SWIM Chip User's Reference

Revision 1.5 (11-Jan-88)

written by S C

Copyright © 1987 by Apple Computer, Inc

CONFIDENTIAL

CONTENTS

Introduction 3

SWIM Pinouts 3

GCR Encoding 5

GCR Track Format 6

MFM Encoding 7

  MFM Track Format 8

  CRC Generation/Detection 9

Using the IWM Hardware 10

  The IWM Register Set 11

Using the ISM Hardware 13

  Parameter RAM 13

  Trans-Space Machine 18

  Error Correction 19

  ISM Register Set 20

IWM State/ISM Register Mapping 26

IWM Register Summary 27

ISM Register Summary 28

Code Examples 30

Bibliography 34

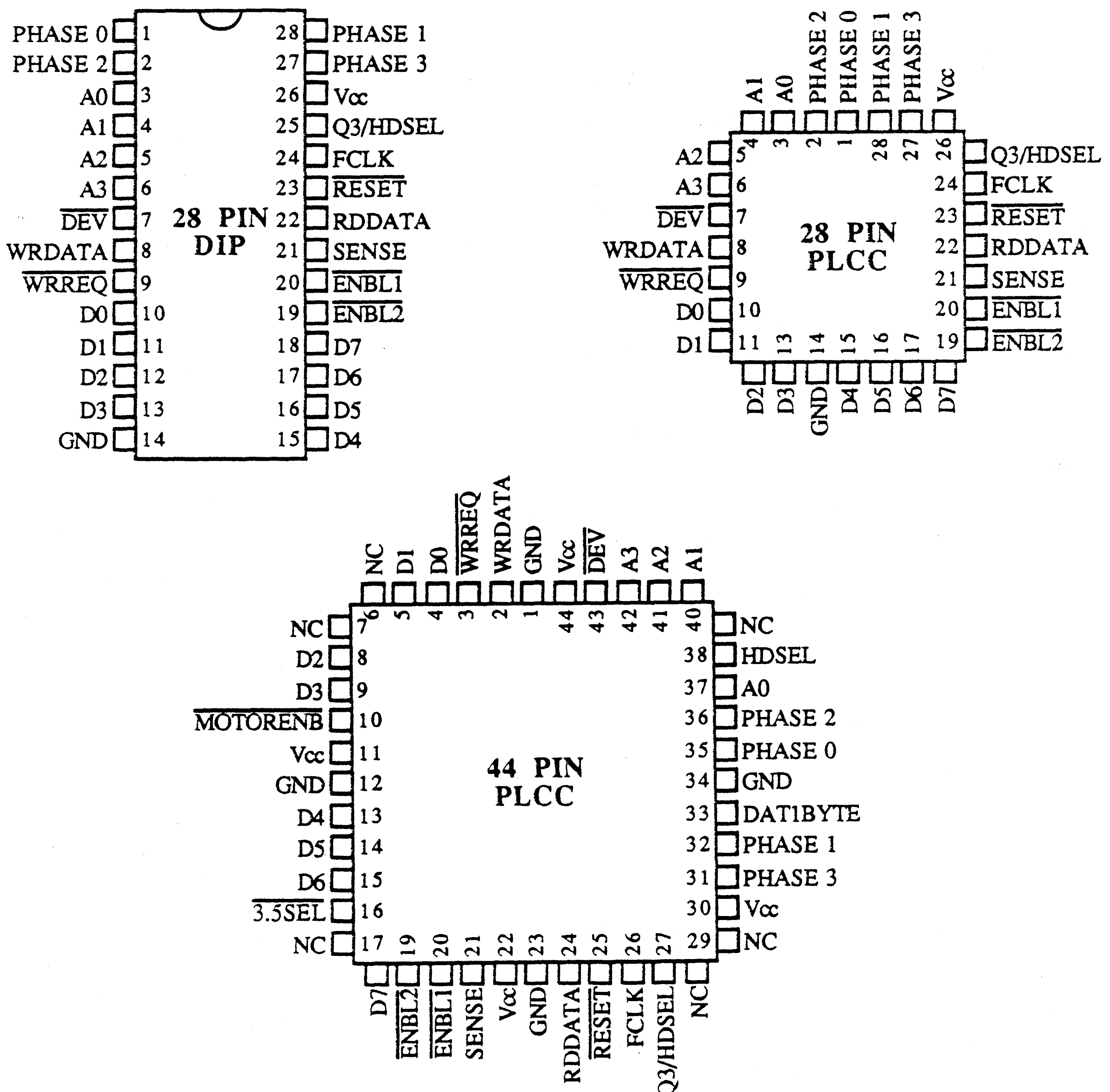
 Copyright © 1987-1988 by Apple Computer, Inc.

