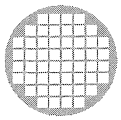


IMS M212 disk processor

The IMS M212 is the first device in a range of intelligent peripherals in the transputer family. The device contains hardware and interface logic to control disk drives as well as a 16-bit processor with on-chip memory and communication capability, and is consistent with the INMOS transputer architecture described in the transputer architecture manual.

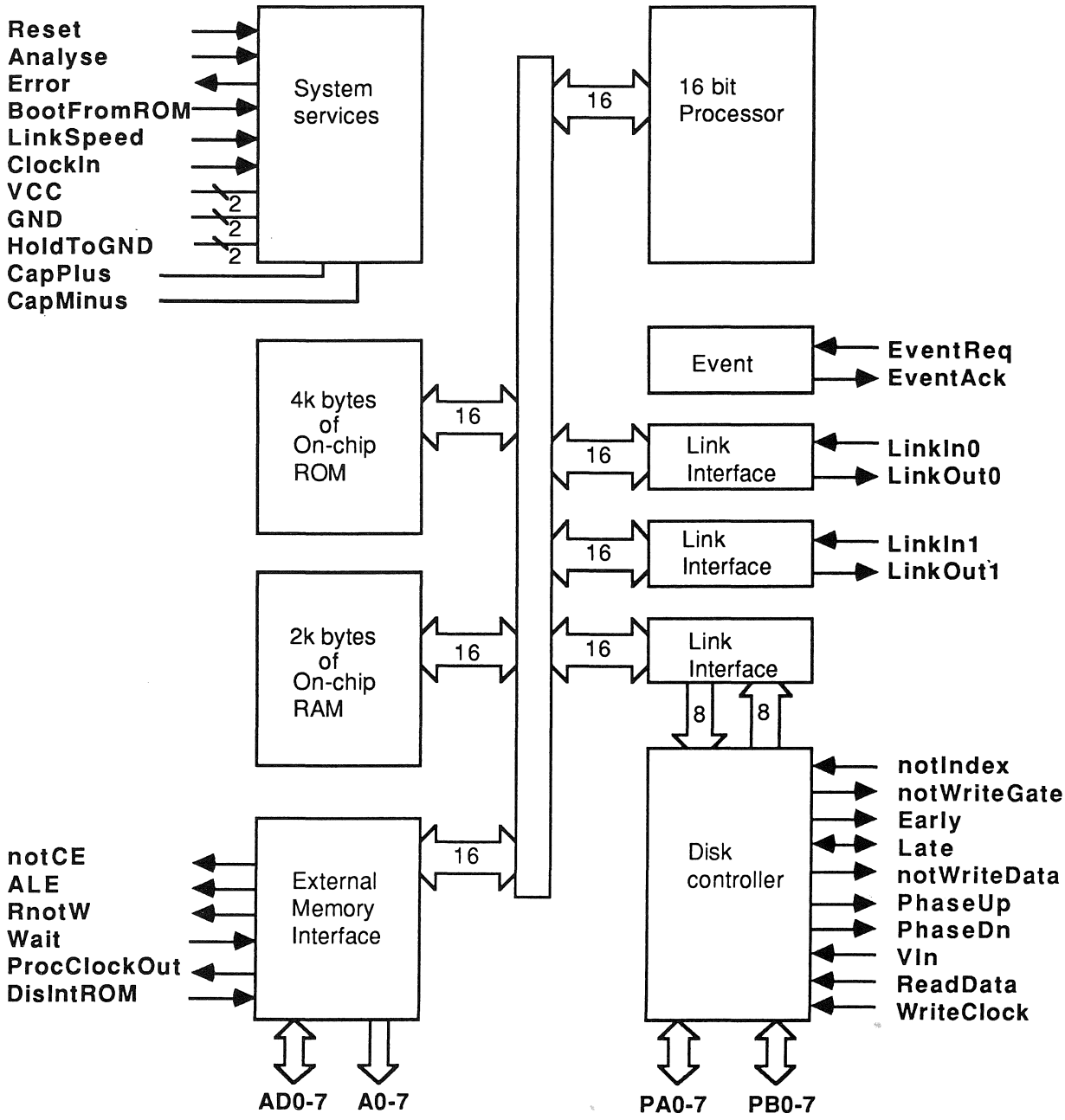
This manual details the product specific aspects of the IMS M212 and contains data relevant to the engineering and programming of the device.

Other information relevant to transputer products is contained in the occam programing manual (supplied with INMOS software products and available as a separate publication), and the transputer development system manual (supplied with the development system).

The examples given in this manual are outline design studies and are included to illustrate various ways in which transputers can be used. The examples are not intended to provide accurate application designs.

The following sections assume a familiarity with the arrangement and operation of disk drives and disk drive control and an appendix (A) has been provided for the reader requiring a brief overview of these topics.

This edition of the manual is dated September 4, 1987.


Reset
Analyse
Error
BootFromROM
LinkSpeed
ClockIn
VCC
GND
HoldToGND
CapPlus
CapMinus
2
2
2
System services
16 bit Processor
16
4k bytes of On-chip ROM
2k bytes of On-chip RAM
16
16
16
16
16
Event
EventReq
EventAck
Link Interface
LinkIn0
LinkOut0
Link Interface
LinkIn1
LinkOut1
Link Interface
8
8
Disk controller
notIndex
notWriteGate
Early
Late
notWriteData
PhaseUp
PhaseDn
VIn
ReadData
WriteClock
notCE
ALE
RnotW
Wait
ProcClockOut
DisIntROM
External Memory Interface
AD0-7
A0-7
PA0-7
PB0-7

IMS M212

The IMS M212 disk processor is a transputer optimised for connection to soft-sectored winchester and floppy disk drives. The block diagram on the previous page outlines the major functional units of the device.

A high performance 16 bit processor provides overall control and satisfies demand for increasing intelligence in disk controllers. The processor also maintains a high level of flexibility allowing the system designer the possibility of modifying the controller function without hardware modifications.

The disk control function has been designed to provide easy connection, with minimal external hardware, to a standard winchester and/or floppy interface. Two byte-wide programmable bidirectional ports are provided to control and monitor disk functions such as head position, drive selection and disk status. A dedicated port is provided for serial data interfaces and critical timing signals. The disk control logic is controlled by the processor via on-chip hard channels using simple command sequences and appears to the processor as a standard INMOS link.

The host interface of the M212 is via two INMOS standard links which provides simple connection to any transputer based system or, via a link adaptor, to a conventional microprocessor system.

The 2 Kbyte on-chip RAM can be used for program or data storage, as a sector buffer or to store parameter and format information. The memory can be extended off chip up to the full 64 Kbyte address range using the external memory interface.

In order to facilitate initial system development and to provide a low part count system an on-chip ROM is provided which contains software, written in occam, which enables the IMS M212 to interface directly to winchester and/or floppy disk drives with no additional software development and the minimum of external hardware. This mode of operation is called 'mode 1' and the details of the controlling process are given in section 2 of this document. When used in this mode a set of commands are available to allow basic disk operations such as Format and Read Sector. This mode also allows up to four disks to be controlled with each disk having different attributes which are automatically retrieved when a disk is selected. When used in this mode no external memory is required as the program is contained in the on-chip ROM and all the necessary workspace and data storage is contained in the on-chip RAM. See section 9 for circuit details of the basic system. Also included in this mode is an automatic boot from disk facility which is in addition to the normal boot from link and boot from ROM operations.

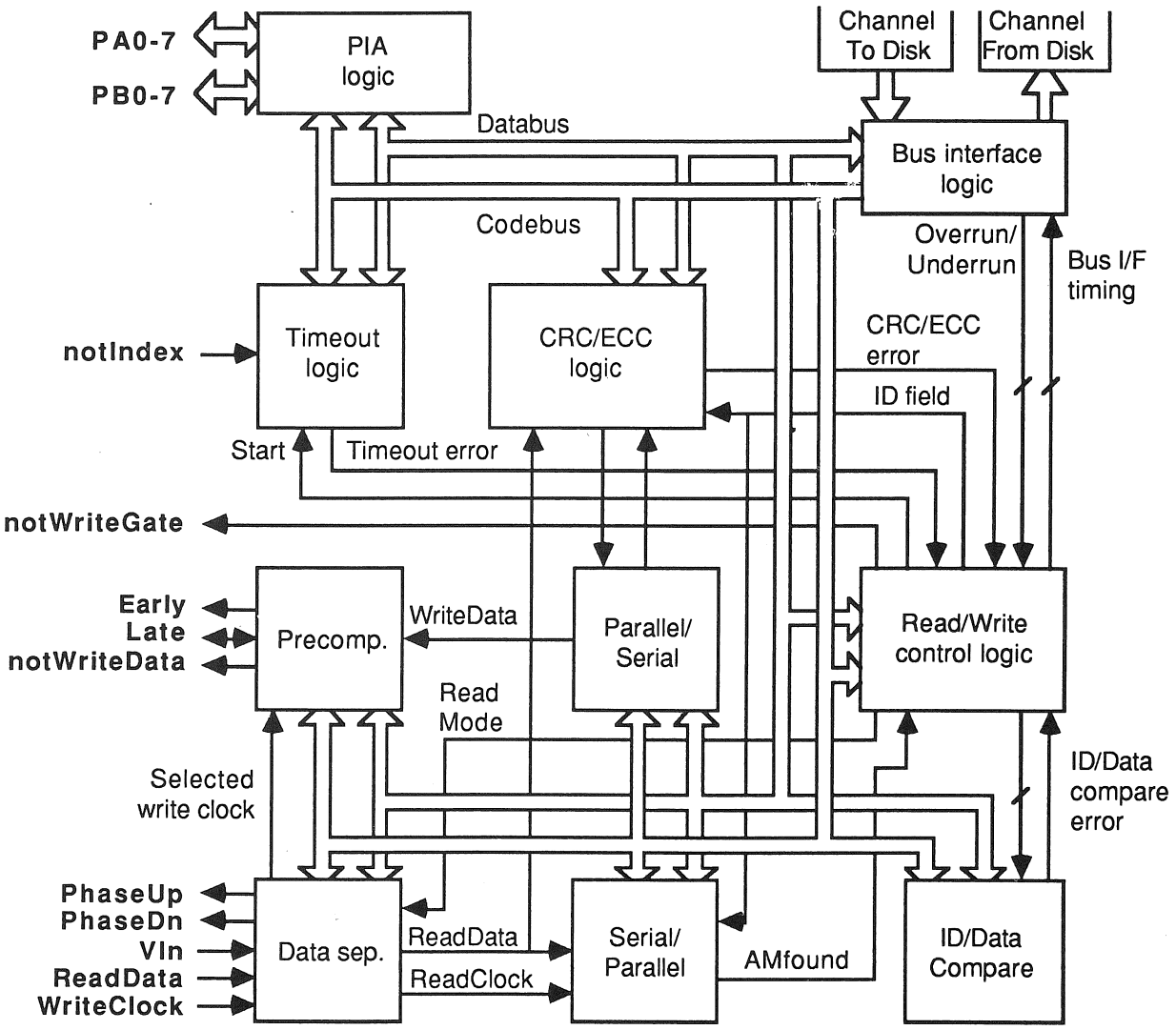
Additional software may be provided by the system designer to enhance the functionality of the IMS M212. This mode of operation is called 'mode 2' and functional details of the on-chip disk hardware and the command sequences used to communicate with it are provided in section 3. This mode would typically be used to enhance the basic procedures of 'mode 1' at the cost of an external RAM or ROM. Typical examples would be the inclusion of disk access sorting procedures to improve the random access time of the disk by minimising the number of head movements or comparing disk data against specified data (disk searching) which will avoid transferring large quantities of data to the host.

When used in 'mode 2' the programmable ports are under the control of the processor enabling the IMS M212 to be used for a wide range of peripheral functions as well as providing a disk interface.

Disk interface

The disk interface has been designed to provide simple interconnection to **ST506/ST412** and **SA400** compatible disk drives.

The IMS M212 has been arranged so that as many disk operations as possible are controlled by the processor and so that the interface to the disk control logic is via a pair of occam channels. A block diagram of the disk logic is shown below and a full description of the functional units can be found in section 3.



Information from the processor is input to the disk hardware via the bytewide 'Channel To Disk'. This information consists of control and data information in specified sequences. The Bus Interface logic interprets this information and outputs the results onto the byte-wide 'Codebus' and 'Databus'. These signals connect to all the other major modules in the disk hardware. Each of these modules contain registers which control the overall function of the hardware. The registers can be programmed and interrogated through the 'ChannelToDisk' and 'ChannelFromDisk' and are summarised in Appendix B.

When writing data to the disk the hardware serialises the data and then encodes it into a Frequency Modulated (FM) or Modified Frequency Modulated (MFM) data stream and any necessary precompensation is performed before outputting the data together with the necessary control signals. Any necessary modification of the data, for instance writing the Address Marks (AM) or inserting the Cyclic Redundancy Check / Error Correction Code (CRC/ECC) bytes, is automatically performed by the hardware.

When reading data from the disk the raw read data is input and the function known as data separation is performed internally. The hardware examines the data stream for an address mark to achieve byte synchronisation and then searches for the desired sector information. When the required data is located it is decoded and a serial to parallel conversion is performed. The data is then transferred to the processor via the bytewide 'Channel From Disk'.

Processor

The 16 bit processor used in the IMS M212 is the same as that used in the IMS T212. This processor achieves compact programs, efficient high level language implementation and provides direct support for the occam model of concurrency. Procedure calls and process switching are sub-microsecond and high performance arithmetic is supported.

Links

The IMS M212 uses a DMA block transfer mechanism to transfer messages between its memory and other devices via two INMOS links. The link interfaces and the processor all operate concurrently, allowing processing to continue while data is being transferred over the links.

Memory

2 Kbytes of single cycle static RAM are available on-chip providing a maximum data rate of 40 Mbytes/s accessible by both the processor and links. 4 Kbytes of on-chip three cycle ROM is also provided which contains software to enable the IMS M212 to be used as a stand alone disk processor.

External Memory interface

The IMS M212 can directly access a linear address space up to 64 Kbytes, with a maximum data rate of 8 Mbytes/s across the external memory interface. 16 bits of address and 8 bits of data are multiplexed to provide a simple interface to byte organized fast static RAM. A wait pin is provided to allow access to slower memory.

Peripheral interface

Links may be interfaced to byte wide peripherals via an INMOS link adaptor. Peripherals can request attention via an event input. The external disk control ports can also be programmed to interface to peripherals as these pins can be controlled by the processor. Alternatively peripherals can be memory mapped into the external memory address space.

Time

The processor provides timers for an unlimited number of high and low priority processes.

Error handling

High-level language execution is made secure with array bounds checking, arithmetic overflow detection etc. A flag is set when an error is detected. The error can be handled internally by software or externally by sensing the **Error** pin. System state is preserved for subsequent analysis.

2.1 Overview

Mode 1 operation uses the code in on-chip ROM to control the disk controller hardware, allowing winchester and floppy disks to be used with little knowledge of the hardware being required. The programming interface to all drive types is identical and there is sufficient flexibility to allow a wide variety of formats and drive types to be used.

Both ST506/412 compatible winchester and SA400/450 compatible floppy drives are supported with standard double density formats. This includes common 5.25 and 3.5 inch drives. Up to 4096 cylinders are allowed and floppy drives can have up to 8 heads while winchesters, which do not require the **MotorOn** signal, can have up to 16 heads. There can be between 1 and 256 sectors per track with sector sizes of 128 to 16384 bytes (in powers of 2). Drives with or without seek complete and ready lines are supported, as are step rates between 64us to 16ms. A range of non-standard formats can also be set up for user-specific requirements.

In mode 1 there is a separate data area in on-chip RAM for each of the four possible drives which contains all of the required control information about that drive. The individual bytes of control information are called parameters and all the parameters for a given drive are kept in a block called a parameter file. The individual parameters may be read from or written to and contain such information as the capacity of the disk, the current position of the heads, the desired sector for reading or writing and drive type and timing details. See section 2.3 for further information on parameters.

Command **BYTE**s and data **BYTE**s are accepted down either of the M212's links and any results are returned down the same link. Examples of commands available are **FormatTrack**, **ReadSector**, **ReadBuffer** and **WriteParameter**. In order to prevent commands from different links from interfering with each other when a command has been accepted from a link then the other link is ignored until an **EndOfSequence** command is received. This stops, for example, one link from trying to use the same area of sector buffer as the other while there is still valid data in it.

Commands which access the disk will implicitly select the drive, perform a seek and select the head. If an ECC or CRC error is found during a **ReadSector** command then a programmable number of automatic retries are performed and a subsequent correction attempted if possible.

The on-chip RAM is used as general workspace for the mode 1 process and to provide 1280 bytes of sector buffer. The process will automatically use any contiguous external RAM which it finds immediately past the internal RAM (i.e. from address #8800 upwards) to extend the size of the sector buffer.

As well as the traditional way of addressing sectors using cylinder, head and sector, it is possible to access the sectors using a logical address from 0 to N-1, where N is the total number of sectors on the disk. It is also possible to auto-increment this logical address each time a command which accesses the disk is performed, and to auto-increment the desired sector buffer each time the sector buffer is accessed.

In addition to the normal boot options provided by the **BootFromROM** pin (see section 5.3), the monitor process also provides two power-on boot options. With one of these options the M212 will boot itself with code which has been read off a disk and this code then replaces the mode 1 process. The other option sends a standard boot message read off a disk out of link 0 and the mode 1 process then continues as normal. It is also possible to send a command at any time to boot the mode 1 M212 from code in the sector buffer.

Throughout the section describing mode 1 operation, the phrase "user process" is used to denote the process(es) which are communicating along the M212's links with the mode 1 process. To distinguish between the whole sector buffer and a particular area of it which is being referred to by a command, the latter is called a desired sector buffer. All words in **bold** are the names of commands, parameters (or fields within parameters), error codes or M212 pins.

2.2 Mode 1 programming interface

This section gives a description of the mode 1 programming interface and ways of using it. The information given in this section is expanded upon in further sections if more detail is required, but there should be most of the information required to be able to control a standard winchester or floppy.

In mode 1 the M212 accepts a command **BYTE** down one of its links which may be followed by one or more data **BYTE**s. In the case of a **ReadParameter** or **ReadBuffer** commands a number of **BYTE**s will be returned along the same link. Once a command has been accepted down one of the links, then the other is ignored until an **EndOfSequence** command is received on the active link. This allows two processes to use the same M212 without interfering with each other.

When the mode 1 process has initialised itself all the soft parameters are set to 0 (except for **NumBuffer-BytesBy256** and **M212Version**) and all the hard parameters are set to the winchester initialise defaults (see section 2.11). This means that all drives have **DriveExists** unset and the hardware will be running at a high speed. Before the controller can be used the parameter files for each physical drive have to be initialised to appropriate values. All the parameters could be individually set using **WriteParameter** commands, but the **Initialise** command is provided for initialising all the parameters in a file to either floppy or winchester type values.

All the data to or from the disk has to go through the sector buffer. This is an area of RAM extending from the top 1280 bytes of internal RAM and into external RAM. The size is determined by **NumBufferBytesBy256** and is initialised to the amount of RAM available as described in section 2.12. A particular area of the sector buffer is specified in terms of being the n'th block of size 2^**SectorSizeLg2** where n is the contents of **DesiredSectorBuffer**. The various drives' parameter files can point to overlapping areas if required. This can be useful when transferring data between different drives since no data has to be transferred to the user process.

There are two sector addressing modes and two auto-increment options that can be used in mode 1 and these are specified in the **Addressing** parameter. The required sector can be specified in terms of either **DesiredCylinder1-0**, **DesiredHead** and **DesiredSector** or as **LogicalAddress2-0** which maps all the sectors in a disk into a linear address space of sectors. In logical addressing mode the logical address can be auto-incremented each time the drive is accessed. Also available in either mode is auto-increment of **DesiredSectorBuffer** which is useful for multiple sector transfers. All these modes are more fully explained in section 2.6. The default mode after an **Initialise** command is auto-increment of logical address.

Most of the commands get the information they require from the parameter files. Therefore the relevant parameters must be set up before the command is issued. Note that all parameters are **BYTE**s. There are separate parameter files for each of the 4 possible drives (numbered 1 to 4), the appropriate one being automatically restored into the accessible parameter file and the old one being saved. Before updating or reading any parameters the appropriate parameter file must have been swapped in. This is either done automatically by a command which accesses a drive or it can be done explicitly by using a **SelectDrive** command after setting up **DesiredDrive**. If **DesiredDrive** is invalid then the drive 0 parameter file is selected. The current parameter file may be ascertained by reading the **CurrentDrive** parameter.

The parameter file for each drive is logically one block, but in fact some of the parameters (soft parameters) are held in on-chip RAM while others (hard parameters) are held in the disk control hardware itself. The programming interface to both types of parameter is the same i.e. the **ReadParameter** and **WriteParameter** commands. In the case of drive 0, which means that no drives are selected, the soft parameters are not saved and restored but the hard parameters are. This ensures that the PIA's and other disk control hardware is in a known and consistent state.

For most uses the hard parameters should not need to be explicitly changed from the mode 1 defaults, except possibly during initialisation. Whenever hard parameters need updating during normal operation of the M212 this is automatically done by the mode 1 process.

A list of all the soft parameters and their use is given in section 2.3. Most of these only require modifying once for the initialisation of disk type, size and similar parameters which differ from the defaults provided. The parameters that will be most used are those concerned with drive access (either **DesiredSector**, **DesiredHead** and **DesiredCylinder1-0** or **LogicalSector2-0**; **DesiredSectorBuffer** and **DesiredDrive**) and also the error

parameters (**Error**, **Reason** and **ErrorDrive**).

Other parameters are concerned with drive characteristics (**DriveType**, **RWCCylinderBy4**, **PCCylinderBy4**, **HeadStepRateIn64us**, **HeadSettleTimeIn64us**, **HeadLoadTimeIn0.5ms** and **MotorStartTimeIn4ms**); drive size (**NumberOfHeads** and **NumberOfCylinders1-0**) and drive format (**SectorSizeLg2**, **NumberOfSectors**, **Interleave**, **Skew**, **NumGap3Bytes**, **NumGap4BytesBy256** and **NumEccCorrectableBits**). Of the remaining parameters **Addressing** is used to select the disk addressing modes, **SectorRetries** and **SeekRetries** are used to control how many attempts are made to read or write a sector before giving up and **NumBufferBytesBy256** gives the size of the available sector buffer. **NumBufferBytesBy256** is set up after a reset as described in section 2.12 and should not be altered except to reserve space at the top of the sector buffer. The remaining 3 parameters (**CurrentCylinder1-0** and **CurrentDrive**) must not be altered by the user process.

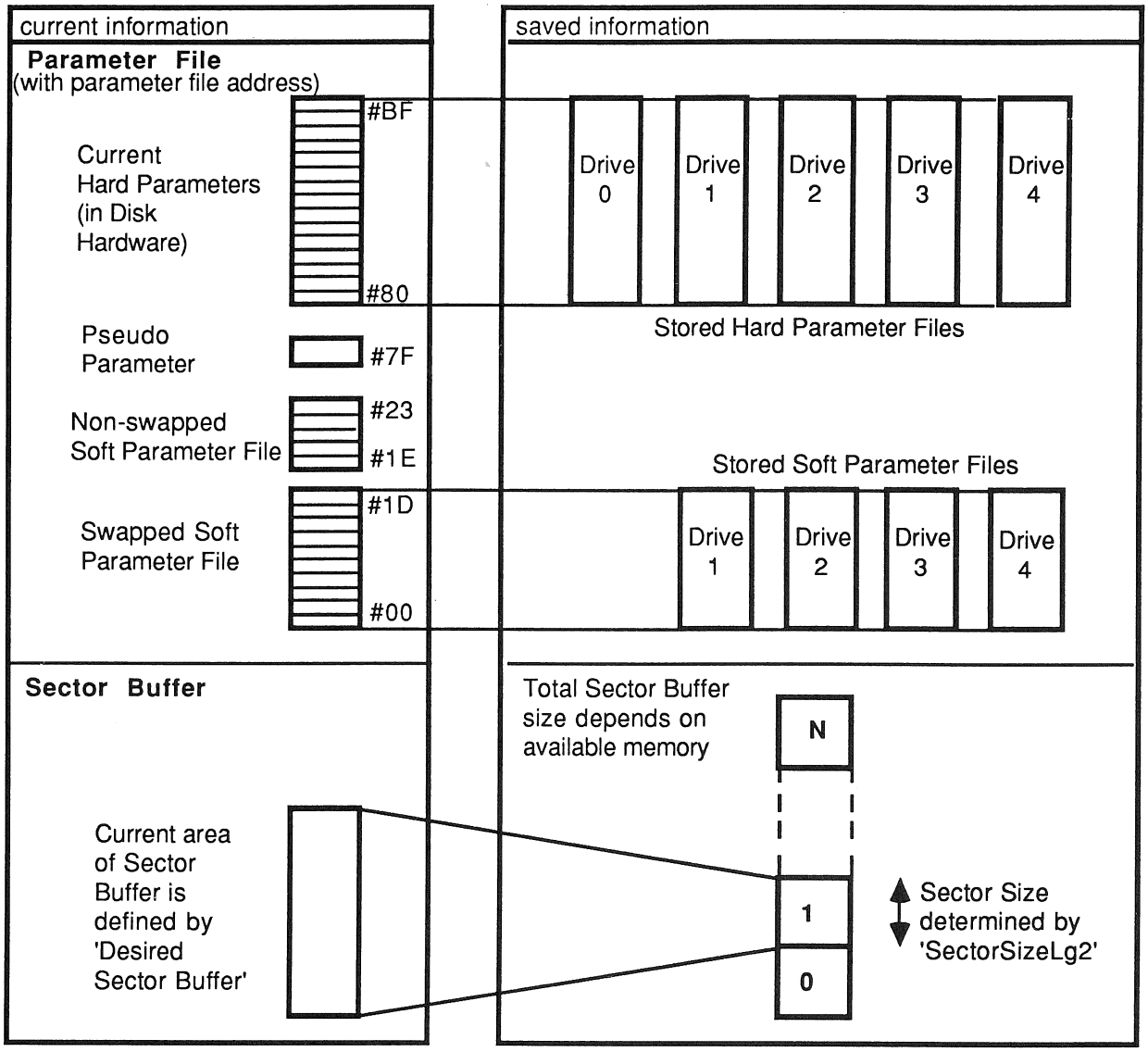
In the following table the command name is followed by the command value and the data **BYTE**s expected or returned. The table is followed by a list of the commands giving a brief description. The words in **bold type** (apart from the command names) are the names of parameters. A more detailed explanation of each command is provided in section 2.4.

Command Name	Command Value	Data Bytes Sent		Data Bytes Returned
		First	Second	
EndOfSequence	#00	—	—	—
Initialise	#01	Drive Type	—	—
ReadParameter	#02	Parameter Address	—	Value
WriteParameter	#03	Parameter Address	New Value	—
ReadBuffer	#04	—	—	Buffer Bytes
WriteBuffer	#05	Buffer Bytes	—	—
ReadSector	#06	—	—	—
WriteSector	#07	—	—	—
Restore	#08	—	—	—
Seek	#09	—	—	—
SelectHead	#0A	—	—	—
SelectDrive	#0B	—	—	—
PollDrive	#0C	—	—	—
FormatTrack	#0D	—	—	—
Boot	#0F	—	—	—

EndOfSequence ends an indivisible sequence of commands on the link and allows both links to be examined.
Initialise will initialise the drive specified in **DesiredDrive** to be a winchester or a floppy.
ReadParameter reads the value of the specified parameter from the current parameter file.
WriteParameter writes the new value into the specified parameter in the current parameter file.
ReadBuffer returns 2^**SectorSizeLg2** **BYTE**s from the **DesiredSectorBuffer**.
WriteBuffer accepts 2^**SectorSizeLg2** **BYTE**s into the **DesiredSectorBuffer**.
ReadSector reads the specified sector from **DesiredDrive** into the **DesiredSectorBuffer**.
WriteSector writes the specified sector on **DesiredDrive** from the **DesiredSectorBuffer**.
Restore moves the heads of **DesiredDrive** towards the outermost track until the **notTrack0** signal is TRUE.
Seek moves the heads of **DesiredDrive** to **DesiredCylinder1-0**.
SelectHead selects the **DesiredHead** on **DesiredDrive**.
SelectDrive selects **DesiredDrive** as the current drive.
PollDrives looks for the first drive to become ready and sets **DesiredDrive** to this drive.
FormatTrack initialises the specified track on **DesiredDrive** to a required format for the controller.
Boot allows code in the sector buffer to take control of the M212.

Section 2.14 shows a typical set of declarations and command sequences that can be used in mode 1.

Mode 1 Programming Model



2.3 Parameters

Each of the possible drives (numbered 1 to 4) has an associated parameter file which contains information about the characteristics of the drive and the current status of the drive. Each drive's parameter file is automatically swapped when the drive is selected. This can be done by setting **DesiredDrive** to the required drive and then either issuing a command which accesses a drive or by explicitly issuing a **SelectDrive** command. The latter case will be necessary if it is required to update any of the parameters prior to performing a disk accessing operation.

If no drive is selected (**CurrentDrive** = 0) then the values of soft parameters which will be read are those of the previously selected drive. They should not be written to since the actual parameter values will already have been saved. The new values would be overwritten when the next drive is selected, even if it is the same drive that was previously selected.

The following parameters are maintained for each drive and are listed with the corresponding parameter address:

#00 DesiredSector
specifies the number of the sector to be accessed by a **ReadSector** or **WriteSector** command. The valid range of this parameter is either 1 to **NumberOfSectors** if **SectorsFrom1** is set or 0 to **NumberOfSectors** - 1 if **SectorsFrom1** is not set. A value for **DesiredSector** of 256 is indicated by 0.

#01 DesiredHead
specifies the head to be used by a **ReadSector**, **WriteSector** or **FormatTrack** command. The valid range of values is between 0 and **NumberOfHeads** - 1 inclusive.

#02 DesiredCylinder0, #03 DesiredCylinder1
form a double length parameter (**DesiredCylinder0** = LS byte, **DesiredCylinder1** = MS byte) specifying the cylinder to seek to for a **ReadSector**, **WriteSector**, **Seek** or **FormatTrack** command. The allowed range is between 0 and **NumberOfCylinders1-0** - 1 inclusive.

#04 LogicalSector0, #05 LogicalSector1, #06 LogicalSector2
form a triple length parameter (with **LogicalSector0** = LS byte and **LogicalSector2** = MS byte). If **LogicalAddressing** is set then these parameters are used instead of the above to specify the sector to be accessed by a **ReadSector** or **WriteSector** command or the track to be accessed by a **FormatTrack** command. The valid range is between 0 and N-1 inclusive where N is the total number of sectors on the disk i.e. **NumberOfSectors** * **NumberOfHeads** * **NumberOfCylinders1-0**.

#07 Addressing
the bits in this parameter enable the following modes when set:

bit 0 - LogicalAddressing
If this bit is set, then instead of addressing a sector in terms of **DesiredSector**, **DesiredHead** and **DesiredCylinder1-0**, a logical sector number is set up in **LogicalSector2-0**. This specifies a sector which is between 0 and N-1 where N is the total number of sectors on the disk i.e. **NumberOfSectors** * **NumberOfHeads** * **NumberOfCylinders1-0**.
See section 2.6 for further information.

bit 1 - IncrementLogical mode.
If this bit as well as **LogicalAddressing** is set, then each time a **ReadSector** or **WriteSector** command is issued **LogicalSector2-0** is incremented by one. Thus a sequence of **ReadSector** or **WriteSector** commands may be sent without having to update the required parameters. In addition, using this mode with **FormatTrack** commands will increment by **NumberOfSectors** so that each successive command will format the next logical track, again without having to update the required parameters. See section 2.6 for further information.

bit 2 - **IncrementBuffer** mode.
If this bit is set, then each time the sector buffer is accessed, **DesiredSectorBuffer** is incremented by one unless an error has occurred. If the **DesiredSectorBuffer** is incremented beyond the end of the total available buffer then an error will be indicated when it is used. The commands which access the sector buffer are **ReadBuffer**, **WriteBuffer**, **ReadSector** and **WriteSector**.

bits 3-7 are reserved by INMOS and should not be used.

#08 DriveType
the bits in this parameter have the following meanings when set:

bit 0 - **Winchester** format. If this bit is not set then floppy format is used.

bit 1 - **WriteProtect**. If set then no **WriteSector** or **FormatTrack** commands are allowed. This is in addition to any hardware write protect which may or may not be set.

bit 2 - **SectorsFrom1**. If this bit is set then sectors are numbered from 1, otherwise they are numbered from 0.

bit 3 - **LengthBy128Lg2** indicates whether the sector length bits in the ID fields are a direct indication of the sector size, or are offset as required by some formats. The sector size bits used for the two modes are as follows:

sector size	**LengthBy128Lg2** set	**LengthBy128Lg2** unset
128	000	011
256	001	000
512	010	001
1024	011	010
2048	100	111
4096	101	100
8192	110	101
16384	111	110

bit 4 - **HasReady** should be set if the drive has a ready line. If this bit is not set then the drive is always assumed to be ready and so **MotorStartTimeIn4ms** must be set to an appropriate value.

bit 5 - **HasSeekComplete** should be set if the drive has a seek complete line. If this bit is not set then **HeadSettleTimeIn64us** must be set to an appropriate value.

bit 6 - **PollThisDrive** if set indicates that when performing a multiple **PollDrives** command (i.e. **DesiredDrive** is zero) then this drive should be examined for ready status. If not set then the drive will be ignored during a multiple **PollDrives** command even if it is ready.

bit 7 - **DriveExists** must be set to indicate that the current drive is accessible. If it is not set then a **DriveDoesNotExist** error will be generated if an attempt to access the drive is made. It could be reset to indicate that a drive is off-line or dismounted.

#09 SectorSizeLg2
specifies the sector size as the log base 2 i.e. sector size = $2^{SectorSizeLg2}$. Valid sector sizes are:

SectorSizeLg2	sector size
#07	128
#08	256
#09	512
#0A	1024
#0B	2048
#0C	4096
#0D	8192
#0E	16384

#0A NumberOfSectors
is the number of sectors per track, which may be between 1 and 256 inclusive. 256 is indicated by the value 0.

#0B NumberOfHeads
is the number of heads on the drive and may be between 1 and 8 inclusive for a floppy and between 1 and 16 inclusive for a winchester.

#0C NumberOfCylinders0, #0D NumberOfCylinders1
are the number of cylinders specified as a double parameter (**NumberOfCylinders0** = LS byte, **NumberOfCylinders1** = MS byte). The number of cylinders may be between 1 and 4096 inclusive.

#0E CurrentCylinder0, #0F CurrentCylinder1
specifies the current position of the heads and must not be altered by the user process. If the value is changed by the user process then the heads may be stepped beyond the allowable range of cylinders for the drive. A **Restore** command will re-synchronise these parameters with the drive, but if the heads have already moved beyond the number of cylinders specified in **NumberOfCylinders1-0** then two **Restore** commands will be required with **Error** being reset between them.

#10 RWCCylinderBy4
is the first cylinder for which reduced write current should be set, divided by 4. The value may be between 0 and 255 specifying cylinders of 0 to 1020.

#11 PCCylinderBy4
is the first cylinder for which pre-compensation should be used, divided by 4. The value may be between 0 and 255 specifying cylinders of 0 to 1020.

#12 SectorRetries
is the number of times the software will attempt to retry a **ReadSector** or **WriteSector** command before giving up and may be between 0 and 255 inclusive. If a re-seek is performed then the number of sector retries performed so far is reset to zero. See section 2.7 for further information.

#13 SeekRetries
is the number of times the software will attempt to re-seek the heads during **ReadSector** or **WriteSector** command before giving up and may be between 0 and 255 inclusive. If a re-seek is performed then the number of sector retries performed so far is reset to zero. See section 2.7 for further information.

#14 HeadStepRateIn64us
is the head step rate used during a **Restore** or **Seek**, in multiples of 64us. The value can be 0 to 255 inclusive specifying step rates of 64us to just over 16ms. A value of 0 will give the shortest possible step rate but it will not actually be 0us due to program and logic delays which are affected by both the speed of the part and the speed at which the disk controller logic is running. For an M212-15 the quickest step rate is 50us for the default winchester and 64us for the default floppy settings. The step pulse width similarly varies and is 12us for the default winchester and 33us for the default floppy settings.

#15 HeadSettleTimeIn64us
is the head settle time used after a **Restore** or **Seek**, in multiples of 64us. The value can be 0 to 255 inclusive specifying settle times of 0ms to just over 16ms. If **HasSeekComplete** is set then this value must be zero, otherwise an unnecessary delay will be incorporated.

#16 HeadLoadTimeIn0.5ms
is the head load time used after a **SelectHead**, in multiples of approximately 0.5ms (actually 0.512ms). The value can be 0 to 255 inclusive specifying load times of 0ms to just over 130ms.

#17 MotorStartTimeIn4ms
is the motor start time or time for the drive to become ready in multiples of approximately 4ms (actually 4.096ms) and is used after a **SelectDrive**. The value can be between 0 and 255 specifying start times of 0ms to just over 1s. If **HasReady** is set then this value must be zero, otherwise an unnecessary delay will be incorporated.

#18 Interleave
is used during a **FormatTrack** command and indicates the staggering of sectors around the track. The value of **Interleave** may be between 1 and **NumberOfSectors** inclusive but a value of **NumberOfSectors** will give an interleave of 1. See section 2.9 for further information.

#19 Skew
is similar to interleave in that it is used during a **FormatTrack** command, but **Skew** is used to optimise performance across tracks. The value of **Skew** may be between 0 and **NumberOfSectors** - 1 inclusive. See section 2.9 for further information.

#1A NumGap3Bytes
is used during a **FormatTrack** command and specifies the gap in terms of bytes between successive physical sectors. The value may be between 0 and 255 inclusive. In general it will not be necessary to change the value of this parameter. Possible uses are to increase the value slightly instead of **Interleave** if a sector is just being missed, or to decrease it slightly to squeeze in an extra sector on a track. See section 2.9 for further information.

#1B NumGap4BytesBy256
is the last of the parameters used specifically for **FormatTrack** commands. This parameter gives the number of bytes to put at the end of the sector information to fill the track completely. The value may be between 0 and 255 inclusive specifying a gap 4 of 0 to 65280 bytes. See section 2.9 for further information.

#1C NumEccCorrectableBits
If the Data field uses an ECC polynomial then **NumEccCorrectableBits** must be set to the correction span of that polynomial. This is then used while performing an automatic correction if two consecutive ECC syndromes are the same during a **ReadSector** command. The allowable values are 0 to 24 inclusive, with 0 implying that no correction is possible. See section 2.10 for further information.

#1D DesiredSectorBuffer
indicates the area of the sector buffer from which data is to be read or to which data should be written. This is expressed as an offset from the beginning of the sector buffer in terms of the sector size specified by 2^**SectorSizeLg2**. In terms of occam 2, the area selected is the **BYTE** slice `[SectorBuffer FROM DesiredSectorBuffer*SectorSize FOR SectorSize]`. The allowed values are those between 0 and 255 which specify a desired sector buffer that is completely contained in the available sector buffer. This means that **DesiredSectorBuffer** must be less than the integer part of (256 * **NumBufferBytesBy256**) / (2^**SectorSizeLg2**). Note that there is no valid desired sector buffer if the sector size is greater than the amount of sector buffer available and also, if 128 byte sectors are being used, that if **NumBufferBytesBy256** is greater than #80 then the top of the sector buffer will be unavailable.

In addition to the above parameters for each of the four drives, there are 6 parameters that are not swapped with the drives:

#1E DesiredDrive
is the drive upon which any disk access commands should be performed. Valid drives are 1 to 4 inclusive. A value of 0 is also allowed in which case all drives will be de-selected and any command that attempts to access a disk will give a fault (except for **PollDrives**).

#1F CurrentDrive
is the currently selected drive must not be altered by the user.

#20 Error
is the main status register and if non-zero indicates that an error has occurred. Once an error has been set, then subsequent errors do not change **Error**, **Reason** or **ErrorDrive** and commands which access a disk are ignored. See section 2.5 for further information.

#21 Reason
provides extra information about the error code in **Error**. See section 2.5 for further information.

#22 NumBufferBytesBy256
indicates the amount of RAM available for the sector buffer. The value in this parameter is set up automatically to include all contiguous external RAM. See section 2.12 for further information on how this is done. It may be reduced by the user to reserve space at the top of the sector buffer if required, but must not be increased beyond the end of RAM otherwise data may be lost. The default value, if no external RAM is used, is #05 i.e. 5 * 256 = 1280 bytes and valid values are in the range #01 to #ED inclusive.

#23 ErrorDrive (M212Version)
is initially set on reset to the version of the M212. This initial value may be changed in order to indicate either hardware or software revisions. At other times this parameter is used to indicate the **CurrentDrive** at the time an error occurred, or if no error has occurred then the drive for which the last **Reason** bit was set.

On reset all the above parameters, except for **NumBufferBytesBy256** and **M212Version (ErrorDrive)** are initialised to zero.

There is also one special pseudo-parameter:

#7F ControllerAccess
can be used to read or write data directly to or from the disk controller hardware so that, for example, control codes and commands may be sent. Great care should be taken when using this parameter. This parameter allows the mode 1 user the ability to transfer commands and data to the disk hardware in a manner that is similar to the operation in mode 2.
See section 2.4.3, 2.4.4 and 2.13 for further information on this parameter's use in mode1 and section 3 for a description of the disk hardware.

As well as the above parameters which are held in RAM, there are a number of parameters which are held in the disk controller hardware. These are referred to in this section as "hard parameters" and further information can be found in section 3. These hard parameters are automatically maintained in mode 1 and it is only necessary to change them if non-standard formats are used. See section 2.9 for futher information on non-standard formats. The hard parameters may be accessed by using **ReadParameter** and **WriteParameter** in the same way as for soft parameters, but using special addresses (the mode 2 addresses with bit 7 set). See section 2.4.3 and 2.4.4 for further information. The hard parameter values are set up by **Initialise** and the values for each drive are remembered in the same way as for ordinary parameters, except that the values for when no drive is selected (**DesiredDrive** = 0) are actually saved and restored. On reset the values in these parameters are those specified for the **Initialise** of a winchester disk.

2.4 Commands

This section contains a list of all the commands implemented in mode 1. It outlines the commands purpose, an example of its use and indicates the errors that may be indicated in the **Error** parameter. In the examples shown in the "Use" section of each command, statements which are shown as comments are intended to show possible code sequences that may be used before or after a command is issued. They are not required so long as the appropriate parameters etc. have already been set up. No automatic output of the errors will be indicated, the user process must interrogate the **Error** parameter using a **ReadParameter** command. A description of the errors can be found in section 2.5.

