the network instruction, where to store the data. It then moves onto the next transaction.

On receipt of the first READ in a group the input will take ownership of a buffer (On the initial ELAN there is one buffer shared by two inputters). It will retain this ownership until the EOP is received or a NACK is forced.

B.5 *Input Context Filtering*

The input context filter is an architectural feature which enables a number of contexts to be ignored by the inputer. Any such match on the input causes a NAck to be sent immediately without touching the store system. Elan 1.0 implementation has only one such context filter available as an external memory location. The filter is enabled with the MMU and so should be set to a non-existent context when not required, for example 0xffff.

The context filtering mechanism might be used by an operating system to reduce the activity of a context on a node during resolution of an exception.

The context filter can be read and written as an external register, InputContextFilter.

THREAD PROCESSOR STRUCTURES AND INSTRUCTIONS

C.1 *Run Queue Entry*

The processors run queue is a front and back pointer queue. Each entry is 32 bytes. The entry together with the de-scheduled register window stored in the stack frame enables any of the processor state to be reloaded.

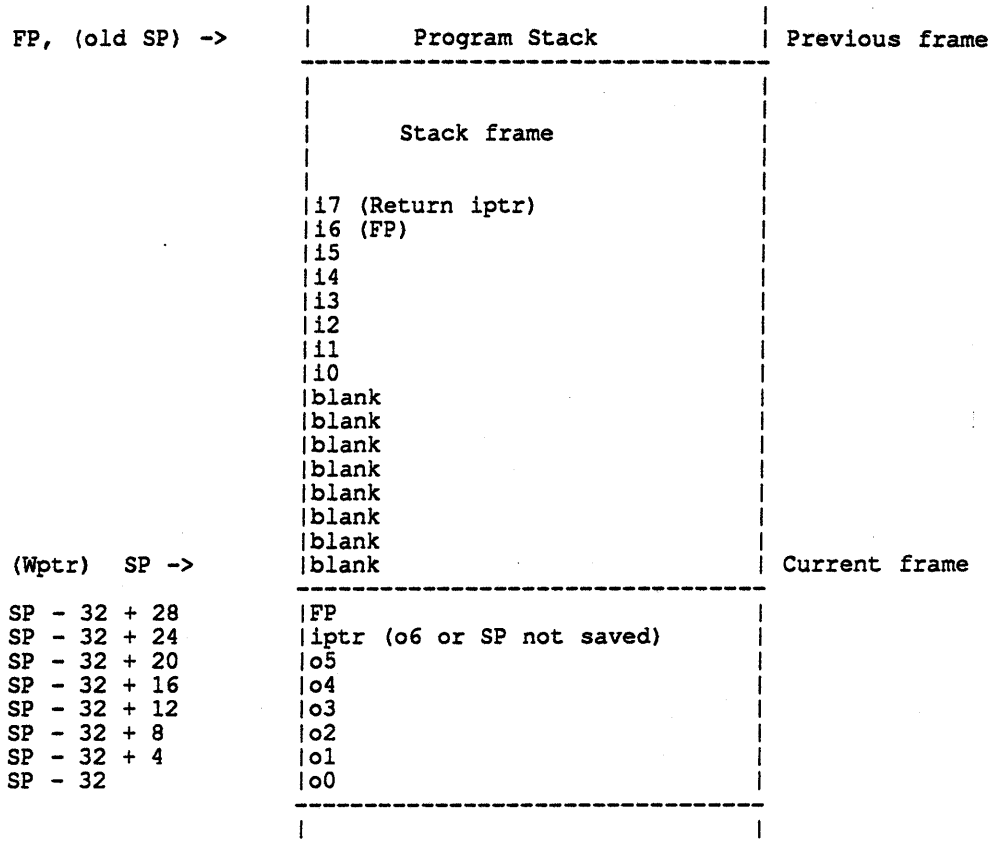
```
STRUCT run_queue_entry
{
   unsigned int context, address;
   unsigned int nPC_inc, cc;
   unsigned int reserved0, reserved1;
   unsigned int reserved2, reserved3;
}
```

On running a process off the queue the nPC_inc is added to the PC + 4 gained from the memory image of the suspended process. The nPC_inc and cc are set to zero if the process had suspended by waiting, or the suspend instruction. However on an exception restart the queue entry can be altered to reflect any instruction having been executed or about to be executed dependent on the way an exception was handled.

C.2 *Software*

C.2.1 *Thread Processor User Stack Frame*

Upon de-scheduling the processor state must be saved. This is stored relative to the SP. The SP is quoted as being the Wptr.



C.3 *Instruction Set*

C.3.1 *Local Instructions*

The local instructions are optimised for communications with additions for the local event control, locked memory operations, scheduling and DMAs.

General Instructions

LD	ST	
ADD (ADDcc)	SUB (SUBcc)	
AND (ANDcc)	ANDN (ANDNcc)	
OR (ORcc)	ORN (ORNcc)	
XOR (XORcc)	XNOR (XNORcc)	
SL	SR	SRA
SETHI		
SAVE	RESTORE	
Bicc		

```
CALL          JMPL
```

Scheduling Instructions

```
SUSPEND   BREAK   RUN
```

Local Event Control

```
WAITEVENT   SETEVENT
```

Locked Memory Instructions

```
SWAP   ATOMICADD   TESTSTORE
```

On the Elan revisions A, B and C only the SWAP instruction is implemented.

DMA Control

```
DMA
```

C.3.2 *Network Instructions*

Packet generation instructions.

```
OPEN   CLOSE   SENDTRANS
```

C.3.3 *Instruction Definitions*

General Instructions

```
ADD
ADDcc
SUB
SUBcc
```

These instructions implement arithmetic operations. `ADDcc` and `SUBcc` modify all the condition codes.

Traps :

(None)

```
AND
ANDcc
ANDN
ANDNcc
OR
ORcc
```

ORN
ORNcc
XOR
XORcc
XNOR
XNORcc

These instructions implement bitwise logical operations. ANDcc, ANDNcc, ORcc, ORNcc, XORcc and XNORcc modify all the condition codes.

Traps :

(None)

SLL
SRL
SRA

These instructions implement logical shift left SLL, logical shift right SRL and arithmetic shift right SRA. The shift count is the five least significant bits of r[rs2] if the *i* field is zero or simm13 if the *i* field is one.

These instructions do not modify the condition codes.

Traps :

(None)

SAVE

The SAVE instruction saves the current window frame into the stack. Otherwise the SAVE instruction acts like an ADD that always writes its result to SP. The amount by which the SP is adjusted by should always be a multiple of 32, since the window frames are stored on 32 byte boundaries.

The out registers of the current frame become the in registers of the new frame.

Traps :

DATA_ACCESS_ERROR, MEM_ADDRESS_NOT_ALIGNED

RESTORE

The RESTORE instruction resorts to the previous window frame on the stack.

Otherwise the RESTORE instruction acts like an ADD that always writes its result to SP. The amount by which the SP is adjusted by should always be a multiple of 32, since the window frames are stored on 32 byte boundaries.

The in registers of the current frame become the out registers of the new frame.

Traps :

DATA_ACCESS_ERROR, MEM_ADDRESS_NOT_ALIGNED

LD

This instructions loads a word from memory into the r register defined by the rd field. If the load traps the register is unchanged. The effective address for the load is either "r[rs1] + r[rs2]" if the i field is zero or "r[rs1] + simm13" if the i is one.

Traps :

DATA_ACCESS_ERROR, MEM_ADDRESS_NOT_ALIGNED

ST

This instructions stores a word from the r register defined by the rd field memory into . If the load traps the register is unchanged. The effective address for the load is either "r[rs1] + r[rs2]" if the i field is zero or "r[rs1] + simm13" if the i is one.

Traps :

DATA_ACCESS_ERROR, MEM_ADDRESS_NOT_ALIGNED

Scheduling Instructions

SUSPEND

SUSPEND de-schedules the current process. The value of the `iptr` is placed in `o6`, which normally contains the `SP`. The ins are written into the stack frame and the augmented outs to the eight words below the stack frame

Traps :

RUN_QUEUE_OVERFLOW, DATA_ACCESS_ERROR

BREAK

BREAK places the process (SP) on the back of the run queue, and SUPENDs it. A break in execution is guaranteed, but note that context0 processes will not BREAK.

Traps:

RUN_QUEUE_OVERFLOW, DATA_ACCESS_ERROR

RUN (rs2)

RUN takes the value in rs2 and assuming it to be a valid thread for the current context places it on the rear of the run queue.

Traps:

RUN_QUEUE_OVERFLOW, MEM_ADDRESS_NOT_ALIGNED, DATA_ACCESS_ERROR.

Local Event Control

Local events are executed by the output process. These instructions are atomic to operations occurring at the input.

WAITEVENT (rs2)

Test the event location pointed to by rs2. If the event is set it is cleared and the process continues. If the event is not set then the thread suspends itself on the event. Multiple threads can be suspended on queueing events.

Traps :

EVENT_QUEUE_OVERFLOW, MEM_ADDRESS_NOT_ALIGNED, DATA_ACCESS_ERROR, DATA_ACCESS_EXCEPTION.

```
SETEVENT (rs2)
```

Set the event pointed to by rs2. If a local thread is waiting then the event is cleared and the thread run. If the suspend location contains a remote event location, then a remote set is queued on the reply processor.

Traps:

RUN_QUEUE_OVERFLOW, MEM_ADDRESS_NOT_ALIGNED, DATA_ACCESS_ERROR, DATA_ACCESS_EXCEPTION.

Locked Memory Instructions
NOT YET IMPLEMENTED

```
ATOMICADD (rs1, rs2, rd)
```

ATOMIC_ADD does an atomic read add write operation with the value at address rs1 and the value in rs2. The read value is placed in rd.

Traps:

MEM_ADDRESS_NOT_ALIGNED, DATA_ACCESS_ERROR, DATA_ACCESS_EXCEPTION.

```
TESTSTORE (rs1, rs2, rd)
```

TESTSTORE does an atomic read test write operation with the value at address rs1, the test value in rs2 and the new value in rd. The value of the read is placed in rd. If the test value is equivalent to the value of the read then it is replaced by writing the new value else the location is unchanged.

Traps:

MEM_ADDRESS_NOT_ALIGNED, DATA_ACCESS_ERROR, DATA_ACCESS_EXCEPTION.

Network Instructions

All network transactions are sent in the currently open packet. If a packet is not open a packet sequence trap will occur.

```
OPEN (rs2)
```

Opens a packet to be transmitted to the virtual process number indicated by rs2. A translation of the virtual process number to physical processor and context is first executed. This is done by indexing into the virtual process table and returning the destination physical processor number and context. The context is held in a temporary register during the outputing of the packet and added to each transaction. The processor number is used to index into a table of route data. These are fetched and placed at the outputer.

Traps:

```
PACKET_OPEN_FAILED, DATA_ACCESS_ERROR,
PACKET_SEQUENCE_ERROR.
```

Implementation Note: route bytes maybe cached on a last used basis.

```
CLOSE (rd)
```

CLOSE is matched to open and delimits a remote packet. *rd* is set to 1 or 0 according to the value of the acknowledge returned by the packet. CLOSE waits (without de-scheduling) for the receipt of the acknowledge and authorises sending of the EndOfPacket signal. CLOSE returns with 1 in rd if an ACK was received and 0 otherwise.

The value of a conditional transaction may be reflected in the value set by a CLOSE.

Traps:

```
PACKET_SEQUENCE_ERROR
```

```
SENDTRANS (imm, rs2, rd)
```

Send transaction instruction, used to launch transactions into the network.

Generate the transaction specified by *imm* to the context currently opened, with the address in *rd*. Cause a trap if not currently open. If rs2 is not %g0 the parameters for the transaction come from a block of memory pointed to by rs2 else come from registers %o0 to %o1 as required. Only the number specified in the SIZE field of the transaction get used, so if the SIZE is zero, both %o0 and %o1 are ignored. For transactions which need more than two word parameters ie those with a SIZE field larger than 0 or 1 (double words), the block version must be used. In practice this is only required for the tr_DMArequest and tr_testandwrite.

The `imm` field is a 16 bit field, bits [20:5] and as such is fixed at compile time.

Traps:

`PACKET_SEQUENCE_ERROR, MEM_ADDRESS_NOT_ALIGNED, DATA_ACCESS_ERROR, DATA_ACCESS_EXCEPTION.`

DMA Control

DMAs are performed by a separate engine with its own work queue. The DMA instruction places DMA descriptors on this queue. The DMA engine and the thread processor are of equal priority so that where both have work to do both make progress. DMA processes are timesliced to prevent long DMAs from hogging resource. The timeslice interval is chosen so that the overhead on memory bandwidth of rescheduling the DMA descriptor is small compared to the amount of data transferred in the timeslice period. Synchronisation with the completion of a DMA is achieved by passing the descriptor of an event location to the DMA engine, then executing a WAIT_EVENT. The DMA engine on completion will cause a SET_EVENT.

`DMA (rs2)`

DMA places a store to store DMA descriptor on the DMA queue. The descriptor is pointed to by rs2 and is 32 byte aligned.

Traps:

`DMA_QUEUE_OVERFLOW, DATA_ACCESS_ERROR`

C.4 *Opcodes*

C.4.1 *Format 1 Opcodes*

```
op

01 CALL
```

C.4.2 *Format 2 Opcodes (op = 00)*

```
op2

\begintable{}
opcode & usage & op2

SENDTRANS & rs2 rd & 001
Bicc & none & 010
SETHI & rd & 100
\endtable

\sectionthree{Format 3 Opcodes (op = 10)}

\beginprogram{}
op3

000000 ADD
000001 AND
000010 OR
000011 XOR
000100 SUB
000101 ANDN
000110 ORN
000111 XNOR
010000 ADDcc
010001 ANDcc
010010 ORcc
010011 XORcc
010100 SUBcc
010101 ANDNcc
010110 ORNcc
010111 XNORcc
100101 SLL
100110 SRL
100111 SRA
```

```
110110 CPop1
111000 JMPL
111100 SAVE
111101 RESTORE
```

C.4.3 *Format 3 Opcodes (op = 11)*

```
op3

000000 LD
000100 ST
001111 SWAP
```

C.4.4 *CPop1 Opcodes (op = 10, op3 = 110110)*

opcode	usage	opc	Hex Opcode
BREAK	none	000000000	0x81b00000
SUSPEND	none	000000001	0x81b00020
CLOSE	rd	000001000	0x81b00100
RUN	rs2	000010000	0x81b00200
OPEN	rs2	000010001	0x81b00220
SETEVENT	rs2	000010010	0x81b00240
WAITEVENT	rs2	000010011	0x81b00260
DMA	rs2	000010100	0x81b00280
ATOMICADD	rs1 rs2 rd	000011000	0x81b00300 (Not implemented)
TESTSTORE	rs1 rs2 rd	000011001	0x81b00320 (Not implemented)

Bits [3:4] of opc define register usage

PROGRAMMER'S INFORMATION — DMA PROCESSOR

D.1 *DMA Descriptor*

The DMA descriptor is a command to the DMA processor, it causes a DMA to be executed, and after successful completion a number of event locations to be set. A number of fields in the descriptor define control of the DMA, these are:

DMASize
The number of bytes required to be written to destination.