Introduction

This document is a guide to writing software that uses the SWIM (Sander-Wozniak Integrated Machine) disk controller chip. The SWIM combines two [virtually] independent disk controller chips into a single package: the IWM (Integrated Woz Machine), currently used in all Macintosh and some Apple II systems, and the ISM (Integrated Sander Machine), an independently-developed controller. The IWM hardware is capable of reading and writing disks using GCR (Group Coded Recording) encoding only. The ISM hardware is somewhat more flexible and can not only read and write GCR disks, but also the MFM (Modified Frequency Modulation) disks used in MS-DOS systems.

IC Pinouts

Pinouts in a software guide? Well, yes. Writing code for this chip inevitably leads to hooking the thing up to a logic analyzer to figure out what's *really* going on relative to what you think is going on. The SWIM chip comes in three different packages: a 28 pin DIP, a 28 pin PLCC, and a 44 pin PLCC. What's inside is exactly the same for all packages, but in the 44 pin PLCC some extra [ISM] signals come out. The 28-pin chips have exactly the same pinout as the two existing IWM versions.



 Copyright © 1987-1988 by Apple Computer, Inc.

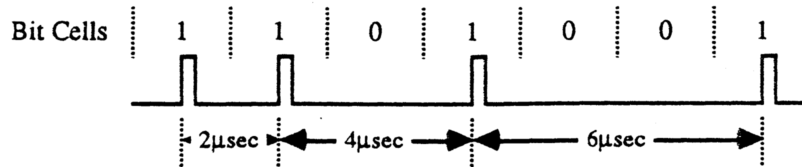
Pin Name	Description
/RESET	Initializes the registers in the chip and resets any current operation.
FCLK	The chip's clock (usually 14.3MHz for Apple II or 15.6672MHz for Macintosh).
Q3/HDSEL	On reset this signal becomes a Q3 input which is used to latch data on systems where the data is not valid on the rising edge of /DEV. If this signal is not needed for latching data, it can be used as a HDSEL output by setting bit 0 in the ISM's Setup register.
/DEV	This line is used to select the chip for reading or writing data from the processor.
A0–A3	These lines are used by the processor to select which register to access. Since the SWIM doesn't have a read/write line, the address lines determine whether an access is a read or a write. When the IWM register set is selected, A0=0 selects a read register and A0=1 selects a write register. When the ISM register set is selected, A3=0 selects a write register and A3=1 selects a read register.
D0–D7	These lines contain the data that is read from or written to the chip.
PHASE 0–PHASE 3	The phase lines are used for communication with a disk drive. When the ISM register set is selected, they can be individually programmed to be either inputs or outputs. When the IWM register set is selected, they are forced to be outputs regardless of how the direction was set when the ISM registers were selected.
/ENBL1	This output is used to select drive number 1.
/ENBL2	This output is used to select drive number 2.
RDDATA	This input contains the serial data being read from the disk.
WRDATA	This output contains the serial data being written to the disk.
/WRREQ	When this output is active (low), the drive will be able to accept data from the WRDATA output.
SENSE	This is a general-purpose input. In Apple systems with Sony floppy disk drives, it is tied to the RDDATA input to be able to read drive status information.
/MOTORENB	This output is active whenever the MotorOn bit (ISM mode register bit 7) is set or the 1/2 second timer (@16MHz) is still timing out (*44-pin PLCC only*).
/3.5SEL	This is a programmable output line that may be used as desired (*44-pin PLCC only*).
DAT1BYTE	This output goes high whenever the ISM register set is selected and a byte can be read from or written to the FIFO. It is the same as bit 7 of the ISM Handshake register (*44-pin PLCC only*).
HDSEL	This output may be used to select the disk side to read to or write from on some drives, or may be used as a general-purpose output (*44-pin PLCC only*).

Copyright © 1987-1988 by Apple Computer, Inc.

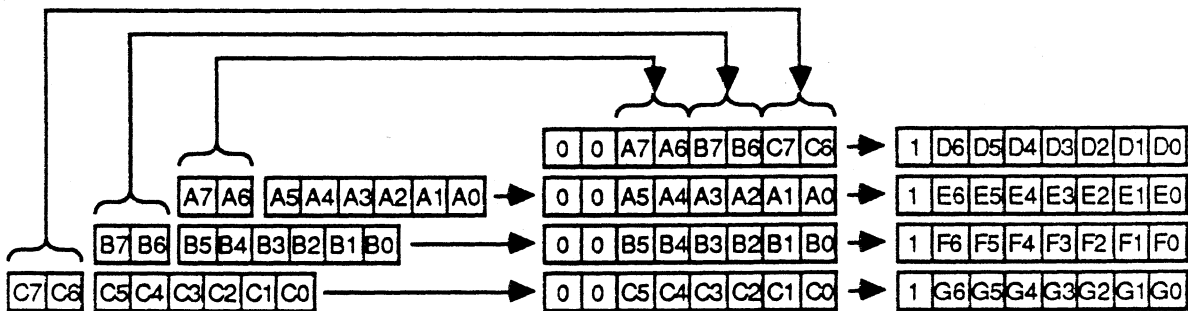
GCR Encoding

The IWM hardware, and for that matter, the ISM hardware as well, are essentially parallel-to-serial converters (and vice-versa). You write a byte of data to the chip, which in turn shifts out the byte to the disk drive one bit at a time. To read the disk, the hardware groups sets of 8 bits together into a byte and presents it to you to read.