Note that all commands and data in mode 1 are transmitted as `BYTE`s.

The following commands are implemented:

2.4.1 #00 EndOfSequence

Purpose

Once a command has been accepted on one of the links, the other link is ignored until an end of sequence command is received. At this point both links are again examined and the first link to send a command gains control. This allows an indivisible sequence of commands to be performed if, for example, two user processes are sharing a single M212. The only other action performed by this command is to reset the **Status** and **Error** parameters to zero.

Use

```
ToM212 ! EndOfSequence
```

Errors

None are set. **Error** and **Reason** are reset to zero.

2.4.2 #01 Initialise

Purpose

Initialise may be issued for each drive being controlled as a convenient way of setting up default parameter values. Note that if initialise is not used, then the hardware parameters will also have to be initialised. Following this command any of the parameters may be changed and the new values will be retained for **DesiredDrive**. See section 2.11 for the default values set up.

Use

DesiredDrive should be set to the required drive and then the command should be issued followed by a single data `BYTE` which specifies whether the **DesiredDrive** is to be initialised as a floppy (data byte = #00) or as a winchester (data byte = #01).

```
-- ToM212 ! WriteParameter; DesiredDrive; 3 (BYTE)
ToM212 ! Initialise; 0 (BYTE) -- sets drive to floppy
-- ToM212 ! Restore
```

or

```
-- ToM212 ! WriteParameter; DesiredDrive; 1 (BYTE)
ToM212 ! Initialise; 1 (BYTE) -- sets drive to winchester
-- ToM212 ! Restore
```

Errors

BadParameterValue
DriveDoesNotExist

2.4.3 #02 ReadParameter

ReadParameter

Purpose

to read a value from a parameter of the currently selected parameter file. The currently selected parameter file is indicated by the value of **CurrentDrive**.
It also has two special uses. It allows data to be read from a hardware parameter by OR'ing the hard parameter address with #80 and it also allows data to be directly read from the disk controller hardware by using **ControllerAccess** as the parameter. See section 2.3 for a list of soft parameters and section 2.11 for a full list of soft parameters, hard parameters and their mode 1 addresses.

Use

Read parameter must be followed by a single data **BYTE** specifying the parameter to be read and returns the value in that parameter.

```
ToM212 ! ReadParameter; SoftParameter
FromM212 ? DataByte
```

or

```
ToM212 ! ReadParameter; BYTE (#80 \/ HardParameter)
FromM212 ? DataByte
```

or

```
ToM212 ! ReadParameter; ControllerAccess
FromM212 ? DataByte
```

Errors

BadParameterForRead

Comments

Care must be taken when using this command to read from **ControllerAccess** since if no data is or will become available then the process will hang. See section 2.13 for possible uses of this version of the command.

2.4.4 #03 WriteParameter

Purpose

to write a value into a parameter of the currently selected parameter file. The currently selected parameter file is indicated by the value of **CurrentDrive**.
It also has two special uses. It allows data to be written into a hardware parameter by OR'ing the hardware parameter address with #80 and it also allows data to be directly written to the disk controller hardware by using **ControllerAccess** as the parameter. See section 2.3 for a list of soft parameters and section 2.11 for a full list of soft parameters, hard parameters and their mode 1 addresses.

Use

Write parameter is followed by two **BYTE**s specifying the parameter which is to be written and the value to be written into that parameter.

```
ToM212 ! WriteParameter; SoftParameter; Value
```

or

```
ToM212 ! WriteParameter; BYTE (#80 \/ HardParameter); Value
```

or

```
ToM212 ! WriteParameter; ControllerAccess; Value
```

Errors

BadParameterForWrite

Comments

Care must be taken when using this command to write to **ControllerAccess** since if data cannot be accepted then the process will hang. It can also cause data to be corrupted by, for example, a format operation being started. See section 2.13 for possible uses of this version of the command.

2.4.5 #04 ReadBuffer

Purpose

to read data stored in the sector buffer as specified by **DesiredSectorBuffer** and 2^**SectorSizeLg2**: **[SectorBuffer FROM DesiredSectorBuffer*SectorSize FOR SectorSize]**.

Use

The command returns the specified number of **BYTE**s.

```
-- ToM212 ! WriteParameter; DesiredSectorBuffer; BufferToRead
-- ToM212 ! WriteParameter; SectorSizeLg2; SizeOfReadBufferLg2
ToM212 ! ReadBuffer
FromM212 ? [MyBuffer FROM 0 FOR ReadBufferSize]
```

Errors

BadParameterValue

Comments

If **SectorSizeLg2** is outside the permitted range of sector sizes (128 - 16384) then 128 bytes will be returned. Also if **DesiredSectorBuffer** is such that the area extends past the end of the available sector buffer then data starting from the beginning of the buffer is returned.

2.4.6 #05 WriteBuffer

Purpose

to write data into the sector buffer as specified by **DesiredSectorBuffer** and 2^**SectorSizeLg2**: **[SectorBuffer FROM DesiredSectorBuffer*SectorSize FOR SectorSize]**.

Use

The command must be followed by the required number of **BYTE**s.

```
-- ToM212 ! WriteParameter; DesiredSectorBuffer; BufferToWrite
-- ToM212 ! WriteParameter; SectorSizeLg2; SizeOfWriteBufferLg2
ToM212 ! WriteBuffer
ToM212 ! [MyBuffer FROM 0 FOR WriteBufferSize]
```

Errors

BadParameterValue

Comments

If **SectorSizeLg2** is outside the permitted range of sector sizes (128 - 16384) then 128 bytes will be accepted. If **DesiredSectorBuffer** is such that the area extends past the end of the available sector buffer then data is placed starting from the beginning of the buffer.

2.4.7 #06 ReadSector

Purpose

to read data into the desired sector buffer from a specified sector on **DesiredDrive**. The area is specified as in **ReadBuffer**.

Use

The sector may either be specified by using **DesiredCylinder1-0**, **DesiredHead** and **DesiredSector** or by using logical addressing mode and specifying **LogicalSector2-0**. See section 2.6 for further information.

```
-- ToM212 ! WriteParameter; DesiredSector; Sector
ToM212 ! ReadSector
-- ToM212 ! ReadBuffer
-- FromM212 ? [MyBuffer FROM 0 FOR SectorSize]
```

Errors

Even if **Error** is zero then warnings may be given in **Reason**. See section 2.5 for information on possible warnings.

BadParameterValue
BadPolyType
UncorrectableEccError
TimedOut
DriveNotSelected
DriveHasBecomeNotReady
DriveDoesNotExist
TooManySectorRetries
TooManySeekRetries

Comments

An automatic **SelectDrive**, **SelectHead** and **Seek** are performed at the start of this command. The number of retries as specified in **SectorRetries** and **SeekRetries** are automatically performed. See section 2.7 for further information. Also an automatic correction is performed if using ECC and a correctable error is found. See section 2.10 for further information.

2.4.8 #07 WriteSector

Purpose

to write data from the desired sector buffer onto a specified sector of **DesiredDrive**. The area is specified as in **WriteBuffer**.

Use

The sector may either be specified by using **DesiredCylinder1-0**, **DesiredHead** and **DesiredSector** or by using logical addressing mode and specifying **LogicalSector2-0**. See section 2.6 for further information.

```
-- ToM212 ! WriteBuffer; [MyBuffer FROM 0 FOR SectorSize]
-- ToM212 ! WriteParameter; DesiredSector; Sector
ToM212 ! WriteSector
```

Errors

Even if **Error** is zero then warnings may be given in **Reason**. See section 2.5 for information on possible warnings.

BadParameterValue
BadPolyType
TimedOut
DriveNotSelected
DriveHasBecomeNotReady
DriveDoesNotExist
DriveReadOnly
TooManySectorRetries
TooManySeekRetries

Comments

An automatic **SelectDrive**, **SelectHead** and **Seek** are performed at the start of this command. The number of retries as specified in **SectorRetries** and **SeekRetries** are automatically performed. See section 2.7 for further information.

2.4.9 #08 Restore

Purpose

to set the heads of **DesiredDrive** back to a known place (track 0). It is not possible to tell directly where the heads of a drive are except when they are on track 0. This command is used to set the drive to track 0 so that the controller knows where the heads are.
If the **notTrack0** signal is TRUE at the start of the command then the heads are stepped inwards until **notTrack0** is FALSE. This is to ensure that the drive is not on a negative track. The heads are then stepped outwards until **notTrack0** is TRUE. **CurrentCylinder1-0** is reset to zero.

Use

```
-- ToM212 ! WriteParameter; DesiredDrive; DriveToRestore
ToM212 ! Restore
```

Errors

BadParameterValue
TimedOut
DriveNotSelected
DriveHasBecomeNotReady
DriveDoesNotExist
TooManySteps

2.4.10 #09 Seek

Purpose

to move the heads of **DesiredDrive** to the track specified in **DesiredCylinder1-0**.

Use

```
-- ToM212 ! WriteParameter; DesiredCylinder0; 1 (BYTE)
ToM212 ! Seek
```

Errors

BadParameterValue
TimedOut
DriveNotSelected
DriveHasBecomeNotReady
DriveDoesNotExist

Comments

If the drive has a seek complete line (and **HasSeekComplete** is set) then a **Seek** command can be used to allow overlapped seeks on several drives at once. The **PollDrives** command (with **DesiredDrive** set to zero) can then be used to find the first drive to become ready.

2.4.11 #0A SelectHead

Purpose

to select the head specified in **DesiredHead**.

Use

```
-- ToM212 ! WriteParameter; DesiredHead; 1 (BYTE)
ToM212 ! SelectHead
```

Errors

BadParameterValue

2.4.12 #0B SelectDrive

Purpose

to select the drive specified in **DesiredDrive** which should be between 1 and 4. If **DesiredDrive** is set to zero then all drives will be de-selected.
This command also saves the parameter file of the previously selected drive and swaps in the parameter file for **DesiredDrive**. It must be used if it is desired to update some parameters before issuing a command which accesses the disk.

Use

```
-- ToM212 ! WriteParameter; DesiredDrive; 1 (BYTE)
ToM212 ! SelectDrive
-- ToM212 ! WriteParameter; DesiredSector; 2 (BYTE)
-- ToM212 ! ReadSector
```

Errors

BadParameterValue

2.4.13 #0C PollDrives

Purpose

is used for two similar purposes. Firstly, if **DesiredDrive** is in the range 1 to 4 then the command waits until the specific drive is ready or the command times out. Secondly, if **DesiredDrive** is set to zero then all drives which have **PollThisDrive** in **DriveType** set will be polled until one becomes ready or the command times out. In this case then **DesiredDrive** will be left set to the drive which became ready or which was the last to be looked at if the command timed out.

Use

```
-- ToM212 ! WriteParameter; DesiredDrive; 1 (BYTE)
ToM212 ! PollDrives -- poll drive 1
```

or

```
-- ToM212 ! WriteParameter; DesiredDrive; 0 (BYTE)
ToM212 ! PollDrives      -- perform multiple drive poll
-- ToM212 ! ReadParameter; DesiredDrive
-- FromM212 ? ReadyDrive -- find out which drive is ready
-- ToM212 ! ReadParameter; Error
-- FromM212 ? ErrorCode  -- and whether timeout occurred
```

Errors

BadParameterValue
TimedOut
DriveDoesNotExist

2.4.14 #0D FormatTrack

Purpose

to initialise a track on **DesiredDrive** to a required format for the controller. It may also be used to wipe any data from the track.

Use

The track may be specified by either **DesiredCylinder1-0** and **DesiredHead**, or by **LogicalSector2-0** if the **LogicalAddressing** bit in **Addressing** is set. See section 2.6 for further information.
The parameters associated with formats such as **NumberOfSectors**, **SectorSizeLg2** and **Interleave** must have been set up prior to issuing this command. See section 2.9 for further information.

```
-- ToM212 ! WriteParameter; DesiredHead; 0 (BYTE)
-- ToM212 ! WriteParameter; NumberOfSectors; 8 (BYTE)
-- ToM212 ! WriteParameter; SectorSizeLg2; Lg512
ToM212 ! FormatTrack
```

Errors

BadParameterValue
BadPolyType
TimedOut
DriveNotSelected
DriveHasBecomeNotReady
DriveDoesNotExist
DriveReadOnly
FormatUnderrun

Comments

An automatic **SelectDrive**, **SelectHead** and **Seek** are performed at the start of this command. The first bytes of the sector buffer are used by this command. The command bytes are stored in the first 128 bytes, followed by **NumberOfSectors** bytes of interleave buffer. Thus for tracks with 128 sectors or fewer only the first 256 bytes will be used, and a maximum of 384 bytes will be used. The interleave table (and command table) may be read after the completion of **FormatTrack** by using **ReadBuffer** to check, for example, on the location of sectors on the track.

2.4.15 #0F Boot

Purpose

to allow user code to control the M212. For example, it may be useful to include high level file handling commands in the M212 to relieve the host processor of this overhead, or to provide a cache of more commonly used sectors such as the directory. The process may have up to three **VAL INT** parameters which can be set up at the time the process is started.

Use

The code must be stored in the sector buffer and **DesiredSectorBuffer** set to point to the beginning of the code (the code may either have been directly read off the disk, or it may have been loaded down one of the links using **WriteBuffer** commands. Additional information required by the command should be stored in the following parameter pairs, LS parameter first:

RWCCylinder/PCCylinder
byte offset from **DesiredSectorBuffer** of the entry point for the code.

SectorRetries/SeekRetries
workspace requirement of the code in words.

HeadStepRateIn64us/HeadSettleTimeIn64us
optional parameter 1.

HeadLoadTimeIn0.5ms/MotorStartTimeIn4ms
optional parameter 2.

Interleave/Skew
optional parameter 3.

```
-- ToM212 ! ReadSector
-- ToM212 ! ReadSector
-- ToM212 ! WriteParameter; DesiredSectorBuffer; BYTE CodeBegin
-- ToM212 ! WriteParameter; RWCCylinder; BYTE (Entry \ 256)
-- ToM212 ! WriteParameter; PCCylinder;  BYTE (Entry / 256)
-- ToM212 ! WriteParameter; SectorRetries; BYTE (WSReq \ 256)
-- ToM212 ! WriteParameter; SeekRetries;   BYTE (WSReq / 256)
-- ToM212 ! WriteParameter; HeadStepRateIn64us;   BYTE (P1 \ 256)
-- ToM212 ! WriteParameter; HeadSettleTimeIn64us; BYTE (P1 / 256)
ToM212 ! Boot
```

Errors

If **Error** is non-zero then the mode 1 process **STOP**s and the **Error** pin is set. Possible causes of non-zero **Error** are **SectorSizeLg2** or **NumBufferBytesBy256** being invalid, **DesiredSectorBuffer** and **SectorSizeLg2** specifying an invalid area of the sector buffer, not enough workspace available as specified by **SectorRetries/SeekRetries** or **Error** being non-zero before the command is started.

Comments

The workspace is put immediately beneath the beginning of the code and in order to maximise the amount of workspace available, the code should be placed as near the end of the sector buffer as possible. The amount available can be calculated by
361 + ((**DesiredSectorBuffer** * (1 << **SectorSizeLg2**)) / 2) words.
Note that following a **Boot** command the mode 1 process will **STOP**, even if a return is performed.

2.5 Errors and reasons

Error is the main status register and it must be explicitly read each time the status is required. If non-zero it indicates that an error has occurred. **Reason** contains either warning information about non-serious errors which have occurred or, if **Error** is non-zero, then further information about the error. **ErrorDrive** contains the **DesiredDrive** which was selected at the time that the last error or warning was set.

Once **Error** has been set, then subsequent errors do not change **Error**, **Reason** or **ErrorDrive**, nor are any commands which access a disk obeyed (**ReadSector**, **WriteSector**, **Restore**, **Seek**, **PollDrives** and **FormatTrack**).

To clear **Error** and **Reason** either they must be explicitly set to zero using the **WriteParameter** command, or they can be cleared by using the **EndOfSequence** command, but in the latter case control of the M212 may be lost to the other link. **ErrorDrive** is not cleared by an **EndOfSequence** command and, if desired, has to be cleared explicitly. However, it is only valid if either **Error** or **Reason** is non-zero in which case it will have been set again, so for most purposes it will not be necessary to clear this parameter.

#00 AllOk

no fatal error has occurred.
Reason may contain warning information that can be used for error logging procedures:

bit 0 - **BadDataCompareByte** if set indicates that a **ReadSector** command successfully finished but that the data compare byte was not as specified in **DataCompare**.

bit 1 - **SectorCorrected** if set indicates that when doing a **ReadSector** command an ECC error was found, but that the data was successfully corrected.

bit 2 - **SectorRetries** if set indicates that when doing a **ReadSector** or **WriteSector** command retries had to be performed, although the command successfully completed.

bit 3 - **SeekRetries** if set indicates that when performing a **ReadSector** or **WriteSector** command re-seeks were performed, although the command successfully completed.

bits 4-7 are not used for warning information.

#01 BadCommand

a command byte was received that was not a valid command.
Reason will contain the erroneous command code.

#02 BadParameterForRead

the parameter specified for a **ReadParameter** command was not a valid parameter.
Reason will contain the erroneous parameter number.

#03 BadParameterForWrite

the parameter specified for a **WriteParameter** command was not a valid parameter.
Reason will contain the erroneous parameter number.

#04 BadParameterValue

an invalid value has been found in a parameter or double parameter.
Reason will contain the parameter number which has the erroneous value. Note that if this specifies the LS parameter of one of the double parameters then it means that the double parameter value as a whole is in error. Also an error in a logical sector number (when in **LogicalAddressing** mode) will manifest itself as an error in **DesiredCylinder0**. If the error was in **DesiredDrive** then all drives will be de-selected and **CurrentDrive** will be set to zero.

#05 BadPolyType

either the ID or Data polynomial was not set to CRC or ECC.
Reason will contain the mask of the polynomial type that was in error i.e. #30 for the Data polynomial and #C0 for the ID polynomial. See section 2.10 for further information.

#06 UncorrectableEccError

two consecutive non-zero ECC syndromes were the same and an attempt was made to correct the error but it was found to be an uncorrectable error.
Reason will be zero.

#07 TimedOut

more than 1 second has elapsed while waiting for a drive to become ready. If this happens during a multiple drive **PollDrives** command then **DesiredDrive** and **CurrentDrive** will be left at an arbitrary drive between 1 and 4 which may be a non-pollable drive or a drive that does not exist.
Reason will be zero.

#08 DriveReadOnly

an attempt has been made to write to a disk that is either hardware or software write protected i.e. either the **notWriteProtect** pin is TRUE or the **WriteProtect** bit in **DriveType** is set.
Reason will contain 1 if the drive was hardware write protected, otherwise 0.

#09 DriveNotSelected

an attempt has been made to access drive 0.
Reason will be zero.

#0A DriveHasBecomeNotReady

during a command which accessed a disk the drive ready signal became de-asserted.
Reason will be zero.

#0B DriveDoesNotExist

either an attempt was made to access a drive for which the **DriveExists** bit in the **DriveType** parameter was not set or an invalid drive type byte was specified during an **Initialise** command.
Reason will be zero in the former case or will be the value of the drive type byte in the latter.

#0C TooManySectorRetries

during a **ReadSector** or **WriteSector** command the number of retries specified in **SectorRetries** was exhausted.
Reason will be zero.

#0D TooManySeekRetries

during a **ReadSector** or **WriteSector** command the number of retries specified in **SeekRetries** was exhausted.
Reason will be zero.

#0E TooManySteps

during a **Restore** command, the track 0 indicator had not reached the required value after **NumberOfCylinders1-0** steps had been taken.
Reason will indicate the logical value of the DirectionIn signal at the time. Thus it will be one if doing the initial steps towards the centre of the disk or zero if doing the restore proper towards track 0.

#0F FormatUnderrun

during a **FormatTrack** command if an under-run error is detected. See section 2.9 for further information on formats and under-runs.
Reason will be zero.

2.6 Addressing and auto-increment modes

Two modes of sector addressing are available in mode 1. Firstly by the "traditional" method of specifying cylinder, head and sector, or secondly by specifying a single logical sector number in **LogicalSector2-0**. This second method maps all of the sectors on a disk into a single linear address space of sectors. The required cylinder, head and sector are calculated by the mode 1 process when a **ReadSector**, **WriteSector** or **FormatTrack** command is performed. Note that **Seek** and **SelectHead** do NOT use the logical address, but always **DesiredCylinder1-0** and **DesiredHead** respectively. The calculated values of **DesiredSector**, **DesiredHead** and **DesiredCylinder1-0** may be examined after the command has completed.

Logical addressing mode is specified by setting the **LogicalAddressing** bit of the **Addressing** parameter. If unset then normal addressing is used. If **LogicalAddressing** is set then an additional bit, **IncrementLogical**, may be set to automatically increment the logical address. It will be incremented by 1 for **ReadSector** and **WriteSector**, or by **NumberOfSectors** for **FormatTrack**. This allows successive commands to access the next sector or track without updating the required address.

In order to make error recovery easier, if **Error** is non-zero i.e. an error has occurred, then no sector auto-increment will take place and **LogicalSector2-0** will be left as it was.

Also useful in this mode is **IncrementBuffer** (see below) which will allow a sequence of **ReadSector** or **WriteSector** commands to be issued which will read or write a block of consecutive sectors from consecutive desired sector buffers without any updating being required.

In logical addressing mode the order of access is sector, then head, then cylinder. This requires the minimum of head switching and stepping. For example, suppose we have a small disk with **NumberOfSectors** = 2, **NumberOfHeads** = 2 and **NumberOfCylinders1-0** = 2. Then the logical sectors are as follows (assuming sectors are numbered from 1 i.e. **SectorsFrom1** is set):

logical sector	desired sector	desired head	desired cylinder
0	1	0	0
1	2	0	0
2	1	1	0
3	2	1	0
4	1	0	1
5	2	0	1
6	1	1	1
7	2	1	1

If a **FormatTrack** is issued when in logical addressing mode, then the track which is formatted is the one which will contain the specified logical sector. If **IncrementLogical** is set then **LogicalSector2-0** is always incremented by **NumberOfSectors** (rather than to the beginning of the next track).

The other addressing mode available is **IncrementBuffer** which automatically increments **DesiredSector-Buffer** each time the sector buffer is accessed (by a **ReadSector**, **WriteSector**, **ReadBuffer** or **WriteBuffer** command). Thus a sequence of **ReadSector** or **WriteSector** commands which will read or write a block of consecutive sectors, or a sequence of **ReadBuffer** or **WriteBuffer** commands which will read a large area of the sector buffer can be issued without having to update **DesiredSectorBuffer**.

In order to make error recovery easier, if **Error** is non-zero i.e. an error has occurred, then no buffer auto-increment will take place and **DesiredSectorBuffer** will be left as it was.

2.7 Retries

Retries are performed during **ReadSector** and **WriteSector** commands if certain errors which mean that the operation cannot be performed correctly are detected.

Sector retries

These retries are performed if the ID field is correctly found, but an error is found in the Data field apart from a **DataCompareError**. i.e. if any error other than **Timeout** or **DataCompareError** is set in hard parameter **StatusRegister1**. In this case another attempt is made to perform the operation straight away unless a correction is being tried. The **SectorRetries** parameter is used to specify the number of sector retries that will be attempted before the command is terminated with a non-zero **Error**. A value of zero in **SectorRetries** disables this form of retry.

If an ECC is being used and **NumEccCorrectableBits** is not zero, then corrections will be attempted after two successsive syndromes are equal. Thus at least 1 sector retry must be allowed in order for corrections to be tried. The more sector retries that are allowed, the greater the chance that two successive syndromes will be equal, but the chance of a mis-correction may become greater.

Seek retries

These retries are performed when the correct ID field is not found within the time allowed by the hardware timeout i.e. the **Timeout** bit of hard parameter **StatusRegister1** is set. It is therefore possible that the heads are over the wrong track and so a restore, followed by a re-seek, is performed before an attempt is made to do the operation again. The **SeekRetries** parameter is used to specify the number of seek retries that will be attempted before the command is terminated with a non-zero **Error**. A value of zero in **SeekRetries** disables this form of retry. If a seek retry is performed then the number of sector retries performed is reset to zero.

The M212 supports hardware retries, but these have been disabled in mode 1 to allow error logging bits to be set in **Reason**. They may be re-enabled by setting the **TimeoutIndexNumberLess2** field of the hard parameter **TimingRegister0** to a value between #1 and #7. See section 3.5.1 for further information. This will allow more hardware retries to be performed before the mode 1 retries are invoked, but the fact that hardware retries have been attempted is not logged. It may, however, be advantageous to enable hardware retries in order to stop seek retries from happening too often. This is because the restore can take a long time to perform, particularly on a winchester with a large number of cylinders and a seek complete line.

2.8 Auto-booting

On reset, if **PA6** (**notHeadSelect[2]**) is low, then an autoboot will be attempted. **PA5-4** (**notHeadSelect[1-0]**) are examined and used as a drive number and type as follows:

PA5	**PA4**	drive number	drive type
high	high	1	winchester
high	low	2	winchester
low	high	3	floppy
low	low	4	floppy

The drive is selected, initialised and restored and an attempt is then made to read sector 1 on head 0 of cylinder 0 (logical sector 0). If this is unsuccessful then the **Error** pin will be set and the process will **STOP**. If read correctly then the first byte of the sector is examined and one of two courses of action taken. If the first byte is between 2 and 255, then the byte followed by that many bytes of the sector are transferred out of link 0. Thus the first sector can be used as a boot message for a transputer at the other end of the link. The boot message can be a primary boot that will read subsequent boot code off the disk. However, if the first byte of the sector is 0 or 1 then the sector must contain details about code which will be read from the disk and used to boot the M212 itself. The layout of the sector bytes in the case of a self boot is as follows:

Bytes

0 must be 0 or 1.

1 is not used.

2-3 contain an INT (LS byte first) specifying the length of the code in bytes.

4-5 contain an INT (LS byte first) specifying the entry point of the code in terms of bytes from the beginning of the code.

6-7 contain an INT (LS byte first) specifying the workspace requirement of the code in words.

8-9 contain an INT (LS byte first) specifying the first optional VAL INT parameter for the code.

10-11 contain an INT (LS byte first) specifying the second optional VAL INT parameter for the code.

12-13 contain an INT (LS byte first) specifying the third optional VAL INT parameter for the code.

14-255 are not used

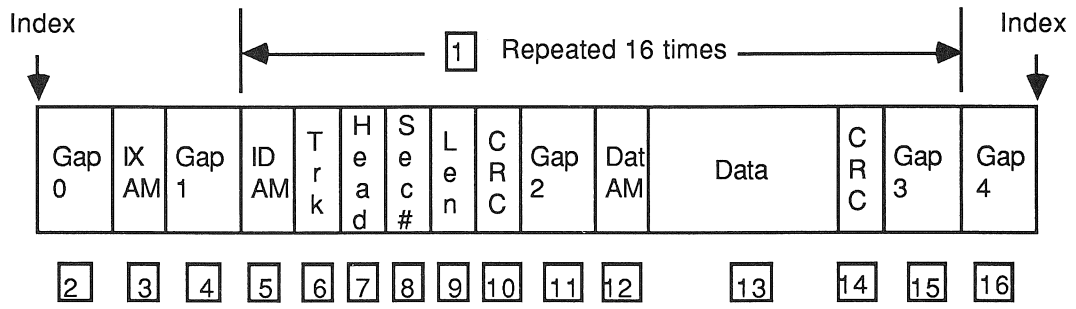
The appropriate number of sectors will be read from logical sector 1 onwards in logical sector number order into the top of the available sector buffer (it must be able to fit into the sector buffer otherwise an error will occur). If no error has occurred and there is enough workspace available, then the code will be entered with the three optional parameters set up. If an error occurs then the **Error** pin will be set and the process will **STOP**. If the boot code returns back to the mode 1 code then it will also **STOP**. The amount of workspace available is 361 + (NumberOfFree256ByteSectorBuffers * 128) words and it is placed immediately beneath the code. Sometimes it may be required to move the code and workspace down from the top of the sector buffer, for example to move them into fast internal RAM. The code can be moved down by making bytes 2-3 (the code length) artificially large and the workspace can be moved down by placing some dummy array declarations at the top of the program before any other declarations.

The booting features of mode 1 assume that the disk uses the default format and disk size. If this is not the case then either the default format must be used for the boot sectors or the format must be compatible for the required number of sectors. The latter case can be achieved, for example, with **SectorsFrom1** unset if the number of sectors read does not exceed the default **NumberOfSectors** - 1. Another possibility is having the actual number of sectors per track greater than than the default, although in this case there will be some unused sectors at the end of each boot code track. If necessary then the first few tracks required for booting could use a different format from that used by the rest of the disk. The write clock provided must also be the required frequency of 20 MHz for a 5 Mbits/s winchester or 16 MHz for a 250 kbits/s floppy.

2.9 Formats available in mode 1

Floppy formats

The standard floppy format (as set up by **Initialise**) uses double density (MFM) and is:



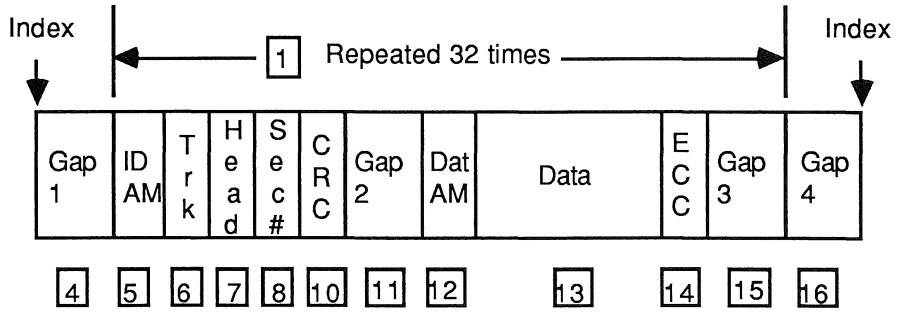
Notes.

1)		16	records of 372 bytes
2)		80	bytes of #4E
3)		12	bytes of #00 followed by
		3	bytes of #C2 with clock bit 3 missing followed by
		1	byte of #FC
4)		50	bytes of #4E
5)		12	bytes of #00 followed by
		3	bytes of #A1 with clock bit 2 missing followed by
		1	byte of #FE-#F1 depending on **`cylinder / 256`**
6)		1	byte of **`cylinder REM 256`**
7)		1	byte of **`head`**
8)		1	byte of **`sector`** between 1 and 16
9)		1	byte of **`length`** which is #01 (256 byte sectors)
10)		2	bytes of preset CRC
11)		22	bytes of #4E followed by
	*	12	bytes of #00
12)	*	3	bytes of #A1 with clock bit 2 missing followed by
	*	1	byte of #FB (data compare byte)
13)	*	256	bytes of #E5
14)	*	2	bytes of preset CRC
15)	*	1	byte of #4E followed by
		53	bytes of #4E
16)		???	bytes of #4E to fill remainder of track

The lines marked with a * are those updated during a **WriteSector**.

Winchester formats

The standard winchester format (as set up by **Initialise**) uses double density (MFM) and is:



Notes.

1)		32	records of 316 bytes
4)		16	bytes of #4E
5)		13	bytes of #00 followed by
		1	byte of #A1 with clock bit 2 missing followed by
		1	byte of #FE-#F1 depending on **`cylinder / 256`**
6)		1	byte of **`cylinder REM 256`**
7)		1	byte of **`head \/ (lengthbits << 5)`** lengthbits = #0
8)		1	byte of **`sector`**
10)		2	bytes of preset CRC
11)		3	bytes of #00 followed by
	*	13	bytes of #00
12)	*	1	byte of #A1 with clock bit 2 missing followed by
	*	1	byte of #F8 (data compare byte)
13)	*	256	bytes of #00
14)	*	4	bytes of preset ECC
15)	*	3	byte of #00
		15	bytes of #4E
16)		???	bytes of #4E to fill remainder of track

The lines marked with a * are those updated during a **WriteSector**.

The byte in field 5 which depends on **cylinder / 256** is used to indicate both that this is an ID field and also which group of 256 cylinders the heads are currently over. For historical reasons of disk controllers, this byte is not a direct indication of the most significant bits, but is encoded according to the following table.

cylinder	byte value
0 - 255	#FE
256 - 511	#FF
512 - 767	#FC
768 - 1023	#FD
1024 - 1279	#FA
1280 - 1535	#FB
1536 - 1791	#F8
1792 - 2047	#F9
2048 - 2303	#F6
2304 - 2559	#F7
2560 - 2815	#F4
2816 - 3071	#F5
3072 - 3327	#F2
3328 - 3583	#F3
3584 - 3839	#F0
3840 - 4095	#F1

The above values may be calculated by **(cylinder / 256) >< #FE**.

Note that if there are more than 1535 cylinders on a winchester or 1279 cylinders on a floppy then one of the above bytes will clash with the default data compare byte. Therefore in these circumstances it may be desirable to change the data compare byte to a value which will not be used.

There are a number of parameters which can be changed which will affect the overall format in some way.

Interleave
the interleave can be altered by changing this register.

Skew
the skew can be altered by changing this register.

RWCCylinderBy4
the **notReducedWriteCurrent** pin (**PB0**) will be asserted if **CurrentCylinder1-0 >= RWCCylinderBy4** * 4.

PCCylinderBy4 and **Precompensation1-0**
pre-compensation will be used on tracks for which **CurrentCylinder1-0 >= PCCylinderBy4** * 4. The amount of pre-compensation may be adjusted by altering the values in **Precompensation1-0** and may be disabled by setting **OntimeDelay** = **EarlyDelay** = **LateDelay** = 0. See section 3.11 for further information.

MFMNotFM in **Control**
if this bit is reset then M212 will use FM encoding and decoding. Although this is quite valid, the format generated is not that required by standard single density (FM) floppies.

DataSeparation or **WriteClock**
changing either of these will alter the data rate on the disk. Again, although this is quite valid, the format generated is not that required by standard drives.

Interleave

When it is desired to read or write multiple sectors on a track, either the user process or the controller may not be ready to read the next physical sector on the track. If this sector is just missed, then there will be a wait of a whole disk revolution before the sector can be read. However, the interleave factor can be set up so that the appropriate sector is the next to be read when the user process or controller is ready for it. This value may be optimised by formatting a track with a varying value for **Interleave** and timing how long it takes to process the whole track. If **Interleave** is initially set to 1 and then increased by 1 each time, up to a maximum of **NumberOfSectors** - 1, then a sudden decrease in access time will be noticed for some value of **Interleave**. This is the point at which sectors are just being read in time. Note that this will give the optimum value of **Interleave** only for the particular processing that is being performed for the test, if this increases slightly then the performance cliff edge may be reached, so it may be useful to use a slightly higher value for **Interleave** than indicated by the test. The same **Interleave** should be used for all tracks on a given drive which have a similar format (i.e. those where **SectorSizeLg2** and **NumGap3Bytes** are the same).

As an example, if a track with 16 sectors starting from sector 1 is formatted with an **Interleave** of 3 the sectors will be in the following order on the track (assuming **Skew** is 0):
1 12 7 2 13 8 3 14 9 4 15 10 5 16 11 6.

Skew

When the head is changed or a seek performed, the user process or controller may just miss the required sector. **Skew** is used to provide an offset, in terms of sectors, for where the first logical sector on a track occurs. With **Skew** set to 0, the first logical sector is also the first physical sector on the disk, when **Skew** is set to 1 it will be the second, etc. This value will, in general, be different for different tracks unlike **Interleave** which will remain the same.

As an example, if a track with 16 sectors starting from sector 1 is formatted with an **Interleave** of 3 (as above) but with a **Skew** of 4 then the sectors will be in the following order on the track:
5 16 11 6 1 12 7 2 13 8 3 14 9 4 15 10.

Gap 3

NumGap3Bytes defines the gap between subsequent sectors on a track, and stops the motor speed variation of the drive causing the end of one sector to overwrite the beginning of the next. In general it will not need to be changed from the default value, but there are a few circumstances in which it may be desirable. If the drive has a high motor speed variation then the **NumGap3Bytes** may have to be increased. If, however, the drive has a close motor speed tolerance then **NumGap3Bytes** may be reduced to squeeze in an extra sector. **NumGap3Bytes** might also be made as large as possible to give the controller or user program a bit more time before the sector becomes available, thus allowing a slightly smaller interleave or skew to be used.

Suppose a winchester drive with a motor speed variation of ± 1% is being used with the standard format. This has a total of 301 bytes per sector excluding the gap 3 bytes. If the disk is formatted while it is rotating at its slowest and then a sector written while it is rotating at its fastest then (((301 * 1.01) - (301 * 0.99)) * 1.01) = 6.08 gap 3 bytes will be needed so that **NumGap3Bytes** should be set to a minimum of 7.

Gap 4

This gap is used to fill up the end of a track after all the sector information has been written. The value of **NumGap4BytesBy256** must be sufficient to take motor speed variations into account. If this value is not great enough then a **FormatUnderrun** error will be shown in **Error**. It may mean that previously recorded data is not erased possibly leaving an old sector intact, or that a section of the track has no flux transitions recorded on it which may cause the data separator PLL to lock to an incorrect frequency. The default value should be enough to cope with any standard winchester (10 Kbytes per track) or floppy (6 Kbytes per track) format. If **NumGap4BytesBy256** is too great, then no error will occur, but there will be a longer than necessary delay before any further commands will be accepted by the disk controller hardware following the **FormatTrack** command.

Suppose that a floppy disk with a rotational period of 200 ms ± 3% is being used with the default format. This uses a total of 6098 bytes excluding the gap 4 bytes at 32 us per byte. Thus if the disk is being formatted while it is running slowly then (((200E-3 / 32E-6) * 1.03) - 6098) = 339.5 gap 4 bytes will be needed so that **NumGap4BytesBy256** should be set to a minimum of 2.

Non-standard formats

Some of the above fields may be changed from their default values:

1) the number of sectors per track may be altered by changing **NumberOfSectors**.

3) the missing clock bits in the Index AM (which is #C2) may be changed by altering the value in the hard parameter **IndexAMMissingClock** (the clock pattern of #C2 is #1C). Also the number of AM bytes may be altered by changing the **ValidAddressMarkNumberLess1** field of hard parameter **TimingRegister0**, although all address mark fields use the same number of AM's.

5) the values of the ID AM data and clock may be changed by altering the values in the hard parameters **IDFieldAMData** and **IDAMMissingClock** respectively. Although not used during the format operation itself, **IDFieldAMClock** will have to be changed to reflect the new clock pattern when performing a **ReadSector** or **WriteSector** command. Also the number of AM bytes may be altered by changing the **ValidAddressMarkNumberLess1** field of hard parameter **TimingRegister0**, although all address mark fields use the same number of AM's.

7) the length bits of the winchester head byte may use the alternative numbering scheme by resetting **SectorSizeBy128Lg2** in **DriveType**. In the case of the default 256 byte sectors then the winchester length bits will become #1. See the description of **DriveType** in section 2.3 for further information.

8) sectors may be numbered from 0 instead of 1 by resetting the **SectorsFrom1** bit in **DriveType**.

9) the length byte of a floppy ID may use the alternative numbering scheme by resetting **SectorSizeBy128Lg2** in **DriveType**. In the case of the default 256 byte sectors then the floppy length byte will become #00. See the description of **DriveType** in section 2.3 for further information.

10) ECC may be used by setting **IDFieldCrcEccMode** in the hard parameter **Control** to #1 or CRC may be used by setting **IDFieldCrcEccMode** to #3. The syndrome may be reset instead of preset by resetting **IDPresetNotReset** also in **Control**. The polynomial may be changed by altering **ECCPolynomial3-0** for ECC or **CRCPolynomial1-0** for CRC, although both ID and Data fields share the same ECC or CRC polynomial.

12) the values of the Data AM data and clock may be changed by altering the values in the hard parameters **DataFieldAMData** and **DataAMMissingClock** respectively. Although not used during the format operation itself, **DataFieldAMClock** will have to be changed to reflect the new clock pattern when performing a **ReadSector** or **WriteSector** command. Also the number of AM bytes may be altered by changing the **ValidAddressMarkNumberLess1** field of hard parameter **TimingRegister0**, although all address mark fields use the same number of AM's. The data compare byte may be altered by changing the hard parameter **DataCompare**.

13) the sector size may be changed by altering **SectorSizeLg2**.

14) ECC may be used by setting **DataFieldCrcEccMode** in the hard parameter **Control** to #1 or CRC may be used by setting **DataFieldCrcEccMode** to #3. The syndrome may be reset instead of preset by resetting **DataPresetNotReset** also in **Control**. The polynomial may be changed by altering **ECCPolynomial3-0** for ECC or **CRCPolynomial1-0** for CRC, although both ID and Data fields share the same ECC or CRC polynomial.

15) the number of gap 3 bytes may be changed by using **NumGap3Bytes**.

16) the nominal number of gap 4 bytes may be changed by using **NumGap4BytesBy256**.

Note that if a new format has been decided upon then the disk must be re-formatted using the new format. It is quite possible to format different parts of a disk with different formats so that programs or operating systems requiring different formats can reside in different partitions of a disk.

2.10 ECC, CRC and polynomials

Mode 1 supports two of the four M212 ECC/CRC modes - ECC and CRC. Both provide a check on the integrity of the data read off a disk. ECCs (Error Correcting Codes) allow certain classes of errors to be corrected as well as detected, while CRCs (Cyclic Redundancy Codes) only allow detection. The ECC provided on the M212 is four bytes long and the CRC is two bytes long. The ECC mode can be used to provide a four byte CRC if required, but in mode 1 operation **NumEccCorrectableBits** must be set to zero to indicate that correction is not possible.

Different registers are provided for the ECC and CRC polynomial definition thus allowing ECC and CRC to be used together in a format. For example, the standard winchester format has a CRC on the ID field and an ECC on the Data field. This arrangement is quite convenient since it is not possible to perform a correction on-the-fly for the ID field, but the Data field may be corrected once it has been read. Mode 1 operation allows the formats to be specified with either CRC or ECC in either of the ID or Data fields. This makes it possible to have floppies with correctable Data fields. In order to change the type of polynomial for a field it is necessary to change some bits in the **Control** hard parameter. For example, to use ECC for the Data field, bits 5-4 must be set to #1. See section 3.8 for further information. If either of these fields is not set to ECC or CRC, then an error will be set during a **ReadSector**, **WriteSector** or **FormatTrack** operation.

The syndrome must be initialised to a consistent value for ECC and CRC to work, on the M212 this may either be to all 0's (reset) or all 1's (preset). Presetting is preferable to resetting because if the syndrome is reset, and by accident no data is transmitted, then the ECC/CRC will stay at 0 and so when the syndrome is examined it will appear as though the transfer has completed correctly. With a preset syndrome however, the syndrome will be non-zero indicating that an error has occurred. The default state for all syndromes in mode 1 is preset.