SourceAddress
Virtual address indicating where in the local store the bytes are to be read from.

DestAddress
Virtual address indicating the start address of where the bytes are to be written. This address is in the memory space of the destination virtual process.

DestProcess
To allow DMAs to act across the network DMA nominates a virtual process number of the destination process. This will be translated in the context of the descriptor. If the destination process is negative the DMA happens locally and the the destination event is set in the local memory space.

LocalEvent
After successful completion of the DMA this event location is set. Null is taken to nominate no event location.

DestEvent
Nominates virtual address of second event to be set on destination process.

Context
On the queue this indicates the context the DMA is to be run in locally. When a descriptor is created the context is not included, but is added by the enqueueing process. This is done by taking the current context, shifting up the type field which must appear in the lower 16 bits, and ORing it in. The completed descriptor is then added to the queue. The current context is taken from the CurrContext register for the enqueueing process. This is a security feature disabling users from faking contexts and is used by the other process queues. When on the queue the context of the queue entry is in the lower 16 bits of the first descriptor word.

D.2 *DMA Queue*

The DMA queue is physically addressed, and all DMA block descriptors are aligned to 32 byte boundaries. Descriptors on the queue conform to the format below.

D.2.1 *DMA Descriptor*

```
| Type   | Context  |
| DMASize           |
| SourceAddress     |
| DestAddress       |
| LocalEvent        |
| DestProcess       |
| DestEvent         |
| SPARE             |
```

DMA Type

Type [1:0] Data type, Byte, Short, Word, Double Word (0,1,2,3)

Type [2] The set location given is on the destination process.

Type [8:3] DMA Opcode

Type [14:9] Fail count. How many packets can fail before DMA is trapped.

Type [15] Reserved

D.2.2 *DMA Descriptor ELAN 1.2*

DestProcess must be negative -5 to give previous local DMA behaviour. This frees up process number 0 as a valid process number.

D.3 *DMA Status Information*

The state of the DMA processor visible to the main processor is,

```
Status Register
Current Descriptor
DMA\_FPTR
DMA\_COUNT
```

The status register, DMA_FPTR ande DMA_COUNT give information about the state of the DMA queue. The state of a descriptor being worked on is held in the internal register as detailed in Appendix N. This descriptor is only updated as the DMA is definitely done. The descriptor read from the internal registers on a trap may detail some part of the DMA which has already been done.

D.4 *DMA Processor Opcodes*

operation	opcode
d_dma_secure	000
d_dma	001

All other opcodes trap as unimplemented DMA instructions.

COMMAND PROCESSOR INSTRUCTIONS

E.1 *Commands*

Each command can be executed with a different context selected from the command context table.

E.1.1 *Queue Commands*

Places item pointed to by address onto the queue. The address is assumed to be virtual and is translated through the MMU using the indicated context. The address is assumed to point to a 32byte aligned block. The first word is shifted up discarding the top 16bits, the context, from the table is ORed in such that user code cannot insert contexts.

Notes

The Remote/Local source bit for the DMA descriptor is not altered from the value in the descriptor passed in.

The context is filled in from the vector of contexts.

cp_REPLY_C0	6'h04
cp_REPLY_C1	6'h05
cp_REPLY_C2	6'h06
cp_REPLY_C3	6'h07
cp_run_C0	6'h08
cp_run_C1	6'h09
cp_run_C2	6'h0a
cp_run_C3	6'h0b
cp_DMA_C0	6'h0c
cp_DMA_C1	6'h0d
cp_DMA_C2	6'h0e

cp_DMA_C3 6'h0f

E.1.2 *Read Comms Processor Registers Commands*

Internal and external register reads and RmW can be performed through the command port. The address of the register is taken from the immediate field directly. In this way all the comms processors registers can be accessed.

cp_Read_internal 6'h10
cp_RmW_internal 6'h14
cp_Read_external 6'h18
cp_RmW_external 6'h1c

E.1.3 *Event Commands*

Data register defines address of an event in current context.

cp_SETEVENT_C0 6'h20
cp_SETEVENT_C1 6'h21
cp_SETEVENT_C2 6'h22
cp_SETEVENT_C3 6'h23

cp_CLEAREVENT_C0 6'h24
cp_CLEAREVENT_C1 6'h24
cp_CLEAREVENT_C2 6'h24
cp_CLEAREVENT_C3 6'h24

TIMER, ALARM AND HUSH PERIPHERALS

The timer is a memory mapped peripheral which resides on the Elan processor device. This peripheral can create interrupts which are directed through the interrupt logic. Their are three parts to the timer a clock, an alarm and a hush location.

F.1 *Clock*

nS time counter incremented in 200nS counts
S time counter (0x3b9aca00 comparator on nS counter)

The clock provides a readable nS counter. This is implemented as two 32 bit registers the first counting in nano-seconds and the second incrementing in seconds. The nano-seconds count is arranged to zero itself at 1,000,000,000nS and increment the Sec counter. The timer is normally read as a double word value. It can however be accessed as two word locations but aliasing will be observed.

The counter increments in 200nS counts, from a 5MHz crystal. This is intended to provide a granularity of less than 10 instruction cycles in the system where the clock rate is 40MHz.

F.2 *Alarm*

A 32 bit register which is decremented by 200 at 200nS intervals. If the value is or becomes negative (top bit set) an interrupt will be signalled.

F.3 *Hush Register*

The hush register is a memory mapped device, used to prevent the communications processor accessing the MBus. It is used to allow a host processor to atomically access data used by the comms processor.

The hush register when written to 1'b1, prevents the comms processor accessing the memory bus. If it is not written to 1'b0 within approximately 0.5mS, or it is written to 1'b1 again before being reset then a HushError interrupt will occur and the comms processor is again allowed access to the memory bus. The hush bit however remains set for the interrupt routine to clear. The state of the comms processor hush register is readable by the main processor.

```
Bit 0 Hush bit
Bit 1 HushError interrupt bit
```

On a HushError the hush bit is cleared and the HushError interrupt bit set. The comms processor can proceed but the interrupt routine is required to clear the interrupt bit. The interrupt routine can be executed on the comms processor.

EXCEPTIONS, TRAPS, AND INTERRUPTS

During operation of the Elan Communications processor various exceptions may occur. These exceptions are divided into two classes: processor exceptions, and interrupts.

G.1 *Processor Exceptions*

Each of the six processors may meet an exception. This is indicated by a six bit code in the bottom bits of the processor's status register. These bits are ORed together to form an interrupt line in the interrupt register. In this way an exception can cause an interrupt on the the main processor or comms processor, and have a trap handler deal with the exception.

Table of Exceptions by Processor on which they Occur

	Cmd	Input	Reply	Thread	DMA	Code
NO_TRAP	X	X	X	X	X	0x00
DATA_ACCESS_EXCEPTION	X	X		X	X	0x04
OUTPUT_INVALID_PROCESS			X	X	X	0x08
OUTPUT_INVALID_ROUTE			X	X	X	0x09
OUTPUT_TIMEOUT			X	X	X	0x0B
EVENT_QUEUE_OVERFLOW		X		X		0x0C
EVENT_INTERRUPT	X	X		X	X	0x12
QUEUE_OVERFLOW	X	X		X	X	0x0D
UNIMPLEMENTED		X		X	X	0x10
PACKET_SEQUENCE_ERROR				X		0x11
DPROC_FAIL_COUNT_ERROR					X	0x14
IPROC_NACK_AFTER_ACK		X				0x06
RPROC_NACKED			X			0x0C

NOTE

The MEM_ADDRESS_NOT_ALIGNED exception maps onto a DATA_ACCESS_EXCEPTION with the fault type indicating an alignment error.

G.2 *Interrupts*

The communications processor generates a single external interrupt and an internal interrupt. These are both maskable and have the same source interrupt register, the interrupt being the logical OR of the masked interrupt register. The interrupt register is made up from the following bits.

Bit 0 RESERVED

Bit 1 command processor exception

Bit 2 input0 processor exception

Bit 3 input1 processor exception

Bit 4 reply processor exception

Bit 5 thread processor exception

Bit 6 DMA processor exception

Bit 7 (Alarm register < 0) See Appendix F

Bit 8 Hush Time Out See Appendix F

Bit 9 External Device Interrupt 1

Bit 10 External Device Interrupt 2

Bit 11 External Device Interrupt 3

Bit 12 External Device Interrupt 4

Bit 13 External Device Interrupt 5

Bit 14 External Device Interrupt 6

Bit 15 External Device Interrupt 7

Bit 16 (TProc & RProc & DProc) all halted

Bit 17 Both inputters finished processing packets and halted

MMU USER GUIDE

The Elan MMU is based on a paged MMU. The protections required over the ELAN network are different to those of a standalone memory system.

H.1 *MMU Fault Status Register Bits*

The following table defines the meaning of bits in the MMU FSR.

Bit no	Name	meaning
31to29	Not used	read as zero
28	LastError	any type of MBus Bus error
27to22	Not used	read as zero
21	Swapped Since Last Access	Set if memory error was detected after a uCode swap.
20	BlockAccess	32 byte block access
19	WordAccess	4 byte access
18	InstructionAccess	Read of instructions
17to13	Not used	read as zero
12	Error3	MBus Error type 3 Uncorrectable
11	Error2	MBus Error type 2 Timeout
10	Error1	MBus Error type 1 Bus Error
9to8	Level	MMU level value. Zero if bus error
7	Event access	Set if event access
6	Remote access	Set if Read or write for an inputter
5	WriteNotRead	Set if a Write access
4to2	Fault type	MMU fault type (FT code)
1	FAV	Fault address valid
0	Not used	Read as zero

The following table defines the meaning of the MMU fault type bits in the MMU FSR.

FT code	Fault types
0	Not used
1	Invalid address error
2	Protection error
3	Not used
4	Translation error
5	Access bus error
6	Not used
7	Address alignment Error

H.2 *Access permissions*

Access type is encoded as a three bit field:

Bit 0 means it is a write not a read.

Bit 1 means it is remote not local.

Bit 2 means it is occurring for event handling.

There are two levels of local access permission: readonly and readwriteevent. There are five levels of remote access permission: noaccess, readonly, readonly-orevent,readwrite, readwriteevent. Remote permission can never be higher than local permission, so the combined codes are:

Access Type	000 local read	001 local write	010 remote read	011 remote write	100 local evread	101 local evwrite	110 remote evread	111 remote evwrite
Permission								
Null	f	f	fl	fl	f	f	fl	fl
localRead	.	f	f	f	f	f	f	f
read	.	f	.	f	f	f	f	f
noremote	.	.	f	f	.	.	f	f
remoteread	.	.	.	f	.	.	f	f
remotewrite	.	.	.	.	.	.	f	f
remoteevent	.	.	.	f	.	.	.	.
remoteall	.	.	.	.	.	.	.	.

Key: f → fault
fl → Acknowledge but ignore (used for broadcasts)

Accesses marked as fl causes the memory cycle to fail but return a seperate error code. If an input process receives this error code it treats the operation as if it had succeeded and will cause an ACK to be sent. The access does not occur. This is used to implement non-contiguous broadcast sets. This is required to enable the recombining ACK/NACK logic to work successfully.

H.3 *Flushing*

The ELAN MMU is flushed by writing 32'h0 to the `FLUSH` external register. Writing values other than zero will cause destructive operations to the TLB but without disabling the entries, these operations being reserved for testing the TLB.

More for this section Please

ELAN CONTROL WORD

I.1 *Introduction*

Several control configuration parameters are required to be set in the elan processor. Some of these are relevant to the individual processes and appear in the respective status registers. Other overall control parameters are set in the Control Word, this document describes these parameters and their default behaviour.

The control word bits have the following meaning:

```
Bit [0]     R/W       MMUEnabled
```

Active High. Reset to zero.

```
Bit [1]     R/W       SER
```

Software External Reset. SER is connected to an output pin on the device an can be used as a single output bit.

```
Bit [2]     R/W       SIRout
```

Causes internal reset of Elan processor. The control word is not reset except for this bit.

Software Internal Reset

```
Bit [3]     R/W       CProcReset
```

Resets CProc. Used during CProc exception trap handler to clear pending exception. Reset to zero

```
Bit [4]      RO       CProcError
```

CProc has trapped. This is cleared using CProcReset, to enable the processor to be restarted the command port needs to be emptied by reading and saving the active commands.

```
Bit [5]     R/W       HaltOthers
```

Setting this bit will halt the three queue processors, rproc, tproc and dproc. The halt points for the three processors are the same, Wakeup Always AND the output's not open OR Wakeup Runnable, if the processor gets de-scheduled and the above is true the processor will NOT be rescheduled till HaltOthers is written low.

Reset value is zero.

```
Bit [6]      R/W      HaltInputters
```

Setting this bit will halt both inputters The halt points for the both processors are the same, process EOP. ie Only trap when not in middle of packet, if the processor gets de-scheduled and the above is true the processor will NOT be rescheduled till HaltInputters is written low.

Reset value is zero.

```
Bit [7]       RO      OthersHalted
```

When the HaltOthers bit is set this flag indicates whether the processes have halted. Only when ALL three processes have halted in the state required by the halting function will this bit be set.

```
Bit [8]       RO      InputsHalted
```

When the HaltInputters bit is set this flag indicates whether the processes have halted. Only when BOTH input processes have halted in the state required by the halting function will this bit be set.

```
Bit [10:9]    R/W      PerfCont
```

Performance Meter control, to enable in situ speed testing of the Elan processor the device has a number of ring oscillators which can be multiplexed to drive the Seconds counter instead of the FiveMegClock input. These oscillators are designed to each exercise a different parts of the silicon processing of the Elan chip. The rate of oscillation will vary from device to device, and for a particular device temperature and supply voltage.

Reset value is 00.