GCR is the type of data encoding used in all Apple floppy disk drives. The rules for encoding the serial data are very simple: a "1" bit produces a pulse and a "0" bit doesn't. Since each bit is 2µsec long (4µsec long in slow mode), this gives pulse-to-pulse times of 2µsec, 4µsec or 6µsec for the three possible bit combinations of 11, 101 or 1001, as shown:



More than two zeroes in a row is not allowed because the IWM hardware starts to have trouble telling how far apart the previous "1" bit and the next one are. Also, the IWM has a rule that the most significant bit of a data byte must be a "1" so that it knows where bytes begin. These two restrictions lead to a solution that covers both. Obviously there are many byte values (0...255) that have more than two zero bits in a row. To get around this, the data to be written to a disk is "nibblized". This involves splitting three consecutive bytes each into 2-bit and 6-bit parts, gathering up the three 2-bit parts into a fourth 6-bit nibble, and encoding them into 8-bit nibblized bytes that have the most significant bit set to "1" and no more than two zeroes in a row. The process is shown as follows:



There are 81 bytes that satisfy the two above-mentioned restrictions. Out of that, two values ($D5 and $AA) are reserved as mark bytes (to be discussed), and 64 bytes (2^6 bytes—sound familiar?) are used for the actual data encoding. These 64 values have no more than one pair of "0"s and are hilighted in black:

$92	$93	$94	$95	$96	$97
		$A4	$A5	$A6	$A7
$B2	$B3	$B4	$B5	$B6	$B7
$D2	$D3	$D4	**$D5**	$D6	$D7
		$E4	$E5	$E6	$E7
$F2	$F3	$F4	$F5	$F6	$F7

$99	$9A	$9B	$9C	$9D	$9E	$9F
$A9	**$AA**	$AB	$AC	$AD	$AE	$AF
$B9	$BA	$BB	$BC	$BD	$BE	$BF
$C9	$CA	$CB	$CC	$CD	$CE	$CF
$D9	$DA	$DB	$DC	$DD	$DE	$DF
$E9	$EA	$EB	$EC	$ED	$EE	$EF
$F9	$FA	$FB	$FC	$FD	$FE	$FF

To get the original data byte back, the nibblized byte read from the disk is decoded back to a 6-bit nibble, and then when four of them have been read, the three 2-bit parts are recombined with their 6-bit parts to form an 8-bit byte.

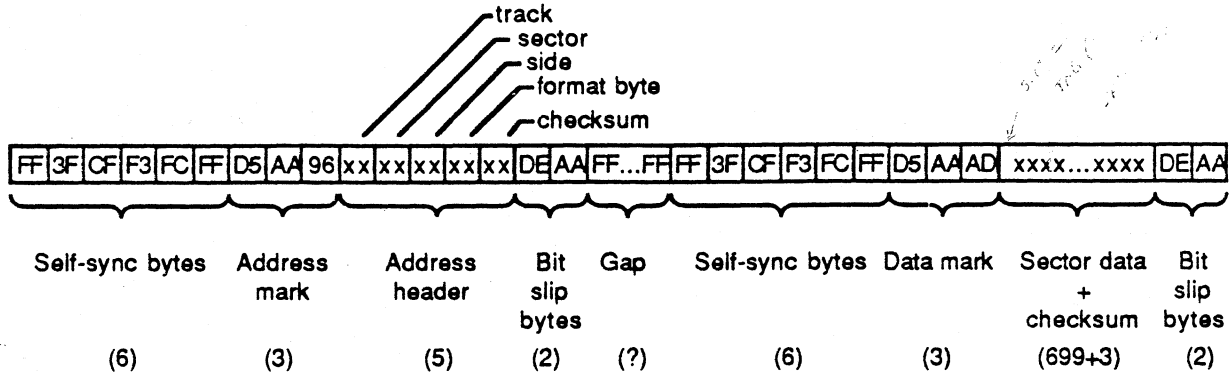
 Copyright © 1987-1988 by Apple Computer, Inc.