The default ECC provided in mode 1 is $x^{32} + x^{23} + x^{21} + x^{11} + x^2 + x^0$ and the default CRC is $x^{16} + x^{12} + x^5 + x^0$ (the CRC-CCITT polynomial). The polynomial used for the ECC or CRC may be changed by setting **ECCPolynomial3-0** and **CRCPolynomial1-0** respectively. See section 3.8 for further information. For the mode 1 ECC correction algorithm to work the ECC polynomial used must be of the form $x^{32} + ... + x^0$. If the ECC polynomial is changed then **NumEccCorrectableBits** must be changed to the correction span of the polynomial. Note that the correction span may change for different sector lengths and for the required probability of a mis-correction being performed. The ECC provided by default should not be used for sectors longer than 4 Kbytes since its length is 42987 (bits). If it is used for longer sectors then the value in **NumEccCorrectableBits** must be set to zero to prevent a correction being performed in the wrong place.

In mode 1, corrections are attempted during a **ReadSector** command if an ECC syndrome is non-zero and the syndrome is equal on two successive retries. Note, therefore, that the value in **SectorRetries** must be one or greater in order to allow the possibility of a correction being attempted.

2.11 Initialise defaults

The drive defaults set up by **Initialise** are:

characteristic	Default floppy	winchester
number of cylinders	80	612
number of heads	2	4
number of sectors	16	32
has ready	TRUE	TRUE
has seek complete	FALSE	TRUE
head step rate	3 ms	64 us
head settle time	15 ms	0 s
head load time	0 s	0 s
motor start time	0 s	0 s
RWC cylinder	44	132
PC cylinder	0	0
pre-compensation	+/- 125 ns	+/- 12 ns
read data polarity	negative	negative
addressing	logical	logical
increment logical	TRUE	TRUE
increment buffer	FALSE	FALSE
poll	FALSE	TRUE
sector retries	16	16
seek retries	2	2
desired sector buffer	0	0
motor on	TRUE	FALSE
sector size	256	256
first sector	1	1
length by 128 Lg2	TRUE	FALSE
interleave	3	6
skew	0	0
number of gap 3 bytes	54	15
number of gap 4 bytes	3072	3072
encoding	MFM	MFM
data rate	250 kbits/s	5 Mbits/s
number of address marks	3	1
ID address mark	data=#A1, clock=#0A	data=#A1, clock=#0A
ID polynomial type	CRC	CRC
ID polynomial initial value	preset	preset
Data polynomial type	CRC	ECC
Data polynomial initial value	preset	preset
Data address mark	data=#A1, clock=#0A	data=#A1, clock=#0A
Data compare byte	#FB	#F8
CRC polynomial	$x^{16} + x^{12} + x^5 + x^0$	$x^{16} + x^{12} + x^5 + x^0$
ECC polynomial	$(x^{11} + x^2 + x^0)\ (x^{21} + x^0)$	$(x^{11} + x^2 + x^0)\ (x^{21} + x^0)$
number of correctable bits	11	11
write clock required	16 MHz	20 MHz

Note that unless an auto-boot has been performed at reset the M212 will be set up to use its internal clock as the write clock. If it is required to use an external write clock then an **ExternalWriteClock** command must be sent to the disk controller. See section 2.13 for an example of how to do this.

The default soft parameter values after an **Initialise** command are:

Parameter no.	Parameter Name	Default floppy	Default winchester
#00	**DesiredSector**	#00	#00
#01	**DesiredHead**	#00	#00
#02	**DesiredCylinder0**	#00	#00
#03	**DesiredCylinder1**	#00	#00
#04	**LogicalSector0**	#00	#00
#05	**LogicalSector1**	#00	#00
#06	**LogicalSector2**	#00	#00
#07	**Addressing**	#03	#03
	bit 0 - **LogicalAddressing**	#1	#1
	bit 1 - **IncrementLogical**	#1	#1
	bit 2 - **IncrementBuffer**	#0	#0
	bits 3-7 are reserved by INMOS	#0	#0
#08	**DriveType**	#9C	#F5
	bit 0 - **Winchester**	#0	#1
	bit 1 - **WriteProtect**	#0	#0
	bit 2 - **SectorsFrom1**	#1	#1
	bit 3 - **LengthBy128Lg2**	#1	#0
	bit 4 - **HasReady**	#1	#1
	bit 5 - **HasSeekComplete**	#0	#1
	bit 6 - **PollThisDrive**	#0	#1
	bit 7 - **DriveExists**	#1	#1
#09	**SectorSizeLg2**	#08	#08
#0A	**NumberOfSectors**	#10	#20
#0B	**NumberOfHeads**	#02	#04
#0C	**NumberOfCylinders0**	#50	#64
#0D	**NumberOfCylinders1**	#00	#02
#0E	**CurrentCylinder0**	#00	#00
#0F	**CurrentCylinder1**	#00	#00
#10	**RWCCylinderBy4**	#0B	#21
#11	**PCCylinderBy4**	#00	#00
#12	**SectorRetries**	#10	#10
#13	**SeekRetries**	#02	#02
#14	**HeadStepRateIn64us**	#2F	#01
#15	**HeadSettleTimeIn64us**	#EB	#00
#16	**HeadLoadTimeIn0.5ms**	#00	#00
#17	**MotorStartTimeIn4ms**	#00	#00
#18	**Interleave**	#03	#06
#19	**Skew**	#00	#00
#1A	**NumGap3Bytes**	#36	#0F
#1B	**NumGap4BytesBy256**	#0C	#0C
#1C	**NumEccCorrectableBits**	#0B	#0B
#1D	**DesiredSectorBuffer**	#00	#00

The following parameters are not altered by **Initialise** but are included here to complete the list of parameters.

#1E	**DesiredDrive**
#1F	**CurrentDrive**
#20	**Error**
#21	**Reason**
#22	**NumBufferBytesBy256**
#23	**ErrorDrive (M212Version)**
#7F	**ControllerAccess**

The default hard parameter values after an **Initialise** command are:

Parameter no.	Parameter Name	Default floppy	Default winchester
#80	**PIAPortAData**	#7F	#FF
	bits 3-0 - **notDriveSelect[4-1]**	#F	#F *
	bits 6-4 - **notHeadSelect[2-0]**	#7	#7
	bit 7 - **notMotorOn / notHeadSelect[3]**	#0	#1
#81	**PIAPortADirection**	#FF	#FF
#82	**PIAPortAChange**	#00	#00
#83	**PIAPortAPin**	#00	#00
#84	**PIAPortBData**	#FF	#FF
	bit 0 - **notReducedWriteCurrent**	#1	#1
	bit 1 - **notStep**	#1	#1
	bit 2 - **notDirectionIn**	#1	#1
	bit 3 - **notSeekComplete**	#1	#1
	bit 4 - **notTrack000**	#1	#1
	bit 5 - **notReady**	#1	#1
	bit 6 - **notWriteFault**	#1	#1
	bit 7 - **notWriteProtect**	#1	#1
#85	**PIAPortBDirection**	#07	#07
#86	**PIAPortBChange**	#00	#00
#87	**PIAPortBPin**	#00	#00
#88	**DataFieldAMData**	#A1	#A1
#89	**DataFieldAMClock**	#0A	#0A
#8A	**IDFieldAMData**	#A1	#A1
#8B	**IDFieldAMClock**	#0A	#0A
#8C	**DataAMMissingClock**	#04	#04
#8D	**IDAMMissingClock**	#04	#04
#8E	**IndexAMMissingClock**	#08	#00
#90	**IDCompare0**	#00	#00
#91	**IDCompare1**	#00	#00
#92	**IDCompare2**	#00	#00
#93	**IDCompare3**	#00	#00
#94	**IDCompare4**	#00	#00
#95	**IDCompare5**	#00	#00
#96	**IDCompare6**	#00	#00
#97	**IDCompare7**	#00	#00
#98	**DataCompare**	#FB	#F8
#9A	**StatusRegister0**	#00	#00
#9B	**StatusRegister1**	#00	#00
#A0	**CRCPolynomial0**	#21	#21
#A1	**CRCPolynomial1**	#10	#10
#A4	**ECCPolynomial0**	#05	#05
#A5	**ECCPolynomial1**	#08	#08
#A6	**ECCPolynomial2**	#A0	#A0
#A7	**ECCPolynomial3**	#00	#00
#A8	**SyndromeByte0**	#00	#00
#A9	**SyndromeByte1**	#00	#00
#AA	**SyndromeByte2**	#00	#00
#AB	**SyndromeByte3**	#00	#00
#AC	**Control**	#FF	#DF
	bit 0 - **DataPresetNotReset**	#1	#1
	bit 1 - **IDPresetNotReset**	#1	#1
	bit 2 - **MFMNotFM**	#1	#1
	bit 3 - **EnablePrecompensation**	#1	#1
	bits 5-4 - **DataFieldCrcEccMode**	#3	#1
	bits 7-6 - **IDFieldCrcEccMode**	#3	#3

#B0	**Precompensation0**	#12	#52
	bits 3-0 - **OntimeDelay**	#2	#2
	bit 4 - **DivideBy2Not4**	#1	#1
	bit 5 - **TestMode**	#0	#0
	bit 6 - **WinchesterNotFloppy**	#0	#1
	bit 7 - **ExternalPrecompensation**	#0	#0
#B1	**Precompensation1**	#40	#40
	bits 3-0 - **EarlyDelay**	#0	#0
	bits 7-4 - **LateDelay**	#4	#4
#B2	**DataSeparation**	#2F	#0A
	bits 2-0 - **VCOFrequencyDivision**	#7	#2
	bits 5-3 - **WriteClockReferenceDivision**	#5	#1
	bit 6 - **PositiveReadData**	#0	#0
	bit 7 - **ExternalDataSeparation**	#0	#0
#B4	**TimingRegister0**	#12	#0C
	bits 1-0 - **ValidAddressMarkNumberLess1**	#2	#0
	bits 4-2 - **CheckableIDBytesLess1**	#4	#3
	bits 7-5 - **TimeoutIndexNumberLess2**	#0	#0
#B5	**TimingRegister1**	#09	#09
	bits 2-0 - **SectorSize**	#1	#1
	bit 3 - **DataCompareEnable**	#1	#1
	bits 7-4 - **SoftwareCrcEccSize**	#0	#0
#B6	**DelayAfterIDLess3**	#13	#00
#B7	**RESERVED**	#00	#00
#B8	**UncheckableIDBytes**	#00	#00
#B9	**Operation**	#00	#00
#BA	**ReadSpecial0**	#00	#00
#BB	**ReadSpecial1**	#00	#00
#BC	**ValidAMCount**	#00	#00

Note that the addresses above are those that must be used for mode 1 operation, the true register addresses used when in mode 2 or using **ControllerAccess** do not have bit 7 set.

* The value of this field will depend on the drive number being initialised. The value shown is that for drive 0, other values are as follows:

drive number	**notDriveSelect**
1	#E
2	#D
3	#B
4	#7

2.12 Hardware requirements

The M212 can operate with very little external hardware, particularly in mode 1 since no external RAM or ROM is required. See section 9 for a circuit of a minimal and an enhanced system.

A write clock may be required, but this can either be shared with the processor clock if the frequencies are appropriate e.g. with a 5 Mbits/s winchester a 10 or 20 MHz write clock can be divided down to give the 5 MHz processor clock or, if an M212-20 is being used, ProcClockOut can be used as the write clock for a 5 Mbits/s winchester.

A few passive filter components are required if either winchesters only or floppies only are being used, with a switchable filter being required if both drive types are to be used together.

The disk control outputs are all negative-true, thus the controller can be directly connected to a drive with no buffers being required (except for a differential driver and receiver for winchester write and read data). However, the output pins' drive capability must not be exceeded and the connections must be kept short to avoid impedance matching problems. If buffers are required then non-inverting, open collector versions should be used. In all cases the input signals to the controller should be terminated with resistors in an appropriate manner.

The mode 1 process will use 1280 bytes of the M212 internal memory for its sector buffer, but if external RAM is provided from address #8800 upwards then the M212 will extend its sector buffer into this area in blocks of 256 bytes. The method used is to try writing a value (#656A) into the top word of each block (#88FF-E, #89FF-E, #8AFF-E, ...). If the value read back is the same then the block is added to the sector buffer and the next block tested. This is repeated until either the test does not succeed or the bottom of the internal ROM is reached (#7000). If there is any memory mapped hardware, then the process must still be able to recognise the end of RAM. Therefore, if there is a device memory mapped into the top word of the block following the end of RAM, then it must not matter if the value is written to it and it must not return the same value when read. Alternatively, if the user process knows the true amount of RAM available then it can reset **NumBufferBytesBy256** to the correct value.

There is one spare I/O signal which may be used, **notWriteFault** (**PB6**). It is not used by the mode 1 process, but can be used to directly monitor a drive's Write Fault or other signal. It is initially set up as an input, but alternatively it may be programmed as an output. Note, however, that the PIA control registers get swapped with the drives and so it is necessary to ensure that the appropriate direction is set for all drives that will be selected (including drive 0). If outputting then the data register may also have to be set up for all drives that will be selected.

The following table gives the translation of PIA port bits to mode 1 functions and whether the pin is an input (I) or an output (O).

pin	PIA port A		PIA port B	
0	(Output)	**notDriveSelect[1]**	(Output)	**notReducedWriteCurrent**
1	(Output)	**notDriveSelect[2]**	(Output)	**notStep**
2	(Output)	**notDriveSelect[3]**	(Output)	**notDirectionIn**
3	(Output)	**notDriveSelect[4]**	(Input)	**notSeekComplete**
4	(Output)	**notHeadSelect[0]**	(Input)	**notTrack0**
5	(Output)	**notHeadSelect[1]**	(Input)	**notReady**
6	(Output)	**notHeadSelect[2]**	(Input)	**notWriteFault**
7	(Output)	**notMotorOn** *	(Input)	**notWriteProtect**

* This signal becomes **notHeadSelect[3]** for a drive with **Winchester** set. For winchester drives with 8 or less heads this signal will always be FALSE while the winchester is selected, therefore any floppy drives will have **MotorOn** also FALSE. However, if a winchester with more than 8 heads is being used then any floppies attatched may be switched on when heads 8-15 of the winchester are selected. If this happens and it is undesirable then the **MotorOn** signal will have to be gated with the floppy drive select signal(s).

2.13 Disk controller access

This section shows a few possiblities of useful code sequences which access the disk controller hardware directly using **ReadParameter** and **WriteParameter** of **ControllerAccess**.

Up to eight bytes of data can be stored in the disk controller output FIFO before it need be read or before an overrun occurs. Great care should be taken if using these versions of the commands since if no data transfer can take place then the process will hang.

Complicated read hard parameter

This performs the same function as a **ReadParameter** of HardParameter but by sending the explicit commands required by the disk controller hardware.

```
ToM212 ! WriteParameter; ControllerAccess; BYTE (#C0 \/ HardParam)
ToM212 ! ReadParameter; ControllerAccess
FromM212 ? HardParameterDataByte
```

Complicated write hard parameter

This performs the same function as a **WriteParameter** of HardParameter but by sending the explicit commands required by the disk controller hardware.

```
ToM212 ! WriteParameter; ControllerAccess; BYTE (#80 \/ HardParam)
ToM212 ! WriteParameter; ControllerAccess; HardParameterDataByte
```

Switch over to using an external write clock

```
ToM212 ! WriteParameter; ControllerAccess; ExternalWriteClock
```

Read ID

This assumes that there are less than eight ID bytes so that they can be fitted into the FIFO.

```
ToM212 ! WriteParameter; Operation; ReadIDField
ToM212 ! WriteParameter; ControllerAccess; Start
SEQ IDByte = 0 FOR NumIDBytes
  SEQ
    ToM212 ! ReadParameter; ControllerAccess
    FromM212 ? IDByteArray[IDByte]
```

Erasing a whole track of data

This fills the whole track with bytes of #00 i.e. not even any sectors. It assumes that there are less than 32 Kbytes per track. A similar effect could be achieved by using a standard mode 1 **FormatTrack** command but this would leave initialised sectors on the disk.

```
ToM212 ! WriteParameter; Operation; Format
ToM212 ! WriteParameter; ControllerAccess; Start
ToM212 ! WriteParameter; ControllerAccess; BYTE (RepeatData \/ #1F)
ToM212 ! WriteParameter; ControllerAccess; 0 (BYTE)
```

2.14 Sample declarations and code sequences

A typical set of declarations might include the following:

```
VAL EndOfSequence     IS #00 (BYTE) :
VAL Initialise        IS #01 (BYTE) :
VAL ReadParameter     IS #02 (BYTE) :
VAL WriteParameter    IS #03 (BYTE) :
... other commands
VAL DesiredDrive      IS #1E (BYTE) :
VAL ECCPolynomial0    IS #A4 (BYTE) :
... other parameters
VAL FloppyDrive       IS   3 (BYTE) :
VAL WinnieDrive       IS   1 (BYTE) :
VAL Floppy            IS   0 (BYTE) :
VAL Winnie            IS   1 (BYTE) :
VAL Lg1024            IS  10 (BYTE) :
VAL f.DriveType.std   IS #9C        :
VAL HasReady          IS #10        :
... other values
VAL LinkToM212        IS 0          :
VAL LinkFromM212      IS 4          :
CHAN OF BYTE ToM212, FromM212       :
PLACE ToM212   AT LinkToM212        :
PLACE FromM212 AT LinkFromM212      :
```

and a typical sequence of commands to initialise a winchester and a floppy might be:

```
-- initialise drive 1 as a standard winchester
-- with 500 cylinders but using an ECC polynomial of
-- x^32 + x^28 + x^26 + x^19 + x^17 + x^10 + x^6 + x^2 + x^0
ToM212 ! WriteParameter; DesiredDrive; WinnieDrive
ToM212 ! Initialise; Winnie
ToM212 ! WriteParameter; NumberOfCylinders1; BYTE (500 / 256)
ToM212 ! WriteParameter; NumberOfCylinders0; BYTE (500 \ 256)
ToM212 ! WriteParameter; ECCPolynomial3; #14 (BYTE)
ToM212 ! WriteParameter; ECCPolynomial2; #0A (BYTE)
ToM212 ! WriteParameter; ECCPolynomial1; #04 (BYTE)
ToM212 ! WriteParameter; ECCPolynomial0; #45 (BYTE)
ToM212 ! WriteParameter; NumEccCorrectableBits; 8 (BYTE)
ToM212 ! Restore

-- initialise drive 3 as a standard 5.25" floppy without a ready line,
-- a motor start time of 0.5s, 4 sectors each of 1024 bytes per track
ToM212 ! WriteParameter; DesiredDrive; FloppyDrive
ToM212 ! Initialise; Floppy
ToM212 ! WriteParameter; DriveType; BYTE (f.DriveType.std /\ (~HasReady))
ToM212 ! WriteParameter; MotorStartTimeIn4ms; 125 (BYTE)
ToM212 ! WriteParameter; NumberOfSectors; 4 (BYTE)
ToM212 ! WriteParameter; SectorSizeLg2; Lg1024
ToM212 ! Restore

ToM212 ! ReadParameter; Error
FromM212 ? ErrorByte
... check for errors
```

This code formats the winchester.

```
-- initialise drive 1 as a winchester
ToM212 ! WriteParameter; DesiredDrive; WinnieDrive
ToM212 ! Initialise; Winnie
ToM212 ! Restore

-- format the complete winchester
SEQ Track = 0 FOR 4 * 500
  ToM212 ! FormatTrack

-- check status to see if it all worked
ToM212 ! ReadParameter; Error
FromM212 ? Status
```

This code retrieves the contents of the floppy onto a winchester (both as above). Note that it has not been necessary to transfer any data to or from the host. If any external RAM is available then this can be done by transferring more than one sector at a time e.g. a whole floppy track. The size and format of the drives could be read in from the drives' parameter files for more generalised code.

```
-- set winchester to increment buffer mode and set logical address
-- to wherever we want which is sector #003F12 onwards in this example
ToM212 ! WriteParameter; DesiredDrive; WinnieDrive
ToM212 ! SelectDrive
ToM212 ! WriteParameter; Addressing; 7 (BYTE)
ToM212 ! WriteParameter; LogicalSector2; #00 (BYTE)
ToM212 ! WriteParameter; LogicalSector1; #3F (BYTE)
ToM212 ! WriteParameter; LogicalSector0; #12 (BYTE)

SEQ i = 0 FOR (2 * 80) * 4
  SEQ
    -- select floppy and read a sector
    ToM212 ! WriteParameter; DesiredDrive; FloppyDrive
    ToM212 ! ReadSector

    -- select winnie and write 4 sectors
    ToM212 ! WriteParameter; DesiredDrive; WinnieDrive
    SEQ Sector = 0 FOR 4
      ToM212 ! WriteSector
    ToM212 ! WriteParameter; DesiredSectorBuffer; 0 (BYTE)

-- check status to see if it all worked
ToM212 ! ReadParameter; Error
FromM212 ? Status
```

The drives have been initialised they can now be used for reading and writing:

```
-- change floppy to use cylinder/head/sector addressing
-- and read sector 2 on head 1 cylinder 36 into buffer
ToM212 ! WriteParameter; DesiredDrive; FloppyDrive
ToM212 ! SelectDrive
ToM212 ! WriteParameter; Addressing; 0 (BYTE)
ToM212 ! WriteParameter; DesiredSector; 2 (BYTE)
ToM212 ! WriteParameter; DesiredHead; 1 (BYTE)
ToM212 ! WriteParameter; DesiredCylinder1; 0 (BYTE)
ToM212 ! WriteParameter; DesiredCylinder0; 36 (BYTE)
ToM212 ! ReadSector
ToM212 ! ReadBuffer
FromM212 ? [ByteBuffer FROM 0 FOR 1024]
```

```
-- write winchester logical sector 18 from buffer
ToM212 ! WriteParameter; DesiredDrive; WinnieDrive
ToM212 ! SelectDrive
ToM212 ! WriteParameter; LogicalSector2; 0 (BYTE)
ToM212 ! WriteParameter; LogicalSector1; 0 (BYTE)
ToM212 ! WriteParameter; LogicalSector0; 18 (BYTE)
ToM212 ! WriteBuffer; [ByteBuffer FROM 0 FOR 256]
ToM212 ! WriteSector

ToM212 ! ReadParameter; Error
FromM212 ? ErrorByte
... check for errors
```

Note that if two processes are sharing the M212 then care must be taken that the correct parameter file is being used or updated. If one process has changed the **CurrentDrive** between 2 sequences of another process then **DesiredDrive** must be updated and, before any parameters are changed, a **SelectDrive** command issued. The provision of indivisible sequences on the links allows, for example, a read-modify-write cycle to be easily implemented which could be useful for multi-user database applications.

2.15 Getting started

This section is intended to provide a few hints on possible problems that may arise when first using the M212 in mode 1.

Communication with M212

All commands and data communicated to and from the M212 must be in terms of **BYTE**'s.

Write clock

The write clock is used as the disk controller logic clock whenever it is not actually reading from the disk and so must be present at all times. It can be selected from one of two sources - internal or external. Because the external write clock will not always be present, when first reset the M212 uses its internal clock as the write clock. If it is required to use an external clock then this must be explicitly selected. See section 2.13 for an example of how to do this. Note that if an auto-boot is performed then the external clock is automatically selected and the correct frequency must be available. The internal write clock frequency available is 30 MHz for an M212-15 and 40 MHz for an M212-20.

Drive selection

Check that only one drive is selected at a time, that the correct drive is selected and that the parameter file is initialised for the correct type of drive (winchester or floppy). Also check that the drive signals are connected to the correct M212 pins and are of the correct polarity. In particular **ReadData** and **notIndex** must be connected since they are used for timing functions by the M212 and if these signals do not occur then the device may hang during a **ReadSector**, **WriteSector** or **FormatTrack** command.

Drive parameters

Check that the parameters are set up appropriately for the drives in use, particularly the bits in **DriveType** and stepping rates etc. If attempting to read disks which have been formatted elsewhere, check that the expected format is correct, including parameters such as the CRC and ECC polynomials and whether they are reset or preset.

DelayAfterIDLess3

This hard parameter defines the delay after the end of an ID before the write update is started (in the case of a **WriteSector**) or before the data field will be looked for (in the case of a **ReadSector**). Accessing disks with standard formats will cause no problems, but if a non-standard delay has been used by a controller on the disk then the value of **DelayAfterIDLess3** may have to be changed. If the disk has too short a delay then two things can happen. Firstly, if doing a read then the data field AM will not be looked for quickly enough and will be missed and secondly if doing a write then the old AM will not be over-written but the following sector may be. If the disk has too long a delay then the same problems will occur in reverse when the disk is transferred back to the other controller.

Auto-boot selection

The auto-boot pins (**notHeadSelect[2-0]**) must be biassed to the correct state by resistors, not hard-wired, although if auto-boot will never be used then **notHeadSelect[1-0]** need not be biassed. If the auto-boot option is being used then the boot disk must have compatible characteristics with the drive defaults. See section 2.8 and 2.11 for further information. Note that the auto-boot drive must be ready within 1 second of **Reset** being de-asserted otherwise it will time out. Thus the boot winchester should be allowed to get up to speed or the boot floppy should already be inserted in the drive before **Reset** is de-asserted. Also the code must be written on the disk in logical address number order with the appropriate logical sector 0 header information.

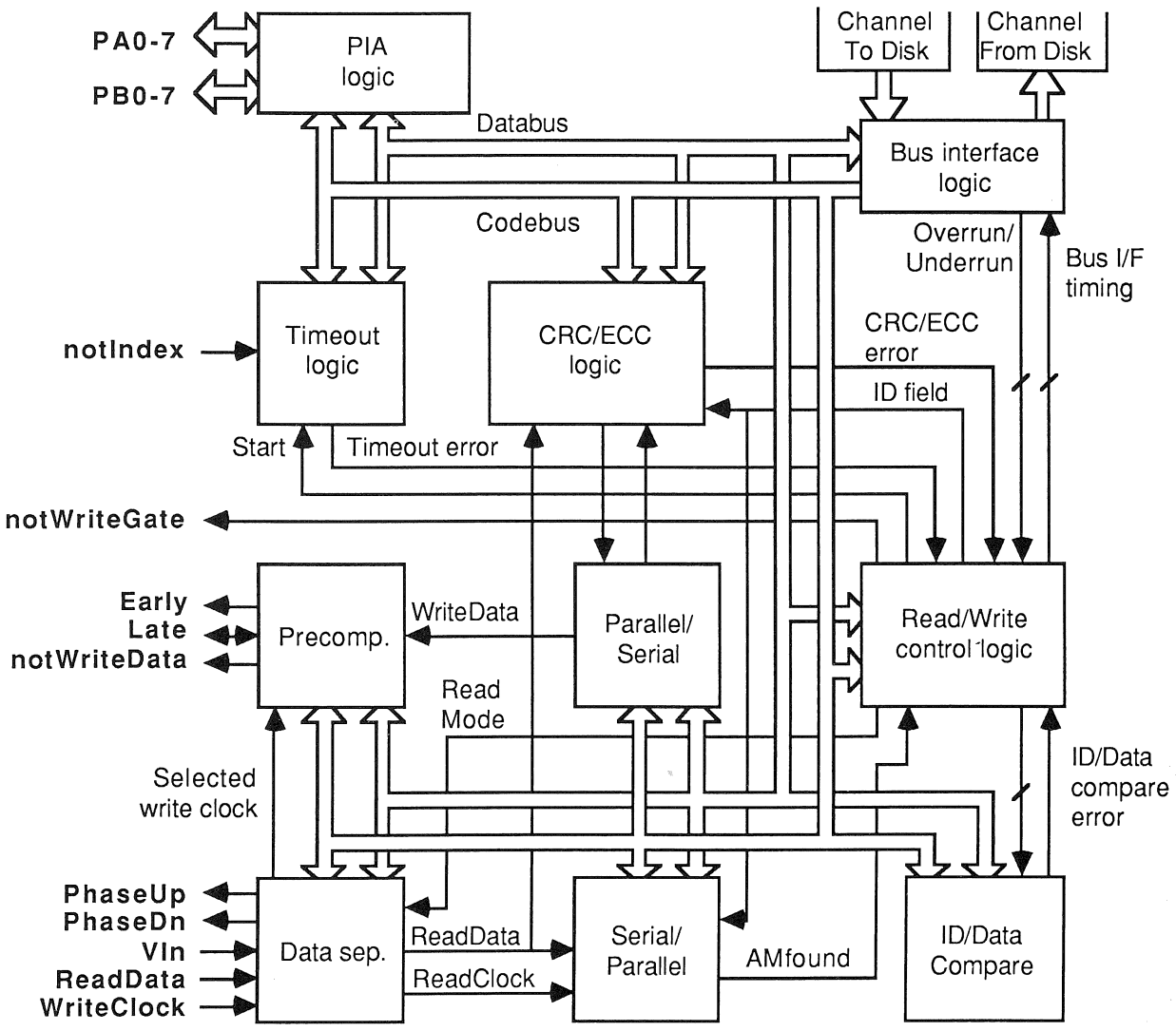
3.1 Overview

In mode 2 operation the M212 internal ROM is bypassed, allowing the device to be used with user-defined software. This software can be held in external ROM (with the internal ROM being disabled), or can be booted from a floppy or winchester disk (2.4.15, 2.8). Alternatively the software can be provided by booting from a link into external or internal RAM. The programmable disk controller registers (hard parameters) and control logic are then directly accessed via a pair of high bandwidth dedicated channels, designated as "ChannelToDisk" and "ChannelFromDisk". These appear to the M212 processor as a normal pair of hardware channels, and might be defined in an occam program as:

```
CHAN OF BYTE ChannelToDisk, ChannelFromDisk :
PLACE ChannelToDisk   AT 2 :
PLACE ChannelFromDisk AT 6 :
```

Sequences of control codes and data bytes are sent to the disk controller logic via ChannelToDisk, and read data is sent back to the M212 processor via ChannelFromDisk. The M212 has been designed so that the on chip processor performs as many functions as possible, which maintains flexible operation and minimises the on chip specific hardware. Each functional unit on the following block diagram has a section in this chapter (3.3 to 3.12) explaining its function and outlining the registers contained in that section. The disk hardware contains 49 registers (summarised in appendix B) all of which are accessible from the processor. The control sequences issued by the processor which control the disk hardware are detailed in section 3.2.

Disk Controller Hardware Logic



3.2 Control code format

Each control code is a single byte, and may be followed by one or more data bytes. Note that undefined codes should not be used as they have an unspecified effect. The format of the control codes is as follows:

3.2.1 Read register: <11xxxxxx>

Reads from the disk controller register addressed by the least significant six bits of the code. The resultant single data byte is sent to the M212 processor via the ChannelFromDisk. Reading from a non-existent register returns a value of #00. A summary of the disk controller registers is given in Appendix B.

3.2.2 Write register: <10xxxxxx>, <data>

Writes the subsequent data byte to the disk controller register addressed by the least significant six bits of the code. Writing to a non-existent register has no effect on the logic. A summary of the disk controller registers is given in Appendix B.

3.2.3 Command code (zero data): <000xxxxx>

These codes perform the function defined by the least significant five bits of the code. There are nine valid 'zero data' command codes:

#00 NOP
The disk controller logic performs no operation when this command is received.

#01 Start
Causes the disk controller logic to perform an access to the disk, as defined in the **Operation** register. If a Write Sector or Format operation is defined, this command will be followed by further write data and control bytes. Any of the 'Read' operations will result in data being sent back to the M212 processor.

#04 SwitchOffWriteGate
Issued during a Write Sector operation to switch off the write gate. This command effectively terminates the Write Sector operation.

#05 SetTestMode
Forces both PIA ports to output for test purposes. The disk controller databus is driven onto the **PA0-7** pins, and the disk controller codebus is driven onto the **PB0-7** pins.

#06 ResetTestMode
Restores the PIA ports to their normal operation. This is the default state after the **Reset** pin is asserted.

#07 ExternalWriteClock
Causes the disk controller logic to be clocked from an external write clock reference frequency, which is driven onto the **WriteClock** pin.

#08 InternalWriteClock
Causes the disk controller logic to be clocked from an internal reference frequency which is twice the frequency of the M212 processor clock. This is the default state after the **Reset** pin is asserted.

#12 Append2SyndromeBytes
Issued during a Write Sector or Format operation to write two hardware generated CRC bytes to the disk.

#14 Append4SyndromeBytes
Issued during a Write Sector or Format operation to write four hardware generated ECC bytes to the disk.

3.2.4 Command code (single data): <001xxxxx>, <data>

These codes perform the function defined in the least significant five bits of the code, and are followed by a

single data byte. There are four valid single data command codes:

#20 ToggleIDField

This command code is used during a format operation to toggle the state of the CRC/ECC logic when either the ID field or the Data field has a software generated syndrome.
During a normal Format operation (when both ID field and Data field CRC/ECC syndromes are being hardware generated) a bit of internal state is toggled, which indicates whether the current field is ID or Data, whenever an **Append2SyndromeBytes** or **Append4SyndromeBytes** command is issued. However if one of the fields has a hardware generated syndrome and the other has a software generated syndrome then the toggling will not be performed on every field because no **AppendnSyndromeBytes** command is issued for the software generated syndrome. In this case the internal state must be toggled explicitly using a **ToggleIDField** command.
The **ToggleIDField** command must be followed by a single data byte, which will be written to the disk in the normal way as the command is being executed. As this command will be issued at the end of an ID field or at the end of a Data field during a Format operation then the value of this data will normally be that of a gap byte.

#31 WriteDataAM

Used during a Write Sector or Format operation to write a Data field Address Mark (AM) byte. This command causes the following data byte to have its FM or MFM coding violated according to the contents of the DataAMMissingClock register. Each bit set in that register will cause a clock pulse to be deleted from the corresponding bit cell in the AM byte. See Appendix A for details of AM coding.

#35 WriteIDAM

Similar to the **WriteDataAM** command, except that the pattern in the IDAMMissingClock register masks out clock pulses from the following address mark data byte.

#39 WriteIndexAM

Similar to the **WriteDataAM** command, except that the pattern in the IndexAMMissingClock register masks out clock pulses from the following address mark data byte. This command can be used for writing Index field address marks, which certain disk formats require.

3.2.5 Repeat data code: <010xxxxx>, <data>

Used during a Write Sector or Format operation, when the following data byte is repeatedly written to the disk. The number of times that the data byte is written to disk is defined by the count value in the least significant five bits of the code, as shown below:

Bits 4-0	Count Value
0 0 0 0 0	16
0 0 0 0 1	1
0 0 0 1 0	2
. . . . .	.
. . . . .	.
0 1 1 1 1	15
1 0 0 0 0	1
1 0 0 0 1	2
1 0 0 1 0	4
1 0 0 1 1	8
. . . . .	.
. . . . .	.
1 1 1 1 1	32768

Note that, when bit 4 of the code is low, the count value is interpreted as the binary value of bits 3-0 of the code, except for the value '0000' which is interpreted as a count value of 16. When bit 4 of the code is high, the count value is interpreted as 2 to the power of the least significant four bits of the code. For example, to write fourteen #00 sync bytes to the disk during a write operation, the byte sequence **#4E**, **#00** would be issued to the disk controller logic.

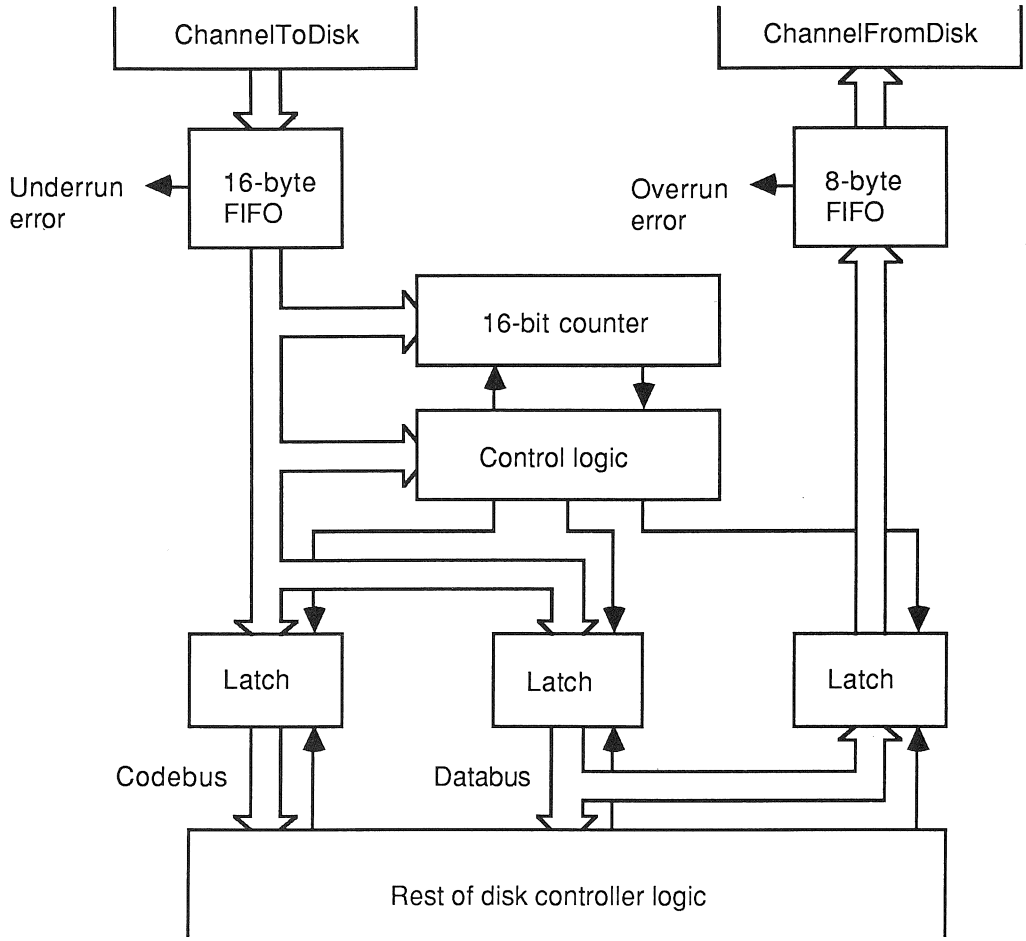
3.2.6 Multiple data code: <011xxxxx>, <data>, ... <data>

Used during a Write Sector or Format operation, when a number of differing data bytes are to be written to disk. The number of data bytes to be written is defined by the count value in the least significant five bits of the code, which is interpreted in the same way as for **repeat data** codes. For example, to write the bytes #00, #01, #02 to the disk during a write operation, the byte sequence **#63**, **#00**, **#01**, **#02** would be issued to the disk controller logic.

3.3 Bus interface logic

All message transfers between the M212 processor and the disk controller logic are controlled by the bus interface logic.

Bus Interface Logic Block Diagram



Messages sent by the M212 processor are first buffered in a 16-byte FIFO register. The bus interface control logic then decodes the message bytes, and constructs codes and code/data sequences according to the M212 control code format. Each code or code/data sequence is then synchronised to the disk controller clock domain, and driven onto the disk controller codebus and bidirectional databus for one byte-long period.

A 16-bit counter is incorporated in the bus interface logic to automatically count data bytes during a **repeat data** or **multiple data** type code.

When a **read register** code is issued, or a 'Read' operation is performed on the disk, data bytes are sent from the disk controller databus to the ChannelFromDisk via an 8-byte FIFO register in the bus interface logic.

3.4 PIA ports

The M212 has two general purpose byte-wide PIA ports, independently controlled via the disk controller codebus and databus, designated as PIA port A and PIA port B. The port pins **PA0-7** and **PB0-7** are used to interface to the disk drive control lines, and are configured by writing to and reading from eight byte-wide registers. These registers are:

PortAData	**PortBData**
PortADirection	**PortBDirection**
PortAChange	**PortBChange**
PortAPin	**PortBPin**

The direction registers control the direction of each individual PIA pin, a '1' indicating output, and a '0' indicating input. These registers are reset to #00 when the **Reset** pin is asserted.

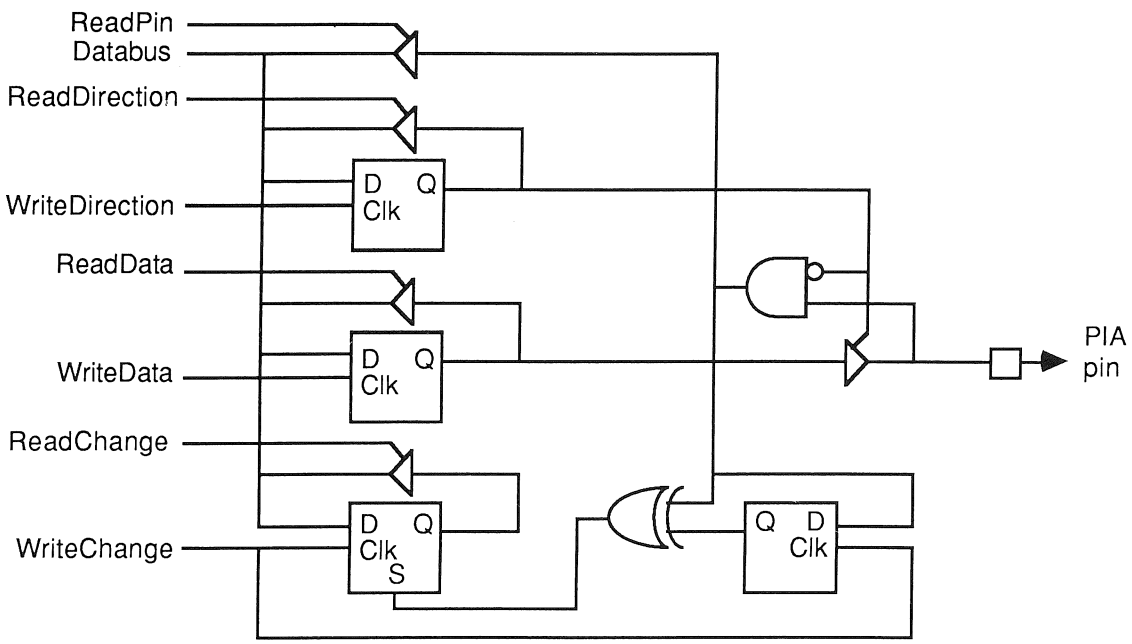
When a particular PIA pin is configured to output, the data in the corresponding bit in the data register is driven onto the output.

The PIA ports can be read by reading the **PortAPin** and **PortBPin** registers. This does not actually read the contents of a physical register, but directly reads the data on the PIA pins (writing to these 'registers' has no effect). Any pin which is configured to output will return a value of '0' when read.

When a change register is written, the data on the associated input pins is simultaneously sampled into the change detection circuitry. By subsequently reading the change register, a program is able to detect transitions or pulses that have occurred on those input pins. Hence writing to a change register is used as both a means of resetting or altering bits within a change register, and as a means of re-priming the change detection circuitry.