PerfCont	Function	Tick Rate
00	Normal Clock	1 Second
01	NAND	< 2.942 uS
10	NOR	< 2.982 uS
11	INV and track	< 7.863 uS

The NAND and NOR functions are ring oscillators built from NAND into INV and NOR into INV primitives respectively. The NAND function will test the strength of the n-type transistors and the NOR function the p-type transistors.

The INV and track test is simply inverters driving near maximum ratio track load. This test calibrates the drive to track load for the process.

The given tick rates are the maximum simulated. For devices within process specification the tick rate will be less than the above.

```
Bit [11]    RO          Link0InReset
Bit [12]    RO          Link1InReset
```

These bits output the Reset state of the link. If a link is in reset for any reason, then its LinkNInReset bit will be set.

```
Bit [13] R/Clearable  Link0Error
Bit [14] R/Clearable  Link1Error
```

These bits will become set if either a line data error or a line phase error is detected. The error is cleared by writing a '1' to the same bit location.

```
Bit [15]    R/W         SyncCProc
```

If this bit is set then before each command is executed a check is made to ensure that there are no outstanding MBus errors. This will guarantee that when wakeup never is written to a status register the value returned by the ext reg read mod write includes all possible traps. Trap code will not update the status register after the ext write.

```
Bit [31:16]       Reserved
```

Read as zero.

COMMUNICATIONS PROCESSOR MEMORY MAP

The communications processor memory map is divided into two parts, slave device addresses and external data structures.

The slave locations appear in the Mbus mapping for the Elan device. These locations control functionality and give access to the clock and alarm registers.

The external data structures are areas of store used by the Elan when running for queue and trapping information.

J.1 *Slave Device Locations*

All slave device locations are fixed with respect to physical address 0xFFi000000, where i is the 4 bit MBus ID driven onto the Elan ID pins. The Elan does not respond to all addresses in this range. Those which are unmapped will cause a MBus timeout when accessed across the MBus.

J.1.0.1 Configuration and ID registers

0xFFiFFFFFC	Mbus Device ID	Read Only
0xFFiFFFFF0	Control register	Read/Write

The Mbus Device ID defines the implementation and revision of the device. This 32 bit value has 4 fields defined by the MBus specification.

Bits Wide	Name	Arev	B Rev	C Rev
16	Implementation	0	0	0
8	Device type	9	9	9
4	Revision	0	1	2
4	Vendor code	0xf	0xf	0xf

The MBus Device ID for the C revision Elan is therefore 0x0000092f.

J.1.0.2 Interrupt ID registers

```
      Main processor interrupt mask
         Single word of store
         0xFFiFFFFE8
      Interrupt register
         Single word of read only store
         0xFFiFFFFD8

   Comms processor store objects
      Command port
         256K area of store
         Mappable to 4K boundaries providing 63 commands
one
         in each page + page 0 (user read only) which
provides
         info Single page non cacheable
         0xFFiF80000

      Clock Hi/Lo
         Two words of store
         0xFFiFFFFC0 Seconds word access R/W
         0xFFiFFFFC8 nS word access R/W
         0xFFiFFFFE0 {Seconds, nS} double word access,
READ ONLY
         Note little endian addressing is same as big
endian

      Hush register
         4K area of store (User read/write/remote)
         Single page non cacheable
         0xFFiFFE000   (Read/write)

      Alarm register
         Single word decrementing counter
         0xFFiFFFFD0
```

J.2 *Main Store Used by Comms Processor*

J.2.1 *Queues and Exception Areas*

The QUEUE_STORE_BASE register defines the area of store used by ELAN. This register is is split into two fields, the upper 27 bits, QUEUE_STORE_BASE[31:5] are masked and shifted up 4 bits to become the 36 bit physical address, Q_STORE_BASE[35:0]. The lower 5 bits QUEUE_STORE_BASE[4:0] are used in to produce Q_SIZE defined as (1 << QUEUE_STORE_BASE[4:0]). Q_SIZE denotes the size of the processor queues in units of the queue descriptor size ie 32 byte chunks.

Memory Usage	Size(Bytes)	Start Address
Reply Queue	Q_SIZE*32	Q_STORE_BASE + 0
Thread Queue	Q_SIZE*32	Q_STORE_BASE + (Q_SIZE * 32)
DMA Queue	Q_SIZE*32	Q_STORE_BASE + 2 * (Q_SIZE * 32)
Command Exception	16	Q_STORE_BASE - (16 * 1)
Input 0 Exception	16	Q_STORE_BASE - (16 * 2)
Input 1 Exception	16	Q_STORE_BASE - (16 * 3)
Reply Exception	16	Q_STORE_BASE - (16 * 4)
Thread Exception	16	Q_STORE_BASE - (16 * 5)
DMA Exception	16	Q_STORE_BASE - (16 * 6)

Example:

After reset the internal register QUEUE_STORE_BASE say is set to 0x045. This gives values to Q_STORE_BASE of 0x400 and Q_SIZE of 0x20. Each queue then has a size of 1K bytes and the first queue, that of the reply processor, appears at the address 0x400. The exception area is just below this address. These values in the above equations give the following addresses in the first 4K of store.

Memory Usage	From	To
DMA Exception	0x0000003a0	0x0000003af
Thread Exception	0x0000003b0	0x0000003bf
Reply Exception	0x0000003c0	0x0000003cf
Input 1 Exception	0x0000003d0	0x0000003df
Input 0 Exception	0x0000003e0	0x0000003ef
Command Exception	0x0000003f0	0x0000003ff
Reply Queue	0x000000400	0x0000007ff
Thread Queue	0x000000800	0x000000bff
DMA Queue	0x000000c00	0x000000fff

J.2.1.1 Queues Structures

A queue item has an entry size of 32 bytes aligned on 32 byte boundary.

The processor queues are accessed by common routines which check for overflow and context0 overflow. The only exceptions which can be generated by these routines are

QUEUE_OVERFLOW, DATA_ACCESS_ERROR.

The three types of descriptor have the following format.

These should be removed from processor descriptions

J.2.2 *Translation Tables*

```
   Comms Process ID tables.
      Context Virtual Base and Size Tables
         2 words per context (8 * 64k bytes => 512K)
      Virtual Process Tables
         (one per active context)
         (Sparse and indirected)
   MMU translation tables.
      Root table (64K * 4) => 256K bytes
         pointed to by context table ptr
      Page Tables (Sparse and indirected)

   Route tables.  (One of)          Variable    (4M max)
      Size defined by contents of Virtual process
tables.
      Base defined by register
```

J.2.3 *Internal Interrupt Event Vector*

Size is 32 events which are all pairs of words. Base of vector is defined by internal register INTERNAL_INTERRUPT_BASE. Each vector element corresponds to an internal interrupt.

J.3 *Memory Map of Slave Word Devices*

0xFFiFFFFFC	Mbus Device ID	Read Only
0xFFiFFFFF8	Mbus Device ID	Read Only
0xFFiFFFFF4	Control register	Read Only
0xFFiFFFFF0	Control register	Read/Write
0xFFiFFFFEC	Main processor interrupt mask	Read Only
0xFFiFFFFE8	Main processor interrupt mask	Read/Write
0xFFiFFFFE4		
0xFFiFFFFE0		
0xFFiFFFFDC	Interrupt register	Read Only
0xFFiFFFFD8	Interrupt register	Read Only
0xFFiFFFFD4	Alarm register	Read Only
0xFFiFFFFD0	Alarm register	Read/Write
0xFFiFFFFCC	Clock Lo	Read Only
0xFFiFFFFC8	Clock Lo	Read/Write
0xFFiFFFFC4	Clock Hi	Read Only
0xFFiFFFFC0	Clock Hi	Read/Write
0xFFiFFE000	Hush register 4K area of store	Read/Write
0xFFiF80000	Command port 256K area of store	Read/Write

J.4 *Memory Map of Slave Double Word Device*

0xFFiFFFFF8	Mbus Device ID (twice)	Read Only
0xFFiFFFFF0	Control register (twice)	Read Only
0xFFiFFFFE8	Main interrupt mask (twice)	Read Only
0xFFiFFFFE0	Clock (Hi,Lo)	Read Only
0xFFiFFFFD8	Interrupt register (twice)	Read Only
0xFFiFFFFD0	Alarm register (twice)	Read Only
0xFFiFFFFC8	Clock (Lo,Lo)	Read Only
0xFFiFFFFC0	Clock (Hi,Hi)	Read Only

MEIKO BYTE-WIDE LINK LINE-PROTOCOL

K.1 *Link Connection*

The basic characteristics of Meiko links is that they are; byte wide; bidirectional; point to point; and high bandwidth (>50 MBytes/s each direction). Each link consists of 20 wires; 10 for the input port, and 10 for the output port. Each port has one clock wire and nine data lines. On both positive and negative transitions of the ChanClkIn wire the ChanIn wires are sampled. The output port sets up a new data pattern on ChanOut at the start of each communications clock period, and toggles ChanClkout in the middle of each period.

```
output port          input port

ChanClkOut           ChanClkIn
ChanOut[8]           ChanIn[8]
...                  ...
...                  ...
ChanOut[0]           ChanIn[0]
```

K.2 *Link Values Encoding*

The line protocol has eight command values, as well 256 data values encoded. No single bit error can change data into a command or visa-versa. No single bit error can change one command into another.

The commands are as follows:

```
Command       Code              Usage

 NULL    {3'h7,3'h0,3'h0}  Nothing to be sent.
 GAP     {3'h7,3'h1,3'h1}  Used for bit stuffing to get
                           receiver in sync with sender.
 SOP     {3'h7,3'h2,3'h2}  Start of packet.
```

```
EOP     {3'h7,3'h3,3'h3}  End of Packet.
TOKEN   {3'h7,3'h4,3'h4}  Receiver can accept 16 more
                          bytes of data.
PNACK   {3'h7,3'h5,3'h5}  Packet Not Acknowledge.
PACK    {3'h7,3'h6,3'h6}  Packet Acknowledge.
RESET   {3'h7,3'h7,3'h7}  Sender is in reset.
```

The order of priority of sending commands and data is shown below:

```
Highest priority                          Lowest priority

RESET   PACK    TOKEN   EOP     Data    GAP     NULL
        PNACK           SOP
```

The outputer attempts to output a GAP every 256 cycles. If having waited 128 cycles a GAP has still not been sent because the line has been continuously busy, then a GAP is sent in preference to data. When a GAP command is transmitted, it must be followed by a NULL command. The NULL following a GAP is higher priority than everything except the RESET command.

The data bytes are encoded in four ranges, as follows:

```
8'h00 - 8'h3f have the value {3'b000, Data[5:0]}
8'h40 - 8'h7f have the value {3'b001, Data[5:0]}
8'h80 - 8'hbf have the value {3'b010, Data[5:0]}
8'hc0 - 8'hff have the value {3'b100, Data[5:0]}
```

At least two of the top bits would have to change before the data byte could possibly be interpreted as a command byte. Errors which change the data values into other data values must be detected by error checking the packet contents at the packets destination.

Packets are made up of route bytes, a SOP, one or more transactions, and one or two EOPs. One PACK or PNACK is returned for each packet sent. The line protocol does not distinguish in any way between packets which have been PACK or PNACKed. If a packet is terminated prematurely by a line data error, the packet is terminated with an SOP EOP. This signals to the inputer that the packet was incomplete.

RESET, TOKEN, GAP and NULL are only used by point to point links and are invisible to higher levels of protocol.

K.3 *Flow Control*

The input port has a FIFO. Data bytes and EOP commands can be stored in the FIFO while a switch or inputer is unable to take the data. The protocol allows the FIFO to be as deep as necessary to take up the delays in the line, but in the first implementations of links it is intended that the FIFO be 48 bytes deep. The size of the byte count register must be sufficient to cope with the largest FIFO it can be connected to, which may be greater than its own FIFO size. In the initial implementation this will be 8 bits, allowing FIFOs up to 256 bytes to be connected.

Any time an input port has 16 or more bytes of space in its FIFO, it instructs its output port to send a TOKEN command and decrements its space available count by 16. This effectively transfers the ownership of 16 bytes of FIFO space from the inputer to the outputter connected to it. Each byte consumed by the inputer frees up one byte of space in the FIFO. Note that the effective size of the FIFO is reduced by between 0 and 15 bytes at any point in time because of the 16 byte granularity of the token.

When an inputer receives a TOKEN command it instructs its paired output port to increment the count of the number of bytes it may send by 16. Each time the output port sends a byte it decrements this count by one. As long as the count is greater than zero the output port is allowed to transmit data, or commands that consume FIFO space, (EOP or SOP).

Data is transmitted as packets. All packets must end with an EOP command. All packets must be either PACKed (Packet Acknowledge), or PNACKed (Packet NOT Acknowledge). Acknowledgements are passed back along the route that the packet took. If the outputting processor traps while it has its output open it will immediately send an EOP. An EOP generated in this way is termed an unsolicited EOP. If an Elite switch chip detects an error, then it will generate a BAD EOP command (this is an SOP immediately followed by an EOP command). A BAD EOP can be generated before a PACK/PNACK and hence be interpreted as an unsolicited EOP. The system must ensure that any PACK or PNACK being returned for that packet is not interpreted as being for a following packet. To ensure this, unsolicited EOP commands are handshaken in the following way. The EOP command can be issued before a PACK/PNACK has been received, but another packet cannot be transmitted along the line before the PACK/PNACK is received. The line is kept open for a packet until both the EOP is sent and the PACK/PNACK is received. If a receiver port receives an EOP before the transmitter has sent a PACK/PNACK then a PNACK is automatically sent by the transmitter. Any PACK or PNACK commands received after an EOP has gone by, and before another packet has started, are deleted.

K.4 *Links and Reset*

A link is held in reset when:

1 The Reset pin of the chip is high.

2 A JTag port holds the link in reset.

3 The input port is not receiving a clock from the line.

4 The input port is receiving RESET commands from the line.

5 The outputter has been disabled by a JTag port.

A link is put into reset for at least 256 clock cycles when :-

1 The value clocked in on ChanIn is neither a command nor Data.

2 The inputer has a phase error. (Due to excessive drift, jitter or double clocking on the ChanInClk)

3 The link needs to be cleared because of a timeout.

4 A JTag port clears the link.

When a link is put into reset the link has a defined state. This is :-

1 No packets are being sent in either direction.

2 No PACK/PNACK is outstanding.

3 The flow control FIFO is empty, but the receiver owns all the space in the FIFO. i.e. TOKEN commands must be sent before any data can be received.