The PIA ports can be set into a test mode for debugging purposes by issuing a **SetTestMode** command. This causes the PIA registers to be ignored, and forces both PIA ports to output. The disk controller databus is then driven onto the **PA0-7** pins, and the disk controller codebus is driven onto the **PB0-7** pins, allowing code and data sequences in the disk controller logic to be examined. Normal operation of the PIA ports is resumed when a **ResetTestMode** command is issued, or when the **Reset** pin is asserted.

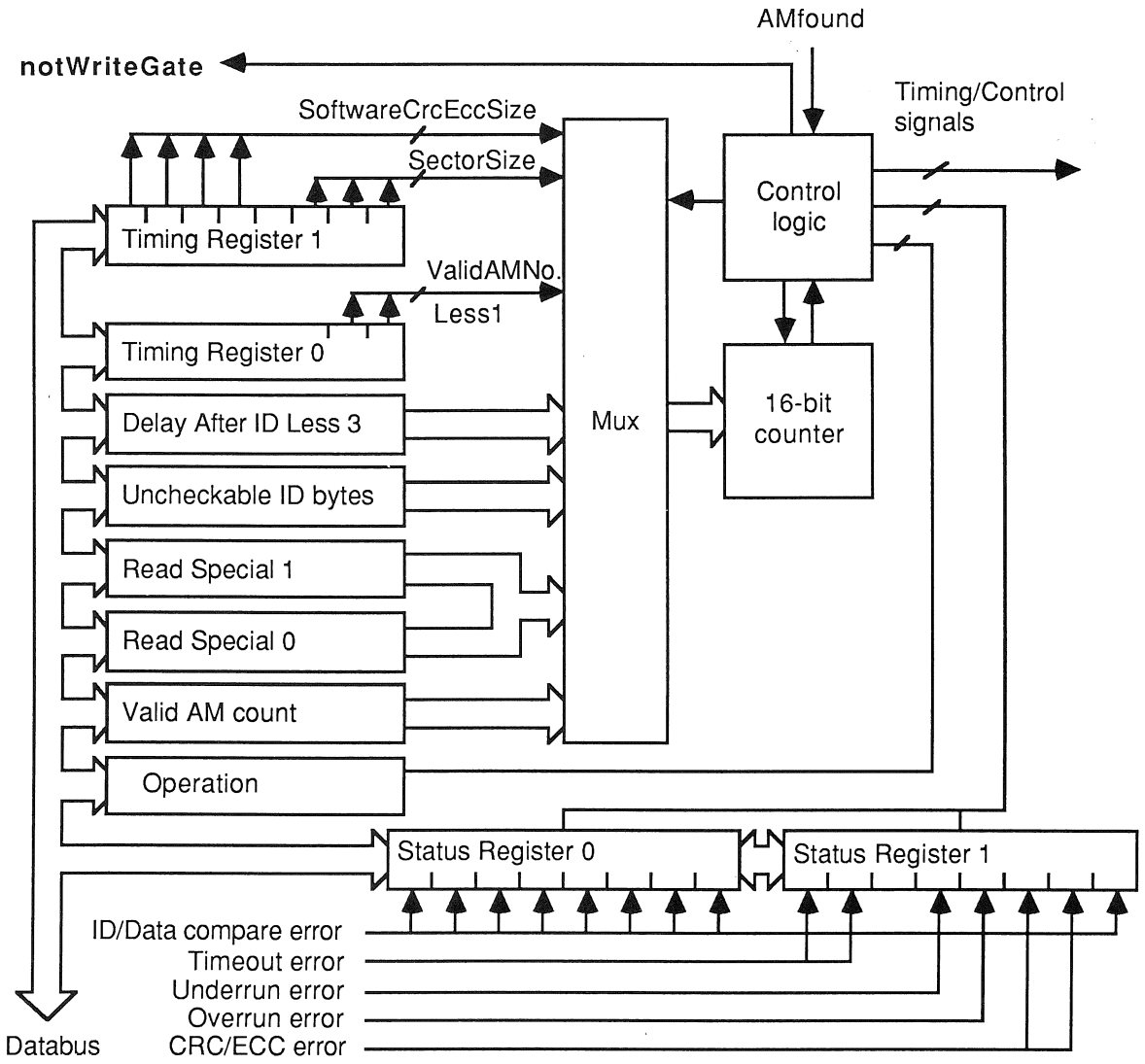
PIA Pin Logic Block Diagram



3.5 Read/Write control

The read/write control logic contains a number of registers whose contents determine the timing and control of read and write operations to the disk. Two byte-wide status registers are also provided, which flag errors that occur during a disk read or write operation. A diagram of the logic is shown below, along with a diagram of a typical sector format, showing the various programmable timing parameters:

Read / Write Control Logic

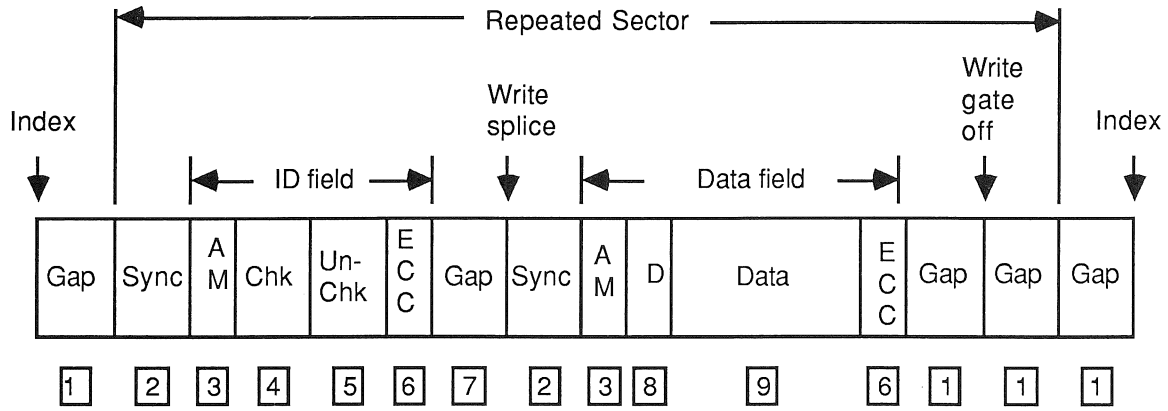


3.5.1 Read/write timing and control registers

Operation
The **Operation** register describes the type of disk read or write operation to be performed when a **Start** command is issued (3.5.2).

TimingRegister0 bits 1-0: ValidAddressMarkNumberLess1
This 2-bit value is one less than the number of AM bytes in a valid AM group. A valid AM group can contain between one and four AM bytes.

Typical Track Format



1 Programmable gap bytes

2 Programmable sync bytes

3 Address mark group (1-4 address mark bytes)

4 1-8 checkable ID bytes

5 0-255 uncheckable ID bytes

6 CRC or ECC bytes

7 3-258 delay bytes before write splice

8 Optional Data Ident byte

9 128-16384 sector data bytes

TimingRegister0 bits 4-2: CheckableIDBytesLess1
This 3-bit value is one less than the number of checkable bytes in an ID field. The ID compare logic can therefore check between one and eight bytes when searching for a particular ID field (3.9).

TimingRegister0 bits 7-5: TimeoutIndexNumberLess2
This 3-bit binary value is two less than the number of index pulses that will occur before an operation is timed out. The number of index pulses before a timeout can therefore be set to between two and nine (3.12).

TimingRegister1 bits 2-0: SectorSize
This 3-bit value 'n' defines a sector size of 2^{n+7} bytes. The sector size can therefore be set to between 128 and 16384 bytes.

TimingRegister1 bit 3: DataCompareEnable
When set, this bit forces a check on the Data ident byte during any of the two Read Sector operations. If no Data ident byte is required, this bit should be reset (3.9).

TimingRegister1 bits 7-4: SoftwareCrcEccSize
This 4-bit value defines the number of extra bytes (between one and fifteen) to be read at the end of an ID or Data field when performing one of the two Read Long operations. This is for when the user wishes to use software-generated CRC/ECC syndromes.

DelayAfterIDLess3
This 8-bit value is three less than the gap between the end of the ID field CRC/ECC bytes and the write splice, which marks the point between an ID field and a Data field when performing any Read Sector or Write Sector operation. This value allows a gap of between 3 and 258 bytes.

UncheckableIDBytes
As well as checkable bytes, the ID field may be configured to contain between 0 and 255 uncheckable bytes before the CRC/ECC bytes, as defined by the value in this register.

ReadSpecial0, ReadSpecial1
This 16-bit value in these two registers defines the number of bytes to be read during a Read Special or Read Special Indexed operation. The **ReadSpecial1** register contains the most significant eight bits of this number, the **ReadSpecial0** register contains the least significant eight bits. Note that a value of #0000 is interpreted as 2^{16}, or 65536.

ValidAMCount
This register holds the number of valid ID field AM groups to be skipped before a read is attempted during a 'Read Special Indexed' operation. This value must not be larger than the number of AM groups on a track less one.

StatusRegister0 bits 7-0: IDCompareError
Each bit set indicates an error in one of the ID field checkable bytes. Between one and eight bytes can be checked during an ID compare sequence.

StatusRegister1 bit 0: DataCompareError
Indicates that the Data ident byte comparison (if enabled) failed during one of the 'Read Sector' operations.

StatusRegister1 bit 1: IDCrcEccError
Indicates that an ID field CRC/ECC error has occurred. This bit will not be set if the CRC/ECC logic is disabled on the ID field.

StatusRegister1 bit 2: DataCrcEccError
Indicates that a Data field CRC/ECC error has occurred. This bit will not be set if the CRC/ECC logic is disabled on the Data field.

StatusRegister1 bit 3: Overrun
During a 'Read' operation, the 8-byte FIFO in the bus interface has overrun. This normally indicates that ChannelFromDisk was not scheduled in time to accept read data from the disk, or that insufficient memory bandwidth was available to ChannelFromDisk. The latter can be caused by a combination of excessively slow RAM, and competition with the other hardware channels of an equal or higher priority. This error should not occur with reasonable hardware/software design.

StatusRegister1 bit 4: Underrun
When writing to the disk, the 16-byte FIFO in the bus interface has underrun. As with the case of overrun errors, this error should not occur if ChannelToDisk is allowed sufficient memory bandwidth.

StatusRegister1 bit 5: InvalidAMGroup
When reading from the disk, an AM group was found which had fewer AM bytes than defined in **TimingRegister0** bits 1-0.

StatusRegister1 bit 6: NoValidAMGroup
Indicates that no valid AM group was found before a timeout error occurred.

StatusRegister1 bit 7: Timeout
Indicates that an operation has not completed within a defined number of disk revolutions, causing a timeout error.

3.5.2 Operation code description

The contents of the Operation register determine the type of disk read/write access that is performed when a **Start** command is issued. The various types of operations that can be performed are:

Code	Operation
#00	Format
#01	Write Sector
#08	Read Special
#0A	Read ID Field
#0B	Read ID Field Long
#0C	Read Sector

#0E Read Sector Long
#18 Read Special Indexed

#0C Read Sector
The Read Sector operation is used to transfer sector data from the disk to the M212 processor. When the **Start** command is issued, the disk controller logic waits for a valid ID field AM group, then performs an ID comparison on the checkable ID bytes. At the end of the ID field, the CRC/ECC bytes will be checked (if enabled). If either the ID comparison or CRC/ECC check fails, the status registers will be reset and another attempt to find the correct ID field will be made. This continues until either the correct ID field has been found, or a timeout occurs (3.12).

Having found the correct ID field, the disk controller logic will wait until a valid Data field AM group has been detected. An optional Data ident comparison will be made on the subsequent byte, then the sector data will be transferred to the M212 processor, the sector size being defined in **TimingRegister0**. Any Data field CRC/ECC bytes will then be checked.

#0E Read Sector Long
The Read Sector Long operation is similar to the Read Sector operation, except that the Data field CRC or ECC bytes are also read. If a software generated ECC is being used (i.e. the CRC/ECC logic is disabled on the Data field) then the number of ECC bytes that are read is defined in **TimingRegister1** bits 7-4.

#0A Read ID Field
The Read ID Field operation is used to read the checkable and uncheckable bytes after the first valid ID field AM group that is encountered on the disk. A normal ID comparison and CRC/ECC check is made on the ID field. This can be useful for testing or checking purposes.

Note that if there is no distinction between a valid ID field AM group and a valid Data field AM group, Data field bytes may be read rather than ID field bytes.

#0B Read ID Field Long
The Read ID Field Long operation is similar to the Read ID Field operation, except that the ID field CRC or ECC bytes are also read. If a software generated ECC is being used (i.e. the CRC/ECC logic is disabled on the ID field) then the number of ECC bytes that are read is defined in **TimingRegister1** bits 7-4.

#08 Read Special
The Read Special operation can be used for testing or verification purposes. It begins by waiting for a single ID field AM to occur. Between 1 and 65536 bytes of data, including the AM byte, will then be read. The number of bytes to be read is defined by the 16-bit binary value in the **ReadSpecial0** register (least significant eight bits), and the **ReadSpecial1** register (most significant eight bits). Note that the value #0000 is interpreted as a count value of 65536.

An attempt to read across a write splice will probably cause a loss of byte-level sychronisation to the disk read data.

#18 Read Special Indexed
The Read Special Indexed operation starts by waiting for the **notIndex** pin to become low, indicating the start of a track. The disk controller logic then waits for between 0 and 255 valid ID field AM groups to occur (this number being defined in the **ValidAMCount** register). The Read Special Indexed operation then continues in the same way as a Read Special operation, by waiting for an ID field AM byte before reading between 1 and 65536 bytes from the disk.

#01 Write Sector

The Write Sector operation is used to write sector data to the disk. When the **Start** command is issued, the disk controller logic will search for the correct ID field, as in the case of the Read Sector operation. If this search is successful, the **notWriteGate** pin will be asserted and data will then be written to the disk. The delay after the ID field CRC/ECC bytes before the **notWriteGate** pin is asserted is defined in the **DelayAfterIDLess3** register, allowing a gap of between 3 and 258 bytes to be defined.

During a Write Sector operation, normal write data is issued to the disk controller logic using repeat data or multiple data codes. This data is then FM or MFM coded, and optionally precompensated, before being written to disk.

For each AM byte in the Data field AM group, a **WriteDataAM** command, followed by the AM data pattern, is issued. Each bit that is set in the **DataAMMissingClock** register will then mask out the corresponding clock bit in the written AM. For example, to write an AM with a data pattern of #A1 and a clock pattern of #0A (with MFM coding enabled), the **DataAMMissingClock** register should be loaded with #04, and the command sequence **#31, #A1** should be issued.

A hardware generated CRC (or ECC) syndrome can be written by issuing an **Append2SyndromeBytes** (or **Append4SyndromeBytes**) command after the last sector data byte has been written.

The Write Sector operation will terminate when a **SwitchOffWriteGate** command is issued. An example of a code/data sequence that would be sent to the disk controller logic during a typical Write Sector operation is shown below. It is assumed that all the necessary registers have been initialised to their correct values, and that the PIA ports have been used to select and position the disk read/write heads properly.

Typical Write Sector Code Sequence

#B9 #01	**Write Operation Register**	- Set Operation to Write Sector
#01	**Start**	
#4D #00	**Repeat data**	-- Thirteen #00 sync bytes
#31 #A1	**WriteDataAM**	-- #A1 address mark
#41 #F8	**Repeat data**	-- #F8 Data ident
#78 #xx	**Multiple data**	-- 256 sector bytes
#xx		
#xx		
...		
...		
#xx		
#14	**Append4SyndromeBytes**	-- Four ECC bytes
#43 #00	**Repeat data**	-- Three gap bytes
#04	**SwitchOffWriteGate**	
#DA	**Read register**	-- Read StatusRegister0
#DB	**Read register**	-- Read StatusRegister1

#00 Format
A Format operation is used to format a single track on the disk. Before performing a Format operation, the selected disk read/write head should be positioned over the required track. When a **Start** command is issued, the disk controller logic will wait for the **notIndex** pin to become low, indicating the start of the track. The **notWriteGate** pin will then be taken low, and data from the disk controller databus will be written to the disk. The track format is controlled by this sequence of data bytes and commands issued by the M212 processor during the format.

The format starts with the CRC/ECC generator waiting to generate an ID field syndrome. Syndrome generation starts when a **WriteDataAM** or **WriteIDAM** command is issued, and halts when an **Append2SyndromeBytes** or **Append4SyndromeBytes** command is issued. The act of appending a syndrome also has the effect of automatically toggling the CRC/ECC logic between ID field and Data field modes.

If an **Append2SyndromeBytes** or **Append4SyndromeBytes** command is not issued at the end of either the ID field or the Data field, as would be the case if a software generated syndrome were being used on that field, the **ToggleIDField** command should be used in its place to keep the CRC/ECC generator in step with the format. The data byte which accompanies the **ToggleIDField** command is written to the disk during the period that the command is being executed.

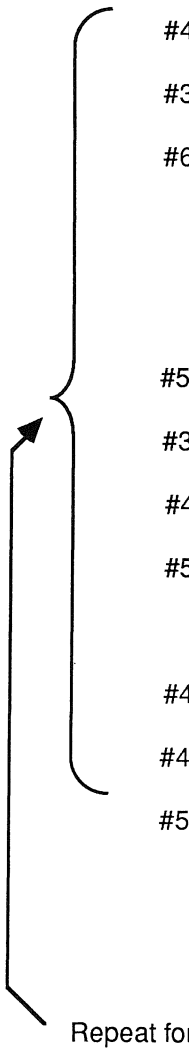
The **notWriteGate** pin will be taken high after one complete revolution of the disk (i.e. when the **notIndex** pin next goes low), or when an underrun error occurs.

Note that a sufficient number of bytes for the end of track gap must be provided to ensure that the track is entirely filled with data, even if the disk revolution frequency varies from its nominal value by a specified amount. Any superfluous gap bytes which are transferred to the disk controller logic after the entire track has been written will be ignored.

An example of a typical winchester disk **Format** operation is shown below.

Typical Format Code Sequence

#B9 #00	**Write Operation Register**	- Set Operation to Format
#01	**Start**	
#54 #4E	**Repeat data**	-- Sixteen #4E gap bytes
#4D #00	**Repeat data**	-- Thirteen #00 sync bytes
#35 #A1	**WriteIDAM**	-- #A1 ID address mark
#64 #FE	**Multiple data**	-- ID field ident
#xx		-- Cylinder number
#xx		-- Head number
#xx		-- Sector number
#12	**Append2SyndromeBytes**	-- Two CRC bytes
#54 #00	**Repeat data**	-- Sixteen #00 sync/gap bytes
#31 #A1	**WriteDataAM**	-- #A1 Data address mark
#41 #F8	**Repeat data**	-- #F8 Data ident
#58 #00	**Repeat data**	-- 256 sector bytes
#14	**Append4SyndromeBytes**	-- Four ECC bytes
#43 #00	**Repeat data**	-- Three gap bytes
#4F #4E	**Repeat data**	-- Fifteen #4E gap bytes
#59 #4E	**Repeat data**	-- 512 end of track gap bytes
#DA	**Read register**	-- Read StatusRegister0
#DB	**Read register**	-- Read StatusRegister1

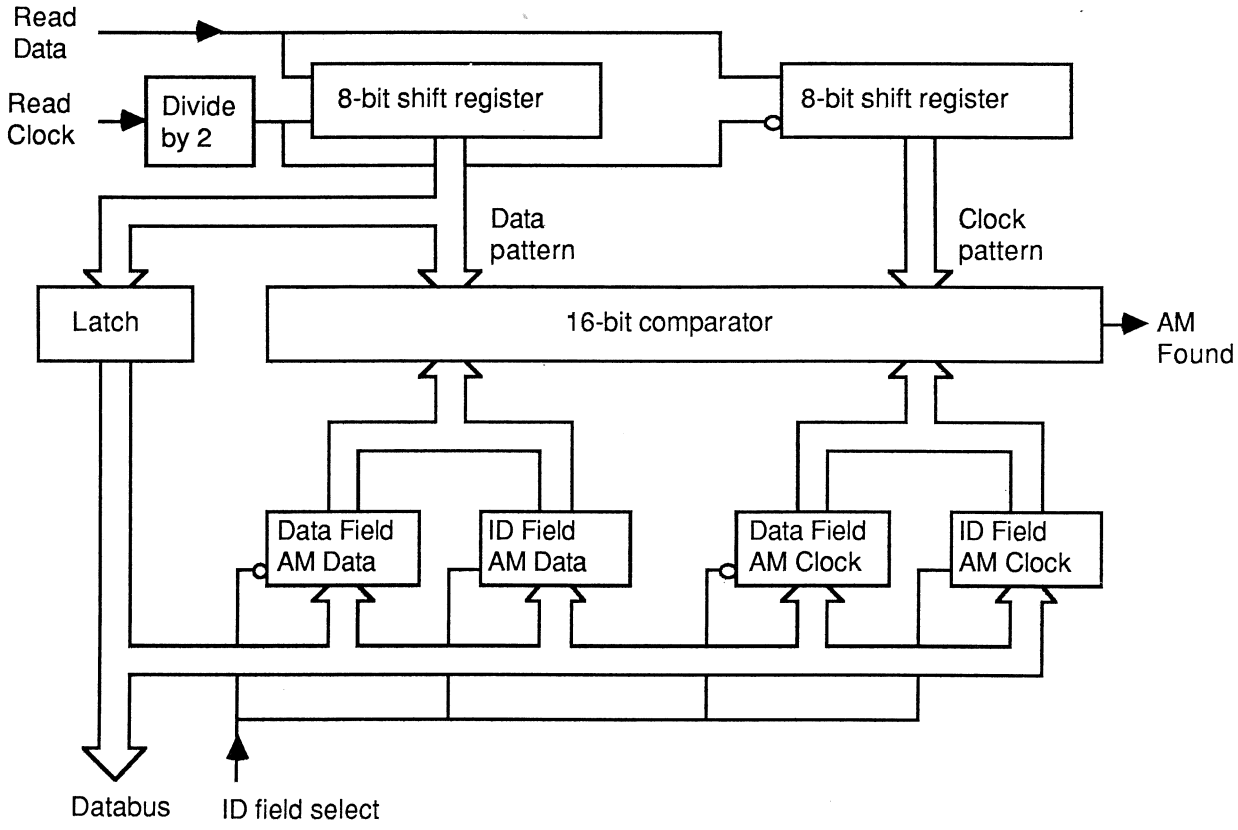


Repeat for each sector
with modified ID field information

3.6 Serial/parallel conversion

When reading data from the disk, the serial/parallel conversion logic searches the serial data stream for an ID field or Data field AM. Having found an AM, data bytes are written onto the disk controller databus at byte-long intervals.

Serial / Parallel Logic Block Diagram



Four registers are used to hold the clock and data patterns for the ID field and Data field AM's, against which the serial data stream is compared. These registers are designated:

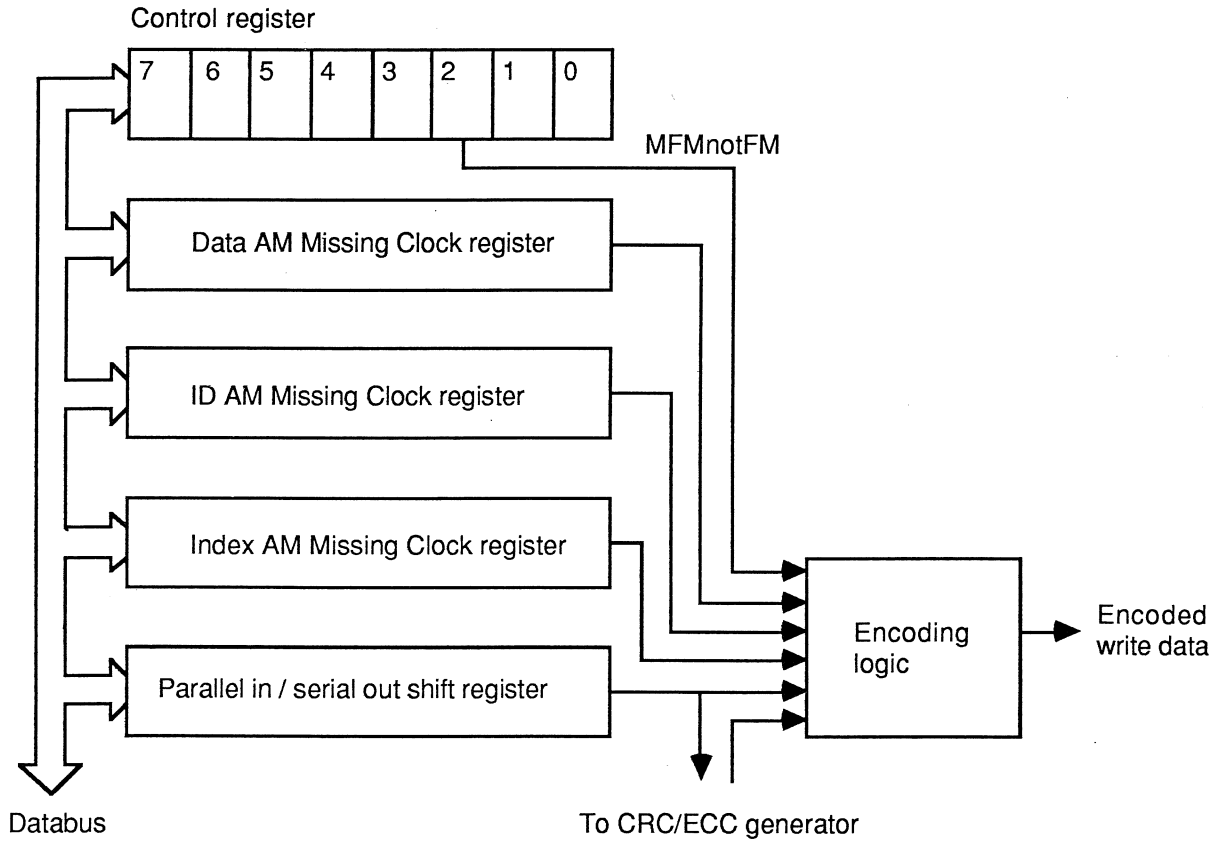
DataFieldAMData
DataFieldAMClock
IDFieldAMData
IDFieldAMClock

For example, if the Data field AM has a data pattern of #A1 and a clock pattern of #0A, the **DataFieldAMData** register should be loaded with #A1, and the **DataFieldAMClock** register should be loaded with #0A. The read/write control logic automatically selects whether an ID or a Data field comparison is to be performed.

3.7 Parallel/serial conversion

During a Write Sector or Format operation, the parallel/serial conversion logic takes byte-wide data from the disk controller databus and serialises it. The serial write data is then encoded into FM or MFM format. The type of coding generated depends upon the state of **Control** register bit 2, which should be low for FM coding and high for MFM coding.

Parallel / Serial Logic Block Diagram

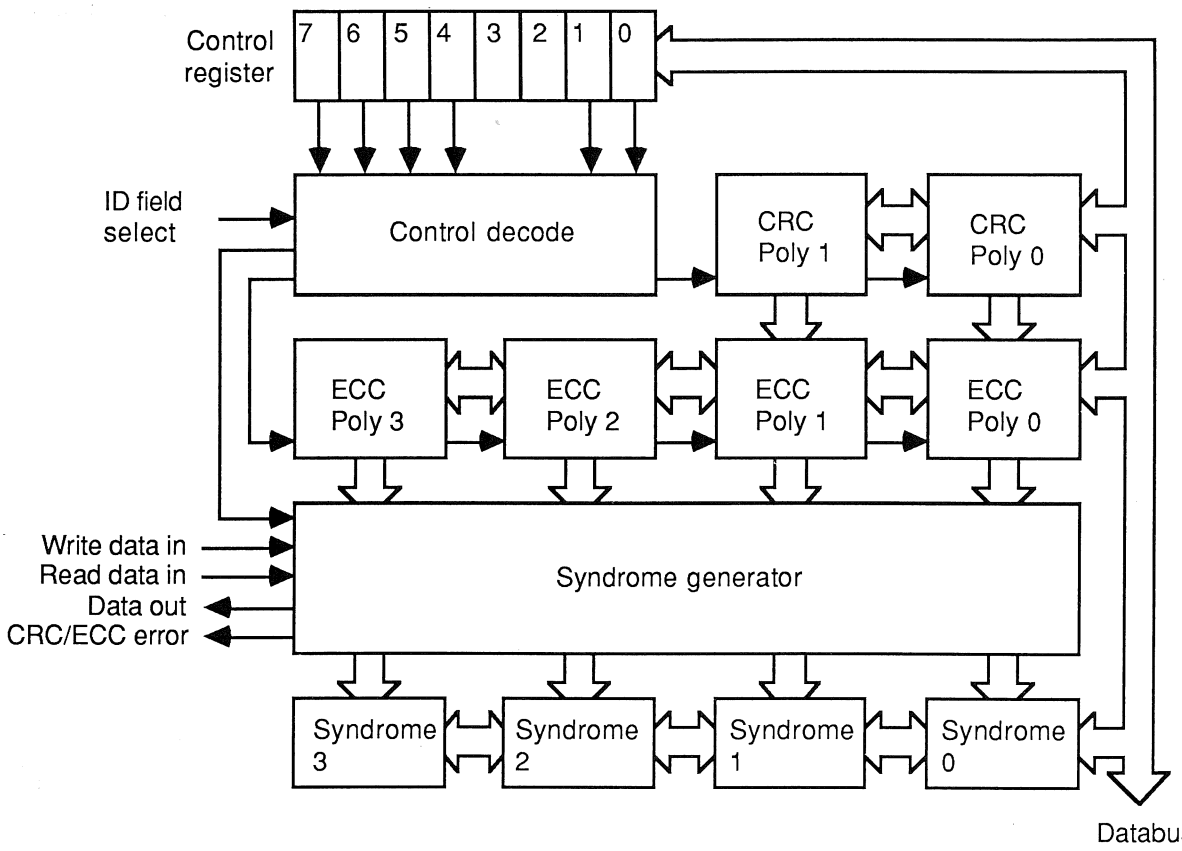


The contents of the **DataAMMissingClock**, **IDAMMissingClock**, and **IndexAMMissingClock** registers are used when writing AM's. For each bit set in the related register when a **WriteDataAM**, **WriteIDAM**, or **WriteIndexAM** command is issued, a clock pulse will be deleted from that bit cell in the following data byte.

When an **Append2SyndromeBytes** or **Append4SyndromeBytes** command is issued during a disk write operation, the hardware-generated CRC/ECC syndrome bytes are inserted into the encoded write data stream.

3.8 CRC/ECC generator

CRC/ECC Logic Block Diagram



The CRC/ECC generator can be enabled on ID and Data fields to provide hardware-generated syndromes for the purpose of error detection and correction. The ID and Data fields can each be configured to use either a 16-bit (CRC) polynomial or a 32-bit (ECC) polynomial, separate user-programmable polynomials being used for CRC and ECC generation. The CRC/ECC shift register is configurable on both ID and Data fields to reset to all zeroes or preset to all ones before generating a syndrome. This is accomplished by setting **Control** register bit 1 (**IDPresetNotReset**) and bit 0 (**DataPresetNotReset**) high for preset and low for reset. Furthermore, when a sector has been written using the industry standard 32-bit ECC polynomial, with **DataPresetnotReset** low, the CRC/ECC generator logic can be configured during read-back to allow the 'Chinese Remainder Theorem' to be used for error correcting. This can result in shorter error correction times when using large sector sizes.

The **Control** register bits 5-4 determine the Data field CRC/ECC mode, and the **Control** register bits 7-6 determine the ID field CRC/ECC mode, according to the following table:

Bits 5-4	Data Mode	Bits 7-6	ID Mode
0 0	Chinese ECC	0 0	Chinese ECC
0 1	32-bit ECC	0 1	32-bit ECC
1 0	Disable	1 0	Disable
1 1	16-bit CRC	1 1	16-bit CRC

When reading an ID or Data field with the CRC/ECC generator enabled, syndrome generation starts when an AM is detected, and stops at the end of the CRC/ECC field. At this point, if any bit is set in the **SyndromeByte0/1/2/3** registers (for ECC) or **SyndromeByte0/1** registers (for CRC), then an error is flagged in **StatusRegister1**. The **SyndromeByte0/1/2/3** registers may be read by the M212 processor to allow a software error correction to be attempted.

When performing a Write Sector or Format operation, syndrome generation starts when a **WriteIDAM** or **WriteDataAM** command is issued, and stops when an **Append2SyndromeBytes** or **Append4SyndromeBytes** command is issued.

Note that the most significant bit of a CRC or ECC polynomial (except when using the Chinese Remainder Theorem) is implicitly set high. For example, to use the CRC polynomial $x^{16} + x^{12} + x^5 + x^0$, the **CRCPolynomial1** and **CRCPolynomial0** registers should be loaded with #10 and #21 respectively.

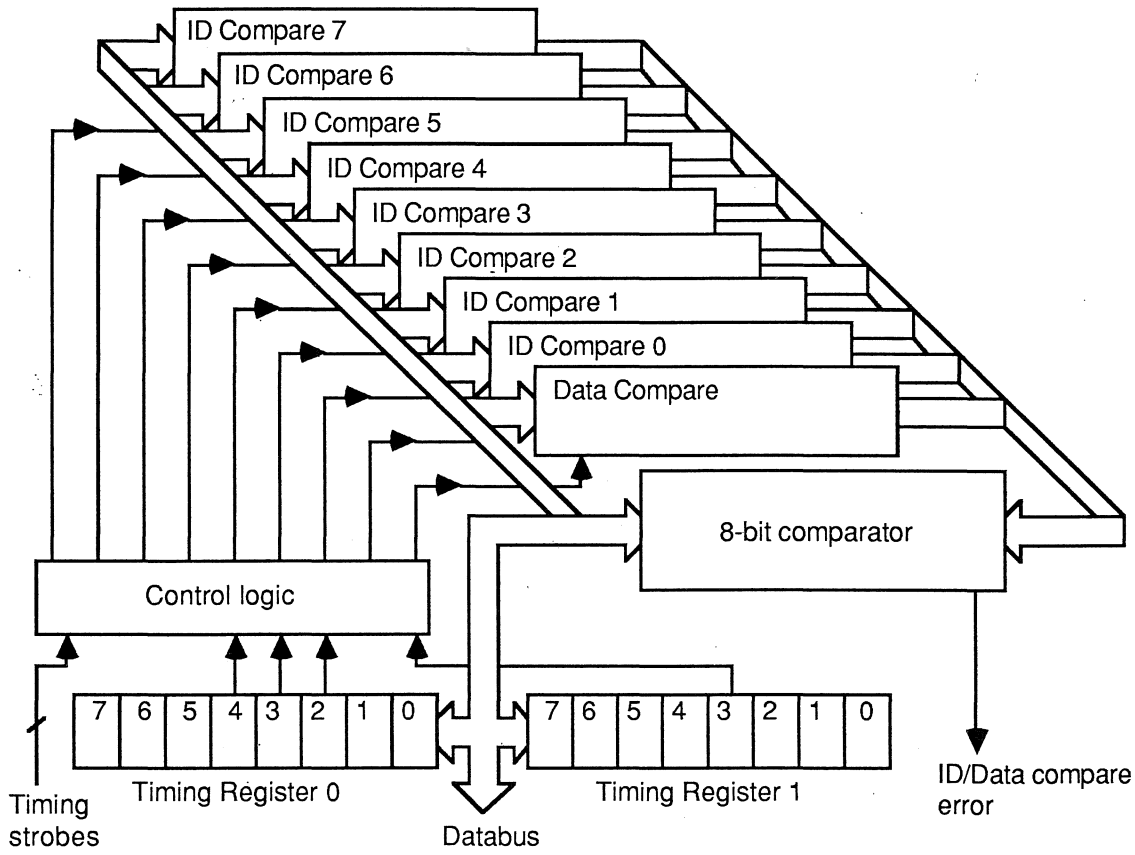
The Chinese Remainder Theorem can be used when reading back a sector that has been written using the industry standard polynomial (with reset enabled) $x^{32} + x^{23} + x^{21} + x^{11} + x^2 + x^0$, which can be factorised as $(x^{21} + x^0)(x^{11} + x^2 + x^0)$. The 32-bit ECC generator is then split into separate 21-bit and 11-bit generators. The terms $x^{21} + x^0$ in the upper (21-bit) part of the polynomial are implicitly set high, as is the term x^{11} in the lower (11-bit) part of the polynomial. To use this mode of operation when reading a sector, therefore, the **ECCPolynomial0/1/2/3** registers should be written with #05, #00, #00, #00 respectively.

Examples of error correction algorithms are given in Appendix C.

3.9 ID/DATA field comparison

The ID/Data field comparison logic is used during a disk read or write operation when searching for a specific ID field or checking the Data field ident byte.

ID/DATA Field Compare Logic Block Diagram



When searching for an ID field, the disk controller logic waits until a valid ID field AM group is recognised. Subsequently, between one and eight ID field bytes are checked against the expected bytes in the **IDCompare0-7** registers. Using a typical track format, these bytes would contain ident, cylinder, head, and sector numbers. The number of checkable bytes in an ID field is defined in **TimingRegister0** bits 4-2, with '000' indicating one checkable byte, and '111' indicating eight checkable bytes. The first ID field byte will be compared with the contents of the **IDCompare0** register, the second byte will be compared with the contents of the **IDCompare1** register, and so on until the required number of bytes have been checked. Any byte that is in error will cause a corresponding bit to be set in **StatusRegister0**. Note that during a Write Sector or Read Sector operation, an ID compare error will cause the disk controller logic to reset the status registers and retry the ID field search, until either the correct IDField is found, or a timeout occurs.

If **TimingRegister1** bit 3 is set, then during a sector read the Data ident byte (the first byte after the Data field AM group) will be checked against the contents of the **DataCompare** register. The sector data will then be read. If the Data ident check fails, **StatusRegister1** bit 0 will be set. If the sector has been formatted with no Data ident byte, **TimingRegister1** bit 3 should be reset. The Data ident comparison will then be disabled, and the sector data will immediately follow the Data field AM group.

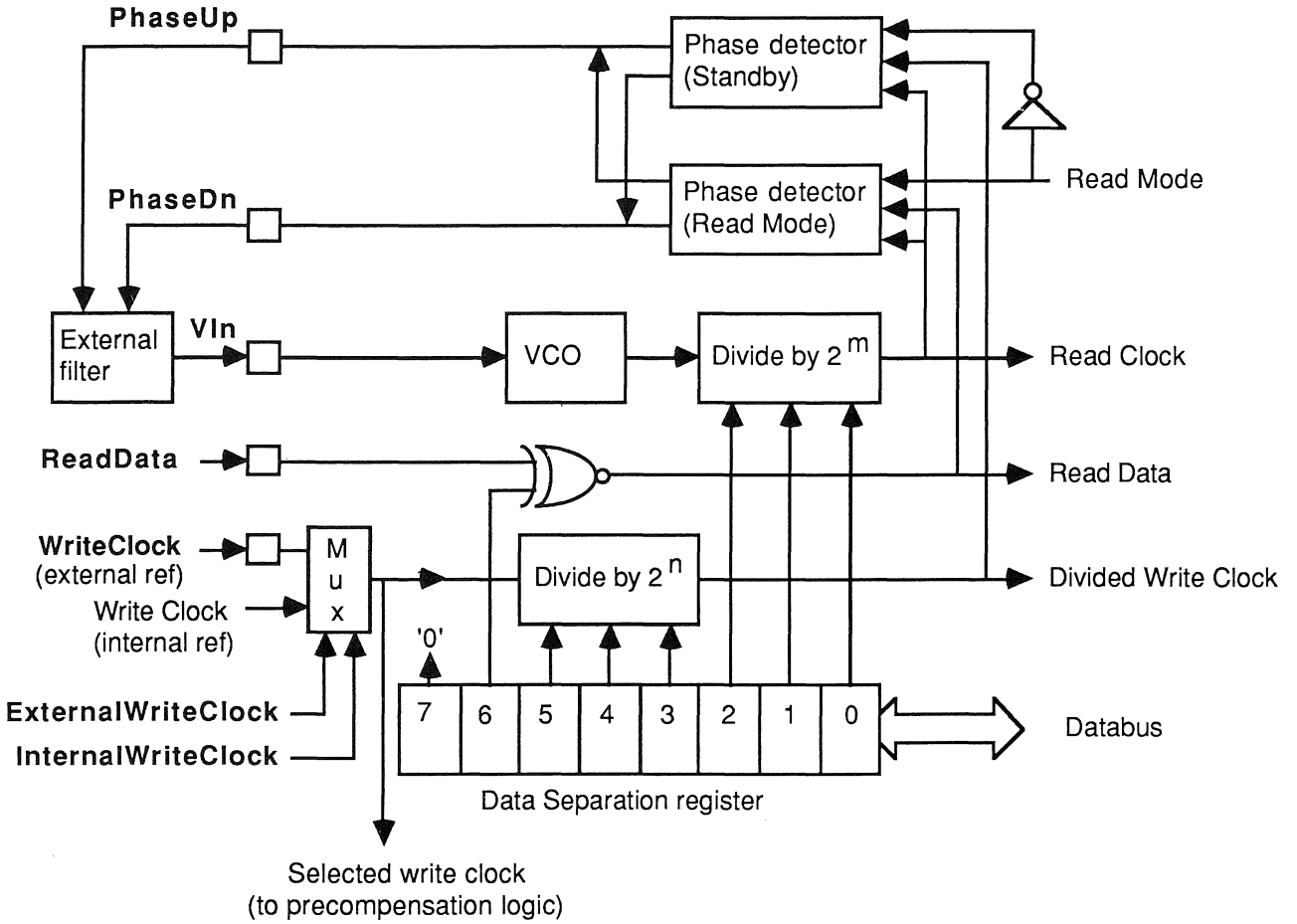
3.10 Data separation

The data separator logic is used to generate a read clock reference frequency and to extract clock/data pulses from a coded FM/MFM read data stream.

The data separator Phase-Locked Loop (PLL) consists of a voltage controlled oscillator, followed by a programmable divider, with a feedback loop consisting of two phase detectors and an off-chip filter. Further details of the required filter characteristics and VCO operating conditions are given in Appendix D.

When not reading from disk, the 'standby' phase detector is enabled and the PLL is phase-locked to the divided write clock frequency, which is nominally equivalent to the expected read clock frequency. When reading from the disk, the 'standby' phase detector is disabled and the 'read mode' phase detector is enabled, allowing the PLL to phase-lock to the incoming data stream, thus providing the read clock and read data signals to the rest of the system.

Data Separation Logic Block Diagram



The data separation logic is configured through the **DataSeparation** register, whose contents default to the value #08 when the **Reset** pin is asserted. A description of this register follows:

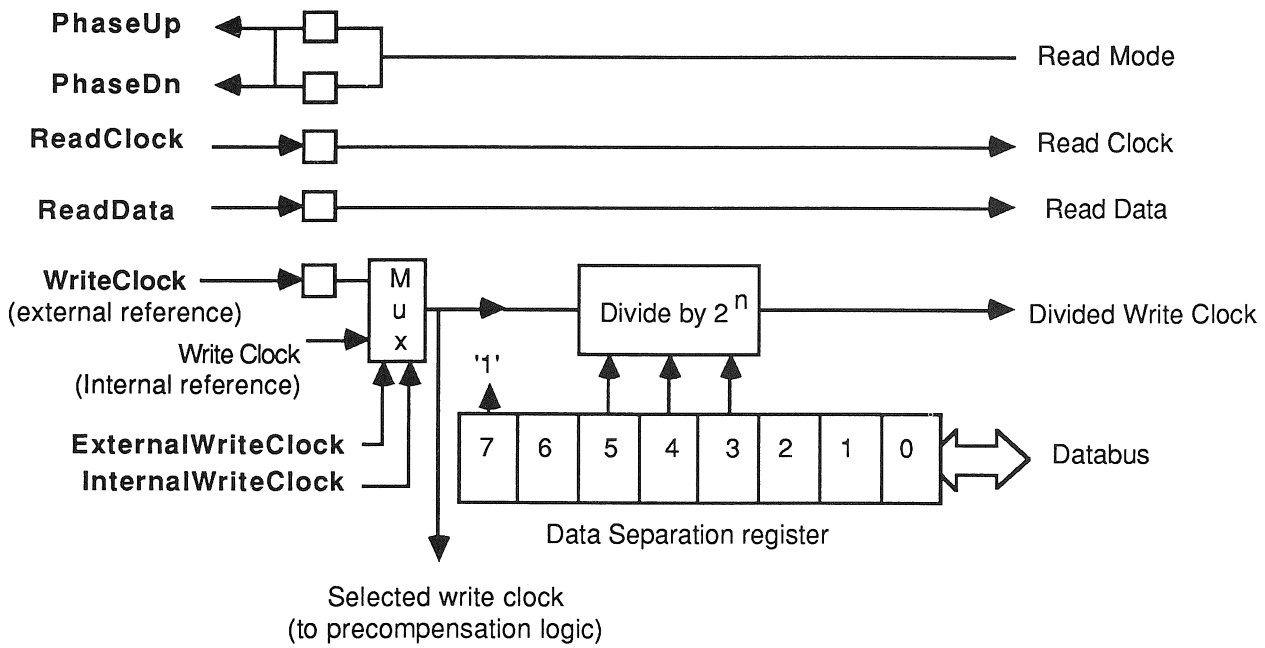
DataSeparation bits 2-0: VCOFrequencyDivision
Provides a 3-bit value to the VCO output divider, allowing the VCO output to be divided by 2^n (0<=n<=7) to provide a read clock frequency to the disk controller logic. This division ratio should be chosen so that the VCO operates within its specified frequency range.

DataSeparation bits 5-3: WriteClockReferenceDivision
Provides a 3-bit value to the write clock divider, allowing the write clock to be divided by 2^n (0<=n<=7) in order to generate the divided write clock frequency to the disk controller logic. The divided write clock frequency should be twice the disk data rate (e.g. 10 MHz for a 5 Mbits/s data rate). The write clock can be chosen from an external source (applied to the **WriteClock** pin) by issuing an **ExternalWriteClock** command, or from an internal frequency (equal to twice the M212 processor clock frequency) by issuing an **InternalWriteClock** command. The selected write clock, before division, is also used as the precompensation logic reference frequency. The internal write clock source is used by default when the **Reset** pin is asserted.

DataSeparation bit 6: PositiveReadData
This bit should be high to allow positive read data pulses to be applied to the device, or low to allow negative read data pulses to be applied.

DataSeparation bit 7: ExternalDataSeparation
Should be set if external separation is to be performed. The PLL is then disabled, and bits 6, 2, 1, and 0 of the **DataSeparation** register are ignored. Certain pins also change function: **PhaseUp** and **PhaseDn** should be tied together to give an indication of Read Mode, and the externally generated read clock should be applied to the **VIn** pin. Read data pulses are applied to the **ReadData** pin.

Logic Configuration for External Data Separation



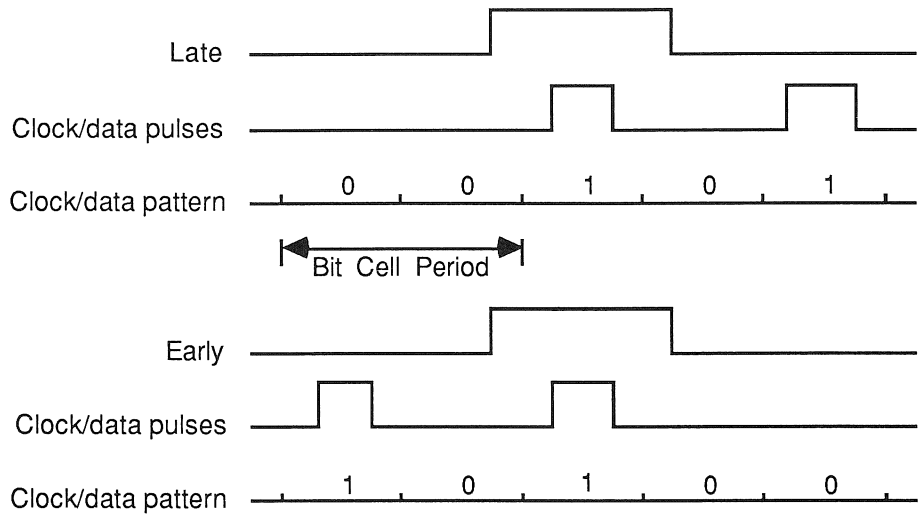
Example:
A winchester with a data rate of 5 Mbits/sec will require a read clock frequency of 10 MHz. We will assume that the disk drive supplies negative read data pulses. We require that the data separation is performed internal to the IMS M212. The recommended VCO range is 32 to 80 MHz so we will select a **VCOFrequencyDivision** of 4 to give a VCO frequency of 40 MHz for a read clock frequency of 10 MHz. If we are using an **External-WriteClock** then we could choose a write clock frequency of 20 MHz and a **WriteClockReferenceDivision** of 2. This would result in the value of the Data Separation Register of #0A.

Note that the frequency of the selected WriteClock, whether derived internally or externally, is important in the generation of the values to be used for precompensation. (See section 3.11)

3.11 Precompensation

Some disk drives require that MFM write pulses be precompensated on certain tracks to allow for bit-shift effects. On the M212, precompensation can be performed either on-chip or off-chip, and is enabled when **Control** register bit 2 (**MFMNotFM**) and **Control** register bit 3 (**EnablePrecompensation**) are set. The patterns that are precompensated are shown below:

Data Patterns Requiring Precompensation



The precompensation logic is configured through the **Precompensation0** and **Precompensation1** registers. The logic consists of two delay chains into which coded write data from the parallel/serial conversion logic passes. An eight bit shift register, which is clocked by the precompensation reference clock and tapped at every half-bit shift, is used for floppy disk precompensation, and a calibrated 16-stage analogue delay chain, allowing finer delay increments, is used for winchester disk precompensation. Early, Ontime, and Late write data are tapped from these delay chains.

The analogue delay chain is calibrated by using a similar 8-stage delay chain as voltage controlled delay element in a phase-locked loop, which has an off chip filter. The control voltage which locks the PLL frequency to that of the precompensation reference clock is also used as the control voltage into the 16-stage delay chain. This ensures that the delay through the 16-stage delay chain will effectively be equal to the period of the precompensation reference clock, irrespective of the device process parameters or operating conditions. The precompensation reference clock is generated from the selected write clock frequency, after division by either two or four. Details of the phase locked loop used for precompensation are shown in Appendix E.

When the **Reset** pin is asserted, the precompensation registers are both reset to #00. The function of the precompensation registers is as follows:

Precompensation0 bits 3-0: OntimeDelay
This 4-bit binary value determines from which output of the selected 16-stage delay chain that ontime write data pulses are tapped. A value of '0000' selects the first output of the delay chain, '1111' selects the last output.

Precompensation0 bit 4: DivideBy2Not4
The precompensation reference clock is derived from the write clock frequency divided by two when this bit is high, or divided by four when this bit is low.

Precompensation0 bit 5: TestMode (Reserved)
This bit is reserved by INMOS for test purposes, and must be set low.

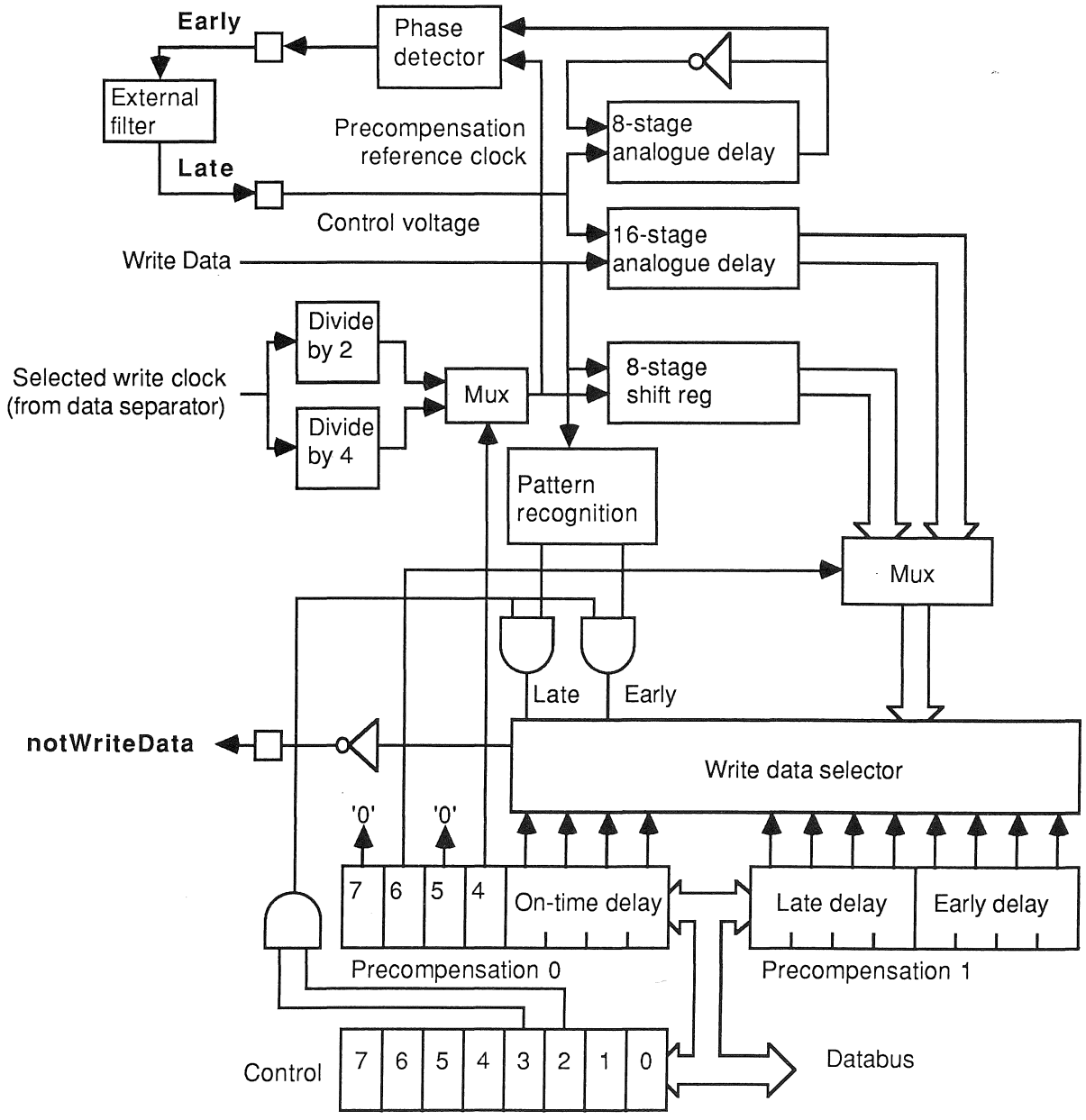
Precompensation0 bit 6: WinchesterNotFloppy
Set high to select the analogue delay chain (for winchester disks), low to select the shift register delay chain (for floppy disks).

Precompensation0 bit 7: ExternalPrecompensation
This bit should be high if off-chip precompensation is to be performed. The write data stream, along with the early and late signals, are then output from the device on the **notWriteData**, **Early**, and **Late** pins. If this bit is low, the write data stream will be precompensated on-chip. The **Early** pin should then be connected to the input of the precompensation PLL filter, and the **Late** pin connected to the filter output.

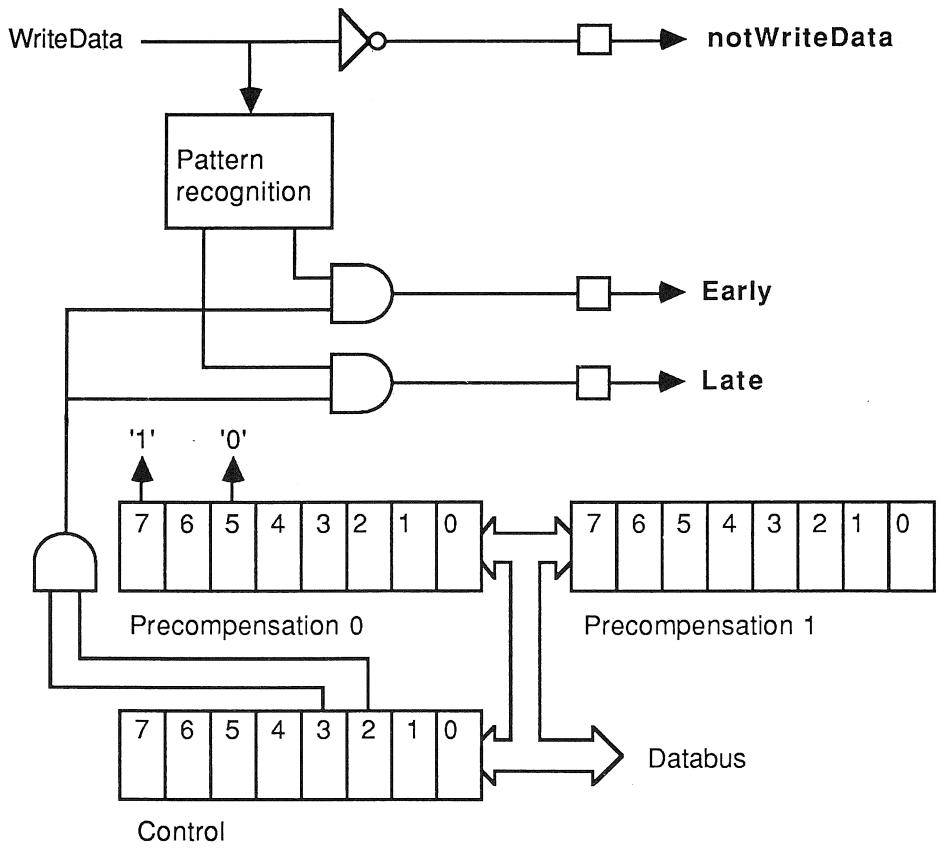
Precompensation1 bits 3-0: EarlyDelay
This 4-bit binary value determines from which output of the selected 16-stage delay chain that early write data pulses are tapped. A value of '0000' selects the first output of the delay chain, '1111' selects the last output.

Precompensation1 bits 7-4: LateDelay
This 4-bit binary value determines from which output of the selected 16-stage delay chain that late write data pulses are tapped. A value of '0000' selects the first output of the delay chain, '1111' selects the last output.

Precompensation Logic Block Diagram



Logic Configuration for External Precompensation



Example 1: Winchester disk precompensation
Suppose the write clock frequency is 20 MHz, and a precompensation value of 12 ns is required. The write clock can be divided by two to generate a precompensation clock period of 100 ns. The delay through the analogue delay line will then be approximately 6 ns per stage. An Early delay of '0' could then be chosen, with an Ontime delay of '2', and a Late delay of '4', giving the required precompensation value of 12 ns. The **Precompensation0** register would then be loaded with #52, and the **Precompensation1** register would be loaded with #40.

Example 2: Floppy disk precompensation
Suppose the write clock frequency is 16 MHz, and a precompensation value of 125 ns is required. The write clock can be divided by four to give a precompensation clock period of 250 ns. The delay through each half-bit shift of the shift register will then be 125 ns. An Early delay of '0' could then be chosen, with an Ontime delay of '1', and a Late delay of '2', giving the required precompensation value of 125 ns. The **Precompensation0** register would then be loaded with #01, and the **Precompensation1** register would be loaded with #20.

Note that if large values of delay are used then two consecutive clock/data pulses will be in the delay chains at the same time and this may result in erroneous data being output.

3.12 Timeout logic

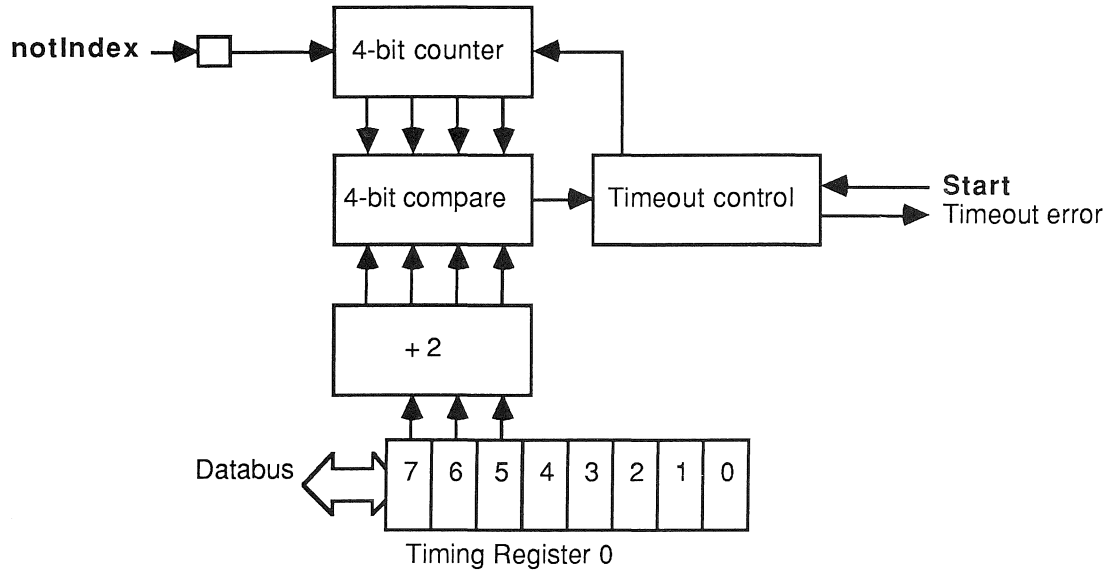
The timeout logic terminates an unsuccessful attempt to read from or write to a disk, caused by a failure to find a valid AM group or the required ID field for instance.

A timeout occurs if a certain number of index pulses occur before a read or write operation terminates. This number of index pulses can be set to any value between two and nine, as defined in **TimingRegister0** bits 7-5, with '000' indicating two index pulses before a timeout, and '111' indicating nine index pulses before a timeout.

When a timeout occurs, **StatusRegister1** bit 7 will be set. If no valid AM group has been found by this time, **StatusRegister1** bit 6 will also be set. The read or write operation will then be forced to continue to completion, including the clearing out of any associated input or output code/data transfers between the disk controller logic and the M212 processor.

Note that the disk will not be written to during an aborted write operation, and unsynchronised data from the disk will be transferred to the M212 processor during an aborted read operation. The status registers should always be read after any disk read/write operation to determine whether the operation completed successfully.

Timeout Logic Block Diagram



The optimal method of programming the IMS M212 processor is to use occam (See the occam programming manual for full details). Several standard languages are also supported, including C, Fortran and Pascal.

The processor provides direct support for the occam model of concurrency and communication. It has a microcoded scheduler which enables any number of concurrent processes to be executed together, sharing the processor time. The scheduler operates in such a way that inactive processes do not consume any processor time. The number of registers which hold the process context is small and this, combined with fast on-chip RAM, provides a sub-microsecond process switch time.

Process communication is implemented by memory to memory block move operations. These utilize fully the bandwidth available from the on-chip RAM.

4.1 IMS M212 types

The implementation of occam for the IMS M212 disk controller supports the following types:

CHAN OF *protocol*	Each communication channel provides communication between two concurrent processes. Each channel is of a type which allows the communication of data according to the specified protocol.
TIMER	Each timer provides a clock which can be used by any number of concurrent processes.
BOOL	The values of type **BOOL** are true and false.
BYTE	The values of type **BYTE** are unsigned numbers **n** in the range $0 \leq \mathbf{n} < 256$
INT	Signed integers **n** in the range $-32768 \leq \mathbf{n} < 32768$
INT16	Signed integers **n** in the range $-2^{15} \leq \mathbf{n} < 2^{15}$
INT32	Signed integers **n** in the range $-2^{31} \leq \mathbf{n} < 2^{31}$
INT64	Signed integers **n** in the range $-2^{63} \leq \mathbf{n} < 2^{63}$
REAL32	Floating point numbers stored using a sign bit, 8-bit exponent and 23-bit fraction in ANSI/IEEE Standard 754-1985 representation
REAL64	Floating point numbers stored using a sign bit, 11-bit exponent and 52-bit fraction in ANSI/IEEE Standard 754-1985 representation

4.2 IMS M212 process multiplexing

The IMS M212 scheduler supports two levels of process priority; high and low. High priority processes are expected to execute for a short time. Two scheduling lists are maintained, one for each priority level. Processes are scheduled by being added to the end of the appropriate list.

If a high priority process becomes able to run, for example as the result of an external event occuring, it will interrupt any low priority process and will then run until it has to wait for a communication, a timer input, or until it completes processing. At this point, providing no other high priority processes are able to proceed, execution of the low priority process may resume.

If one or more high priority processes are able to proceed, then one is selected from the front of the high priority scheduling list and run.

If no process at high priority is able to proceed, but one or more processes at low priority are able to proceed, then one is selected from the front of the low priority scheduling list and run.

Low priority processes are periodically timesliced to provide an even distribution of processor time between

computationally intensive tasks.

If there are **n** low priority processes, then the maximum latency from the time at which a low priority process becomes active to the time when its starts processing is 2**n** - 2 timeslice periods. It is then able to execute for between one and two timeslice periods, less any time taken by high priority processes.

Each timeslice period lasts for 5120 cycles of the input clock **ClockIn** (approximately 1 millisecond at the standard frequency of 5 MHz).

To ensure that each low priority process proceeds, high priority processes should never occupy the processor continuously for a period of time equal to a timeslice period. A good guideline is to ensure that, if there are a total of **n** high priority processes, then each limits its activity to much less than 1/**n**'th of any 1 millisecond period.

Interrupt latency

If a high priority process is waiting for an external channel to become ready, and if no other high priority process is active, then the interrupt latency (from when the channel becomes ready to when the process starts executing) is typically 19 processor cycles, maximum 53 cycles (assuming use of on-chip RAM).

4.3 IMS M212 Error flag

Expressions which cause arithmetic overflow are invalid, and processes which cause array bound violations are invalid. If the compiler is unable to check that a given construct contains only valid expressions and processes, then extra instructions are compiled to perform the necessary checks at runtime. If the result of the check indicates that an invalid expression or invalid process has occurred, then the processor's Error flag is set.

In the IMS M212 implementation of occam, the offending process can be made to stop when the Error flag is set.

The IMS M212 can be initialised so that the processor halts when the Error flag is set. This is done by setting the HaltOnError flag in the CPU. The appropriate initialisation sequence to do this is provided by the development system.

If the processor has been halted as the result of an error, the links continue with any outstanding transfers, and the IMS M212 may be analysed.

When a high priority process pre-empts a low priority process, the current state of the Error flag is saved; the Error flag itself is maintained. When, finally, there are no high priority processes able to run, the current state of the Error flag is lost, and the preserved state is restored as part of commencing execution of the pre-empted low priority process.

This ensures that the state of the Error flag is preserved during the evaluation of an expression or a sequence of assignments.

4.4 IMS M212 memory map

The address space of the IMS M212 is signed and byte addressed. Words are aligned on two-byte boundaries. The 16-bit address lines access addresses in the range #8000 - #7FFF, which is organised as follows:

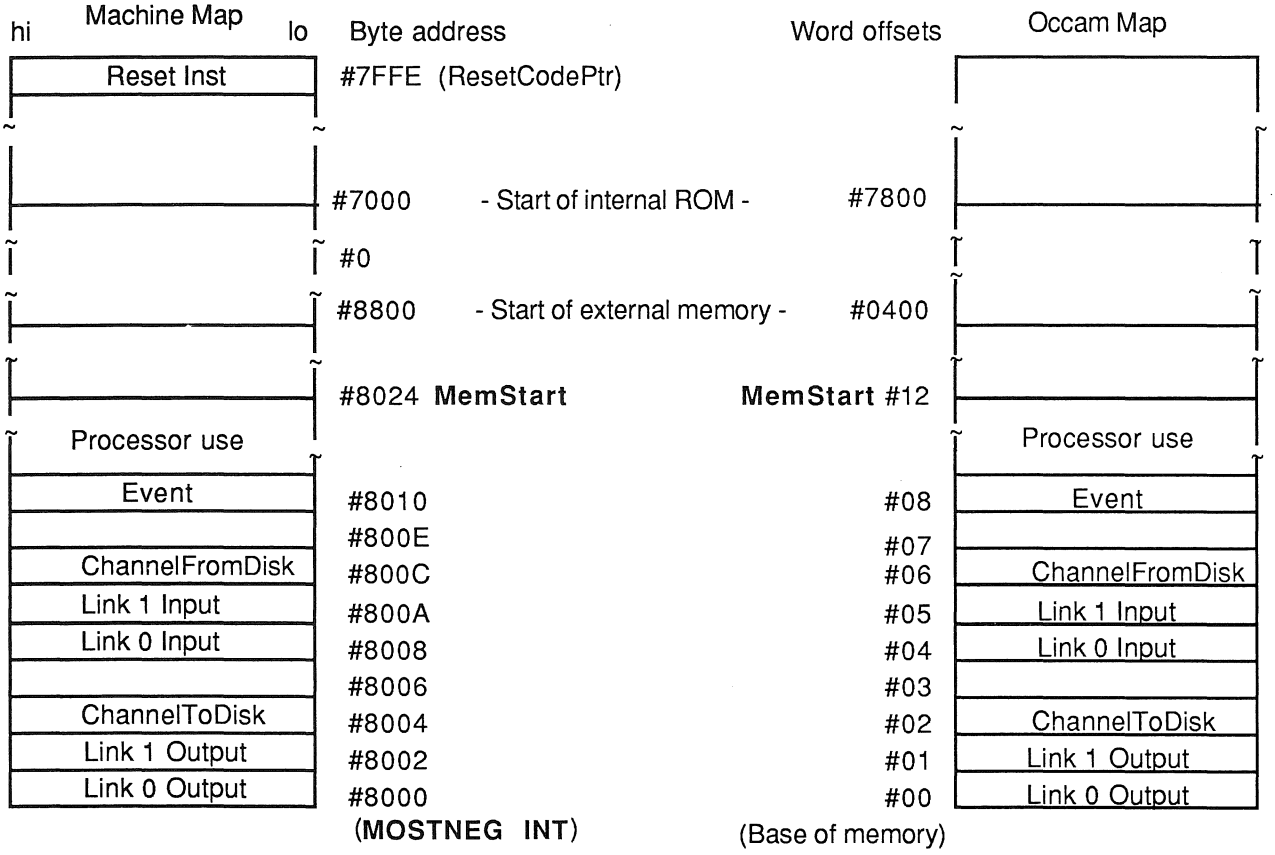
#8000 - #87FF Internal RAM
#8800 - #6FFF External Address Space
#7000 - #7FFF Internal ROM if **DisIntROM** is low, External Address Space if **DisIntROM** is high.

The first 18 words of the address space are used for system purposes. The next available location is, by convention, referred to as **MemStart**.

A suitable declaration of **MemStart**, for example is

```
VAL MemStart IS #12 :
```

The programmer can access locations in memory by using the occam mechanism of placement. This allows variables of any type to be placed at a location specified as a word offset from the base of memory. The placement of a byte array allows access to the byte components of a word. For example, a byte array could be placed in internal memory to optimize access, or a similar array could be placed in external memory to address one or more memory mapped devices.



This schema for calculating addresses is used to provide word length independent code, as though memory were an array of type **INT** (**[]INT**). The link addresses will always be found within the lower bounds of the memory array space.

The top of address space is used, by convention, for ROM based code. If the transputer is configured to bootstrap from ROM, then the processor commences execution from address #7FFE. Tools are supplied with the compiler and development system to enable the generation and placement of ROM images.

4.5 IMS M212 timer

The IMS M212 has two timer clocks. High priority processes use the high priority timer clock for high resolution (1μs) timing, low priority processes use the low priority timer clock (64μs) which allows long periods to be timed.

At the standard **ClockIn** frequency of 5 MHz, the high priority timer clock ticks every microsecond (five **ClockIn** periods) and cycles approximately every 65 milliseconds.

The low priority timer clock ticks every 64 microseconds (320 **ClockIn** periods), therefore one second is exactly equal to 15625 ticks. The low priority timer cycles approximately every four seconds.

4.6 IMS M212 event pins

An attention-seeking peripheral may signal the IMS M212 via the **EventReq** pin, which the IMS M212 handshakes using **EventAck**. The IMS M212 implements a hardware channel to allow a low to high transition on the **EventReq** pin to be communicated to a process as a synchronizing message.

An occam channel (which must already have been declared) may be associated with **EventReq** pin by a channel placement. The conventional name and the value used for this channel is given by

```
PLACE Event AT 8 :
```

Event behaves like an ordinary occam channel, and a process may synchronize with a low to high transition on the **EventReq** pin by using the occam construct:

```
Event ? signal
```

The process waits until the channel **Event** is ready. The channel is made ready by the transition on the **EventReq** pin (this may occur before the process attempts to input).

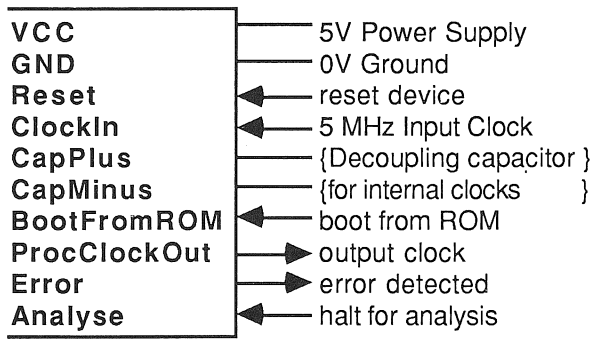
When the process is able to proceed, **EventAck** is taken high to indicate that the process is running. If the process executes at high priority, then it will take priority over any low priority process which may be executing when the transition occurs on the **EventReq** pin.

4.7 IMS M212 link placement

The link addresses will always be found within the lower bounds of the memory array. The conventional names and the values used for these channels are

```
PLACE Link0Output AT 0 :
PLACE Link1Output AT 1 :
PLACE Link0Input  AT 4 :
PLACE Link1Input  AT 5 :

PLACE Link2Output AT 2 : -- channel to the disk
PLACE Link2Input  AT 6 : -- channel from the disk
```

The system services comprise the clocks, power and initialisation logic used by the whole of the transputer. The **Reset** and **Analyse** inputs enable the IMS M212 to be initialised, for example on power up, or halted in a way which preserves its state for subsequent analysis. Whilst the IMS M212 is running both **Reset** and **Analyse** should be held low. The **Error** signal is directly connected to the processor's Error flag.



5.1 Reset

The IMS M212 is reset by taking **Reset** high whilst holding **Analyse** low. Operation ceases immediately and all state information is lost. **Reset** is then taken low after the hold time has elapsed.

The processor then bootstraps. If the **BootFromROM** input is high it will start to execute code starting from address #7FFE. If **BootFromROM** is low it will bootstrap from a link, that is, it will become ready to load a program from a link prior to executing it.

When initialising following power-on, a time is specified during which **VCC** must be within specification, **Reset** must be high, and the input on **ClockIn** must be oscillating. **Reset** is taken low after the specified time has elapsed. During power-on reset all link inputs must be held low. (N.B. all link outputs are made low by reset.)

5.2 Analyse

A system built from transputers may be brought to a halt in a consistent state which is preserved for subsequent analysis. This analysis is performed in a manner similar to bootstrapping. The transputer development system includes appropriate bootstrap and analysis software.

A signal may be applied to the **Analyse** inputs of all the transputers in the system. The system is analysed by first taking **Analyse** high. This causes each transputer to halt, after a short period of time, in a consistent internal state. When the system has halted, **Reset** is taken high for the specified hold time, after which it is taken low. **Analyse** is then taken low, at which time each transputer bootstraps.

When **Analyse** is taken high, the processor will halt within three timeslice periods (approximately three milliseconds), plus the time taken for any high priority process to cease processing unless a disk data input is in progress in which case this will complete before processing ceases. If a disk output is in progress the processor will halt and the disk hardware will probably indicate an underrun has occured . Any outputting links continue until they complete the remainder of the current word. Input links will continue to receive data. Provided that there are no delays in sending acknowledgements, the links in a system will therefore cease activity within a few microseconds. Sufficient time must be allowed to allow the processor to halt and link traffic to cease before **Reset** is asserted.

The system must be designed so that links connected to the external world are quiet during reset.

5.3 Bootstrapping and analysis of a "failed" system

The transputer has two methods of bootstrapping. Firstly, the conventional bootstrap which occurs after **Reset**

and secondly, an analysis bootstrap which occurs after **Reset** plus **Analyse**. This second method allows the state of a system, which could well be a complex network of transputers, to be examined. In both cases the mechanism used is very similar and the bootstrap or analysis code may be provided from either the external memory of the transputer, often in ROM, or if the transputer is a part of a network, via its communication links.

5.3.1 Bootstrapping

It is possible to bootstrap the IMS M212 either by executing code held in ROM or by executing code which has been received on a link. In addition, prior to bootstrapping from a link, it is possible to read or write to any memory location in the transputer's memory map by peeking and poking down a link.

5.3.2 Bootstrapping from ROM

To bootstrap from ROM, the **BootFromROM** input is wired to **VCC**.

The IMS M212 bootstraps from ROM by executing a process at low priority. Control is transferred to the top two bytes in memory (at #7FFE), which will invariably contain a backward jump into ROM. **MemStart** (#8024) is used as the location of the process workspace. If **DisIntROM** is high the bootstrap location is in the external memory space. If **DisIntROM** is low the bootstrap location is in internal memory.

5.3.3 Bootstrapping from a link

To bootstrap from a link, the **BootFromROM** input is wired to **GND**.

The IMS M212 bootstraps from a link by waiting for the first byte (the control byte) to arrive on any of the two link inputs. If the value of the control byte is two, or greater, it is considered to be a count of the number of bytes to be input. The following bytes are then placed into memory at **MemStart** (#8024) and the IMS M212 begins to execute the input code as a process at low priority by transferring control to **MemStart**. The memory space immediately above the loaded code is used as the process workspace.

The mechanism of bootstrapping from any link allows a network of transputers to be bootstrapped without the need for ROM or any external memory.

5.3.4 Peeking and Poking

A unique feature of the transputer allows any location of transputer memory, be it internal or external, to be read from, or written to a link. This allows, with appropriate software, an external memory system to be debugged without the need to run a program on the system under development. All peeking and poking must have ceased before an attempt is made to bootstrap the device.

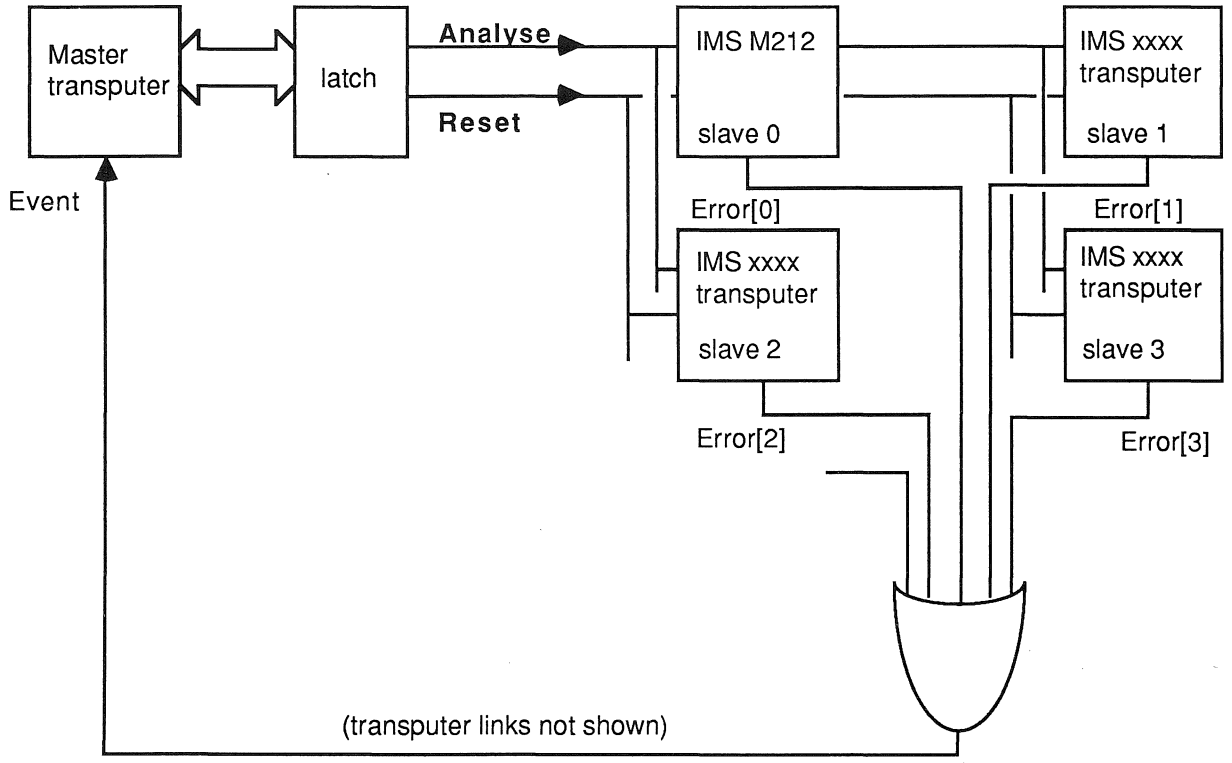
If, whilst the transputer is waiting to boot from a link, it receives a zero byte, then a word address is input, followed by a word of data, which is written to consecutive byte locations from that address. The transputer then returns to the state of awaiting a message from any link.

If the first byte received is one, then a word address is input, a word of data is read from consecutive byte locations from that address and is output down the corresponding output link. The transputer will then return to the state of awaiting a message from any link.

5.4 Using Error and Analyse

Analyse may be used in conjunction with the **Error** output to isolate errors in a multi-transputer system. A transputer which has signalled an error may be halted and subsequently analysed under the control of a 'master' transputer, using the methods of peeking, poking and bootstrapping described above. Alternatively, this could be achieved by connecting the 'OR' of all the **Error** pins to the **EventReq** pin on the master transputer, and connecting the **Analyse** and **Reset** pins to the master transputer as a 'peripheral'.

Error handling in multi-transputer system

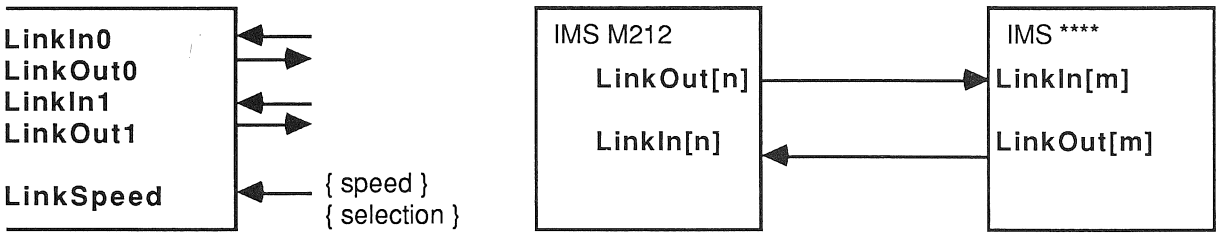


6.1 Standard transputer links

The IMS M212 provides two standard links, each providing two uni-directional point-to-point occam channels.

The links implement the standard inter transputer communications protocol. The links are connected by wiring a **LinkOut[n]** to a **LinkIn[m]** and **LinkIn[n]** to a **LinkOut[m]**. Unused link inputs must be held to ground, **GND**.

Link connection



The IMS M212 links wait until each full byte has been received before outputting the corresponding acknowledge packet. An acknowledge packet can be received at any time following the transmission of a data packet start bit.

At a link speed of 10 Mbits/sec, data is transmitted at about 400 Kbytes/sec in each direction, and the combined (bidirectional) data rate when the link carries data in both directions at once is 800 Kbytes/sec.

At a link speed of 20 Mbits/sec, data is transmitted at about 800 Kbytes/sec in each direction, and the combined (bidirectional) data rate when the link carries data in both directions at once is 1600 Kbytes/sec.

6.1.1 Link speed selection

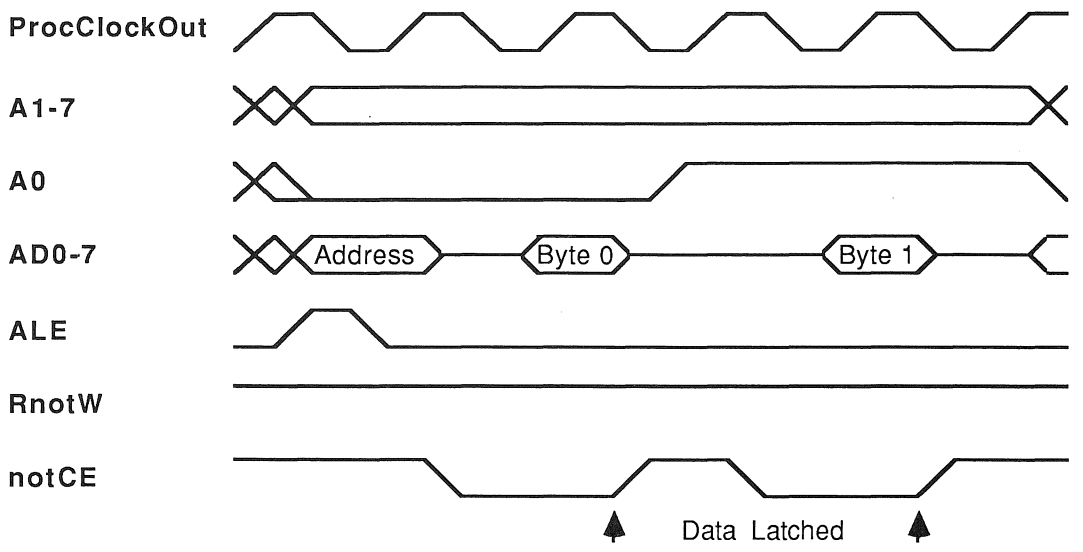
The speed of the links is directly proportional to the frequency supplied on the **ClockIn** signal. The universal speed for all transputer products is twice **ClockIn**, i.e. 10 Mbits/s. A faster speed of 20 Mbits/s can be selected by holding the **LinkSpeed** signal high.