4 The transmitters count of bytes it may send is set to zero.

5 Any packets being sent when the link was put into reset are completely consumed, and if possible, NACKed.

6 Any packets being received when the link was put into reset are ended with a BAD EOP (SOP, EOP);

7 The link is forced to output the RESET command, except if the link is reset by receiving a RESET command. If the link is receiving the RESET command then the transmitter sends the NULL command.

If an outputer is in the midst of sending a packet when it is put into reset, the remainder of the packet is consumed but not transmitted, and a PNACK returned to the sender. The outputter will consume the rest of the packet up to the EOP, even if the link is taken out of reset. New packets arriving at a link which is in reset are held until the link is taken out of reset.

If an inputer is receiving a packet when it is reset, it forwards a SOP, followed by an EOP. If this occurs while route information is being sent this forces the message to terminate. If the error occurs during the data part of the packet the SOP is passed in to the inputing communications processor, which detects it as an error which does not require acknowledgement.

This mechanism insures that reset can be forced at any time on a port in a way which can be detected and recovered from by all devices (either processors or switches) using that port. The reset mechanism will always reset the link in both directions; this is essential as no direction can be deduced from an erroneous command or data item. Reset is however only sent in one direction across the link, form the end that detected the error. Resets are propagated along the currently connected routes so that an entire blocked packet is flushed out.

K.5 *Clock Skew Tolerance*

The output port generates both the data and the Clock. Any variation in voltage, temperature, or process, should not cause the skew between the clock and data, as seen by the receiver, to vary. The data is clocked on both positive and negative edges of the clock. Therefore the maximum frequency of any pin is half the peak data rate.

K.6 *Clock Phase Locking and Control*

Both directions of a link must transmit at the same frequency. To avoid having to distribute a global clock, marginal (<200ppm) frequency variations are permitted. Receivers use the same frequency as their transmitters, and so have an unknown, and slowly changing phase difference with respect to the data coming in on the line. Inputters can correct for this by inserting or removing NULL commands. The points at which corrections can be made are signalled by GAP commands. GAP commands must be sent often enough to insure that the maximum frequency drift is always compensated for before it can cause phase errors.

Each receiver has a phase detecting circuit and a short FIFO. Data is clocked into the FIFO using the clock sent with the data. The data is clocked out of the FIFO using the receivers local clock. The FIFO is three entries deep. The minimum possible latency through the FIFO is zero cycles, and the maximum possible latency through the FIFO is three cycles. The receiver monitors the latency through the FIFO, and tries to maintain a 1 to 2 cycle latency. At regular intervals, the sender transmits a GAP command. The GAP command is always followed by a NULL command. When the receiver receives a GAP then, if the measured latency is greater than two, because of clock drift or transmission delay drift, then in one cycle the receiver can remove both the GAP and the NULL from the FIFO. This will reduce the latency though the FIFO by one cycle. If, when the receiver receives a GAP, the measured latency is less than one cycle, then the receiver does not take anything out of the FIFO for one cycle. This will increase the latency by one cycle.

K.7 *Automatic Link Output Tri-state*

The link has an automatic link output tri-state function. This is included to enable hot insertion of circuit boards within a switch network. When a link is operating normally, it will be receiving an edge on the ChanClkIn pin every clock cycle. If the link is disconnected then the ChanClkIn will stop oscillating. A very weak pull down resistor on the ChanClkIn pin will pull the input to ground. If the ChanClkIn pin is read as zero without being read as a one for at least 256 Comms cycles, then all the output pins of the link out will be tri-stated. The ChanClkOut pin has a weak (but not very weak) pull up resistor. So if a board is re-inserted, and the link connection made again, when the power is restored the ChanClkIn pins of both ends of the link will be read as a one (because the weak pull up resistor wins over the very weak pull down resistor). When the ChanClkIn pin has been read as a one for more than 10ms the link output pads will be taken out of tri-state. While a link is in tri-state, the link is held in reset. Links will be automatically tri-state and untri-state regardless of the state of Chip Reset. The only exception is if the link is being boundary scanned using the TAP. In this case the link will be forced out of tri-state.

STATUS REGISTERS

There is one status register for each of the six processes implemented on the microengine. Each of these status registers is mapped as an external register. This may be read/written via the command port at the external register addresses detailed in Appendix M "Elan External Registers".

On exceptions the status register, at the point at which the error is taken, is written to the exception area for the respective process. The position the exception areas are defined in Appendix J "Communication Processor Memory Map".

`Bit 31` `OutputOpen.` `(Read Only)`

This bit means that the processor has the output channel open. This can only be set for the reply, DMA, or thread process. On an error, the microcode closes any open output devices with an EOP error. If this bit is set in the Status Register as written to the exception area it means that the processor had the output open when the error occurred.

`Bit 30` `Reserved`

`Bit 29` `Reserved`

`Bit 28` `TrapOnTransBit`

This bit only has meaning for the input processors. When this is set each transaction will cause a trap to the main processor, to allow single stepping through a packet. A trap will occur on each subsequent received transaction.

`Bit 27` `Runable.` `(Read Only)`

For queued devices such as the thread, reply and DMA processors, this means that the device may have more work to do. Runnable for these devices means that QueueReady is set, and at least one of the Run bits is set. On seeing Runnable the processor will attempt to run the top item of the queue. If the

queue is empty or the top item is a non context 0 process and only context 0 are runnable then the QueueReady bit is cleared.

Runnable for the command processor means there is a command waiting in one or other of the command registers or an internal interrupt is pending.

Runnable for the input processors means there is a transaction waiting to be processed.

`Bit 26,25  RunNonContext0, RunContext0`

These two bits determine the run-ability of any request to the processors. Each queue item or input transaction has a context associated with it, the action will only be executed if the context type matches with one of these bits which is set.

For queued devices these are used to evaluate whether or not to run the top item of the queue.

For the inputters these bits cause the automatic Nacking of the respective transaction types.

If these bits have to be changed then the following procedure should be used :-

The status register should be written with the wakeup function set to never using cp_RmW_external. The new value of RunNonContext0 and RunContext0 is then inserted in the returned value and then written back again using cp_RmW_external with the original wakeup function. For this to work correctly on iproc0 and iproc1 the SyncCProc bit should be asserted in the Elan control register.

TheRunNonContext0 and RunContext0 bits have no meaning for the command processor.

`Bit 24  QueueReady`

This bit is set any time a process is put on the queue. Any time the main processor asserts one of the Run bits it should also reassert QueueReady, so that the processor can check whether this has caused the queue to become Runnable. Note that this bit DOES NOT MEAN THE QUEUE IS EMPTY/FULL. It is simply an indication to the microengine to attempt to run the item at the top of the queue.

Bit 23 Reject - inputters only (Read Only)

Reject is turned on automatically when a Nack is sent before an AckNow bit has been detected in a packet. This causes all further transactions in a packet to be thrown away.

Bit 22 AckSent - inputters only

An Ack has been sent for a packet. This is important for the input exception handlers as it means that any transactions which occur after the Ack has been sent must be handled by the exception handler, or if they are thrown away, a higher level protocol must insure retransmission.
Read Only.

Bit 21 BadCRC - inputters only

Transaction with a BadCRC has been received. This causes an exception. If no Ack has been sent Reject will be turned on. Exception handler must restart the device by writing appropriate wakeup function and suspend address to device.
Read Only.

Bit 20 SOP error - inputters only

Packet was terminated by reset or another SOP (SOP EOP means end of packet error).
Read Only.

Bit 19 ProcessingPacket - inputters only

This bit will be set in the status register for all transactions other than EOP. ie when TrapOnTrans is asserted it can be used to indicate that the EOP has been received.
Read Only.

Bit [18:16] WakeupFn

Valid functions are:

0 WakeupNever	Sleep
1 WakeupAlways	Swapped out but going to run again
2 WakeupRunable	Waiting for queue item

Following for outputing processors only:

3 WakeupOutputReady	Waiting for output and at least two trans space.
4 WakeupAckNAckOrTimeout	Waiting for Ack on output
5 WakeupTwoTransSpace	Waiting for Space

Programming note. It is possible for a memory error to cause a process to trap with the wakeup function set to WakeupAckNAckOrTimeout. If this is seen then the trap code must write to that processes local external ClearAckNAck location so that the AckNAck buffer that was being used by processes for outputing a packet is freed up again.

`Bit [15:6]  SuspendAddr`

This is the suspend address of the micro-process which is controlling the processor. Valid suspend addresses are defined in L.1. The field is 10 bits wide of which Elan 1.0 only uses 9 bits.

`Bit [5:0]  TrapType`

Meaning of TrapType is discussed in Appendix G on "Exceptions".

The reset state of the status registers is all zero except with the run context0 bits enabled, and the inputters enabled.

L.1 *Micro-Process Suspend Addresses*

The implementation of the six Elan processes is as six micro-processes timesharing the same micro-processor. Each executes micro-code from a ROM, to achieve high-density and functional integrity they share areas of this ROM.

It is necessary for trap handling code to sometimes alter the SuspendAddr, which is the micro-code pointer to where execution will restart. Only certain micro-code addresses are valid restart points, each having a different usage. These are listed in the table below. This is achieved by changing the SuspendAddr field in the respective status register.

`Addr_NullMWord  0x001`

Suspend point for idle process. Setting a process to this value and runnable will cause it to idle but at expense of lower priority processors which will not be able to execute at all.

On an exception the SuspendAddr is set to this value by exception microcode.

`Addr_DequeueDMA 0x010`

Reset value for DMA Processor. Place to restart DMA after removing descriptor. DMA processor will begin executing descriptors off the DMA queue.

`Addr_AfterSwapSPARCEntry 0x018`

Timeslice point while executing thread code. Thread processor timeslices between each instruction. PC, nPC, condition code bits and registers are all valid.

`Addr_DequeueThread 0x020`

Reset value for Thread Processor. Place to restart thread after removing or deleting executing descriptor because of exception. Processor will begin executing descriptors off the thread run queue.

`Addr_ExecuteCommand 0x028`

Reset value for command processor. Only place to restart command processor.

`Addr_DequeueReply 0x030`

Reset value for reply Processor. Place to restart reply processor after removing or deleting executing descriptor because of exception. Processor will begin executing descriptors off the reply queue. If the reply buffer is marked as valid, it will be used as the first reply descriptor.

`Addr_NoThereIsnt 0x034`

Reply suspend value while waiting for output to become free or for enough space to appear in output buffer during execution of a descriptor. All reply descriptors are read into the reply buffer during their execution.

`Addr_DoReply 0x036`

The reply processor waits here when it has dequeued a descriptor into the reply buffer and is waiting for the outputter to have enough space in its buffer, or ownership of the buffer.

`Addr_DoDmaLoopRead 0x06c`

The dma processor waits here, between transactions on a dma packet sent into the network, for output buffer space to become available and to let higher priority processes in.

`Addr_WakeupForOpen 0x040`

The thread processor waits here for space in th output buffer or ownership of the buffer.

`Addr_ReplyCheckAck 0x0ee`

Reply processor suspend address while awaiting an acknowledge in response to a packet. It is unlikely that this value will be seen, as the link logic will always supply an acknowledgement (packet Ack, NAck or timeout). However if this value is seen, and rproc has been stopped, then the trap code must write to rprocs local external ClearAckNAck to free off the Ack buffer.

`Addr_CheckAckNAckTimeout 0x106`

DMA processor suspend address while awaiting an acknowledge in response to a packet. It is unlikely that this value will be seen, as the link logic will always supply an acknowledgement (packet Ack, NAck or timeout). However if this value is seen, and dproc has been stopped, then the trap code must write to dprocs local external ClearAckNAck to free off the Ack buffer.

`Addr_SendDMATransactions 0x10e`

The dma processor waits here for output buffer space and higher priority processes before preparing the first transaction of a dma packet to be sent into the network.

`Addr_DoLoopRead 0x127`

The dma processor waits here during memory memory DMAs to let higher priority processes in.

`Addr_WakeupForSendTrans 0x130`

The threads processor waits here for enough output buffer space before executing a Send trans instruction.

```
Addr_DMASentOK3     0x15b
```

The dma processor waits here for the outputer to become free between packets of a multi packet network dma.

```
Addr_HandleTranscation     0x163
```

Normal wakeup address for inputters. Inputter will wake up a valid transaction appears in the input buffer.

```
Addr_MoveToNextTransaction     0x167
```

Inputter will wake up and move to the next transaction in the input buffer.

```
Addr_WakeupForClose     0x176
```

Thread processor suspend address while awaiting an acknowledge in response to a packet. If this value is seen, and tproc has been stopped, then the trap code must write to tprocs local external ClearAckNAck to free off the Ack buffer.

```
Addr_DoSendDMAEOP     0x1a5
```

The dma processor waits here for enough buffer space to send the eop of a dma packet.

```
Addr_SendNullTransaction     0x1bb
```

DMA processor is waiting to send a null transaction at the end of packet with AckNow bit set to ensure an acknowledge has been sent.

```
Addr_NAckAndMoveToNextTransaction     0x1e3
```

Inputter will wake up and attempt to send a packet NAck in response to the current packet. If successful the pointers are correctly moved on in response to the current transaction. If the packet has already been acknowledged a trap will occur.

```
Addr_DMASentOk     0x1f1
```

The dma processor waits here to let other processes in on a memory memory dma before updating the descriptor.

```
Addr_ResetMWord     0x000
```

This microword is not a suspend address but is used in conjunction with the NullMWord by the idle microengine process.

```
Addr_InputterEntry     0x008
```

Reset value for inputters. Inputters power up in this state but with Runnable set to Always. This location resets the input buffer pointers and sets the wakeup function to Runnable, then suspends itself at Addr_HandleTranscation, ready to receive input transactions.

ELAN EXTERNAL REGISTERS

The following is a list of the external register values. The external registers are accessed by a 32 bit read and 32 bit write bus. Not all of the registers return 32 bit values. Where this is the case the extra bits are returned as zero. As for the internal registers some of the registers can be accessed locally. The local access address being formed with bits [7:5] coming from the processor identification number ProcessID. The read and write addresses are both 8 bits wide, for external register reads however bit [4] is ignored. Global accesses can be made to the external registers, these being made at addresses 0x00 - 0x1f.

The external registers may have different read and write behaviour.

M.1 *External Register Definitions*

The following tables list the external register definitions. The tables consist of 2 columns: the first column defines the registers meaning when it is written to; the second column defines its meaning when it is read. The two meanings may, or may not, be the same

M.1.1 *Global External Registers addresses (0x00 - 0x1f)*

`NoWrite`	`0x00`	`VAddrReg`	`0x00`
No write occurs if a write is made to this address. Maps nowhere.		Virtual address register.	
`ResetOutput`	`0x01`	`CurrAT`	`0x01`
Reset the outputter logic.		Current MBus access type	
`InputContextFilter`	`0x02`	`R/W location`	
Bits [15:0] read and write the 16 bit value of the input context filter. If inputs with this value of context are received they will be NACKed automatically. The NACKing will begin on the next			

packet received.