The memory interface consists of an eight bit data bus which is multiplexed with the most significant eight bits of the sixteen bit address bus. Word reads and word writes are performed in five processor cycles with a processor cycle being defined as one cycle of **ProcClockOut**. An address latch enable **ALE** is provided to latch the multiplexed addresses and a read not write pin **RNotW** and not chip enable pin **notCE** control memory access.

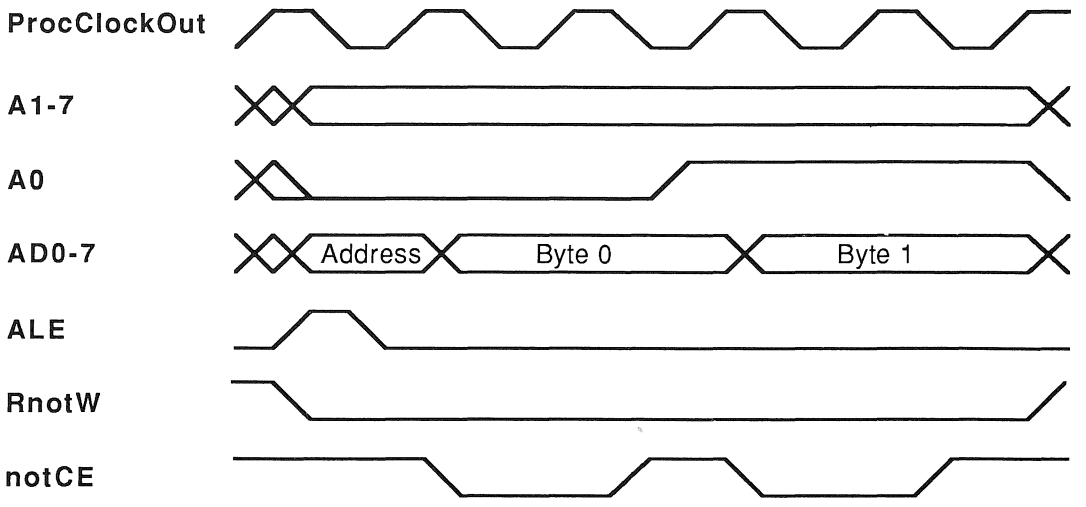
A wait pin is provided to extend the memory cycle in each of the byte accesses. A wait state lasts for a number of whole processor cycles.

The **DisIntROM** signal, when held high stops all access to the internal ROM, allowing access to this address space in external memory.

Read cycle



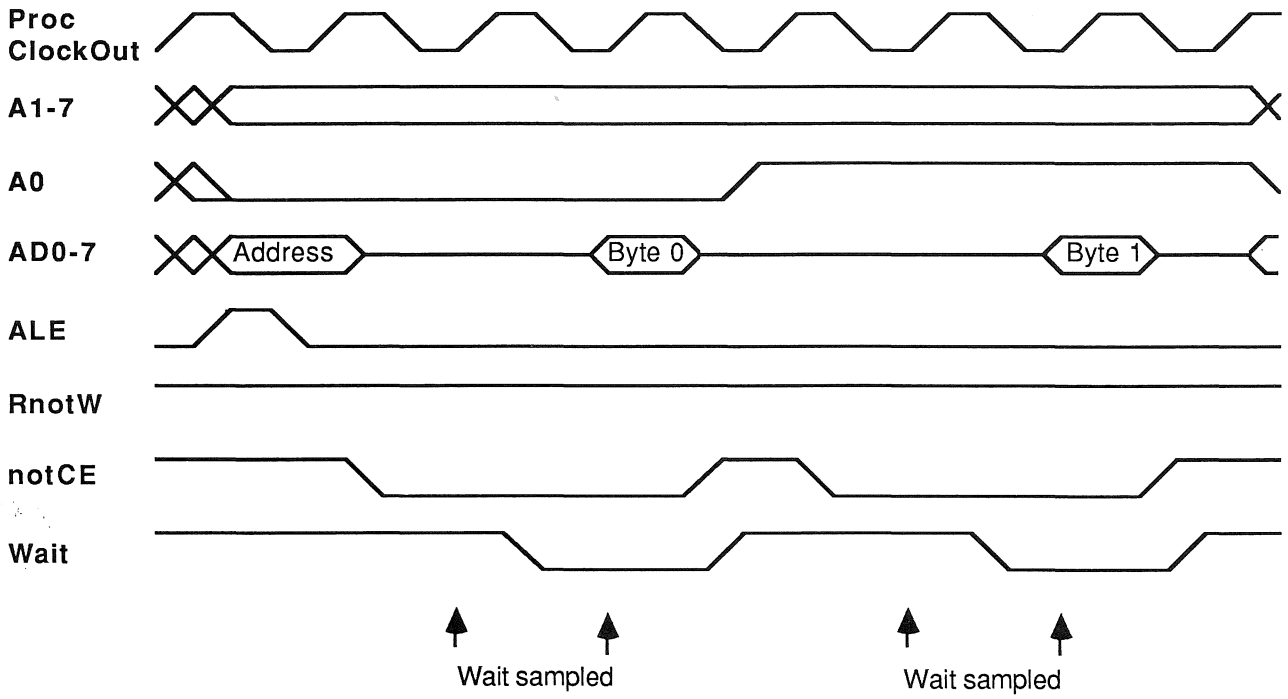
Write cycle



Byte writes are possible with this interface and they are implemented using the **notCE** signal. If the upper byte of a word is to be written, the **notCE** signal is only asserted when **A0** is high. If the lower byte is to be written, the **notCE** signal is only asserted when **A0** is low.

Use of Wait

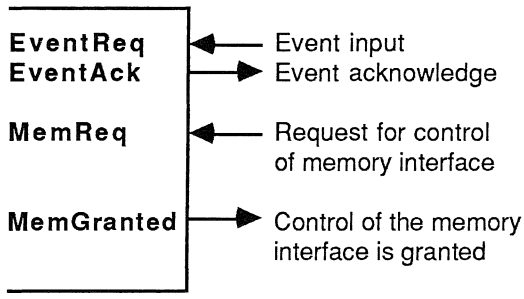
A wait input, **Wait** is provided which may be used with **ProcClockOut** to extend the length of memory cycles thus allowing the external memory system to be designed using devices with differing access times.



When either the internal RAM or the internal ROM, if enabled, is being accessed then the address pins **A1-7** and the address/data pins **AD0-7** will output the value of the internal address bus. The signals **A0**, **RnotW**, **notCE** will remain high and the signal **ALE** will remain low.

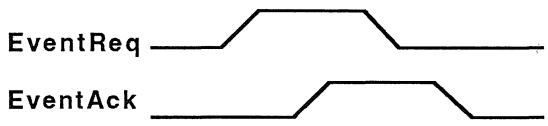
8.1 Event process

Event Request block diagram



The two event signals, **EventReq** and **EventAck**, together provide a handshaken interface with an occam process executing in the processor.

Event request signal protocol



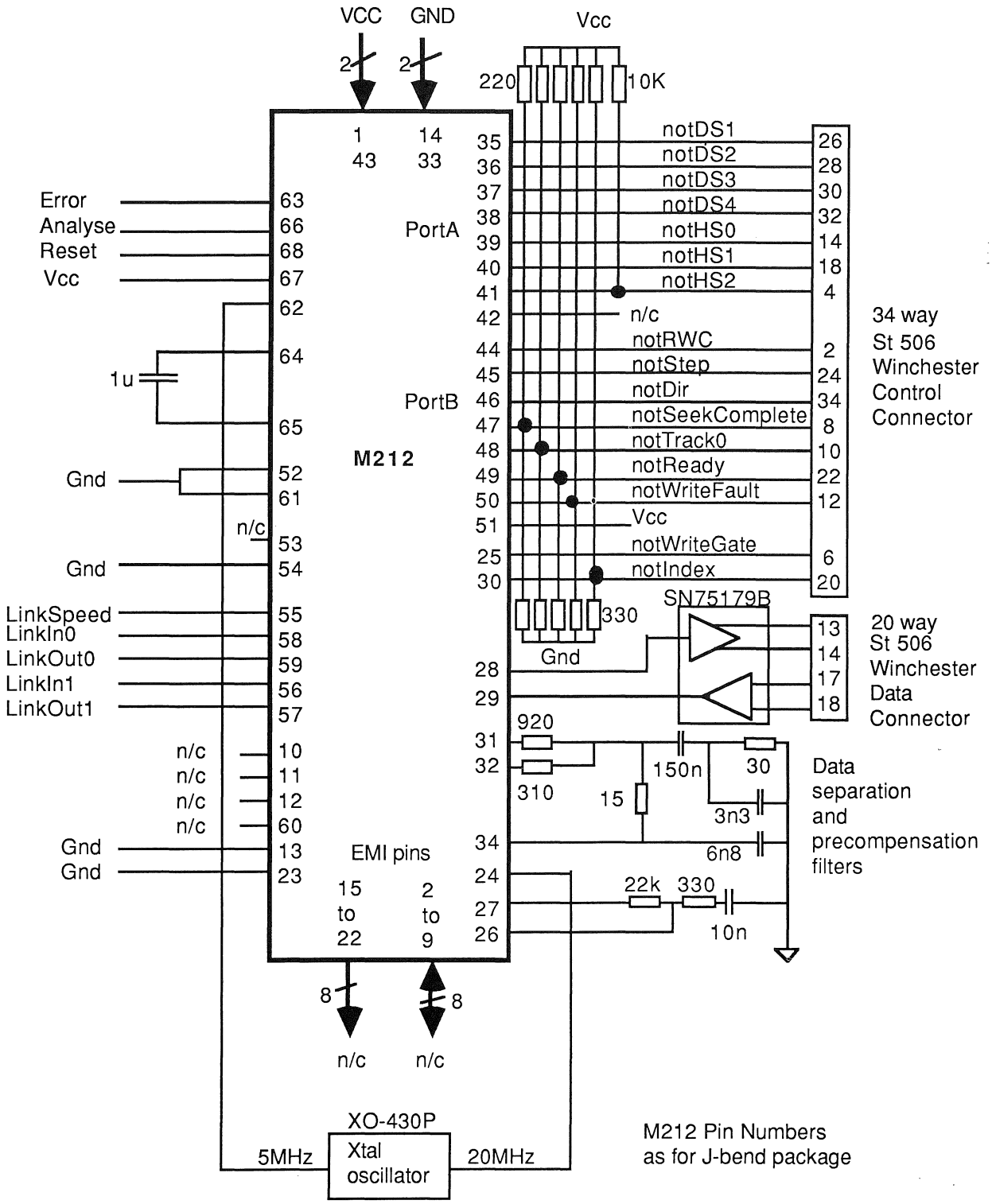
External logic takes **EventReq** high when the logic wishes to communicate with a process in the transputer. The rising edge of **EventReq** makes an external channel ready to communicate with the process. (This channel is additional to the external channels of the links.) When both the channel is ready and a process is ready to input from the channel, then the processor takes **EventAck** high and the process is scheduled. At any time after this point the external logic may take **EventReq** low, following which the processor will set **EventAck** low. After **EventAck** goes low, **EventReq** may go high to indicate the next event. Any further communication or synchronization necessary (for example, to tell external logic that the process has acted in response to the event) must be programmed explicitly.

If the process has high priority, and there is no other high priority process already running, then the maximum latency is 53 processor cycles, assuming that all memory accesses are to on-chip RAM. The typical latency is 19 processor cycles.

EventReq should be held low on reset.

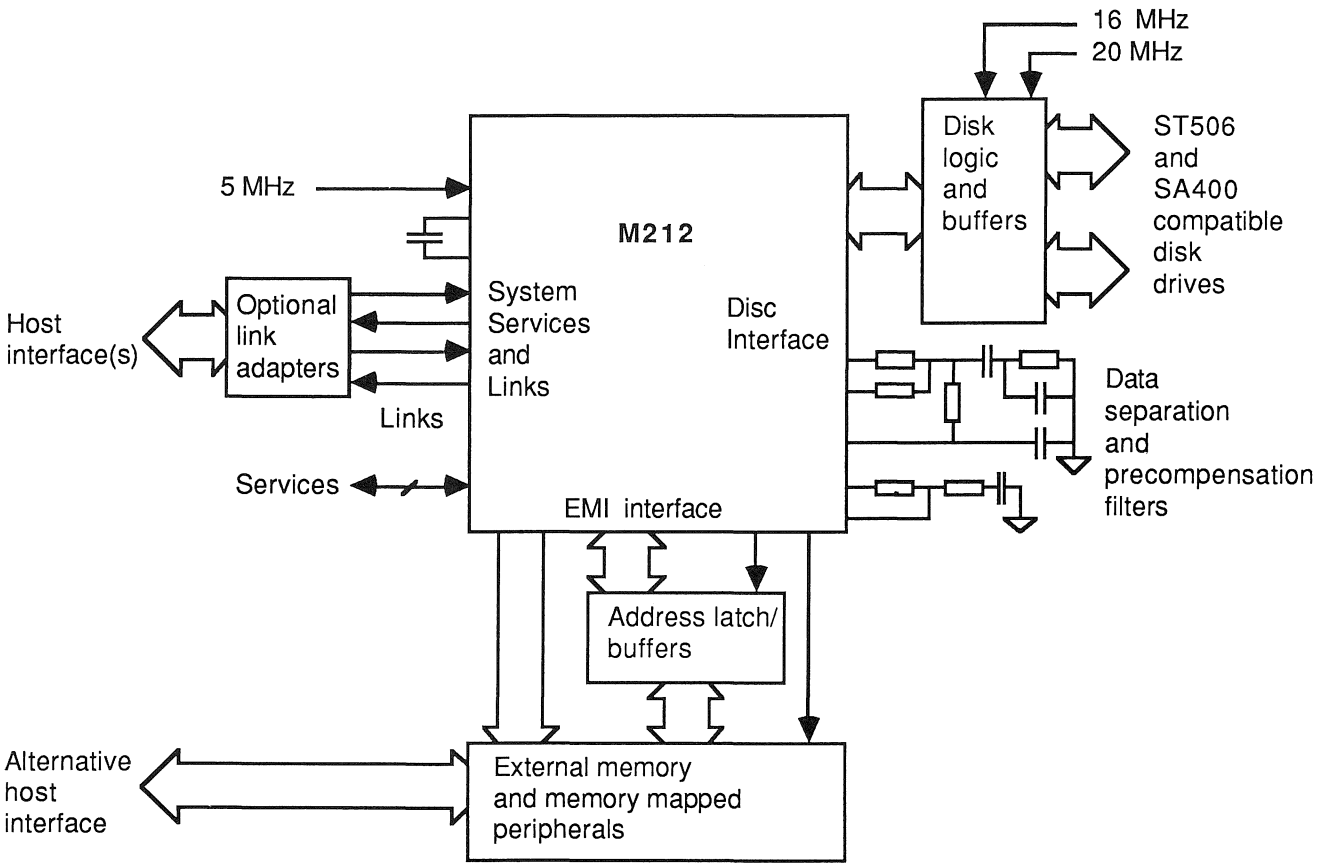
The IMS M212 can interface to a floppy or winchester disk with very little external circuitry required when used in 'mode 1'. A typical arrangement for interfacing to a winchester disk is shown below.

Basic system



If this basic system is enhanced by the addition of external memory the on-chip monitor will automatically use this extra memory space to increase the sector buffer. Alternatively the external memory can be used to provide increased facilities in the disk operating system. An example of a more complex circuit arrangement is shown below.

Enhanced system



The performance of the transputer is measured in terms of the number of bytes required for the program, and the number of (internal) processor cycles required to execute the program. The figures here relate to occam programs. For the same function, other languages should achieve approximately the same performance as occam.

10.1 Performance overview

These figures are averages obtained from detailed simulation, and should be used only as an initial guide; they assume operands are of type **INT**. The following abbreviations are used to represent the quantities indicated.

np number of component processes
ne number of processes earlier in queue
r 1 if INT parameter or array parameter, 0 if not
ts number of table entries (table size)
w width of constant in nibbles
p number of places to shift
Eg expression used in a guard
Et timer expression used in a guard
Tb most significant bit set of multiplier ((-1) if multiplier is 0)

Performance table

	Size (bytes)	**Time** (cycles)
Names		
variables		
in expression	1.1+**r**	2.1+2(**r**)
assigned to or input to	1.1+**r**	1.1+(**r**)
in **PROC** call, corresponding to an **INT** parameter	1.1+**r**	1.1+(**r**)
channels	1.1	2.1
Array Variables (for single dimension arrays)		
constant subscript	0	0
variable subscript	5.3	7.3
expression subscript	5.3	7.3
Declarations		
CHAN OF *protocol*	3.1	3.1
[size]CHAN OF *protocol*	9.4	2.2 + 20.2*__size__
PROC	body+2	0
Primitives		
assignment	0	0
input	4	26.5
output	1	26
STOP	2	25
SKIP	0	0
Arithmetic operators		
+, -	1	1
*****	2	23
/	2	24
REM	2	22
>>, <<	2	3+**p**

	Size (bytes)	**Time** (cycles)
Modulo Arithmetic operators		
PLUS	2	2
MINUS	1	1
TIMES(fast multiply)	1	4+**Tb**
Boolean operators		
OR	4	8
AND, **NOT**	1	2
Comparison operators		
= constant	0	1
= variable	2	3
<> constant	1	3
<> variable	3	5
>, <	1	2
>=, <=	2	4
Bit operators		
/\, \/, ><, ~	2	2
Expressions		
constant in expression	**w**	**w**
check if error	4	6
Timers		
timer input	2	3
timer **AFTER**		
if past time	2	4
with empty timer queue	2	31
non-empty timer queue	2	38+**ne***9
ALT (timer)		
with empty timer queue	6	52
non-empty timer queue	6	59+**ne***9
timer alt guard	8+2**Eg**+2**Et**	34+2**Eg**+2**Et**
Constructs		
SEQ	0	0
IF	1.3	1.4
if guard	3	4.3
ALT (non timer)	6	26
alt channel guard	10.2+2**Eg**	20+2**Eg**
skip alt guard	8+2**Eg**	10+2**Eg**
PAR	11.5+(**np**-1)*7.5	19.5+(**np**-1)*30.5
WHILE	4	12
Procedure call		
	3.5+(nparams-2)*1.1 +nvecparams*2.3	16.5+(nparams-2)*1.1 +nvecparams*2.3
Replicators		
replicated **SEQ**	7.3{+5.1}	(-3.8)+15.1*count{+7.1}
replicated **IF**	12.3{+5.1}	(-2.6)+19.4*count{+7.1}
replicated **ALT**	24.8{+10.2}	25.4+33.4*count{+14.2}
replicated timer **ALT**	24.8{+10.2}	62.4+33.4*count{+14.2}
replicated **PAR**	39.1{+5.1}	(-6.4)+70.9*count{+7.1}

Figures in curly brackets are not necessary if the number of replications is a compile time constant. To estimate performance, add together the time for the variable references and the time for the operation.

10.1.1 Fast multiply, TIMES

The IMS M212 has a fast integer multiplication instruction ('product'). If **Tb** is the position of the most significant bit set in the multiplier, then the time taken for a fast multiply is 4+**Tb**. The time taken for a multiplication by zero is 3 cycles. For example, if the multiplier is 1 the time taken is 4 cycles, if the multiplier is -1 (all bits set) the time taken is 19 cycles. Implementations of high level languages on the transputer may take advantage of this instruction. For example, the occam modulo arithmetic operator **TIMES** is mapped onto the instruction and the right-hand operand is treated as the multiplier.

The fast multiplication instruction is also used in high level language implementations for the multiplication implicit in multi-dimensional array access.

10.1.2 IMS M212 arithmetic

A set of routines are provided within the development system to support the efficient implementation of multiple length and floating point arithmetic. In the following table, **n** gives the number of places shifted and all arguments and results are assumed to be local. Full details of these routines are provided in the occam reference manual supplied as part of the development system and available as a separate publication.

When calculating the execution time of the predefined maths routines, no time needs to be added for calling overhead. These routines are compiled directly into special purpose instructions which are designed to support the efficient implementation of multiple length and floating point arithmetic.

Routine		Cycles	+Cycles for parameter access (Assuming local variables)
LONGADD		2	7
LONGSUM		3	8
LONGSUB		2	7
LONGDIFF		3	8
LONGPROD		18	8
LONGDIV		20	8
SHIFTRIGHT	(**n**<16)	4+**n**	8
	(**n**>=16)	**n**-11	8
SHIFTLEFT	(**n**<16)	4+**n**	8
	(**n**>=16)	**n**-11	8
NORMALISE	(**n**<16)	**n**+6	7
	(**n**>=16)	**n**-9	7
	(**n**=32)	4	7
ASHIFTRIGHT		**SHIFTRIGHT**+2	5
ASHIFTLEFT		**SHIFTLEFT**+4	5
ROTATERIGHT		**SHIFTRIGHT**	7
ROTATELEFT		**SHIFTLEFT**	7
FRACMUL		**LONGPROD**+4	5

10.1.3 Floating point operations

Floating point operations are provided by a run-time package, which requires approximately 2000 bytes of memory for the double length arithmetic operations, and 2500 bytes for the quadruple length arithmetic operations. The following table summarizes the estimated performance of the package.

		Processor cycles (typical)	Processor cycles (worst)
REAL32	+, -	530	705
	*	650	705
	/	1000	1410
	<, >, =, >=, <=, <>	60	60

REAL64	+, -	875	1190
	*	1490	1950
	/	2355	3255
	<, >, =, >=, <=, <>	60	60

10.1.4 Effect of external memory

Extra processor cycles may be needed when program and/or data are held in external memory, depending both on the operation being performed, and on the speed of the external memory. After a processor cycle which initiates a write to memory, the processor continues execution at full speed until at least the next memory access.

The on-chip ROM also requires extra processor cycles as the access time of this memory is equivalent to three processor cycles for each word accessed.

Whilst a reasonable estimate may be made of the effect of external memory, the actual performance will depend upon the exact nature of the given sequence of operations.

External memory is characterized by the number of extra processor cycles per external memory cycle, denoted as **e**. The value of **e** is 4 for no wait states. If program is stored in external memory, the number of extra cycles required for linear code sequences may be estimated at (2**e** - 1)/4 per byte of program. A transfer of control may be estimated as requiring **e** + 3 cycles. The value of **e** for the on chip ROM will be 2. These estimates may be refined for various constructs. In the following table, **n** denotes the number of components in a construct. In the case of **IF**, the **n**'th conditional is the first to evaluate to **TRUE**, and the costs include the costs of the conditionals tested. The number of bytes in an array assignment or communication is denoted by **b**.

	Program off chip	**Data off chip**
Boolean expressions	**e**-1	0
IF	3**en**-1	**en**
Replicated **IF**	6**en**+9e-12	(5**e**-2)**n**+6
Replicated **SEQ**	(4**e**-3)**n**+3e	(4**e**-2)**n**+3-e
PAR	4**en**	3**en**
Replicated **PAR**	(17**e**-12)**n**+9	16**en**
ALT	(4**e**-1)**n**+9**e**-4	(4**e**-1)**n**+9**e**-3
array assignment and communication in one transputer	0	max (2**e**, **eb**)

The following simulation results illustrate the effect of storing program and/or data in external memory. The results are normalized to 1 for both program and data on chip. The first program (Sieve of Erastosthenes) is an extreme case as it is dominated by small, data access intensive, loops; it contains no concurrency, communication, or even multiplication or division. The second program is the pipeline algorithm for Newton Raphson square root computation.

	e	4	**on chip**
Program off chip	(1)	2.1	1.
	(2)	1.6	1
Data off chip	(1)	2.1	1
	(2)	1.6	1
Program and data off chip	(1)	3.0	1
	(2)	2.1	1

10.2 IMS M212 speed selections

The following table illustrates the designation of the IMS M212 speed selections.

Designation	Instruction throughput	Processor clock speed	Processor cycle time	Input clock frequency
IMS M212-15	7.5 MIPS	15 MHz	67 ns	5 MHz
IMS M212-20	10 MIPS	20 MHz	50 ns	5 MHz

Parameters given in this section will be revised as a result of fuller characterization.

11.1 Absolute maximum ratings

Parameter		Min	Max	Unit	Note
VCC	DC supply voltage	0	7.0	V	1, 2, 3
VI,VO	Input or output voltage on any pin	-0.5	**VCC**+0.5	V	1, 2, 3
II	Input current		±25	mA	4
OSCT	Output short circuit time (one pin)		1	S	1
TS	Storage temperature	-65	150	°C	1
TA	Ambient temperature under bias	-55	125	°C	1
PD	Power dissipation rating		1	W	

Notes

1 Stresses greater than those listed may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions beyond those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect reliability.

2 All voltages are with respect to **GND**.

3 This device contains circuitry to protect the inputs against damage caused by high static voltages or electric fields; however it is advised that normal precautions be taken to avoid application of any voltage higher than the absolute maximum rated voltages to this high impedance circuit. Reliability of operation is enhanced if unused inputs are tied to an appropriate logic voltage level such as **GND**.

4 The input current applies to any input or output pin and applies when the voltage on the pin is between **GND** and **VCC**.

11.2 Recommended operating conditions

Parameter		Min	Max	Unit	Note
VCC	DC supply voltage	4.75	5.25	V	1
VI,VO	Input or output voltage	0	**VCC**	V	1,2
CL	Load capacitance on any pin		50	pF	
TA	Operating temperature range	0	70	°C	3

Notes

1 All voltages are with respect to **GND**.

2 Excursions beyond the supplies are permitted but not recommended; see DC characteristics.

3 Ambient temperature in still air.

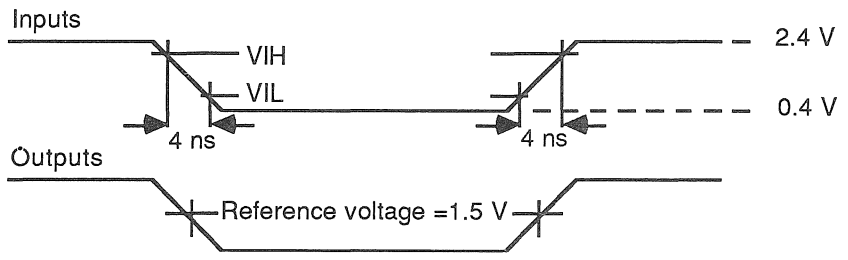
11.3 DC characteristics

Parameter		Conditions	Min	Max	Unit
VIH	High level input voltage		2.0	**VCC**+0.5	V
VIL	Low level input voltage		-0.5	0.8	V
II	Input current	**GND**<**VI**<**VCC**		±10	μA
VOH	Output high voltage	**IOH**=2mA	**VCC**-1		V
VOL	Output low voltage	**IOL**=4mA		0.4	V
IOS	Output short circuit current	**GND**<**VO**<**VCC**		50	mA
IOZ	Tristate output current	**GND**<**VI**<**VCC**		±10	μA
PD	Power dissipation			0.7	W
CIN	Input capacitance	f=1 MHz		7	pF
COZ	Output capacitance tristate	f=1 MHz		10	pF

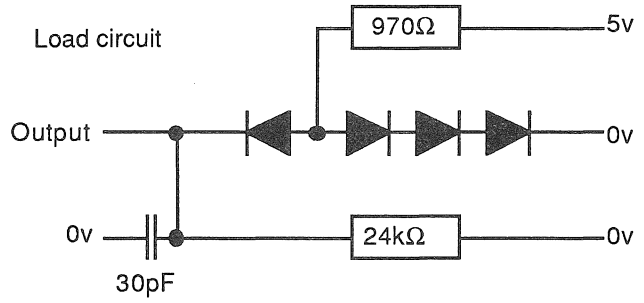
Note :
4.5 V < **VCC** < 5.5 V
0 °C < **TA** < 70 °C
Input clock frequency = 5 MHz
All voltages are with respect to **GND**

11.4 Measurement of AC characteristics

Reference points for AC measurements



Load circuit for AC measurements



The load circuit approximates to two Schottky TTL loads, with total capacitance of 30 pF.

11.5 Connection of INMOS serial links

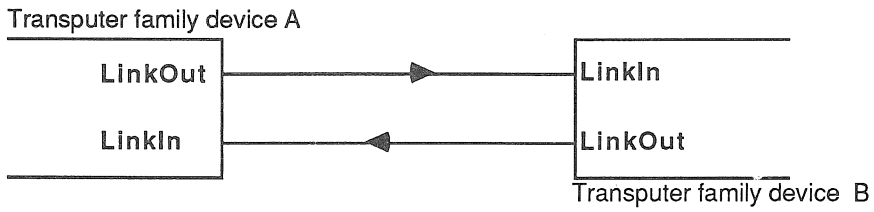
INMOS serial links can be connected in 3 different ways depending on their environment:

1 Directly connected

2 Connected via a series matching resistor

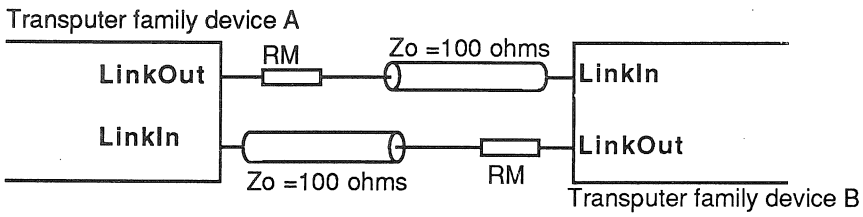
3 Connected via buffers

Direct connection



Direct connection is suitable for short distances on a printed circuit board.

Matched line

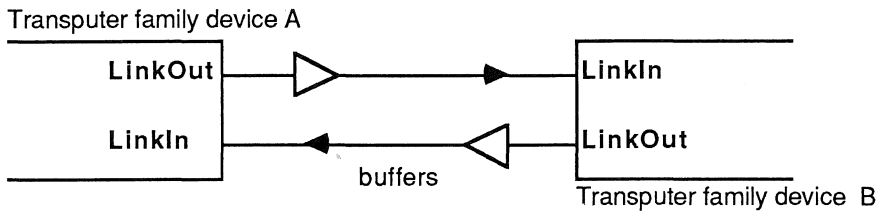


For long wires, approximately >30 cm, then a 100ohm transmission line should be used with series matching resistors.

Parameter		Nom	Max	Unit
RM	Series matching resistor for 100 ohm line.	56		ohm
TD	Delay down line		0.4	bit time

Note that if two connected devices have different values for **TD**, the lower value should be used. With series termination at **LinkOut** the transmission line reflection must return within 1 bit time. Otherwise line buffers should be used.

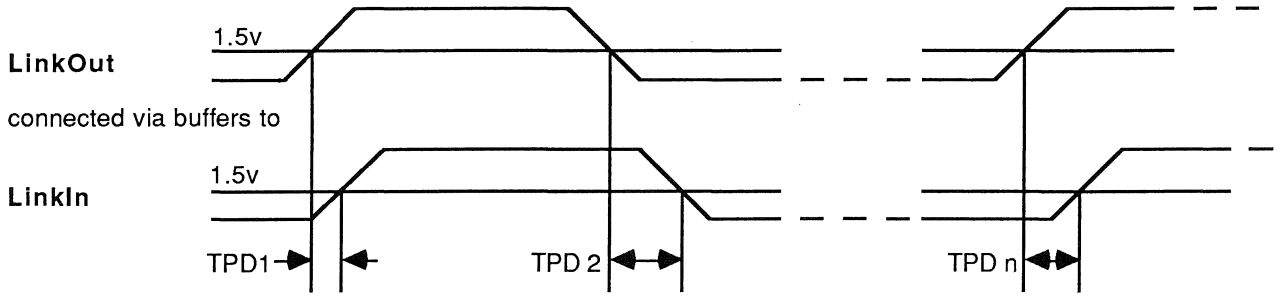
Buffered links



If buffers are used their overall propagation delay, TPD, should be stable within the skew tolerance.

Parameter	Max	Unit
Skew in buffering at 10 Mbits/sec	10	ns
Skew in buffering at 20 Mbits/sec	3	ns
Rise and fall time of **LinkIn** (10% to 90%)	20	ns

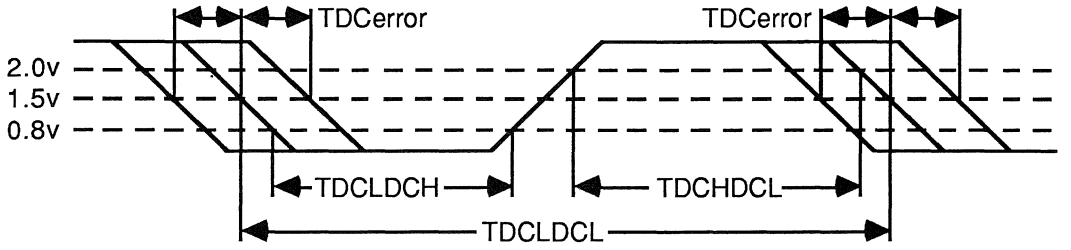
The above figures indicate that buffered links can be realised at 10 and 20 Mbits/sec.



The absolute value of TPD is immaterial because data reception is asynchronous. However, TPD will vary from moment to moment because of ground noise, variation in the power supplies of buffers and the difference in the delay for rising and falling edges. This will vary the length of data bits reaching **LinkIn**. *Skew* is the difference between the maximum and minimum instantaneous values of TPD.

11.6 AC characteristics of system services

ClockIn waveform



Parameter		Min	Nom	Max	Unit	Note
TDCHDCL	Clock pulse width high	40			ns	1
TDCLDCH	Clock pulse width low	40			ns	1
TDCr	Rise time of **ClockIn**			10	ns	1
TDCf	Fall time of **ClockIn** (10% to 90%)			8	ns	1
TDCLDCL	Clock period	200	200	400	ns	2
TDCerror	**ClockIn** timing error			±0.5	ns	7
TDC1DC2	Difference in frequencies of **ClockIn** for two devices connected by a link			400	ppm	6
Cap	Decoupling capacitor between **CapPlus** and **CapMinus**	1			μF	3
TRHRL	**Reset** pulse width high	8			**ClockIn** periods	4
	Time **VCC** must be valid and **Clockin** running before **Reset** goes low	10			ms	5
	BootFromROM setup time before **Reset** or **Analyse** goes low	0				
	BootFromROM hold time after **Reset** goes low	50			ms	
	Analyse pulse width	8			**ClockIn** periods	

Notes

1 The clock transitions must be monotonic within the range between **VIH** and **VIL**.

2 The **TCLCL** parameter is measured between corresponding points on consecutive falling edges.

3 A 1 μF ceramic low inductance, low leakage capacitor must be connected between **CapPlus** and **CapMinus** to decouple the supply to the on-chip clock generator. It should preferably have an impedance less than 3 ohms between 100 KHz and 20 MHz. PCB track lengths to the capacitor should be minimised. No power supply should flow in these tracks.

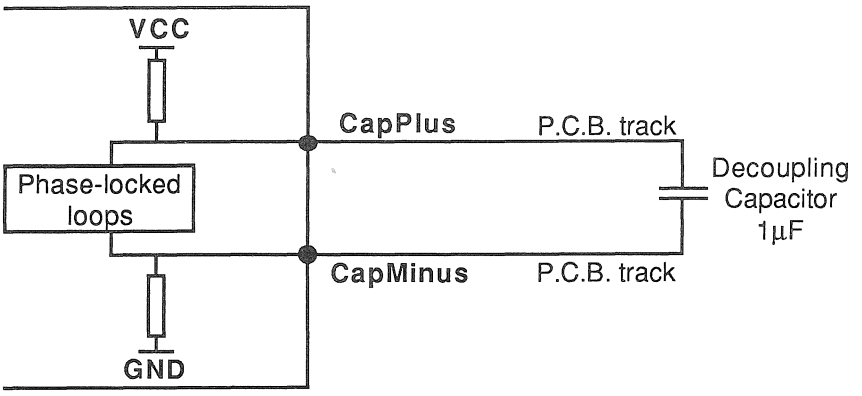
4 Link inputs must be held low during reset. Reset forces link outputs low.

5 At power on reset.

6 This value allows the use of low cost 200ppm crystal oscillators.

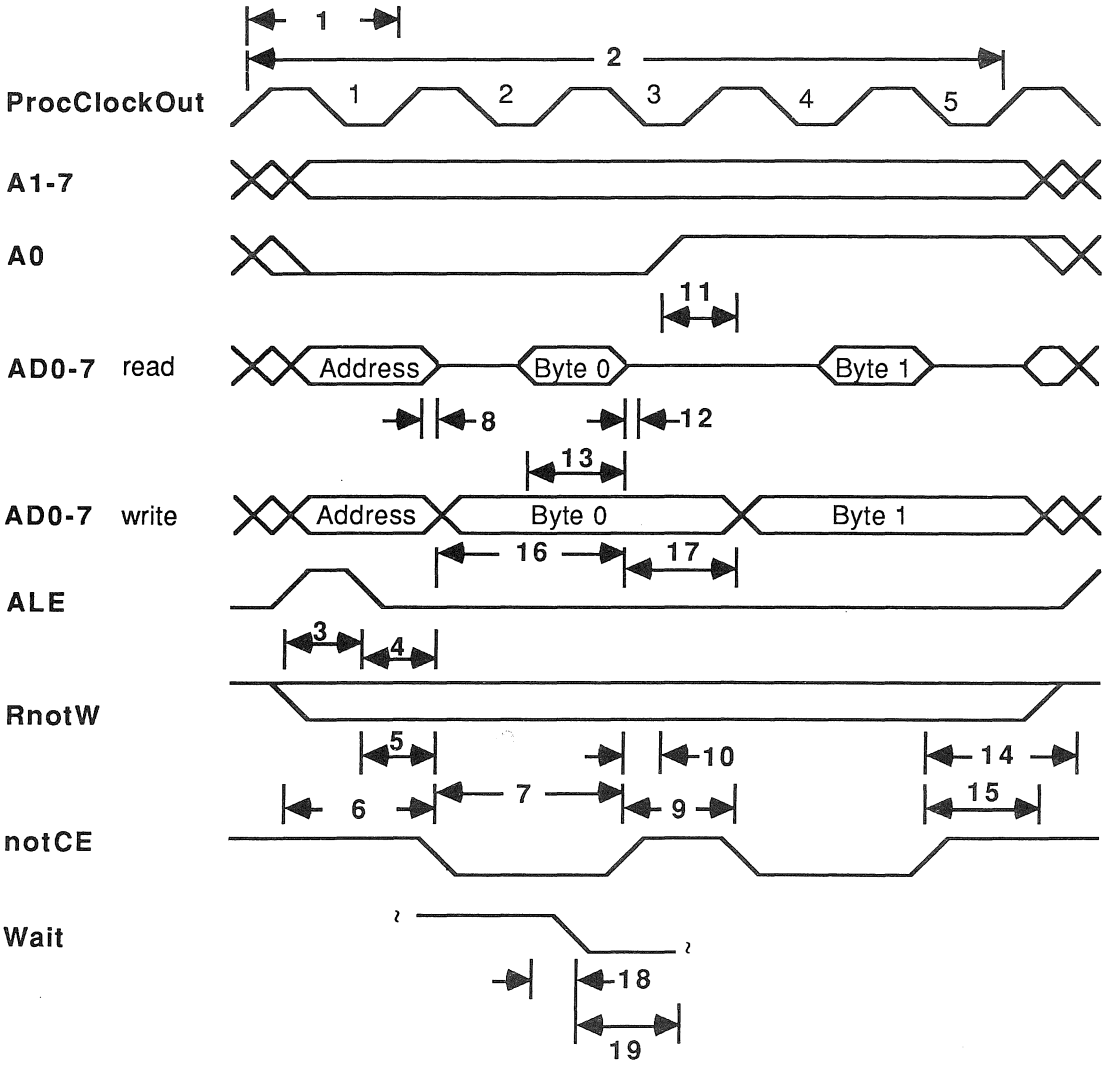
7 Variation of individual falling edges from their nominal times.

Recommended PLL Decoupling.



11.7 Memory interface AC characteristics

The AC characteristics of the memory interface are dependent upon the speed of the transputer and the use of wait states.

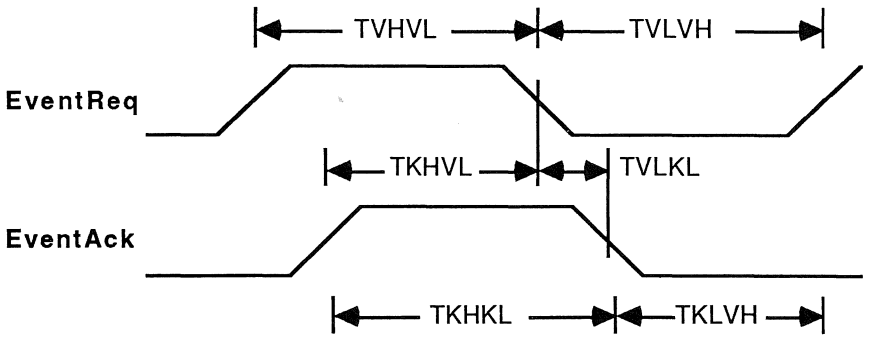


The diagram illustrates the parameters given in the table below. Parameters given in this section will be revised as a result of fuller characterization. All figures are in nanoseconds.