ReplyBufferValid	0x03	R/W location

Bit[31] indicates validity of reply buffer.

CurrContext	0x04	R/W location

Bits[15:0] read and write the current value of the hardware context being used by the MMU.

LineAddr	0x05	BootModeAndLineAddr	0x05

Line address access to TLB

LineData	0x06	R/W location

32 bit line data currently pointed to by TLB.

Flush	0x07	TlbAccessFault	0x07

Invokes flush function on TLB

Reads TLB access fault type.

WriteBlockSize	0x08	R/W location
WriteDataPtr	0x09	R/W location
ExtMemDataReg	0x0a	R/W location
ReadDataPtr	0x0b	R/W location
PhysAddrLo	0x0c	R/W location
PhysAddrHi	0x0d	R/W location

36 bit physical address register. Writing to the high part causes the top 32 bits to get written and the lower four bits to be zeroed. Writing the lower part merely writes the bits [31:0] of the physical address register. Reading the high part returns the bits [35:4] and the low part bits [31:0].

SetMagicBits	0x0e	LastAccess	0x0e

Sets written data bits in magic field

Read last memory access
Bit allocation is
((MMUs FSR as for mem error) << 3)
| (uCode ProcID initiating access)

ClearMagicBits	0x0f	MemException	0x0f

Clears written data bits in

Read type of memory exception

magic field.

PhysWordWrite	0x10	No read mapping
PhysWordRead	0x11	No read mapping
PhysBlockWrite	0x12	No read mapping
PhysBlockRead	0x13	No read mapping
TLBWrite	0x14	No read mapping
TLBRead	0x15	No read mapping

Do physical memory operation.

VAddrRegReadByte	0x18	No read mapping
VAddrRegWriteByte	0x19	No read mapping
VAddrRegReadWord	0x1a	No read mapping
VAddrRegWriteWord	0x1b	No read mapping
VAddrRegReadBlock	0x1c	No read mapping
VAddrRegWriteBlock	0x1d	No read mapping
VAddrRegEventWord	0x1e	No read mapping
VAddrRegLocalWriteWord	0x1f	No read mapping

Set virtual address register with given access type

M.2 *Locally Mapped Registers*

M.2.1 *Command Processor Externally Mapped Registers (0x20 - 0x3f)*

WakeupFunction	0x27	ProcessId	0x27
Bits[2:0] writes wakeup function into status register of Cproc		Read process ID of Command processor ie 0x00000001	
CommandData	0x28	R/W location	
32 bit register used by command port to transfer data.			
CommandFinished	0x29	CommandPort	0x29
Indicate to command port that command processor has finished the current command		Command from command port Bit [31] indicates that an internal interrupt is pending	
No Write mapping		MaskedInterruptReg	0x2c

Interrupt register bit wise ANDed with internal interrupt mask

`CommsProcIntMaskReg 0x2d` R/W location

18 bit Internal interrupt mask register.

`StatusReg 0x2f` R/W location

32 bit Status register for Cproc processor. Fields within status register are defined in Appendix L

M.2.2 *Iproc0 Externally Mapped Registers (0x40 - 0x5f)*

`InputReplyBufferInc 0x40`

Writing will cause input reply buffer to be incremented if it is not owned by Iproc1.

`InputReplyBufferPtr 0x40`

Bits [7:0] read value of current input reply buffer pointer. Bit [31] reads 0 if input reply buffer is in use by Iproc1.

`InputReplyBufferReset 0x41`

Resets input reply buffer pointer logic.

`CondTransResult 0x42`

Send ACK or NACK depending on value of previous subtract. Used to make network conditional fast.

`SendNAck 0x43`

Send a NACK to the processor from whom we are currently receiving.

`CurrContextAndInputCode 0x44`

32 bit write value which sets the

`InputFirstCase 0x44`

Reads microcode case value for

current MMU context for Iproc0 and transaction code being decoded.

transaction being decoded given state of Iproc0.

`TransactionUsed` `0x46` `NextTransFront` `0x46`

The input transaction has been processed

Bits [7:0] read value of pointer to next transaction in Iproc0 buffer.

`WakeupFunction` `0x47` `ProcessId` `0x47`

Bits[2:0] writes wakeup function into status register of Iproc0

Read process ID of Iproc0

`StatusReg` `0x4f` `R/W location`

32 bit Status register for Iproc0 processor. Fields within status register are defined in Appendix L

M.2.3 *Iproc1 Externally Mapped Registers (0x60 - 0x7f)*

`InputReplyBufferInc` `0x60` `InputReplyBufferPtr` `0x60`

Writing will cause input reply buffer to be incremented if it is not owned by Iproc0.

Bits [7:0] read value of current input reply buffer pointer. Bit [31] reads 0 if input reply buffer is in use by Iproc0.

`InputReplyBufferReset` `0x61`

Resets input reply buffer pointer logic.

`CondTransResult` `0x62`

Send ACK or NACK depending on value of previous subtract. Used to make network conditional fast.

`SendNAck` `0x63`

Send a NACK to the processor from whom we are currently receiving.

`CurrContextAndInputCode` `0x64`

32 bit write value which sets the current MMU context for Iproc0 and transaction code being decoded.

`InputFirstCase` `0x64`

Reads microcode case value for transaction being decoded given state of Iproc1.

`TransactionUsed` `0x66`

The input transaction has been processed

`NextTransFront` `0x66`

Bits [7:0] read value of pointer to next transaction in Iproc1 buffer.

`WakeupFunction` `0x67`

Bits[2:0] writes wakeup function into status register of Iproc0

`ProcessId` `0x67`

Read process ID of Iproc1

`StatusReg` `0x6f`

32 bit Status register for Iproc1 processor. Fields within status register are defined in Appendix L

`R/W location`

M.2.4 *Rproc Externally Mapped Registers (0x80 - 0x9f)*

`SendRouteBytes` `0x80`

Makes output logic send route bytes. This together with SendTransaction and SendEOP delimit the sending of transactions and the EOP.

`OutWordBackAnd2TransSpace` `0x80`

Bits [7:0] pointer to output buffer space. Bit[31] indicates whether there is enough free space for another two transactions.

`SendTransaction` `0x81`

`OutByteBackAnd2TransSpace` `0x81`

Makes output logic send transaction.

Bits [7:0] pointer to output buffer space. Bit[31] indicates whether there is enough free space for another two transactions.

`SendEOP` `0x82` `No read mapping`

Makes output logic send EOP.

`No write mapping` `CheckAckNAck` `0x83`

Read output packet status. Bit[31] indicates whether the packet has ended. The following read values are possible,
0x00000000 Ack/Nack not yet received
0x80000001 Ack received
0x80000004 Packet timed out
0x80000005 Nack received

`ClearAckNack` `0x84` `No read mapping`

Clears the Ack/Nack status. This frees up the logic which buffers the Ack/Nack and allows its use by another output packet. This is the final step in sending a packet and acknowledges the reading of the CheckAckNAck value.

`ReserveTrans` `0x85` `No read mapping`

Reserve the space for a transaction to be sent in the output buffer.

`ReplyBufferPtr` `0x88` `R/W location`

Returns pointer into reply buffer. Writing to this location

writes the pointer. Only bits[3:1] get written.

No write mapping

ReplyBufferValidTrue 0x8b

Bit[31] indicates whether reply buffer is valid.

StatusReg 0x8f R/W location

32 bit Status register for Rproc processor. Fields within status register are defined in Appendix L

M.2.5 *Dproc Externally Mapped Registers (0xc0 - 0xdf)*

SendRouteBytes	0xc0	OutWordBackAnd2TransSpace	0xc0
SendTransaction	0xc1	OutByteBackAnd2TransSpace	0xc1
SendEOP	0xc2		
		CheckAckNAck	0xc3
ClearAckNAck	0xc4		
ReserveTrans	0xc5		

Output controls, as for reply processor

WakeupFunction	0xc7	ProcessId	0xc7

Bits[2:0] writes wakeup function into status register of Dproc

Read process ID of DMA processor 0x00000006

DmaResetTimeSlice	0xc8	DmaDoTimeSlice	0xc8

Reset DMA timeslice indicator

Indicates whether or not the DMA timeslice period has been exceeded.

StatusReg 0xcf R/W location

32 bit Status register for Dproc

processor. Fields within status register are defined in Appendix L

DmaWriteBlockSize	0xd8
SkipPos	0xd9
WriteDataPtrDma	0xda
ReadDataPtrDma	0xdb
SourceDestDiff2to0	0xdc
FlushPrevReg	0xdd
DmaDataType	0xde
TLBBlockReadDma	0xdf

These registers control the DMA logic.

M.2.6 *Tproc Externally Mapped Registers (0xa0 - 0xbf)*

SendRouteBytes	0xa0	OutWordBackAnd2TransSpace	0xa0
SendTransaction	0xa1	OutByteBackAnd2TransSpace	0xa1
SendEOP	0xa2		
		CheckAckNAck	0xa3
ClearAckNAck	0xa4		
ReserveTrans	0xa5		

Output controls, as for reply processor

WakeupFunction	0xa7	ProcessId	0xa7
Bits[2:0] writes wakeup function into status register of Tproc		Read process ID of thread processor 0x00000005	

SC_Instruction	0xa8	R/W location

32 bit instruction register. Used by Tproc logic to decode instruction.

SC_InsStatus	0xa9	R/W location

Bit[31] indicates validity of IN registers. Bit[0] indicates whether IN registers are dirty.

SC_cc	0xaa		
Bits[3:0] write Tproc condition code bits.			
SC_op_compat	0xab	SC_rd	0xab
		SC_ANNUL	0xab
Controls execution of Tproc instructions on arithmetic or logic unit.		Returns number of destination register for current Tproc instruction in Bits[5:0]. Bit[31] indicates annul status if instruction is a Bicc.	
		SC_FirstCase	4'hc
		SC_BRANCH	4'hc
No write mapping		Returns case value for Tproc instruction in bits [8:0]. Bit[31] holds branch status if instruction is a Bicc.	
		SC_Immediate	0xad
No write mapping		Value of Immediate field for current Tproc instruction.	
SC_DirtyInsIfDest	0xae	No read mapping	
Indicates that IN register block needs to be marked as dirty ie need to be written back to store if the current dest register is an IN register.			
StatusReg	0xaf	R/W location	

32 bit Status register for Tproc processor. Fields within status register are defined in Appendix L

`VAddrRegInstBlock 0xbc` `No read mapping`

The 32 bit address is to be treated as an instruction address for reading.

ELAN INTERNAL REGISTERS

The following is a list of the internal register contents. The ELAN has 72 internal 32-bit registers which can be accessed as either 32-bit values or 64-bit values.

Each processor has an area of the internal register map which it can access locally without indexing. This is the area where most of the values used by a processor are stored. The entire register file can alternatively be accessed by indexing. The local accesses are made relative to the following local access constants.

```
Thread_Locals    0x20
Reply_Locals     0x48
Command_Locals   0x20
DMA_Locals       0x10
Input0_Locals    0x12
Input1_Locals    0x14
```

In addition to the local accesses, each processor can make global accesses to the first 16 locations, 0x00–0x0f, without indexing.

N.1 *Internal Register Definitions*

```
ROUTE_TABLE_PTR     0x00
```

Pointer to base of route table. Individual entries given by 64 byte offset. Each entry containing four 16 route byte. Pointer is upper 32 bits of a 36 bit physical address.

```
CONTEXT_PTR     0x01
```

Pointer to base of MMU context table. Entries in turn point to MMU entries for each context. Pointer is upper 32 bits of a 36 bit physical address.

```
QUEUE_STORE_BASE   0x02 // Upper
QUEUE_SIZE         0x02 // Lower
```

Bits [31:5] form upper 27 bits of a 36 bit physical address, pointing to queue store base. This is area where the three queues for reply, thread and DMA processors are placed, together with the trap area.

Bits[4:0] form shcnt where (1 << shcnt) is the number of queue entries for each processor.

For example, 0x00000405 indicates that queues begins at address 0x000000400, and that each contains (1 << 5) ie. 32 entries.

```
OUTPUT_CONTEXT      0x03
```

If a processor has the output in use this is the value of context added to each transaction for the packet.

```
REPLY_FPTR      0x04 // Upper
REPLY_COUNT     0x04 // Lower
RUN_FPTR        0x05 // Upper
RUN_COUNT       0x05 // Lower
DMA_FPTR        0x06 // Upper
DMA_COUNT       0x06 // Lower
```

These describe the state of the processor queues for each of the queued processors.

Bits[31:16] form a 16 bit front of the queue index. Bits[15:0] form a 16 bit value giving the number of entries on the queue.

```
DMA_PACKET_SIZE   0x07

AandC_SAVE          0x08 // Two words
TEMP_A              0x08 // Usable between external
TEMP_C              0x09 // memory accesses
BandD_SAVE          0x0a // Two words
TEMP_B              0x0a // Usable between external
TEMP_D              0x0b // memory accesses

FREE_1              0x0c
FREE_2              0x0d
FREE_3              0x0e
FREE_4              0x0f
```

These values are used for temporary space by all processors while executing.

```
DMA_NoOfRdBytes      0x10 // valid when sending Pack
DMA_OUTPUT_CONTEXT   0x10 // valid when not sending a packet
DMA_NoOfWrBytes      0x11 // valid when sending Pack

TEMP_INPUT0          0x12
```

Temporary register used by Iproc0.

```
DMA_TmpSource   0x13 // valid when sending Pack

TEMP_INPUT1     0x14
```

Temporary register used by Iproc1.

```
DMA_TmpDest          0x15

DMA_ROUTE_CACHE0     0x16
DMA_ROUTE_CACHE1     0x17
```

16 byte route cache used by DMA between packets to reduce memory accesses. When no DMA is running these registers are invalid. Used in addition with DMA_ROUTE_CACHE2 and DMA_ROUTE_CACHE3.

```
DMA_DESC_TYPE_CONTEXT   0x18
DMA_DESC_SIZE           0x19
DMA_DESC_SOURCE         0x1a
DMA_DESC_DEST           0x1b
DMA_DESC_EVENT          0x1c
DMA_DESC_DESTPROC       0x1d
DMA_DESC_6              0x1e // Not used
DMA_ROUTE_CACHE2        0x1e // DMA Tmp
DMA_DESC_7              0x1f // Not used
DMA_ROUTE_CACHE3        0x1f // DMA Tmp
```

During DMA execution the 8 word DMA descriptor is held in these registers. This descriptor is not updated till the DMA has succeeded. The currently unused parts of the DMA descriptor, words 6 and 7 are used during remote DMAs as route cache words.

```
GLOBAL_0    0x20
SWAP_PC     0x20 // When thread Swapped out
```

When thread processor is timesliced contains value of PC. During thread processor execution register is set to zero. When no thread is running this value is undefined.