	Parameter	IMS M212-20	IMS M212-15	
1	**ProcClocOut** period	50	66.6	typical
2	EMI cycle (no wait)	250	333	typical
3	Address set up to **ALE**	20	28	minimum
4	Address hold from **ALE**	20	28	minimum
5	**ALE** to **notCE**	22	30	minimum
6	**RnotW** to **notCE**	45	60	minimum
7	**notCE** low period	55	75	minimum
8	Address tristate to **notCE**	0	0	minimum
9	**notCE** high period	32	40	minimum
10	**notCE** to**A0** byte change	10	12	minimum
11	**A0** byte change to**notCE**	22	28	minimum
12	Read data hold after **notCE**	0	0	minimum
13	Read data setup before **notCE**	20	20	minimum
14	**notCE** to **ALE**	32	40	minimum
15	**notCE** to address or **RnotW**	32	40	minimum
16	Write data setup to **notCE**	55	75	minimum
17	Write data hold from **notCE**	32	40	minimum
18	**Wait** hold from **ProcClocOut** high	0	0	minimum
19	**Wait** setup to **ProcClocOut** high	20	20	minimum

11.8 Peripheral interfacing AC characteristics

Event signals waveforms



Parameter		Min	Max	Unit
TVHVL	**EventReq** pulse width high	2		processor cycles
TVLVH	**EventReq** pulse width low	2		processor cycles
TVLKL	Falling edge delay from **EventReq** to **EventAck**	0	2	processor cycles
TKHKL	**EventAck** pulse width high	2		processor cycles
TKHVL	Delay from **EventAck** to falling edge of **EventReq**	0		
TKLVH	Delay from falling edge of **EventAck** to next **EventReq**	0		

11.9 Disk Interface Parameters

Parameter		Min	Typ	Max	Unit	Notes
1	Pulse on **PIA** to affect Change Detect	20			ns	
2	**notIndex** Pulse width	16			Bit cells	
3	**WriteClock** Frequency	0.25		40	MHz	1
4	**Write Clock** Mark Space	40		60	%	2
5	**WriteData** Pulse Width		0.25		Bit cells	3
6	**ReadData** Pulse Width	20			ns	
7	**PhaseUp** On resistance to vcc			100	ohms	
8	**PhaseDn** On resistance to gnd			100	ohms	
9	Jitter on Data Separator			± 10	ns	4

Notes

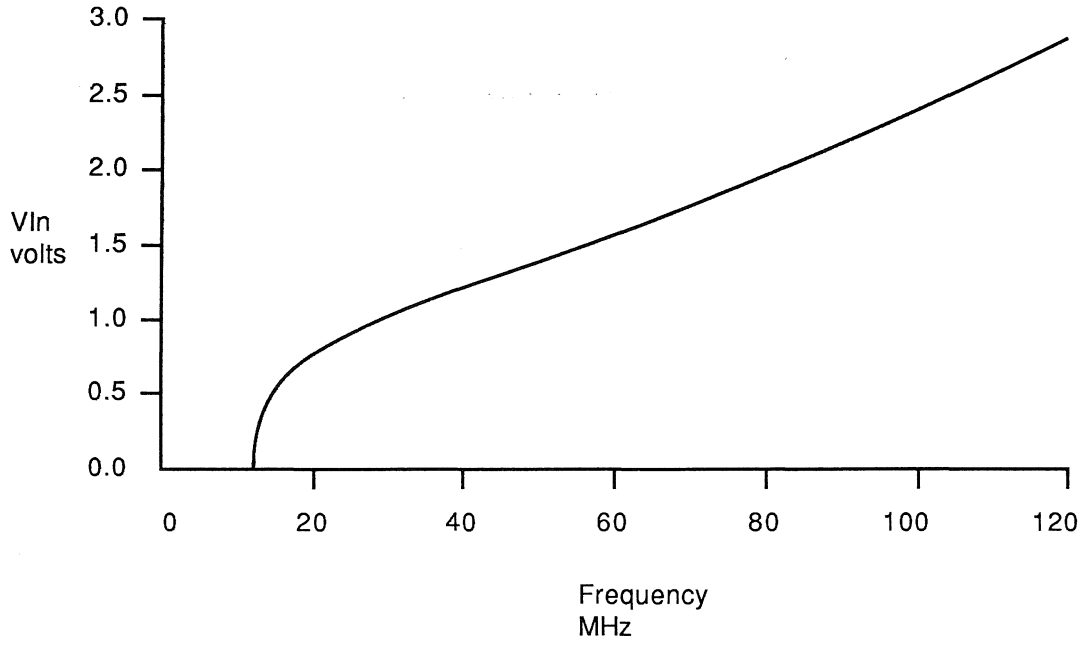
1 **WriteClock** frequency limits depend also on the division ratio to generate the internal reference clock.

2 **WriteClock** mark-space ratio limits only apply if the division ratio to generate the internal reference clock is set to one. With any other division ratio the minimum high and low periods of **WriteClock** is 20 ns.

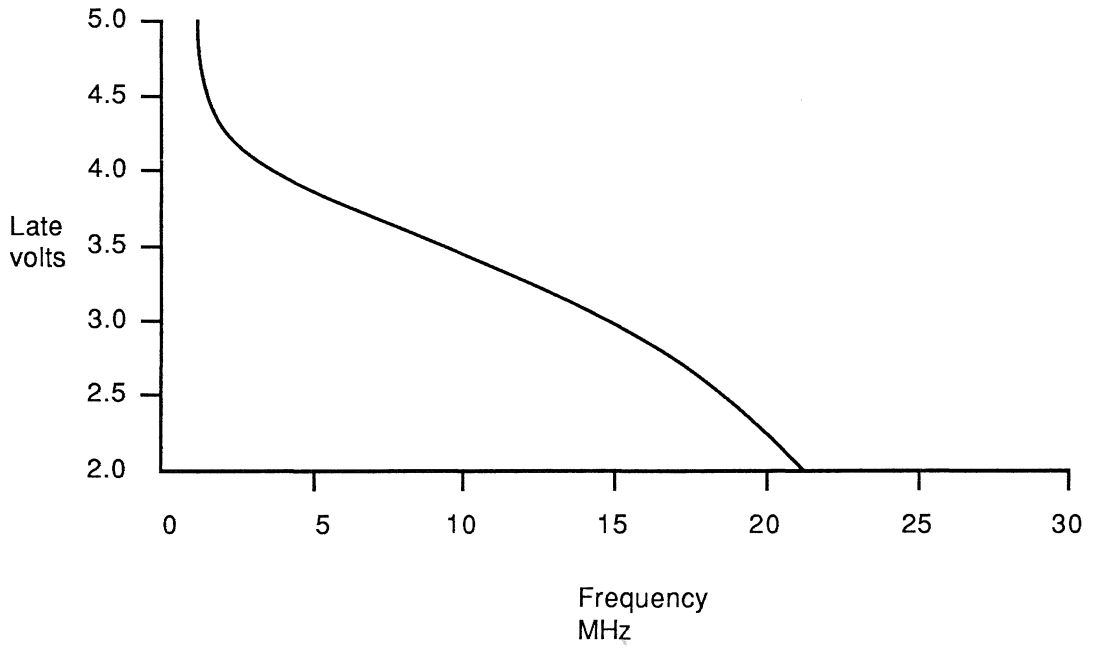
3 **WriteData** pulse width tolerance will depend on the **WriteClock** input and the division ratio to generate the internal reference clock.

4 The jitter on the data separator is a function of the external filter components and the board layout of these components and the device power supply decoupling.

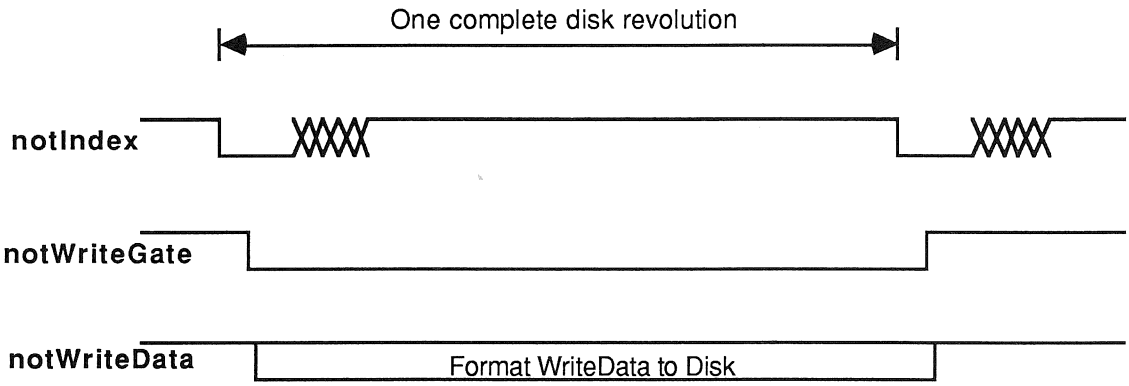
VIn voltage versus Frequency for Data Separation VCO



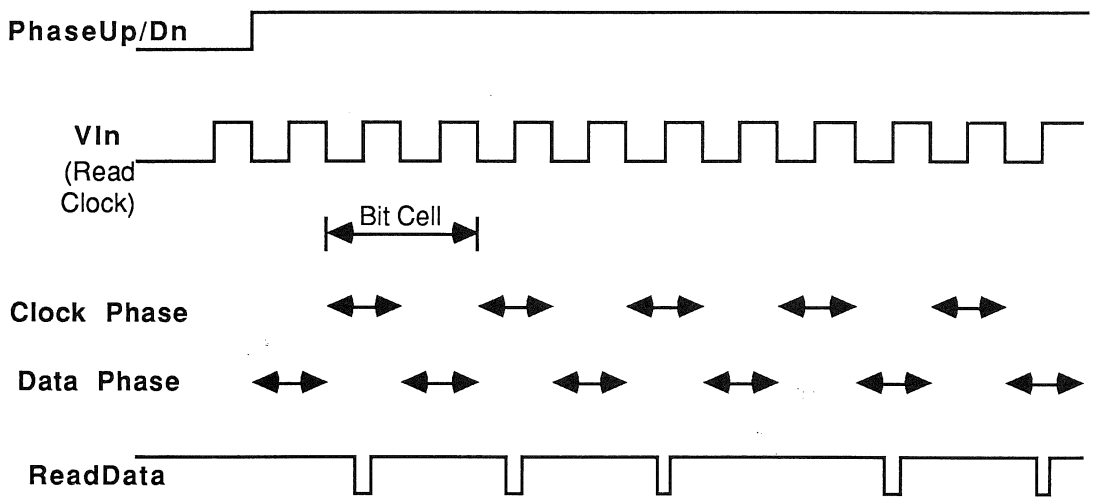
Late voltage versus Frequency for Precompensation VCO



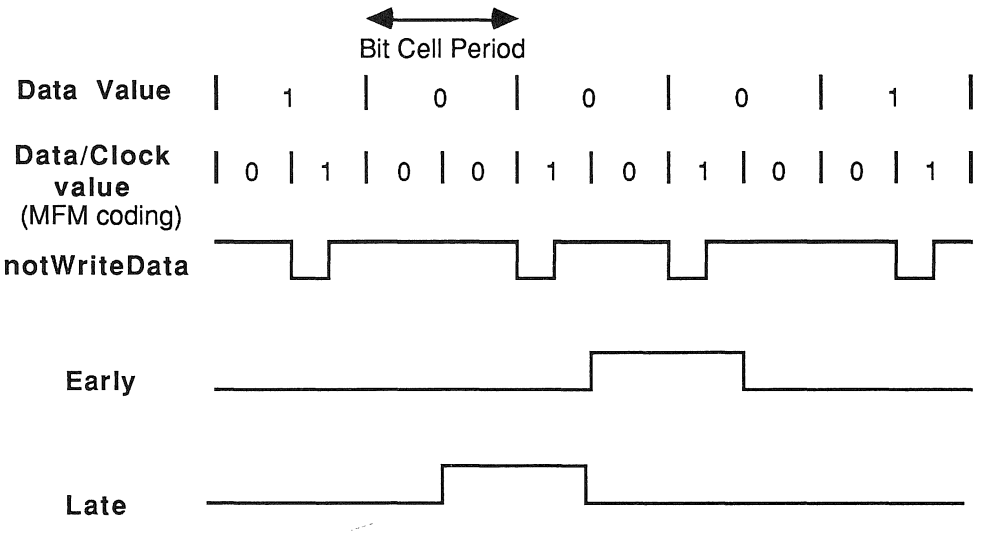
notIndex and notWriteGate Timing during format



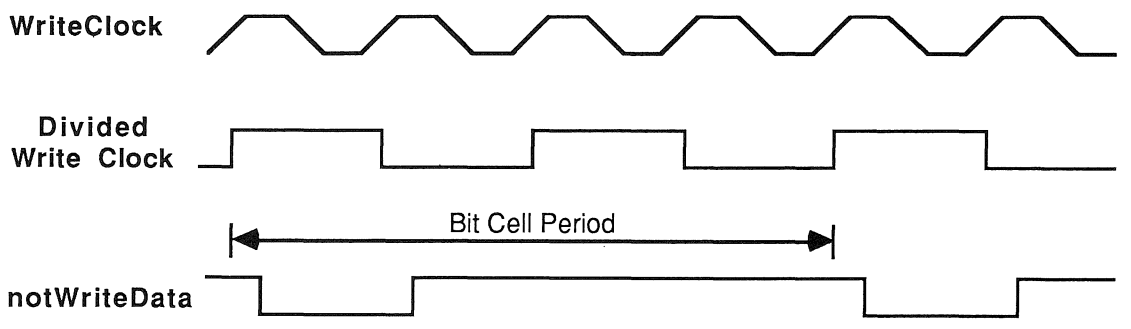
Read Clock / Read Data Timing for external separation



Early / Late Timing to notWriteData for external Precompensation



WriteClock and notWriteData timing with internal Precompensation



Note: Hard Parameter 'Data Separation' (Register #32)
WriteClockReferenceDivision set to divide by two

12.1 Signal list

A0 - 7 8-bit wide lower byte address bus for external memory expansion.
Output

AD0 - 7 8-bit wide multiplexed upper byte address bus and data bus for external memory expansion.
Input/Output

ALE Latches the multiplexed address during an external memory cycle.
Output

Analyse Signal used to investigate the state of a IMS M212 . The signal brings the processor, links and clocks to a halt within approx three timeslice periods (approx 3 milliseconds) depending on the condition of the status of the disk interface. **Reset** may then be applied, which will not destroy state information . After **Reset** has been taken low, **Analyse** may be taken low after which the processor will execute its bootstrap routine.
Input

BootFromROM If this is held to **VCC**, then the boot program is taken from ROM. Otherwise the IMS M212 awaits a bootstrap message from a link.
Input

CapMinus The negative terminal of an internal power supply used for the internal clocks. This must be connected via a 1 microfarad capacitor to **CapPlus**. If a tantalum capacitor is used **CapMinus** should be connected to the negative terminal.

CapPlus The positive terminal of an internal power supply used for the internal clocks. This must be connected via a 1 microfarad capacitor to **CapMinus**. If a tantalum capacitor is used **CapPlus** should be connected to the positive terminal.

ClockIn Input clock from which all internal clocks are generated. The nominal input clock frequency for all transputers, of whatever wordlength and speed, is 5 MHz.
Input

DisIntROM This pin should be held low in order to use the ROM disk procedures.
Input

Early Provides connection to a filter for the on-chip precompensation. See connection diagram in section 9 and Appendix E. This pin also provides an indication of writing a data bit early if external precompensation is selected and enabled. See also **Late**.
Input

Error The processor has detected an error. This pin remains high until **Reset**. Errors may result from arithmetic overflow, array bound violations or division by zero. The Error flag can also be explicitly set by an instruction to allow other forms of software detected error.
Output

EventAck The **EventReq** and **EventAck** are a pair of handshake signals for external events. External logic takes **EventReq** high when the logic wishes to communicate with a process in the IMS M212 . The rising edge of **EventReq** causes a process to be scheduled. The processor takes **EventAck** high when the process is scheduled. At any time after this point the external logic may take **EventReq** low, following which the processor will set **EventAck** low.
Output

EventReq Request external event. See description of **EventAck** above.
Input

GND Power supply return and logic reference, 0 V. There are several **GND** pins to minimize inductance within the package.

HoldToGND These are input pins reserved for INMOS use. They should be held to ground either directly or through a resistor of less than 10K ohms.
Input

Late Provides connection to a filter for the on-chip precompensation. See connection diagram in section 9 and Appendix E. This pin also provides an indication of writing a data bit late if external precompensation is selected and enabled. See also **Early**.
Input/Output

LinkIn0 - 1 Input pins of the standard links. A **LinkIn** pin receives a serial stream of bits including protocol bits and data bits. Link inputs must be connected to another Link output or tied to **GND**. The Link input must not be tied high or left floating.
Input

LinkOut0 - 1 Output pins of each of the standard links. A **LinkOut** pin may be left floating or may be connected to one (and only one) **LinkIn** pin. As long as the skew specification is met, the connection may be via buffers.
Output

LinkSpeed This signal effects the speed of the links. If low link speed is 10 Mbits/sec and if high link speed is 20 Mbits/sec.
Input

MemWait Wait input for the memory interface. If, at the time **MemWait** is sampled, the input is low, the interface cycle proceeds. Otherwise, the interface is held until the input is sampled and found to be low.
Input

notCE This active low pin provides a chip enable for the external memory system.
Output

notIndex This is a timing reference signal from the disk drive occurring once every revolution.
Input

notWriteData This pin is used to output serial data to the disk drive.
Output

notWriteGate This indicates to the disk drive that valid write data is being output from the controller.
Output

PA 0 - 7 Programable input/output pins.
Input/Output

PB 0 - 7 Programable input/output pins.
Input/Output

PhaseUp/Dn These signals provide connection to an external filter for an on-chip data separator. When using external data separation these pins can be connected together to provide an indication of read mode.
Output

ProcClockOut The processor clock which is output in phase with the memory interface. The processor clock frequency is a multiple of the input clock frequency. The multiple differs for different speed parts.
Output

ReadData This is used to input raw data from the disk drive.
Input

Reset The falling edge of **Reset** initialises the IMS M212 and then starts the processor executing a bootstrap routine.
Input

RnotW When high, this signal denotes a read operation, when low, a write.
Output

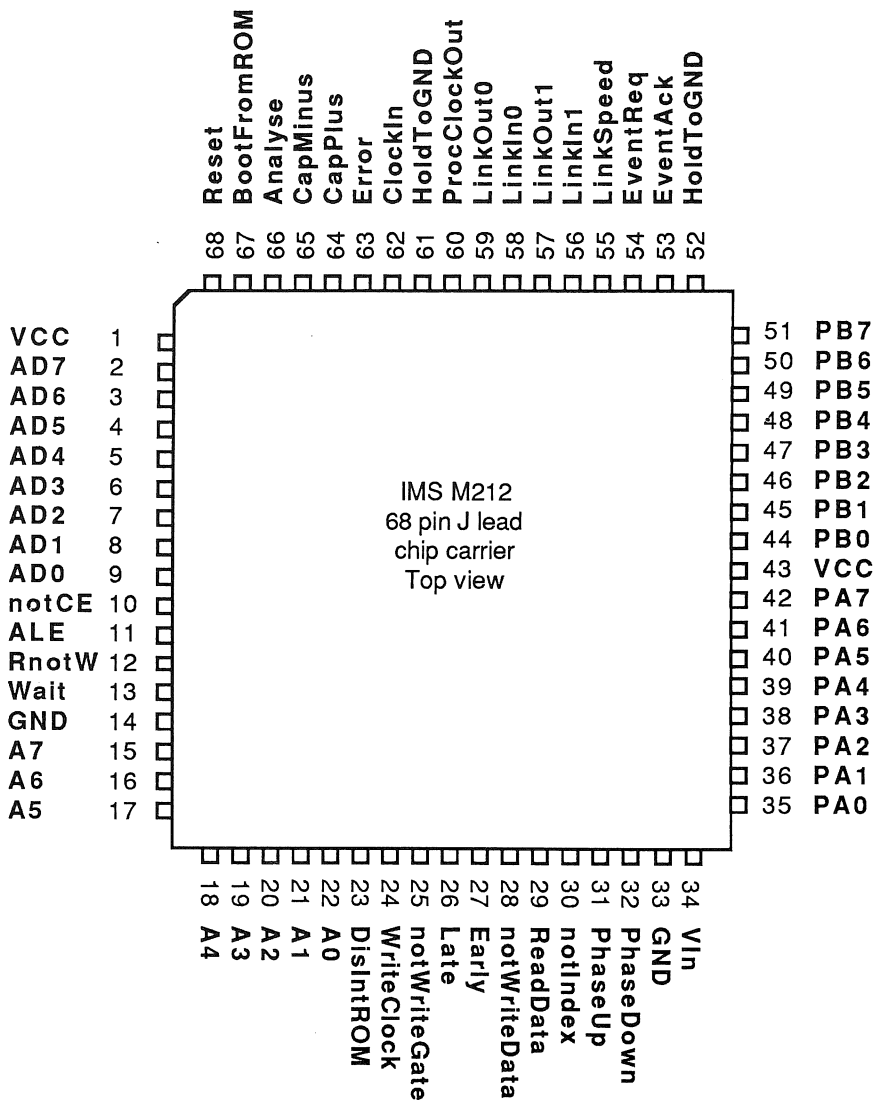
VCC Power supply, nominally 5 V. There are several **VCC** pins to minimize inductance within the package. **VCC** should be decoupled to **GND** by at least one 100 nF low inductance (such as ceramic) capacitor.

Vin This pin provides connection to an external filter for an on-chip data separator. When using external data separation this pin should have a read clock input connected.
Input

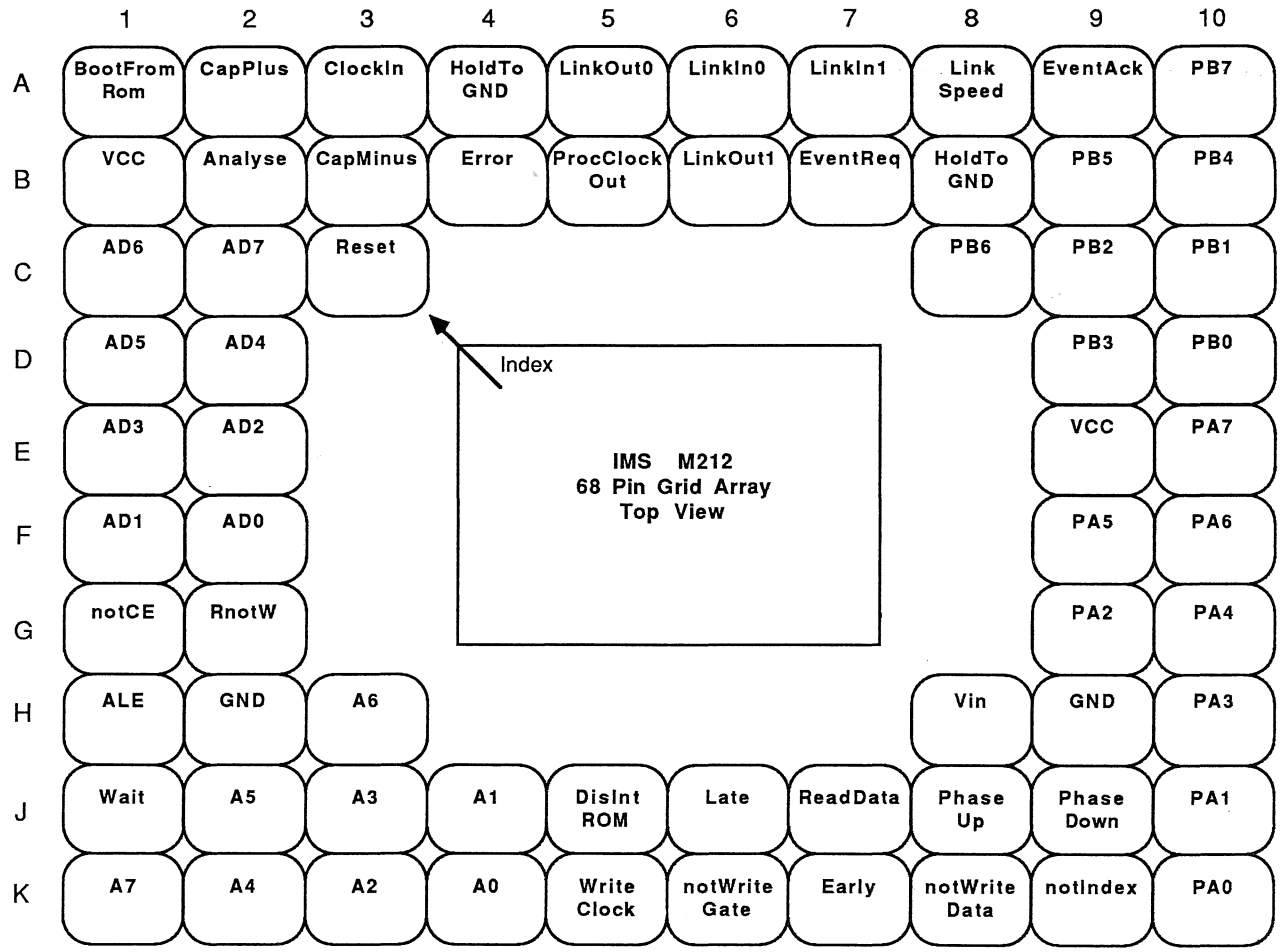
WriteClock This is a reference frequency to the disk hardware during writing. This signal must be applied continuously when an external clock reference is required.
Input

The IMS M212 is available in an 68 pin J-Lead chip carrier or 68 pin Grid Array.

13.1 J-Lead chip carrier

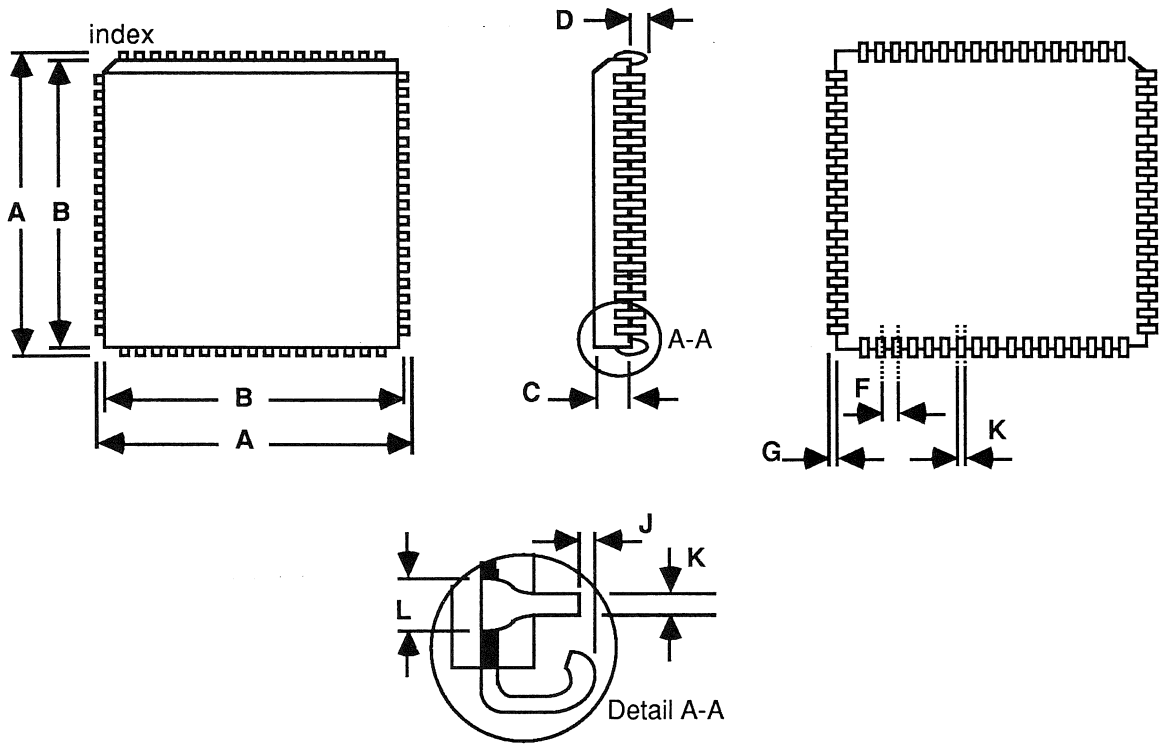


13.2 Pin Grid Array



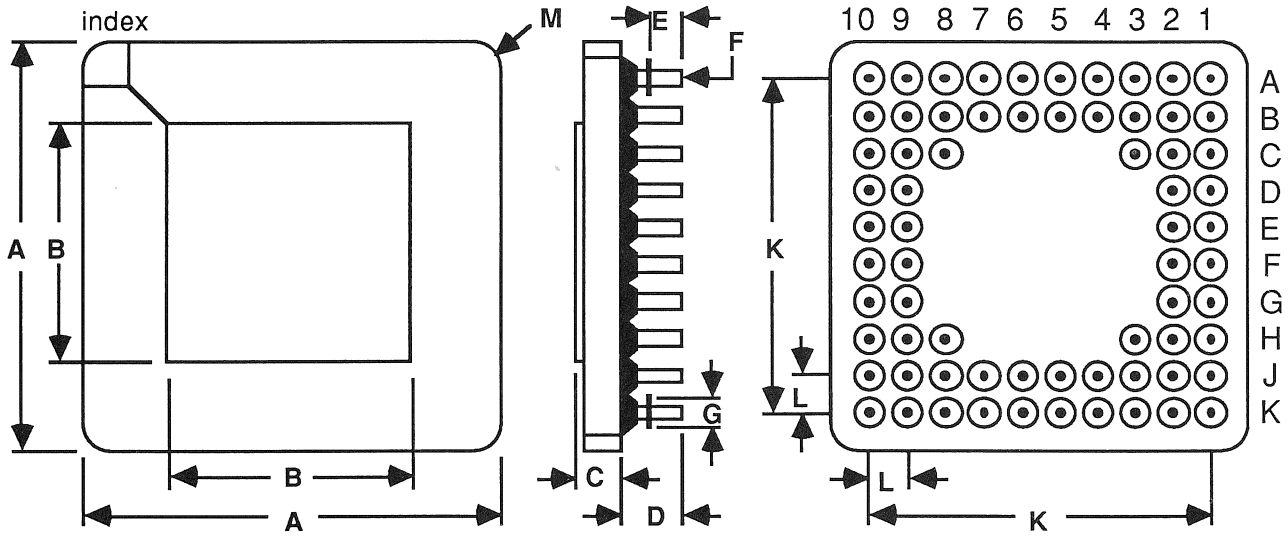
13.3 Package Dimensions

13.3.1 J-Lead Chip Carrier



DIM	Millimetres		Inches		NOTES
	NOM	TOL	NOM	TOL	
A	30.226	±0.127	1.190	±0.005	
B	29.312	±0.127	1.154	±0.005	
C	3.810	±0.127	0.150	±0.005	
D	0.508	±0.127	0.020	±0.005	
F	1.270	±0.127	0.050	±0.005	
G	0.457	±0.127	0.018	±0.005	
J	0.000	±0.051	0.000	±0.002	
K	0.457	±0.127	0.018	±0.005	
L	0.762	±0.127	0.030	±0.005	

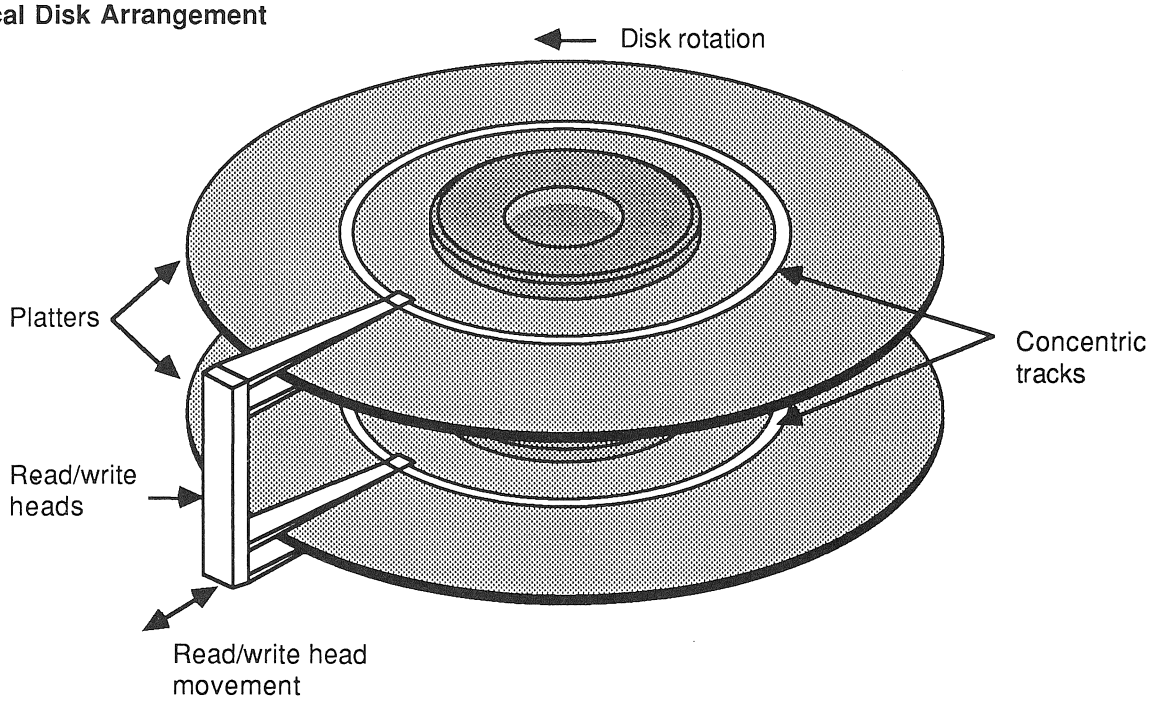
13.3.2 Pin Grid Array



DIM	Millimetres		Inches		NOTES
	NOM	TOL	NOM	TOL	
A	26.924	±0.254	1.060	±0.010	
B	17.019	±0.127	0.670	±0.005	
C	2.456	±0.278	0.097	±0.011	
D	4.572	±0.127	0.180	±0.005	
E	3.302	±0.127	0.130	±0.005	
F	0.457	±0.025	0.018	±0.001	Pin diameter
G	1.143	±0.127	0.045	±0.005	Flange diameter
K	22.860	±0.127	0.900	±0.005	
L	2.540	±0.127	0.100	±0.005	
M	0.508		0.020		Chamfer

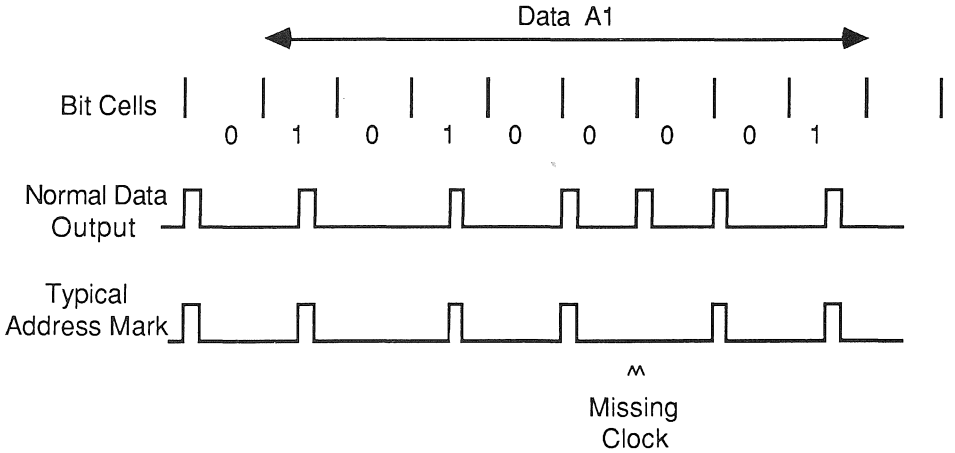
All disks consist of one or more circular platters which rotate about their common centre. The surfaces of the platters are coated with a magnetic substance on which data can be recorded. Each surface to be used for recording data has a read/write head positioned over it, so that as the disk rotates a ring of the surface, known as a track, moves under the head. In order to know where the start of the track should be positioned, once every revolution an index pulse is produced. The heads may be moved in or out across the surfaces (all the heads are moved together) so that a number of concentric tracks may be accessed. This is known as *seeking*. A set of tracks that may be accessed without moving the heads (i.e. one track for each head) is known as a cylinder. Most disk drives have a control signal which indicates when the heads are positioned on the outermost track available, this is generally called Track 0.

Typical Disk Arrangement



Each track has a number of blocks of data written on it. These are known as records and are the smallest addressable unit on a disk. On standard formats there are two fields per record - the Identification (ID) field which is then followed by the Data field. The disk controller must be able to find the start of each field, and so a special pattern of information is pre-written onto each track. This operation is known as formatting the disk. The patterns written during formatting include special markers called address marks (AM). These do not follow the normal rules for encoding data so that they can be distinguished from normal data by the disk controller. Not only are address marks used to identify the position of fields, but because data is written serially they are used to align the data into bytes.

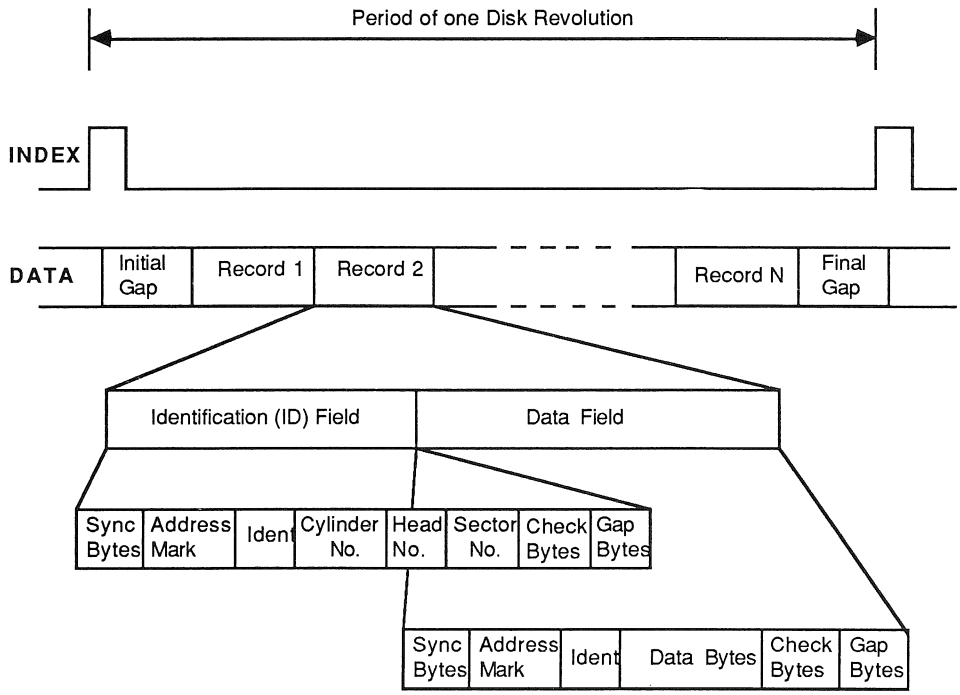
Address Mark Coding (MFM format)



There are two more types of information on tracks - gaps and sync bytes. Gaps are used to provide delays between the ID and Data fields to allow time for the disk controller and drive to change from reading to writing and also to provide protection against drive motor speed variation. Sync bytes precede the address marks and are a regular data pattern that allows the disk controller to synchronise itself to the data rate and phase of the data coming from the disk.

The ID field is used to uniquely identify the particular sector which follows. The ID field will start with an address mark and possibly a byte to indicate that this is an ID field, not a Data field. Standard formats include the sector number and will generally also include the cylinder and head. Some formats also include details such as sector length in the ID field. A check code is also usually appended to the end of the ID field to check that it has been read correctly. The ID field is only written during a format - sector writes only update the Data field which follows the ID Field. This consists of an address mark and possibly a byte to indicate a Data field, followed by the sector data itself and probably a check code. Each time a sector is written the whole Data field including the preceding sync bytes and AM are also written to ensure that it is aligned with the sector data correctly.

Typical Track Format



One additional type of field used on some floppies is the index address mark which is placed at the beginning of each track. Although this is not used by most controllers it is usually written during a floppy format operation as required for compatibility.

Data is written onto the magnetic surface of a disk as a sequence of flux transitions. If these transitions are close together then when the data is read off a disk the transitions appear to move away from each other due the the properties of the recording media and heads. This reduces the margins for data recovery and so to overcome this a technique known as pre-compensation is used. This shifts transitions that are close to each other even closer together so that when read back they appear in the correct place. Different disk manufacturers specify different amounts of pre-compensation to be used, and different tracks on which to use it.

Another problem can occur on the inner tracks of the disk where the same amount of data has to be written onto a shorter length of track. As well as possibly needing pre-compensation, some disks require the write current in the heads to be reduced on the inner tracks. They have an input signal, usually called Reduced Write Current, which must be set for the inner tracks.

Address	Register
#00	**PIAPortAData**
#01	**PIAPortADirection**
#02	**PIAPortAChange**
#03	**PIAPortAPin**
#04	**PIAPortBData**
#05	**PIAPortBDirection**
#06	**PIAPortBChange**
#07	**PIAPortBPin**
#08	**DataFieldAMData**
#09	**DataFieldAMClock**
#0A	**IDFieldAMData**
#0B	**IDFieldAMClock**
#0C	**DataAMMissingClock**
#0D	**IDAMMissingClock**
#0E	**IndexAMMissingClock**
#10	**IDCompare0**
#11	**IDCompare1**
#12	**IDCompare2**
#13	**IDCompare3**
#14	**IDCompare4**
#15	**IDCompare5**
#16	**IDCompare6**
#17	**IDCompare7**
#18	**DataCompare**
#1A	**StatusRegister0**
	bits 7-0: **IDCompareError**
#1B	**StatusRegister1**
	bit 0: **DataCompareError**
	bit 1: **IDCrcEccError**
	bit 2: **DataCrcEccError**
	bit 3: **Overrun**
	bit 4: **Underrun**
	bit 5: **InvalidAMGroup**
	bit 6: **NoValidAMGroup**
	bit 7: **Timeout**
#20	**CRCPolynomial0**
#21	**CRCPolynomial1**
#24	**ECCPolynomial0**
#25	**ECCPolynomial1**
#26	**ECCPolynomial2**
#27	**ECCPolynomial3**

Address	Register
#28	**SyndromeByte0**
#29	**SyndromeByte1**
#2A	**SyndromeByte2**
#2B	**SyndromeByte3**
#2C	**Control**
	bit 0: **DataPresetNotReset**
	bit 1: **IDPresetNotReset**
	bit 2: **MFMNotFM**
	bit 3: **EnablePrecompensation**
	bits 5-4: **DataFieldCrcEccMode**
	bits 7-6: **IDFieldCrcEccMode**
#30	**Precompensation0**
	bits 3-0: **OntimeDelay**
	bit 4: **DivideBy2Not4**
	bit 5: **TestMode**
	bit 6: **WinchesterNotFloppy**
	bit 7: **ExternalPrecompensation**
#31	**Precompensation1**
	bits 3-0: **EarlyDelay**
	bits 7-4: **LateDelay**
#32	**DataSeparation**
	bits 2-0: **VCOFrequencyDivision**
	bits 5-3: **WriteClockReferenceDivision**
	bit 6: **PositiveReadData**
	bit 7: **ExternalDataSeparation**
#34	**TimingRegister0**
	bits 1-0: **ValidAddressMarkNumberLess1**
	bits 4-2: **CheckableIDBytesLess1**
	bits 7-5: **TimeoutIndexNumberLess2**
#35	**TimingRegister1**
	bits 2-0: **SectorSize**
	bit 3: **DataCompareEnable**
	bits 7-4: **SoftwareCrcEccSize**
#36	**DelayAfterIDLess3**
#37	**RESERVED**: must be set to #00
#38	**UncheckableIDBytes**
#39	**Operation**
#3A	**ReadSpecial0**
#3B	**ReadSpecial1**
#3C	**ValidAMCount**

This appendix describes the various different correction algorithms that can be used when a non-zero ECC syndrome has been found. Three methods are described, each of which has its own set of advantages and disadvantages.