B_ADDR 0x21

PC 0x22
COMMAND_TEMP 0x22 // When thread Swapped out

During thread processor execution register contains current PC. When thread isn't running it is used as a temporary value by the command processor.

nPC 0x23

During thread processor execution register contains current nPC. When thread isn't running it is undefined.

COMMAND_CONTEXT_0_1 0x24 // Cmd
COMMAND_CONTEXT_2_3 0x25 // Cmd

Vector of four 16 bit contexts used by command processor in generating event or queue commands. Contexts are held little endian in the words ie command context number 0 appears in COMMAND_CONTEXT_0_1 bits [15:0] and command context number 3 appears in COMMAND_CONTEXT_2_3 bits [31:16].

CONTEXT 0x26

Value of hardware context for current thread processor. When no thread processor is running this value is undefined.

INTERNAL_INTERRUPT_BASE 0x27 // Cmd

Points to base of internal interrupt event vector. Each entry containing two words long. Pointer is upper 32 bits of a 36 bit physical address.

OUTS_0 0x28
OUTS_1 0x29
OUTS_2 0x2a
OUTS_3 0x2b
OUTS_4 0x2c
OUTS_5 0x2d
OUTS_6 0x2e
OUTS_7 0x2f

OUT registers of thread processor.

```
IB_BUFFER_0   0x30
IB_BUFFER_1   0x31
IB_BUFFER_2   0x32
IB_BUFFER_3   0x33
IB_BUFFER_4   0x34
IB_BUFFER_5   0x35
IB_BUFFER_6   0x36
IB_BUFFER_7   0x37
```

32 byte instruction buffer for thread processor.

```
INS_0   0x38
INS_1   0x39
INS_2   0x3a
INS_3   0x3b
INS_4   0x3c
INS_5   0x3d
INS_6   0x3e
INS_7   0x3f
```

IN registers of thread processor.

```
INPUT_REPLY_BUFFER_0   0x40
INPUT_REPLY_BUFFER_1   0x41
INPUT_REPLY_BUFFER_2   0x42
INPUT_REPLY_BUFFER_3   0x43
INPUT_REPLY_BUFFER_4   0x44
INPUT_REPLY_BUFFER_5   0x45
INPUT_REPLY_BUFFER_6   0x46
INPUT_REPLY_BUFFER_7   0x47
```

Space used by inputters to build up reply descriptor prior to placing it on the reply queue.

```
REPLY_BUFFER_0   0x48
REPLY_BUFFER_1   0x49
REPLY_BUFFER_2   0x4a
REPLY_BUFFER_3   0x4b
REPLY_BUFFER_4   0x4c
REPLY_BUFFER_5   0x4d
REPLY_BUFFER_6   0x4e
REPLY_BUFFER_7   0x4f
```

Space for a single reply descriptor. This is used to queue a reply if the reply queue is empty, so reducing MBus memory accesses. An external register location ReplyBufferValid indicates whether this buffer is in use. The buffer is also used during execution of a reply descriptor and so if the reply processor traps the ReplyBufferValid bit must be cleared explicitly by the trap handler.

INPUT0_BUFFER_0	0x50
INPUT0_BUFFER_1	0x51
INPUT0_BUFFER_2	0x52
INPUT0_BUFFER_3	0x53
INPUT0_BUFFER_4	0x54
INPUT0_BUFFER_5	0x55
INPUT0_BUFFER_6	0x56
INPUT0_BUFFER_7	0x57
INPUT0_BUFFER_8	0x58
INPUT0_BUFFER_9	0x59
INPUT0_BUFFER_10	0x5a
INPUT0_BUFFER_11	0x5b
INPUT0_BUFFER_12	0x5c
INPUT0_BUFFER_13	0x5d
INPUT0_BUFFER_14	0x5e
INPUT0_BUFFER_15	0x5f

Input transaction buffer used for Iproc0.

OUTPUT_BUFFER_0	0x60
OUTPUT_BUFFER_1	0x61
OUTPUT_BUFFER_2	0x62
OUTPUT_BUFFER_3	0x63
OUTPUT_BUFFER_4	0x64
OUTPUT_BUFFER_5	0x65
OUTPUT_BUFFER_6	0x66
OUTPUT_BUFFER_7	0x67
OUTPUT_BUFFER_8	0x68
OUTPUT_BUFFER_9	0x69
OUTPUT_BUFFER_10	0x6a
OUTPUT_BUFFER_11	0x6b
OUTPUT_BUFFER_12	0x6c
OUTPUT_BUFFER_13	0x6d
OUTPUT_BUFFER_14	0x6e
OUTPUT_BUFFER_15	0x6f
OUTPUT_BUFFER_16	0x70
OUTPUT_BUFFER_17	0x71
OUTPUT_BUFFER_18	0x72

OUTPUT_BUFFER_19	0x73
OUTPUT_BUFFER_20	0x74
OUTPUT_BUFFER_21	0x75
OUTPUT_BUFFER_22	0x76
OUTPUT_BUFFER_23	0x77
OUTPUT_BUFFER_24	0x78
OUTPUT_BUFFER_25	0x79
OUTPUT_BUFFER_26	0x7a
OUTPUT_BUFFER_27	0x7b
OUTPUT_BUFFER_28	0x7c
OUTPUT_BUFFER_29	0x7d
OUTPUT_BUFFER_30	0x7e
OUTPUT_BUFFER_31	0x7f

Output buffers used by reply, thread and dma processors to formulate packets including route byte headers. Additionally the DMA processor uses the output buffer as a temporary storage during memory memory DMA.

INPUT1_BUFFER_0	0xd0
INPUT1_BUFFER_1	0xd1
INPUT1_BUFFER_2	0xd2
INPUT1_BUFFER_3	0xd3
INPUT1_BUFFER_4	0xd4
INPUT1_BUFFER_5	0xd5
INPUT1_BUFFER_6	0xd6
INPUT1_BUFFER_7	0xd7
INPUT1_BUFFER_8	0xd8
INPUT1_BUFFER_9	0xd9
INPUT1_BUFFER_10	0xda
INPUT1_BUFFER_11	0xdb
INPUT1_BUFFER_12	0xdc
INPUT1_BUFFER_13	0xdd
INPUT1_BUFFER_14	0xde
INPUT1_BUFFER_15	0xdf

Input transaction buffer used for Iproc1.

Elan Elite
Technology

Elite Switch Processor

Reference Manual

meiko

OVERVIEW

1.1 Introduction

The Meiko Elite Packet Switch Chip has eight bi-directional byte wide Elan links, and an 8x8 cross-bar switch with broadcast capability. An Elite routes and checks Elan packets from any input link to one or more output link. An Elite can be intergrated and controlled via an IEEE Standard Test Access Port (TAP). Routing is controlled by protocol sent with the packets. Elite switches within a network attempt to implement a distributed fair round robin arbitration scheme across the whole network, giving all sending processors equal priority to each receiving processor. Each link has a 48 byte flow control FiFo, allowing Elite switches to be connected by up to 25 meters of cable without any loss of bandwidth.

1.2 Major Objectives

The major objectives for the switch processor are:

- To have high link bandwidth. (Total switch bandwidth = 600 Mbytes/sec)
- To pass packets across the switch from link to link with minimum latency.
- To return PACK/PNACKs across the switch from link to link with minimum latency.
- To impliment broadcast communications.
- To impliment a switch network wide fair round robin arbitration scheme.
- To check the integrity of packets transmitted across the switch.
- To handle errors with minimum disruption to other packet traffic on the switch network.

1.3 Elan Links

The links are described in detail appendix A.

8X8 CROSS-BAR SWITCH

2.1 Route byte format

Packets entering a switch chip must start with a route byte. This is used to select an output, or group of outputs, to transmit the packet to. The route byte is the first byte of the packet as seen by the switch chip. The switch chip uses the route byte and then removes the route byte from the packet. If the packet has to travel through a number of switch chips, then the route bytes for each switch chip are placed in the order they will be used as the packet moves from Elite switch to Elite switch. When the packet arrives at the destination processor or processors, all the route byte(s) will have been deleted from the front of the packet. The route byte has four fields as shown below:-

```
RouteByte[2:0]   Lower bound of output ports.
RouteByte[3]     Priority bit. This must be set for
                 broadcast transmissions.
RouteByte[6:4]   Upper bound of output ports.
RouteByte[7]     Odd Parity bit.
```

For normal point to point connections across a switch, both the upper and lower bound values must have the number of the output link. The priority bit is normally reset, and the parity bit is set to odd parity for the route byte. The whole packet will be sent to the addressed output link, and the PAck or PNAck, received at the output link, will be returned back across the switch and sent to the link that received the packet.

e.g.

1 Link 3 receives a packet with a route byte of value 0x91.

2 Both the upper and lower bounds have the value 1.

The packet will be routed to Link1, and the ack received by Link1 will be sent back out of Link3.

For a broadcast connection across the switch the lower bound has the number of the lowest output link and the upper bound has the number of the highest output link. The priority bit must be set, and the parity bit is set to odd parity for the route byte.

e.g.

1 Link 6 receives a packet with a route byte of value 0x49.

2 The upper bound has the value 4.

3 The priority bit is set.

4 The lower bound has the value 1.

The effect is :-

- The packet will be routed and copied to Links 1,2,3 and 4.
- When all the links 1 to 4 have received an ack (PAck or PNAck) then an ack will be sent out of link 6.
- If any of the links 1 to 4 receive a PNAck then a PNAck will be sent from link 6.
- Only if all the links 1 to 4 receive a PAck will a PAck be sent by Link 6.

2.2 *Distributed round robin arbitration*

The Elite switch chip implements a fair 'switch network wide' round robin arbitration scheme. Within each switch the switch output for a link, when a connection is made, round robin arbitrates onto a switch input. The priority of arbitration is as follows: After chip reset switch input 0 is the highest priority and switch input 7 is the lowest priority. So if the first two packets arriving at the switch arrive at the same time and require routing to the same switch output, then the switch input with the lowest number is connected first. On the next arbitration, the switch input that was last connected becomes the lowest priority, and the next switch input (last switch input + 1) becomes the highest priority. The priority increases as the switch input number increases, wrapping from switch input 7 back to switch input 0. A switch output does not always re-arbitrate at the end of a packet. All packets end with one End of Packet (EOP), but only packets ending with two successive EOPs allow a switch output to re-arbitrate. An Elan processor always ends a packet with two EOPs. As the packet moves through the switch network, the second EOP may be deleted by an

Elite switch. This will happen if another switch input is waiting to be connected to the switch output passing the packet. If only one EOP is seen on the end of a packet, then the switch output will not re-arbitrate. In other words, two EOPs means the switch output may re-arbitrate, one EOP means the switch output may not. The second EOP will not be deleted if the arbitration wraps from switch input 7 to switch input 0. e.g. if switch inputs 1, 3, and 5 are all receiving back to back packets destined to the same switch output and each ending with two EOPs, then the order the switch inputs are selected is 1 - 3 - 5 - 1 - 3 - 5 - 1 - 3 - 5... Packets from switch inputs 1 and 3 will have their second EOP deleted, but the packets received at switch input 5 keeps its second EOP as the arbitration at the end of the packet from switch input 5 involves an arbitration back to switch input 1 that wraps from switch input 7 to switch input 0. If a packet is received at a switch input with only one EOP, and the next packet received at the same switch input is not to be routed to the same switch output, then the switch output adds a second EOP to the first packet so that subsequent Elite switches are not locked onto an idle switch input. However, the second EOP is not added if another switch input is waiting to use the switch output following the rules above. The distributed round robin only functions correctly provided all the links involved are trying to communicate to a single Elan processor.

2.3 *Priority arbitration*

An additional round robin arbitor has been included to enable arbitration between incoming broadcast packets. If the priority bit (bit 3 in the route byte) is set then that switch input has priority over normal connections. This mechanism has been introduced to enable broadcast communications to operate correctly. If two broadcast communications arrive simultaneously at two switch inputs and the set of switch outputs that they are to connect to overlaps then it is probable that they will each connect to some, but not all, of their switch outputs and each be left waiting for switch outputs that the other has connected with. In this case the switch will deadlock. In order to prevent this, a priority request round robin arbitor has been added. When the priority bit in the route byte is set then before any requests to connect to any switch outputs is made a priority round robin request is made. When the priority request is acknowledged, all other switch inputs are inhibited from making their requests, and the acknowledged switch input is guaranteed to connect to all the switch outputs after any packets being passed those switch outputs has gone by. Re-arbitration will occur even if the packets being passed only end with one EOP. The priority bit must be set for broadcast communications but must never be set for point to point communications.

2.4 *Broadcast communication*

A switch network made using Elite switch chips and connecting Elan communication processors together can be viewed as a linear array of Elan processors numbered 0 to n connected by a tree of Elite switch chips. Broadcasts are made by sending a packet to the top of the tree and then copying the packet to more than one switch output as the packet moves back down the tree. In this way it arrives at many Elan processors at the bottom of the tree. The broadcast set is contiguous in the range X to Y where 0 <= X <= Y <= n. Each Elite switch has a number of links going up in the network the rest going down towards the Elan processors. The links going down always start with link 0. Broadcasts may only spread out going down the network. Each Elite has a value called its BroadcastTop. This should be set to highest link number going down. So, for example, if links 0 to 3 go down and links 4 to 7 go up, then the BroadcastTop value should be set to 3. While an Elite is in reset the notError pin is tri-state. When an Elite comes out of reset the value on the error pin is sampled. If the value is high then BroadcastTop is set to 3. If the value is low then BroadcastTop is set to 7. The error pin can be pulled by a resistor to the appropriate level. Other values of BroadcastTop can be set by writting to the GlobalReg using the JTag interface. The route bytes required to make a broadcast connection are as follows :-

1 A set of normal point to point route bytes are required to send the packet to the top of the switch network tree.

2 A set of broadcast route bytes back down the tree describe the contiguous set of Elan processors being communicated with.

Each broadcast route byte has a 3 bit upper and a 3 bit lower bound number describing on which links the packet is to leave the Elite. The First broadcast route byte describes which links the packet should leave on the top most Elite switch chip within the switch network. As the network is a tree, and we require a contiguous range of Elan processors to be communicated with, then the middle downward links of this Elite should then pass the packet to every Elan processor on a downward path from these links. For this reason the top Elite modifies all subsequent route bytes for this packet (to be used by all the Elite chips below the top Elite switch chip) to say "output on all downward links". The value chosen for this is "broadcast 7 to 0". Zero is always the lowest downward link from any elite. Seven is the highest possible. When an upper bound of seven is received by an Elite on a broadcast route, then if the BroadcastTop value is not seven, then the higher bound is changed to the BroadcastTop value. The least significant link being broadcast to on the top Elite switch chip only requires the upper bound to be modified on all subsequent route bytes, and the most significant link being broadcast to on the top Elite switch chip only requires the lower bound to be modified on all subsequent route bytes. All other lower Elite switches apply the same rules. Most will receive a route byte with an upper bound of 7 and a lower bound

of 0. These routes cannot be modified further. The Elite switches on the edges of the downward formed broadcast pyramid will have to modify the route bytes leaving them that are no longer on the edge of the pyramid. So the broadcast route bytes sent by the sending Elan processor consist of a list of upper and lower bounds, where the upper bounds describes the most significant edge of the broadcast pyramid, and the lower bounds describes the least significant edge of the broadcast pyramid. As route bytes may be changed then the parity bit may also have to be changed if the new value of route byte requires it. Note, the parity bit is not regenerated, it is only changed. Hence if the route byte was corrupted, then the new value will still say that the route is corrupted.

The sending Elan processor is only allowed to send it's EOP when it receives either an PACK or a PNACK. A receiving Elan processor will only send a PACK when the 'Ack now' bit is set within a transaction. An Elite switch is only allowed to send a PACK or a PNACK when it has received a PACK or PNACK on all of the links it sent the packet out of. A PACK will only be sent back from an Elite switch if a PACK was received on all of the links it sent the packet too. If any of the output links received a PNACK then when all of the other links have received a PACK or PNACK, a PNACK will be sent back to the sending Elan. The packet is held open within an Elite switch until an EOP is received from the sending Elan processor. The full sequence of events for a broadcast communication is as follows :-

1 A set of normal point to point route bytes are sent by the sending Elan processor to take the packet to the top of the switch network tree.

2 A set of broadcast route bytes are sent by the sending Elan processor to take and broadcast the packet back down the tree to all of the Elan processors being communicated with.

3 A Start of packet (SOP) command is sent.

4 One or more transactions are sent. One of them must have the 'Ack now' bit set.

5 When the transaction with the 'Ack now' bit set is received by the Elan processors then they return either a PNACK or PACK.

6 When all the PNACKs and PACKs have been merged and returned back across the switch network a single PNACK or PACK is received by the sending Elan processor.

7 Two EOP commands are sent by the sending Elan processor.

8 As the EOPs move across the Elite switches, the switch connections are disconnected and made available for other communications.

ERRORS, ERRORS HANDLING, AND RESET PROPAGATION

Errors can be observed through an ERROR pin. The precise nature of the error can be found, and if necessary the error may be corrected, using the TAP. When the Elite detects an error, it will close and remove any effected packets, and if possible, will tell the sending and receiving Elan processors that an error has occurred.

Errors on the switch chips fall into three categories

1 Link line protocol errors.

2 Routing and packet protocol errors.

3 Timeout errors.

Each of the eight links has an error flag register. If an error is detected, then the appropriate bit is set in the error flag register. All the error flag bits are ORed together for all the links to form an Elite error value that is output on the _ERROR pin. Once an error bit is set, it can only be reset via the Test Access Port. All error bits are reset by Chip reset.

3.1 *Link line protocol errors*

There are two line protocol errors that can be detected :-

1 Received Phase Error.

2 Received Data Error.

A Received Phase Error occurs when the link receiver was unable to keep phase aligned to incoming link data. This may occur because of :-

1 Excessive frequency drift between the transmitter and the receiver,

2 Excessive line clock jitter, caused by gnd bounce etc.

3 Noise on the line causing multiple clocks to be received.

A Received Data Error occurs when the incoming data is not a valid command value or a valid data value.

When a link line protocol error occurs, both the link transmitter and link receiver are put into reset for at least 256 clock cycles, and any packets being transmitted by the link are correctly terminated and if possible NACKed.

3.2 *Routing and packet protocol errors*

Elan packets consist of route bytes, a SOP, one or more transactions, then an EOP.

The route bytes each have a parity bit. All the route bytes entering an Elite have their parity checked. If any of the route bytes has a parity error then the route parity error bit is set and the packet is terminated. When the packet is terminated a NAck is returned to the sending Elan, the rest of the packet being thrown away, and a Bad EOP (SOP EOP) is sent on to the receiving Elan. This may or may not reach the receiving Elan depending on which Elite switch and in which route byte the error occurred.

Each transaction within a packet ends with a two byte CCITT CRC polynomial. This is used to check the integrity of the transaction. As a packet moves through an Elite, all the transactions are checked. The transaction CRC error bit will be set if either the CRC value does not match the transaction, or the transaction ends prematurely with either an EOP or a Bad EOP. The size of a transaction is defined in bits six to four of the first byte of the transaction.

First Byte [6:4]	Trans size in words	Total in bytes (Inc crc)
0	2	10
1	4	18
2	6	26
3	10	42
4	18	74
5	34	138
6	66	266
7	130	522

Using the TAP the setting of the CRC error bit may be disabled if the Elite is to be used for switching non standard Elan packets. Current Elan processors will only generate transactions of 10, 18, 26 or 42 bytes in size. If an error is detected and enabled then the rest of the packet sent to the Elite is consumed, a Bad EOP is sent on from the

Elite, and if a PACK or PNACK has not already been returned a PNACK is returned to the sending Elan processor. If the error is detected on the last transaction within the packet and the EOP of the packet immediately follows the last transaction, then it is possible for the packet to leave the Elite before a Bad EOP can be placed on the end of the packet. However the transaction CRC error bit for that link input will still be set.

3.3 *Timeout errors*

There are two sources of timeout error for each link. They are :-

1 Packet open timeout.

2 Packet waiting timeout.

An Elite switch input is considered open from the time that the first byte of a packet moves across the switch to the time that switch is able to re-arbitrate to a new input. If this time exceeds 240 to 480ms for a 70MHz switch network then the switch will disconnect and reset the link connected to the output of the switch. It will also set the Input Open Timeout error bit for the link connected to the input of the switch.

If, when a packet arrives at a switch input, the switch is unable to connect to the desired switch output because another switch input is connected to that switch output, then that input is considered to be waiting. Each link input has a two bit Waiting timeout control register. This register controls the behaviour of the switch input if the input is waiting for longer than the waiting timeout value. The register is accessed using the TAP and is defined below :-

1 Bit 0 Enable automatic Nack and gobble.

2 Bit 1 Enable setting of input open timeout error bit.

When bit 0 is set, then if the waiting timeout value is exceeded, the whole packet is consumed and a NACK is returned to the sending Elan processor. Nothing is sent on as the switch did not connect to any switch output. When bit 1 is set, then if the waiting timeout value is exceeded, the Input Open Timeout error bit for the link connected to the switch input is set. If bit 1 is set on its own, then the only action is to set the error bit. i.e. the packet is not terminated, and if the switch output that the input is waiting for becomes free, the packet is transmitted as normal. On chip reset both bits are cleared.

The waiting timeout value is set by loading the appropriate three bit value into the TAP global register. The waiting timeout values for an Elite with a 70MHz comms clock is given below :-

Global reg[2:0]	Timeout period
0	22-29us (Reset value)
1	44-58us
2	88-117uS
3	176-234uS
4	352-468uS
5	704-936uS
6	1408-1873us
7	2816-3745uS

3.4 *Reset Propagation*

If the output link on an Elan processor has been open for more than the timeout duration, then the Elan must clear the packet being communicated. This may have occurred for one of the following reasons :-

1. One or more of the transactions in the packet where invalid.
2. No 'Ack now' bit had been set in any of the packets transactions.
3. The routes used to send the packet where not valid.
4. Part of the network to be used by the packet is broken.
5. The PACK or PNACK was lost by a bad transmission.
6. The inputer on the receiving Elan processor has been disabled.
7. Erronious code on the sending Elan processor has not 'Closed' the packet.
8. Congestion of the switch network prevented the packet from reaching the receiving Elan processor within the timeout period.

For any of the above reasons, and possibly other reasons, the packet must be capable of being cleared. The whole packet could be stored in the flow control FiFos of each of the links used by the packet. When the Elan outputer times out, it puts its outputing link into reset. This causes the RESET command to be sent down link. If an Elite switch chip receives a RESET command, and is connected to one or more output links, then the reset value is propagated across the switch and causes all of the output links connected to the inputing link to put their transmitters into reset and transmit a single RESET command. The output will clear its output byte count and then ignore any TOKEN commands it receieves for the next 64 to 128 cycles. The output will continue to supply GAP commands for line syncronization and also will continue to return PACK and PNACK commands for any packets being sent down the link in the

other direction. If an elite receives a single RESET command then it clears its input fifo and will send on a reset to any outputs that the input is connected to. The input will then wait 256 to 512 cycles before returning and TOKENS back to the outputer. In this way the whole packet is cleared from the switch network. If the reset reaches the receiving Elan processor, then it is interpreted as a BAD EOP by the inputer of the Elan processor.

The RESET command is only transmitted in one direction across a link. Any packets being transmitted in the other direction are not affected. If a packet times out, the whole of that packet is cleared from the switch network without affecting other packets apart from blocking other packets trying to use a link in the same direction as the timed out packet.

If a link is put into reset by either a hard data or phase error, or by the jtag or chip reset pin, then both the sender and receiver are put into reset and a timeout is sent in both directions.

3.5 *Disabled Links*

A link on an Elite switch chip can be disabled by sending a command on the TAP. When a link is disabled, the link is put into reset and outputs a RESET command. Any packet that arrives to be transmitted out of the link from the switch is consumed and PACKed. For most errors it is normal to PNACK and consume a packet, but the disabled link PACKs. This is necessary if portions of the switch network are to be partitioned off and broadcast communications are still able to operate around the partitioned area.

TEST ACCESS PORT. (TAP)

The Elite switch chip is control by and conforms to the IEEE Standard Test Access Port (IEEE Std 1149.1-1990). The TAP has the full five pin interface of :-

1	TCK	Test Clock Input.
2	TMS	Test Mode Select Input.
3	TDI	Test Data Input.
4	TDO	Test Data Output.
5	TRST*	Test Reset Input.

The instruction register is six bits wide. All the mandatory instructions BYPASS, SAMPLE/PRELOAD, and EXTEST are included. The optional IDCODE instruction is also included. Private instructions have been included that perform the following functions :-

1 Global register access.

2 Timer register access.

3 Individual link error value register access.

4 Individual link error bits register access.

5 Individual link reset register access.

6 Individual link timeout control register access.

7 Equivalent instructions to SAMPLE/PRELOAD, and EXTEST for each of the eight links.

8 Eight instructions to read link control state.

The full list of instructions is given in appendix B. The full list of register bit allocation and meanings is given in appendix C.

MEIKO BYTE-WIDE LINK LINE-PROTOCOL

A.1 Link Connection

The basic characteristics of Meiko links is that they are; byte wide; bidirectional; point to point; and high bandwidth (>50 Mbyte/s each direction). Each link consists of 20 wires; 10 for the input port, and 10 for the output port. Each port has one clock wire and nine data lines. On both positive and negative transitions of the ChanClkIn wire the ChanIn wires are sampled. The output port sets up a new data pattern on ChanOut at the start of each communications clock period, and toggles ChanClkout in the middle of each period.

```
output port          input port

ChanClkOut           ChanClkIn
ChanOut[8]           ChanIn[8]
...                  ...
...                  ...
ChanOut[0]           ChanIn[0]
```

A.2 Link Values Encoding

The line protocol has eight command values, as well 256 data values encoded. No single bit error can change data into a command or visa-versa. No single bit error can change one command into another.

The commands are as follows:

Command	Code	Usage
NULL	{3'h7,3'h0,3'h0}	Nothing to be sent.
GAP	{3'h7,3'h1,3'h1}	Used for bit stuffing to get receiver in sync with sender.
SOP	{3'h7,3'h2,3'h2}	Start of packet.

```
EOP     {3'h7,3'h3,3'h3}  End of Packet.
TOKEN   {3'h7,3'h4,3'h4}  Receiver can accept 16 more
                          bytes of data.
PNACK   {3'h7,3'h5,3'h5}  Packet Not Acknowledge.
PACK    {3'h7,3'h6,3'h6}  Packet Acknowledge.
RESET   {3'h7,3'h7,3'h7}  Sender is in reset.
```

The order of priority of sending commands and data is shown below:

```
Highest priority                              Lowest priority

RESET   PACK    TOKEN   EOP     Data    GAP     NULL
        PNACK           SOP
```

The outputer attempts to output a GAP every 256 cycles. If having waited 128 cycles a GAP has still not been sent because the line has been continuously busy, then a GAP is sent in preference to data. When a GAP command is transmitted, it must be followed by a NULL command. The NULL following a GAP is higher priority than everything except the RESET command.

The data bytes are encoded in four ranges, as follows:

```
8'h00 - 8'h3f have the value {3'b000, Data[5:0]}
8'h40 - 8'h7f have the value {3'b001, Data[5:0]}
8'h80 - 8'hbf have the value {3'b010, Data[5:0]}
8'hc0 - 8'hff have the value {3'b100, Data[5:0]}
```

At least two of the top bits would have to change before the data byte could possibly be interpreted as a command byte. Errors which change the data values into other data values must be detected by error checking the packet contents at the packets destination.

Packets are made up of route bytes, a SOP, one or more transactions, and one or two EOPs. One PACK or PNACK is returned for each packet sent. The line protocol does not distinguish in any way between packets which have been PACK or PNACKed. If a packet is terminated prematurely by a line data error, the packet is terminated with an SOP EOP. This signals to the inputer that the packet was incomplete.

RESET, TOKEN, GAP and NULL are only used by point to point links and are invisible to higher levels of protocol.

A.3 *Flow Control*

The input port has a FiFo. Data bytes and EOP commands can be stored in the FiFo while a switch or inputer is unable to take the data. The protocol allows the FiFo to be as deep as necessary to take up the delays in the line, but in the first implementations of links it is intended that the FiFo be 48 bytes deep. The size of the byte count register must be sufficient to cope with the largest FiFo it can be connected to, which may be greater than its own FiFo size. In the initial implementation this will be 8 bits, allowing FiFos up to 256 bytes to be connected.