Throughout this appendix, reference is made to 'the standard polynomial'. This is
$(x^{21} + x^0)(x^{11} + x^2 + x^0) = x^{32} + x^{23} + x^{21} + x^{11} + x^2 + x^0$
and it can correct a sector with a length of up to $21(2^{11} - 1)$ = 42987 bits.

The derivation of the algorithms is beyond the scope of this appendix. In particular, the chinese correction algorithm is specific to the standard polynomial used, although the general form will be the same for others. Also, although any polynomial may be programmed into the M212, many will not have the desired properties for an ECC. For further information on the mathematics of ECC's and suitable polynomials see any book on codes such as Error Correcting Codes by W.Peterson and E.Weldon (MIT Press).

Note that correction is normally only performed after performing several retries, typically 16, all of which have failed, or after two successive retries yield the same value of syndrome.

All the following examples have an undefined process called 'InvertBit' which takes as its parameter a bit index into the data buffer. The numbering of the bits includes all the bits that were ECC'ed and also the ECC itself. As an example, consider a standard 256 byte winchester sector which will have its bits numbered as follows:

Address mark	Data mark	256 data bytes			ECC bytes		
A1	F8	D0	... D255	ECC3	ECC2	ECC1	ECC0
2095-2088	2087-2080	2079-2072	... 39-32	31-24	23-16	15-8	7-0

It would be more efficient to align the error pattern to a byte or word boundary and then exclusive-or the error pattern with the data directly, rather than a bit at a time. 'InvertBit' is just intended to make it clear which bits should be corrected. The other simplification that has been made to keep the algorithms clear is to assume a 32 bit wordlength for the integers.

Note that there is a small but finite probability of an error being mis-corrected. The probability will depend on the polynomial being used and also on the size and distribution of the errors on the disk.

There is also the possibility that an error will be generated which is defined as correctable but when the bit index is calculated it does not point to a valid sector location (in the above example outside the range of 0 - 2095). This type of error should be considered to be uncorrectable.

C.1 Normal correction

This is the standard correction algorithm which will work with any appropriate ECC polynomial.

```
PROC ForwardCorrect( VAL INT Syndrome, BOOL Corrected )

  VAL BurstLength IS 11 :
  VAL RegLength   IS 32 :
  VAL CodeLength  IS 42987 :
  VAL Polynomial  IS #00A00805 :
                     -- x^32 + x^23 + x^21 + x^11 + x^2 + x^0
  VAL RegBits     IS #FFFFFFFF :
  VAL ZeroesMask  IS RegBits >> BurstLength :
  VAL TopBit      IS #80000000 :

  INT Shifts :
  INT ErrorPattern :

  SEQ
    -- get error pattern and number of shifts
    Shifts := 0
    ErrorPattern := Syndrome
    WHILE ((ErrorPattern /\ ZeroesMask) <> 0) AND
          (Shifts < CodeLength)
      SEQ
        IF
          (ErrorPattern /\ TopBit) = 0
            ErrorPattern := ErrorPattern << 1
          TRUE
            ErrorPattern := (ErrorPattern << 1) >< Polynomial
        Shifts := Shifts + 1

    -- try to correct
    IF
      (ErrorPattern /\ ZeroesMask) = 0
        SEQ
          SEQ ErrorPatternBit = 0 FOR BurstLength
            SEQ
              Shifts := Shifts + 1
              IF
                (ErrorPattern /\ TopBit) <> 0
                  InvertBit( CodeLength - Shifts )
                TRUE
                  SKIP
              ErrorPattern := ErrorPattern << 1
          Corrected := TRUE
      TRUE
        Corrected := FALSE
:
```

Unfortunately, this algorithm implies that the sector is a full 42987 bits long (using the standard polynomial). Suppose we use the standard winchester sector which has 2096 bits, then there are an extra 42987-2096 implicit zero bits at the front of the sector. The algorithm finds the errors starting at the most significant bits which are the implicit zeroes, and so at least 40891 shifts will have to be performed before the error can be found. This makes the method extremely inefficient, particularly for small sector sizes, although with a 4 Kbyte sector the overhead will not be too great.

C.2 Reverse correction

To get around the problem of the implicit zero bits, the reverse polynomial can be used. This method finds the bits in error starting from the least significant bit, and so for a standard winchester sector a maximum of 2096 shifts will be required. However, the reverse polynomial must exist to use this method, the restriction being that the polynomial must be of the form $x^{32} + ... + x^0$.

```
PROC ReverseCorrect( VAL INT Syndrome, BOOL Corrected )

  VAL BurstLength IS 11 :
  VAL RegLength   IS 32 :
  VAL CodeLength  IS 42987 :
  VAL Polynomial  IS #00A00805 :
                     -- x^32 + x^23 + x^21 + x^11 + x^2 + x^0
  VAL RegBits     IS #FFFFFFFF :
  VAL ZeroesMask  IS RegBits << BurstLength :
  VAL TopBit      IS #80000000 :

  INT Shifts :
  INT ErrorPattern :
  INT RevPolynomial :

  SEQ
    -- get reverse polynomial
    RevPolynomial := (Polynomial >> 1) \/ TopBit

    -- get error pattern and number of shifts
    Shifts := 0
    ErrorPattern := Syndrome
    WHILE ((ErrorPattern /\ ZeroesMask) <> 0) AND
          (Shifts < CodeLength)
      SEQ
        IF
          (ErrorPattern /\ 1) = 0
            ErrorPattern := ErrorPattern >> 1
          TRUE
            ErrorPattern := (ErrorPattern >> 1) >< RevPolynomial
        Shifts := Shifts + 1

    -- try to correct
    IF
      (ErrorPattern /\ ZeroesMask) = 0
        SEQ
          SEQ ErrorPatternBit = 0 FOR BurstLength
            SEQ
              Shifts := Shifts - 1
              IF
                (ErrorPattern /\ TopBit) <> 0
                  InvertBit( Shifts )
                TRUE
                  SKIP
              ErrorPattern := ErrorPattern << 1
          Corrected := TRUE
      TRUE
        Corrected := FALSE
:
```

An uncorrectable error has been found when the number of shifts performed exceeds the sector size, so it is not actually necessary to complete all 42987 shifts. For large sector sizes, however, the maximum number of shifts can still be large (32816 for a 4 Kbyte sector).

C.3 Chinese correction

This is the quickest algorithm for large sector sizes. The algorithm is more complex, however, and the choice of polynomial is still further restricted - the polynomial used must be capable of being factorised into specific types of polynomials and also the syndrome has to be reset, not preset. In the case of the M212 hardware, the polynomial must be of the form $(x^{21} + ... + x^0)(x^{11} + ... + x^0)$.

The maximum number of shifts ever required by the chinese correction algorithm using the standard polynomial is $21 + (2^{11} - 1) = 2068$. Thus for sectors of 512 bytes or more it will probably be quicker to use the chinese method, but at the expense of more code.

Note that when using chinese correction on the M212, the **ECCPolynomial** registers must be set up for the multiplied-out polynomial and **Control** for ECC during write, but for the factorised polynomial and chinese during read. The term $(x^{21} + ... + x^0)$ occupies the most significant 21 bits of the **ECCPolynomial** registers and the term $(x^{11} + ... + x^0)$ occupies the lower 11 bits. The terms $(x^{21} + x^0)$ and x^{11} are implicit in the **ECCPolynomial** registers in chinese mode and these bits must not be set. For example, using the standard polynomial, the following values would be used:

	write	read
ECCPolynomial3	#00	#00
ECCPolynomial2	#A0	#00
ECCPolynomial1	#08	#00
ECCPolynomial0	#05	#05
DataFieldCrcEccMode	#1	#0
DataPresetNotReset	#0	#0

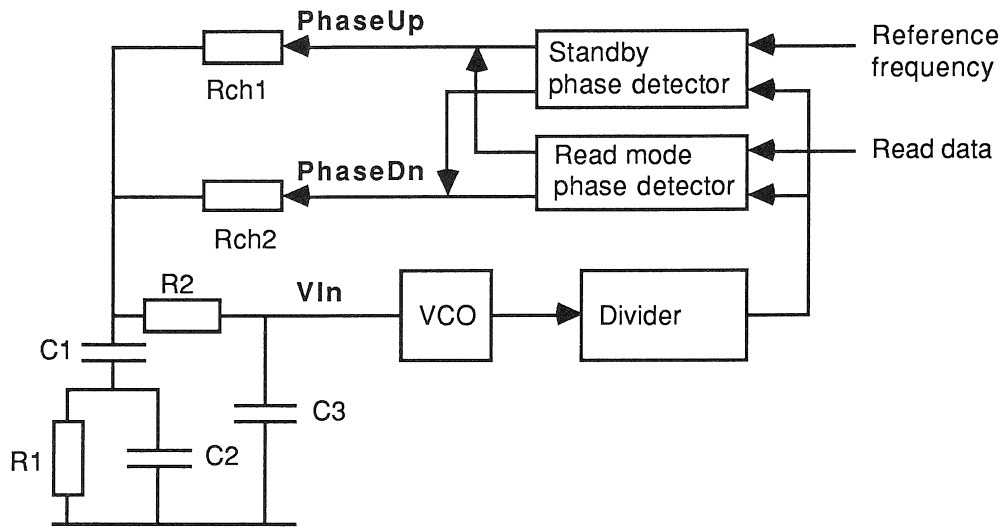
Therefore the write operation is the same whether ECC or chinese mode is being used, only the read is different. Thus it is possible to choose the correction algorithm at the time of the read, so long as the syndrome was reset for the write.

```
PROC ChineseCorrect( VAL INT Syndrome, BOOL Corrected )

  VAL BurstLength IS 11 :
  VAL CodeLength  IS 42987 :

  VAL ARegLength  IS 21 :
  VAL ACodeLength IS 21 :
  VAL APolynomial IS #00000001 : -- x^21 + x^0
  VAL ARegBits    IS #001FFFFF :
  VAL AZeroesMask IS RegBits >> BurstLength :
  VAL ATopBit     IS #00100000 :
  VAL AFactor     IS 38893 :

  VAL BRegLength  IS 11 :
  VAL BCodeLength IS 2047 :
  VAL BPolynomial IS #00000005 : -- x^11 + x^2 + x^0
  VAL BRegBits    IS #000007FF :
  VAL BZeroesMask IS RegBits >> BurstLength :
  VAL BTopBit     IS #00000400 :
  VAL BFactor     IS 4095 :

  INT Shifts, AShifts, BShifts :
  INT ErrorPattern, AErrorPattern, BErrorPattern :
```

```
SEQ
  -- get number of A shifts
  AShifts := 0
  AErrorPattern := (Syndrome >> BRegLength) /\ ARegBits
  WHILE ((AErrorPattern /\ AZeroesMask) <> 0) AND
        (AShifts < ACodeLength)
    SEQ
      IF
        (AErrorPattern /\ ATopBit) = 0
          AErrorPattern := AErrorPattern << 1
        TRUE
          AErrorPattern := (AErrorPattern << 1) >< APolynomial
      AErrorPattern := AErrorPattern /\ ARegBits
      AShifts := AShifts + 1

  IF
    (AErrorPattern /\ AZeroesMask) = 0
      SEQ
        -- get error pattern
        ErrorPattern :=
          AErrorPattern >> (ARegLength - BReglength)

        -- get number of B shifts
        BShifts := 0
        BErrorPattern := Syndrome /\ BRegBits
        WHILE (BErrorPattern <> ErrorPattern) AND
              (BShifts < BCodeLength)
          SEQ
            IF
              (BErrorPattern /\ BTopBit) = 0
                BErrorPattern := BErrorPattern << 1
              TRUE
                BErrorPattern :=
                  (BErrorPattern << 1) >< BPolynomial
            BErrorPattern := BErrorPattern /\ BRegBits
            BShifts := BShifts + 1

        -- reconstruct number of shifts
        Shifts := ((AShifts*AFactor)+(BShifts*BFactor)) REM CodeLength

        -- try to correct
        IF
          BErrorPattern = ErrorPattern
            SEQ
              SEQ ErrorPatternBit = 0 FOR BurstLength
                SEQ
                  Shifts := Shifts + 1
                  IF
                    (ErrorPattern /\ BTopBit) <> 0
                      InvertBit( CodeLength - Shifts )
                    TRUE
                      SKIP
                  ErrorPattern := ErrorPattern << 1
              Corrected := TRUE
          TRUE
            Corrected := FALSE
    TRUE
      Corrected := FALSE
:
```

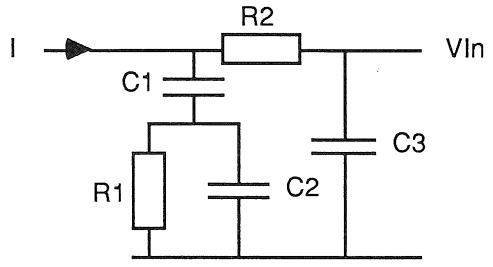
D.1 Basic Equations

Block Diagram



The block diagram above shows the components of the phase locked loop which are used for data separation on the M212. The output of the voltage controlled oscillator, after passing through a programmable divider, provides a read clock frequency. This is used as an input into two phase detectors, one used in standby mode, and the other used when reading from disk. The divided write clock frequency provides the other input to the standby phase detector, and the read data stream from the disk provides the other input to the read mode phase detector. The detected phase errors from the selected phase detector is then output from the M212 on the pins **PhaseUp** and **PhaseDn**. These signals are then connected through the charging resistors Rch1 and Rch2 to the filter which consists of the components R1, R2, C1, C2, C3. The resulting voltage output from the filter then provides the control voltage to the VCO on the input pin **VIn**.

Consider the filter shown below:



The transfer characteristic for this filter is:

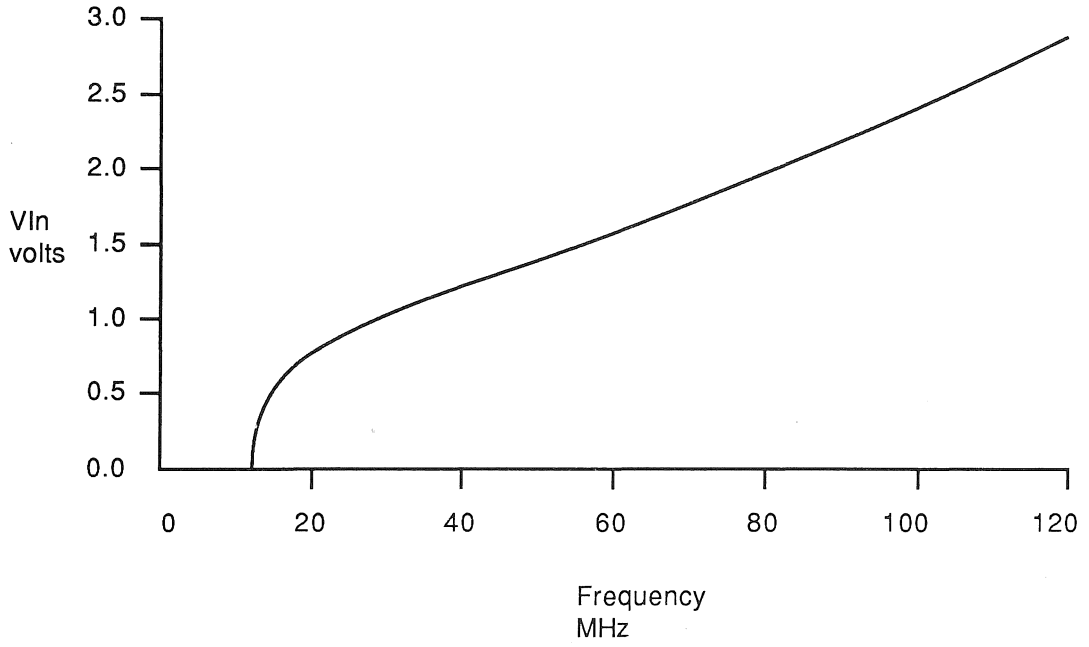
$$Vin = \frac{I\left(1+s\tau_5\right)}{s\tau_1\tau_2 C1\left(s^2+s\left(\frac{1}{\tau_1}+\frac{1}{\tau_2}+\frac{1}{\tau_3}+\frac{1}{\tau_4}\right)+\left(\frac{1}{\tau_1\tau_4}+\frac{1}{\tau_1\tau_2}\right)\right)}$$

where

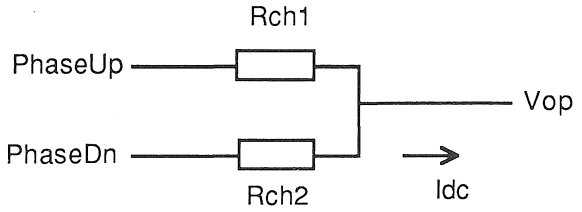
$$\tau_1 = R1C2$$
$$\tau_2 = R2C3$$
$$\tau_3 = R2C2$$
$$\tau_4 = R2C1$$
$$\tau_5 = R1(C1 + C2)$$

Now consider the on-chip components :

The VCO has a transfer characteristic:-



The gain of the VCO, K_v, is defined as the gradient at the operating point, V_{op}, in units of radians/volt-sec. Also, the divider is defined to divide by N, and the phase detector to have an attenuation of M. Therefore the overall gain of the chip components is $\frac{K_v}{NM}$ radians/ volt-sec.
An error voltage is produced on the **PhaseUp** and **PhaseDn** pins depending on the **VIn** voltage. This voltage is converted into an input current for the filter by Rch resistors.



Assuming that Rch1 and Rch2 are greater than the on resistance of the M212 output transistors:

$$Idc = \frac{V_{cc} - V_{op}}{Rch1} = \frac{V_{op}}{Rch2}$$

Hence the actual current which is input to the filter as a result of a phase error is :

$$I = \frac{V_{op}}{2\pi Rch2} Amps/radian$$

Open Loop Gain $G(s) = \left(\frac{K_v}{NM}\right)\left(\frac{V_{op}}{Rch2}\right)\left(\frac{1}{2\pi}\right)\left(\frac{Vin}{I}\right)\left(\frac{1}{s}\right)$

Hence

$$G(s) = \frac{\left(\frac{K_v V_{op}}{2\pi NMRch2}\right)(1 + s\tau_5)}{s^2\tau_1\tau_2 C1\left(s^2 + s\left(\frac{1}{\tau_1} + \frac{1}{\tau_2} + \frac{1}{\tau_3} + \frac{1}{\tau_4}\right) + \left(\frac{1}{\tau_1\tau_4} + \frac{1}{\tau_1\tau_2}\right)\right)}$$

Simplify this equation by letting:

$$a\tau = \left(\frac{1}{\tau_1\tau_4} + \frac{1}{\tau_1\tau_2}\right)^{-1}$$

$$b\tau = \tau_1 + \frac{1}{\tau_3}\left(\frac{\tau_3 + \tau_1}{\frac{1}{\tau_2} + \frac{1}{\tau_4}}\right)$$

Hence

$$G(s) = \frac{K_v V_{op}(1 + \tau_5 s)}{2\pi NMRch2(C1 + C3)s^2(a\tau s^2 + b\tau s + 1)}$$

The value of $\frac{K_v V_{op}}{2\pi NMRch2(C1+C3)}$ is a constant for a given circuit arrangement, and is denoted L_c. Modifying the gain equation to define the poles and zeroes gives:

$$G(s) = \frac{L_c\left(\frac{s}{Z1} + 1\right)}{s^2\left(\frac{s}{P1} + 1\right)\left(\frac{s}{P2} + 1\right)}$$

with the following values:

$$Z1 = \frac{1}{\tau_5}$$

$$P1 = \frac{b\tau - \sqrt{b\tau^2 - 4a\tau}}{2a\tau}$$

$$P2 = \frac{b\tau + \sqrt{b\tau^2 - 4a\tau}}{2a\tau}$$

From this equation the gain and phase margin equations can be derived:

$$Gain|G(\omega)| = \frac{L_c\sqrt{1 + \frac{\omega^2}{Z1^2}}}{\omega^2\sqrt{\left(\left(1 - \frac{\omega^2}{P1P2}\right)^2 + \left(\frac{\omega(P1+P2)}{P1P2}\right)^2\right)}}$$

$$PhaseMargin\phi = arctan\left(\frac{\omega}{Z1}\right) - 180 - arctan\left(\frac{\omega}{P1}\right) - arctan\left(\frac{\omega}{P2}\right)$$

where $\omega = 2\pi f$. f is the frequency in Hertz.

D.2 Typical Winchester Example

Consider a typical winchester disk with a data rate of 5 Mbits/sec. A read clock frequency of twice the data rate is required to separate the data correctly.
Let

$$C1 = 150nF$$
$$C2 = 3.3nF$$
$$C3 = 6.8nF$$
$$R1 = 30\Omega$$
$$R2 = 15\Omega$$

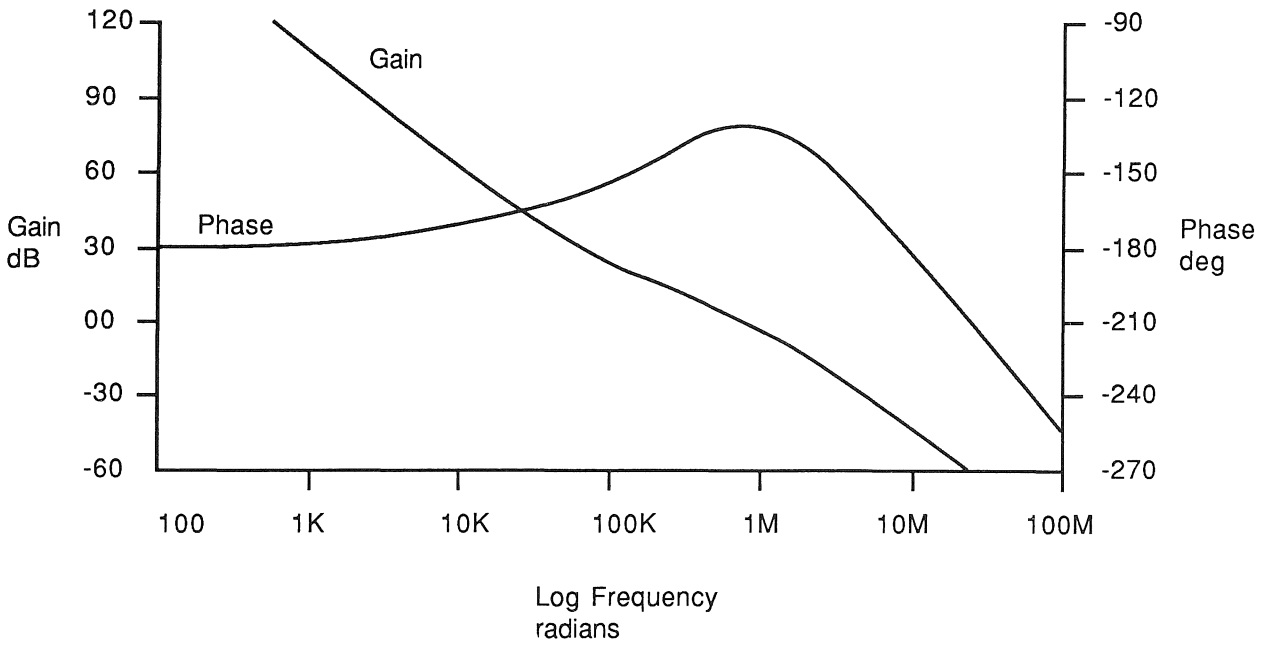
The VCO has a working frequency range of 32 MHz to 80 MHz so choosing f_{op}=40MHz gives N=4 for a 10 MHz read clock frequency. The standby phase detector has an attenuation of 1 and the read mode phase detector has a worst case attenuation of 2 in MFM mode. From the VCO curve V_{op} = 1.1 volts.
If we let $Rch2 = 310\Omega$ and $Rch1 = 920\Omega$
Then

$$\tau_1 = 9.9 \cdot 10^{-8}$$
$$\tau_2 = 1.02 \cdot 10^{-7}$$
$$\tau_3 = 4.95 \cdot 10^{-8}$$
$$\tau_4 = 2.25 \cdot 10^{-6}$$
$$\tau_5 = 4.6 \cdot 10^{-6}$$
$$a\tau = 9.66 \cdot 10^{-15}$$
$$b\tau = 3.92 \cdot 10^{-7}$$

Hence

$$Z1 = 2.17 \cdot 10^{5}$$
$$P1 = 2.74 \cdot 10^{6}$$
$$P2 = 3.78 \cdot 10^{7}$$
$$L_c = 1.62 \cdot 10^{11}$$

The gain and phase versus frequency graphs can now be plotted:



D.3 Typical Floppy Example

Consider a typical floppy disk with a data rate of 250 Kbits/sec. A read clock frequency of twice the data rate is required to separate the data correctly.
Let

$$C1 = 1.0uF$$
$$C2 = 22nF$$
$$C3 = 6.8nF$$
$$R1 = 82\Omega$$
$$R2 = 15\Omega$$

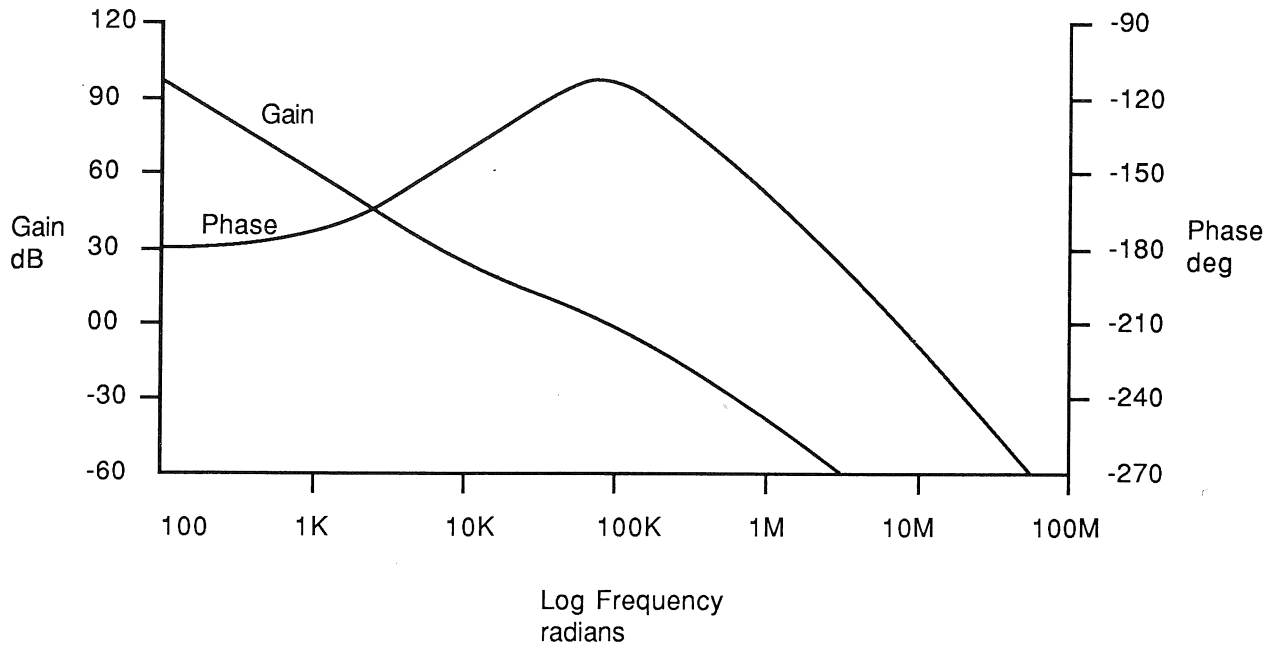
The VCO has a working frequency range of 32 MHz to 80 MHz so choosing f_{op}=64MHz gives N=128 for a 500 KHz read clock frequency. The standby phase detector has an attenuation of 1 and the read mode phase detector has a worst case attenuation of 2 in MFM mode. From the VCO curve V_{op} = 1.6 volts.
If we let $Rch2 = 310\Omega$ and $Rch1 = 920\Omega$
Then

$$\tau_1 = 1.8 \cdot 10^{-6}$$
$$\tau_2 = 1.02 \cdot 10^{-7}$$
$$\tau_3 = 3.3 \cdot 10^{-7}$$
$$\tau_4 = 1.5 \cdot 10^{-5}$$
$$\tau_5 = 8.38 \cdot 10^{-5}$$
$$a\tau = 1.83 \cdot 10^{-13}$$
$$b\tau = 2.46 \cdot 10^{-6}$$

Hence

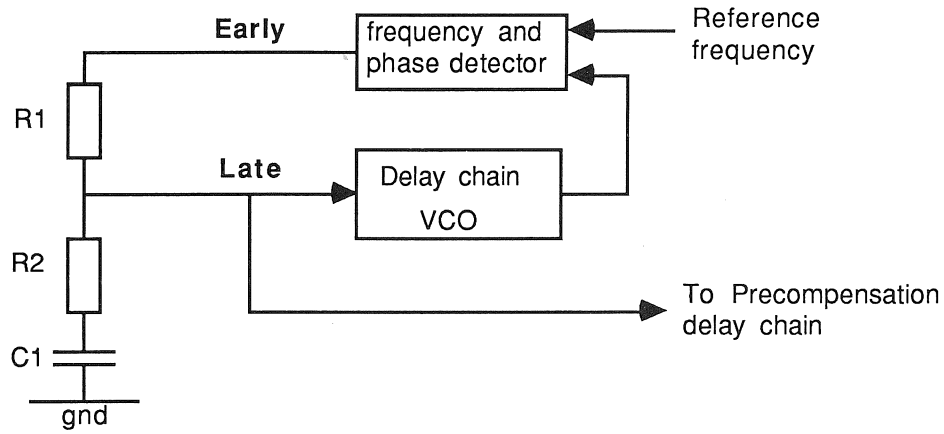
$$Z1 = 1.19 \cdot 10^4$$
$$P1 = 4.2 \cdot 10^5$$
$$P2 = 1.3 \cdot 10^7$$
$$L_c = 1.05 \cdot 10^9$$

The gain and phase versus frequency graphs can now be plotted:



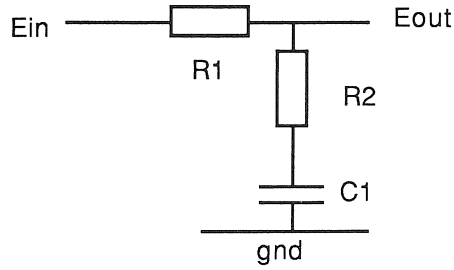
E.1 Basic Equations

Block Diagram



The block diagram shows the components of the phase locked loop used for controlling the precompensation on the M212. The VCO of this phase locked loop consists of an eight stage delay chain, whose output is inverted and then fed back into the delay chain input. The delay element used in the oscillator is the same as an element used in the precompensation delay circuit. The output of the VCO is then input to a phase and frequency detector and compared with a reference frequency. Any resulting error voltages are applied to the **Early** pin. An external filter converts the error voltages into a control voltage which is applied to the **Late** pin and is used to control the VCO and the precompensation delay chain. The components R1, R2, C1 are the external filter components.

Consider the filter shown below:



The transfer characteristic of this filter is:

$$\frac{Eout}{Ein} = \frac{(s\tau_2 + 1)}{(s\tau_1 + 1)}$$

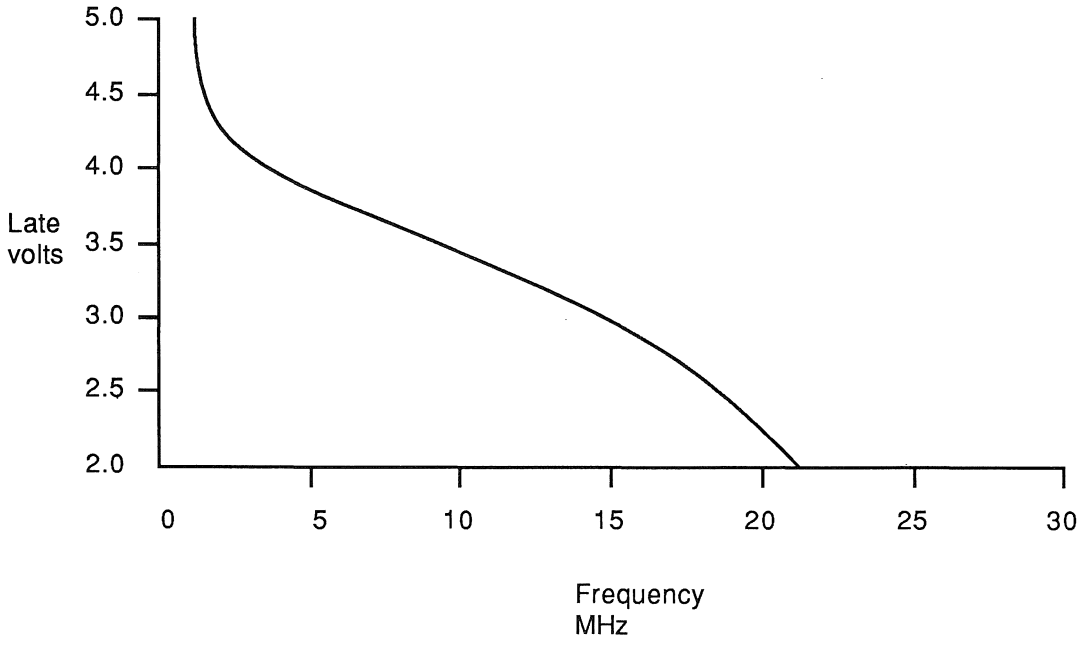
where

$$\tau_1 = (R1 + R2)\, C1$$

$$\tau_2 = R2C1$$

Now consider the on-chip components:

The VCO has a transfer characteristic:-



The gain of the VCO, K_v, is defined as the gradient at the operating point,V_{op}, in units of radians/volt-sec. The phase detector has unity gain.
Hence the chip gain L_c can be expressed as:-

$$L_c = \frac{K_v V_{op}}{2\pi} sec^{-1}$$

Therefore Open Loop Gain $G(s) = \frac{L_c(1+\tau_2 s)}{s(1+\tau_1 s)}$

The Gain and Phase margin equations can be derived:

$$|G(\omega)| = \left(\frac{L_c}{\omega}\right)\sqrt{\left(\frac{1+\omega^2{\tau_2}^2}{1+\omega^2{\tau_1}^2}\right)}$$

$$\phi = arctan(\omega\tau_2) - arctan(\omega\tau_1)$$

where $\omega = 2\pi f$. f is the frequency in Hertz.

E.2 Typical Example

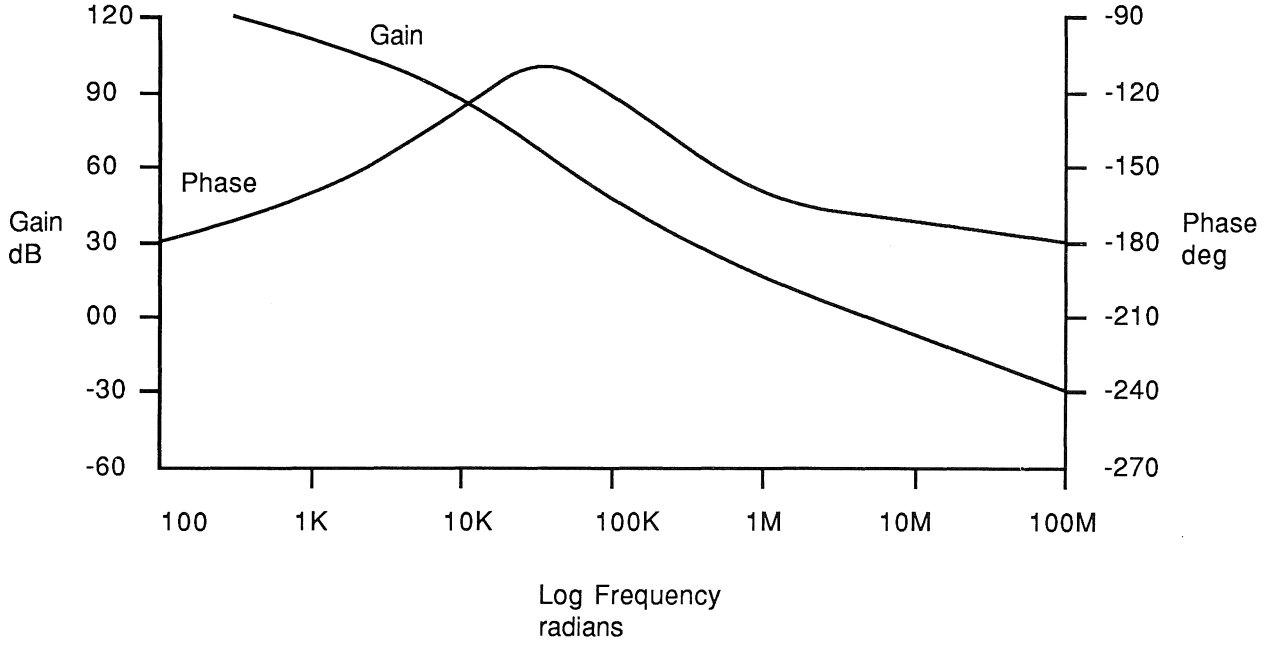
If we consider a typical winchester with the value of Precompensation of 12 ns. A typical delay stage of 6 ns results in a total delay of sixteen stages of 96 ns. The precompensation oscillator needs to be phase locked to a reference frequency of 10 MHz. This is easily achieved by deriving the reference frequency from the 20 MHz **WriteClock**.
Let

$$C1 = 10nF$$
$$R1 = 22K\Omega$$
$$R2 = 330\Omega$$

Hence

$$\tau_1 = 2.23 \cdot 10^{-4}$$
$$\tau_2 = 3.3 \cdot 10^{-6}$$
$$L_c = 43.2 \cdot 10^{6}$$

The gain and phase versus frequency graphs can now be plotted:



Commands

value	command	value	command	value	command
#00	**EndOfSequence**	#05	**WriteBuffer**	#0A	**SelectHead**
#01	**Initialise**	#06	**ReadSector**	#0B	**SelectDrive**
#02	**ReadParameter**	#07	**WriteSector**	#0C	**PollDrives**
#03	**WriteParameter**	#08	**Restore**	#0D	**FormatTrack**
#04	**ReadBuffer**	#09	**Seek**	#0F	**Boot**

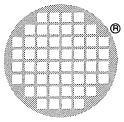
Parameters

address	parameter
#00	**DesiredSector**
#01	**DesiredHead**
#02	**DesiredCylinder0**
#03	**DesiredCylinder1**
#04	**LogicalSector0**
#05	**LogicalSector1**
#06	**LogicalSector2**
#07	**Addressing**
	bit 0 - **LogicalAddressing**
	bit 1 - **IncrementLogical**
	bit 2 - **IncrementBuffer**
#08	**DriveType**
	bit 0 - **Winchester**
	bit 1 - **WriteProtect**
	bit 2 - **SectorsFrom1**
	bit 3 - **LengthBy128Lg2**
	bit 4 - **HasReady**
	bit 5 - **HasSeekComplete**
	bit 6 - **PollThisDrive**
	bit 7 - **DriveExists**
#09	**SectorSizeLg2**
#0A	**NumberOfSectors**
#0B	**NumberOfHeads**
#0C	**NumberOfCylinders0**
#0D	**NumberOfCylinders1**
#0E	**CurrentCylinder0**
#0F	**CurrentCylinder1**

address	parameter
#10	**RWCCylinderBy4**
#11	**PCCylinderBy4**
#12	**SectorRetries**
#13	**SeekRetries**
#14	**HeadStepRateIn64us**
#15	**HeadSettleTimeIn64us**
#16	**HeadLoadTimeIn0.5ms**
#17	**MotorStartTimeIn4ms**
#18	**Interleave**
#19	**Skew**
#1A	**NumGap3Bytes**
#1B	**NumGap4BytesBy256**
#1C	**NumEccCorrectableBits**
#1D	**DesiredSectorBuffer**
	non-swapped parameters
#1E	**DesiredDrive**
#1F	**CurrentDrive**
#20	**Error**
#21	**Reason**
#22	**NumBufferBytesBy256**
#23	**ErrorDrive (M212Version)**
	pseudo parameter
#7F	**ControllerAccess**

Errors and reasons

value	Error	Reason
#00	**AllOk**	bit 0 - **BadDataCompareByte** bit 2 - **SectorRetries** bit 1 - **SectorCorrected** bit 3 - **SeekRetries**
#01	**BadCommand**	erroneous command code
#02	**BadParameterForRead**	erroneous parameter number
#03	**BadParameterForWrite**	erroneous parameter number
#04	**BadParameterValue**	erroneous parameter number
#05	**BadPolyType**	mask of the polynomial type
#06	**UncorrectableEccError**	zero
#07	**TimedOut**	zero
#08	**DriveReadOnly**	drive was software (0) or hardware (1) write protected
#09	**DriveNotSelected**	zero
#0A	**DriveHasBecomeNotReady**	zero
#0B	**DriveDoesNotExist**	access of non-existent drive (0) or invalid drive type
#0C	**TooManySectorRetries**	zero
#0D	**TooManySeekRetries**	zero
#0E	**TooManySteps**	0 - during step out, 1 - during step in
#0F	**FormatUnderrun**	zero



INMOS Limited
1000 Aztec West
Almondsbury
Bristol BS12 4SQ
UK
Telephone (0454) 616616
Telex 444723

INMOS Corporation
PO Box 16000
Colorado Springs
CO 80935
USA
Telephone (303) 630 4000
TWX 910 920 4904

INMOS GmbH
Danziger Strasse 2
8057 Eching
Munich
West Germany
Telephone (089) 319 10 28
Telex 522645

INMOS Japan K.K.
4th Floor No 1 Kowa Bldg
11 - 41 Akasaka 1-chome
Minato-ku
Tokyo 107
Japan
Telephone 03-505-2840
Telex J29507 TEI JN
Fax 03-505 2844

INMOS SARL
Immeuble Monaco
7 rue Le Corbusier
SILIC 219
94518 Rungis Cedex
France
Telephone (1) 46.87.22.01
Telex 201222

September 1987
72 TRN 103 02