Any time an input port has 16 or more bytes of space in its FiFo, it instructs its output port to send a TOKEN command and decrements its space available count by 16. This effectively transfers the ownership of 16 bytes of FiFo space from the inputer to the outputer connected to it. Each byte consumed by the inputer frees up one byte of space in the FiFo. Note that the effective size of the FiFo is reduced by between 0 and 15 bytes at any point in time because of the 16 byte granularity of the token.

When an inputer receives a TOKEN command it instructs its paired output port to increment the count of the number of bytes it may send by 16. Each time the output port sends a byte it decrements this count by one. As long as the count is greater than zero the output port is allowed to transmit data, or commands that consume FiFo space, (EOP or SOP).

Data is transmitted as packets. All packets must end with an EOP command. All packets must be either PACKed (Packet Acknowledge), or PNACKed (Packet NOT Acknowledge). Acknowledgements are passed back along the route that the packet took. If the outputing processor traps while it has its output open it will immediately send an EOP. An EOP generated in this way is termed an unsolicited EOP. If an Elite switch chip detects an error, then it will generate a BAD EOP command (this is an SOP immidately followed by an EOP command). A BAD EOP can be generated before a PACK/PNACK and hence be interprited as an unsolicited EOP. The system must ensure that any PACK or PNACK being returned for that packet is not interpreted as being for a following packet. To ensure this, unsolicited EOP commands are handshaken in the following way. The EOP command can be issued before a PACK/PNACK has been received, but another packet cannot be transmitted along the line before the PACK/PNACK is received. The line is kept open for a packet until both the EOP is sent and the PACK/PNACK is received. If a receiver port receives an EOP before the transmitter has sent a PACK/PNACK then a PNACK is automatically sent by the transmitter. Any PACK or PNACK commands received after an EOP has gone by, and before another packet has started, are deleted.

A.4 *Links and Reset*

A link is held in reset when:

1. The Reset pin of the chip is high.
2. A JTag port holds the link in reset.
3. The input port is not receiving a clock from the line.
4. The input port is receiving RESET commands from the line.
5. The outputer has been disabled by a JTag port.

A link is put into reset for at least 256 clock cycles when :-

1. The value clocked in on ChanIn is neither a command nor Data.
2. The inputer has a phase error. (Due to excessive drift, jitter or double clocking on the ChanInClk)
3. The link needs to be cleared because of a timeout.
4. A JTag port clears the link.

When a link is put into reset the link has a defined state. This is :-

1. No packets are being sent in either direction.
2. No PACK/PNACK is outstanding.
3. The flow control FiFo is empty, but the receiver owns all the space in the FiFo. i.e. TOKEN commands must be sent before any data can be received.
4. The transmitters count of bytes it may send is set to zero.
5. Any packets being sent when the link was put into reset are completely consumed, and if possible, NACKed.
6. Any packets being received when the link was put into reset are ended with a BAD EOP (SOP, EOP);

7 The link is forced to output the RESET command, except if the link is reset by receiving a RESET command. If the link is receiving the RESET command then the transmitter sends the NULL command.

If an outputer is in the midst of sending a packet when it is put into reset, the remainder of the packet is consumed but not transmitted, and a PNACK returned to the sender. The outputer will consume the rest of the packet up to the EOP, even if the link is taken out of reset. New packets arriving at a link which is in reset are held until the link is taken out of reset.

If an inputer is receiving a packet when it is reset, it forwards a SOP, followed by an EOP. If this occurs while route information is being sent this forces the message to terminate. If the error occurs during the data part of the packet the SOP is passed in to the inputing communications processor, which detects it as an error which does not require acknowledgement.

This mechanism insures that reset can be forced at any time on a port in a way which can be detected and recovered from by all devices (either processors or switches) using that port. The reset mechanism will always reset the link in both directions; this is essential as no direction can be deduced from an erronious command or data item. Reset is however only sent in one direction across the link, form the end that detected the error. Resets are propogated along the currently connected routes so that an entire blocked packet is flushed out.

A.5 *Clock Skew Tolerance*

The output port generates both the data and the Clock. Any variation in voltage, temperature, or process, should not cause the skew between the clock and data, as seen by the receiver, to vary. The data is clocked on both positive and negative edges of the clock. Therefore the maximum frequency of any pin is half the peak data rate.

A.6 *Clock Phase Locking and Control*

Both directions of a link must transmit at the same frequency. To avoid having to distribute a global clock, marginal (<200ppm) frequency variations are permitted. Receivers use the same frequency as their transmitters, and so have an unknown, and slowly changing phase difference with respect to the data coming in on the line. Inputers can correct for this by inserting or removing NULL commands. The points at which corrections can be made are signaled by GAP commands. GAP commands must be sent often enough to insure that the maximum frequency drift is always compensated for before it can cause phase errors.

Each receiver has a phase detecting circuit and a short FiFo. Data is clocked into the FiFo using the clock sent with the data. The data is clocked out of the FiFo using the receivers local clock. The FiFo is three entries deep. The minimum possible latency through the FiFo is zero cycles, and the maximum possible latency through the FiFo is three cycles. The receiver monitors the latency through the FiFo, and tries to maintain a 1 to 2 cycle latency. At regular intervals, the sender transmits a GAP command. The GAP command is always followed by a NULL command. When the receiver receives a GAP then, if the measured latency is greater than two, because of clock drift or transmission delay drift, then in one cycle the receiver can remove both the GAP and the NULL from the FiFo. This will reduce the latency though the FiFo by one cycle. If, when the receiver receives a GAP, the measured latency is less than one cycle, then the receiver does not take anything out of the FiFo for one cycle. This will increase the latency by one cycle.

A.7 *Automatic Link Output Tri-state*

The link has an automatic link output tri-state function. This is included to enable hot insertion of circuit boards within a switch network. When a link is operating normally, it will be receiving an edge on the ChanClkIn pin every clock cycle. If the link is disconnected then the ChanClkIn will stop oscillating. A very weak pull down resistor on the ChanClkIn pin will pull the input to gnd. If the ChanClkIn pin is read as zero without being read as a one for at least 256 Comms cycles, then all the output pins of the link out will be tri-stated. The ChanClkOut pin has a weak (but not very weak) pull up resistor. So if a board is re-inserted, and the link connection made again, when the power is restored the ChanClkIn pins of both ends of the link will be read as a one (because the weak pull up resistor wins over the very weak pull down resistor). When the ChanClkIn pin has been read as a one for more than 10ms the link output pads will be taken out of tri-state. While a link is in tri-state, the link is held in reset. Links will be automatically tri-state and untri-state regardless of the state of Chip Reset. The only exception is if the link is being boundary scanned using the TAP. In this case the link will be forced out of tri-state.

ELITE TEST ACCESS PORT INSTRUCTIONS

The Test Access Port instruction register is six bits long. During the Capture-IR state the instruction register is loaded with the following.

```
{2'b00, ChipReset, ErrorFlag, 2'b01}
```

The following is a list all the instructions with the hex code and number of data bits.

Hex Code	No of Data bits	Instruction Name	Extra Function
00	164	ExTest	Also resets all links
01	164	SamplePreload	
02	11	ReadAndWrtGlobalControl	
03	11	ReadGlobalControl	
04	24	ReadAndWrtTimeoutReg	
05	24	ReadTimeoutReg	
06	72	ReadErrorValRegs	
07	72	ReadErrorValRegs	
08	16	ReadAndWrtResetControl	
09	16	ReadResetControl	
0a	16	ReadAndWrtTimeoutControl	
0b	16	ReadTimeoutControl	
0c not used			
0d not used			
0e	40	ReadAndClearErrors	Clear individual error bits
0f	40	ReadErrors	Read all switch error bits
10	20	ExTestLink0	Also resets link 0
11	20	SamplePreloadLink0	
12	20	ExTestLink1	Also resets link 1
13	20	SamplePreloadLink1	

14	20	ExTestLink2	Also resets link 2
15	20	SamplePreloadLink2	
16	20	ExTestLink3	Also resets link 3
17	20	SamplePreloadLink3	
18	20	ExTestLink4	Also resets link 4
19	20	SamplePreloadLink4	
1a	20	ExTestLink5	Also resets link 5
1b	20	SamplePreloadLink5	
1c	20	ExTestLink6	Also resets link 6
1d	20	SamplePreloadLink6	
1e	20	ExTestLink7	Also resets link 7
1f	20	SamplePreloadLink7	
20		Bypass	
21	15	ReadStateLink0	
22		Bypass	
23	15	ReadStateLink1	
24		Bypass	
25	15	ReadStateLink2	
26		Bypass	
27	15	ReadStateLink3	
28		Bypass	
29	15	ReadStateLink4	
2a		Bypass	
2b	15	ReadStateLink5	
2c		Bypass	
2d	15	ReadStateLink6	
2e		Bypass	
2f	15	ReadStateLink7	
30-3c		Bypass	
3d	6	IDCode	
3e		Bypass	
3f	1	Bypass	

ELITE TEST ACCESS PORT REGISTERS

The following is a list of Elite internal registers that can be accessed using the Test Access Port. The registers are not all the same length.

C.1 *External scan shift path.*

The following is used by the instructions SAMPLE/PRELOAD, and EXTEST.

Whole scan path is:-

```
{Link7Scan[19:0], Link6Scan[19:0], Link5Scan[19:0],
 Link4Scan[19:0], Link3Scan[19:0], _ERROR, _RESET,
 Link2Scan[19:0], Link1Scan[19:0], COMMSCLK, _COMMSCLK,
 Link0Scan[19:0]}
```

Each link scan bit order is:-

```
{LinkIn8, LinkIn7, LinkIn6, LinkIn5, LinkIn4,
 LinkInClk, LinkIn3, LinkIn2, LinkIn1, LinkIn0,
 LinkOut0, LinkOut1, LinkOut2, LinkOut3, LinkOut4,
 LinkOutClk, LinkOut5, LinkOut6, LinkOut7, LinkOut8}
```

C.2 *Link Switch State.*

Eight instructions exist to access read only state within each link. One instruction per link. Each register is 15 bits long. The bits are defined below :-

Bit(s)	
6:0	RouteByte. Valid if waiting for connection.
7	Switch input is connected and transmitting packet to one or more outputs.
8	Switch input is waiting to be connected.
9	Switch output is connected to a switch input.
10	Switch output FirstEop. Waiting for next byte to decide if the output should rearbitrate to a new switch input.
11	Receiver flow control fifo is not empty
12	No more fifo space for link transmitter to send data to.
13	Link output pads tri-state for power up.
14	Link is in reset

C.3 *Link Reset Control.*

This is a 16 bit read/write register that is used to give individual link reset control. Two bits per link. The coding is as follows :-

Value	Meaning
0	Normal operation
1	Reset Switch input. i.e. Send reset on to all outputs connected to input, NAck and gobble the rest of the packet.
2	Disable switch output. i.e. PAck and gobble all packets.
3	Hold link in reset

The bit ordering of the whole register is :-

```
{RC7[1:0], RC6[1:0], RC5[1:0], RC4[1:0],
 RC3[1:0], RC2[1:0], RC1[1:0], RC0[1:0]}
```

C.4 *Waiting timeout control.*

This register is used to give individual link control to the behaviour of a switch input that has been waiting to connect to an output for more than the waiting timeout value defined in the Global register. The waiting timeout is a 16 bit read/write register with two bits per link. The meaning of each bit is given below :-

```
Bit 0   Enable automatic Nack and gobble.
Bit 1   Enable setting of input open timeout error bit.
```

The bit ordering of the whole register is :-

```
{TC7[1:0], TC6[1:0], TC5[1:0], TC4[1:0],
 TC3[1:0], TC2[1:0], TC1[1:0], TC0[1:0]}
```

The whole register is zeroed on chip reset.

C.5 *Error value register.*

This is a 72 bit read only register. It is split into 8 * 9 bit registers, one 9 bit register per link. Each register holds the value of the last data error received by the link. The bit ordering of the whole register is :-

```
{EV7[8:0], EV6[8:0], EV5[8:0], EV4[8:0],
 EV3[8:0], EV2[8:0], EV1[8:0], EV0[8:0]}
```

C.6 *Error Flag Register*

This is a 40 bit read / bit clearable register. There are five bits per link. The error bits can be read. They are set to 1 if the error occurs, and are cleared by writing a 1 to the corresponding bit position from the TAP. Writing a 0 to a bit from the TAP causes the value not to change. All error bits are cleared by chip reset. The order of the error bits for each link is given below :-

```
Bit 0   Route Parity Error.  Set by route parity error.
Bit 1   Transaction CRC error. Set by an invalid
        transaction.
Bit 2   Received Phase Error. Set if input was unable
        to keep phase aligned to incoming link data.
Bit 3   Received Data Error. Set if received data was
        not a valid data byte or command.
Bit 4   Input Open Timeout. Set if packet was connected
        to an output for too long, or if enabled the
        packet was waiting to be connected for too
```

long.

Whole reg is:-

```
{Lnk7EF[4:0], Lnk6EF[4:0], Lnk5EF[4:0], Lnk4EF[4:0],
 Lnk3EF[4:0], Lnk2EF[4:0], Lnk1EF[4:0], Lnk0EF[4:0]}
```

C.7 *Global control register.*

This is an 11 bit read/write register. It is divided into 6 fields as follows:-

```
Bit(s)
2:0     Waiting Timeout Control value
 0            22-29us (Reset value)
 1            44-58us
 2            88-117uS
 3            176-234uS
 4            352-468uS
 5            704-936uS
 6            1408-1873us
 7            2816-3745uS

4:3     Perf meter control.
 0      Perf meter is disabled,
 1      Perf meter is testing nand gates,
 2      Perf meter is testing nor gates,
 3      Perf meter is testing inv gates and track load,

5       Mux perf meter output onto the error pin.

6       Disable transaction checking. When set,
        transaction CRC error bits will not be set.

9:7     BroadcastTop value.

10      ChipReset. Ored with the _RESET pin to form
        the chip reset value.
```

C.8 *Perf meter Count register and timeout register.*

This is a 24 bit read/write incrementing register. Under normal operation it is incremented every 256 Comms cycles. It is used to provide timeout periods for both open and waiting packets. It is also used to control the duration of tri-state after a link is reconnected. If the Perf meter control field of the global register is non-zero, then it is incremented by the Perf meter output.