GCR Track Format

GCR disks formatted with Sony drives get the amount of information on a disk that they do by varying the rotational speed of the disk depending on which of five speed zones the current track is in. What this does is increase the number of sectors that will fit on most of the tracks:

Tracks		
Tracks	0 to 15:	12 sectors
	16 to 31:	11 sectors
	32 to 47:	10 sectors
	48 to 63:	9 sectors
	64 to 79:	8 sectors

When the disk is first formatted, the sectors are written so that they are spaced evenly around each track (i.e., the amount of gap between each sector is about the same). The sector itself consists of two parts: an address field that tells where we are, and a data field that contains the actual sector data.



Both parts of the sector begin with six self-sync bytes. This special pattern is such that no matter what bytes preceed it, by the time all six bytes have been read, the hardware is synchronized with the start of an address or data mark.

The address field consists of three mark bytes ($D5, $AA, $96); the track, sector, side, format and checksum bytes (encoded as GCR nibbles); and two bit slip bytes ($DE, $AA). The track and side bytes actually combine into a 16-bit word that puts the side number into bit 11 and the track number into bits 0-10.

The data field consists of three mark bytes ($D5, $AA, $AD), the sector number, sector data, and two bit slip bytes ($DE, $AA). The sector data consists of 12 tag bytes which are used by the operating system (see the *Disk Driver* chapter of Inside Macintosh), 512 data bytes and 3 checksum bytes. When encoded as GCR nibbles, the data takes up 702 bytes: 4 * [(12 + 512) / 3 groups] = 699 data + 3 checksum bytes. The tag and data bytes are mangled a bit in the process. In case you were curious, here are the algorithms. On the left side is the one for transforming the data bytes and calculating the three checksum bytes, and on the right is the one for converting the de-nibblized bytes back into data bytes and re-calculating the checksum.

WRITE TO FD

checksumA = 0; checksumB = 0; checksumC = 0
clear carry

Then repeat the following until all bytes are encoded:

checksumA = checksumA + byteA + carry
byteA = byteA XOR checksumC

checksumB = checksumB + byteB + carry
byteB = byteB XOR checksumC A

checksumC = checksumC + byteB + carry
byteC = byteC XOR checksumB
carry = checksumC[7]
rotate checksumC left one bit, but not through the carry

READ FROM FD

checksumA = 0; checksumB = 0; checksumC = 0

Then repeat for all of the bytes:

carry = checksumC[7]
rotate checksumC left one bit, but not through the carry

byteA = byteA XOR checksumC
checksumA,carry = checksumA + byteA + carry

byteB = byteB XOR checksumA
checksumB,carry = checksumB + byteB + carry

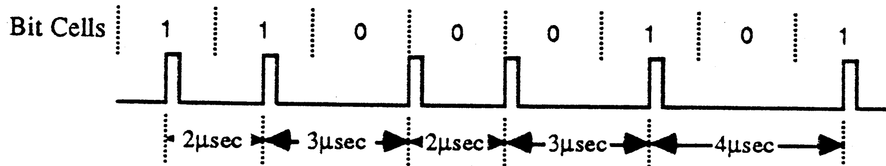
byteC = byteC XOR checksumB
checksumC,carry = checksumC + byteC + carry

 Copyright © 1987-1988 by Apple Computer, Inc.

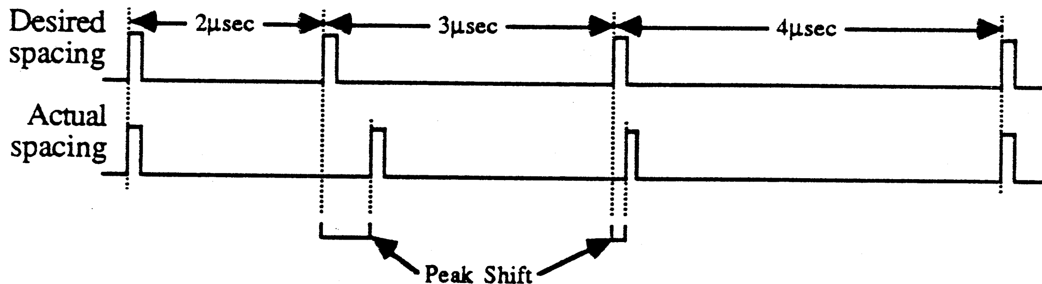
MFM Encoding

MFM encoding is used in most non-Apple floppy disk drives, and in particular, MS-DOS compatible machines. The rules for encoding the serial MFM data are exactly the same as GCR: a "1" bit causes a *data* pulse to occur in the middle of a bit cell and a "0" doesn't. Actually, there is one difference: a pair of "0"s causes a *clock* pulse to occur on the common cell boundary of the two "0"s. What all this means is that there are no restrictions on MFM data, so any data value (0...255) is perfectly legal.

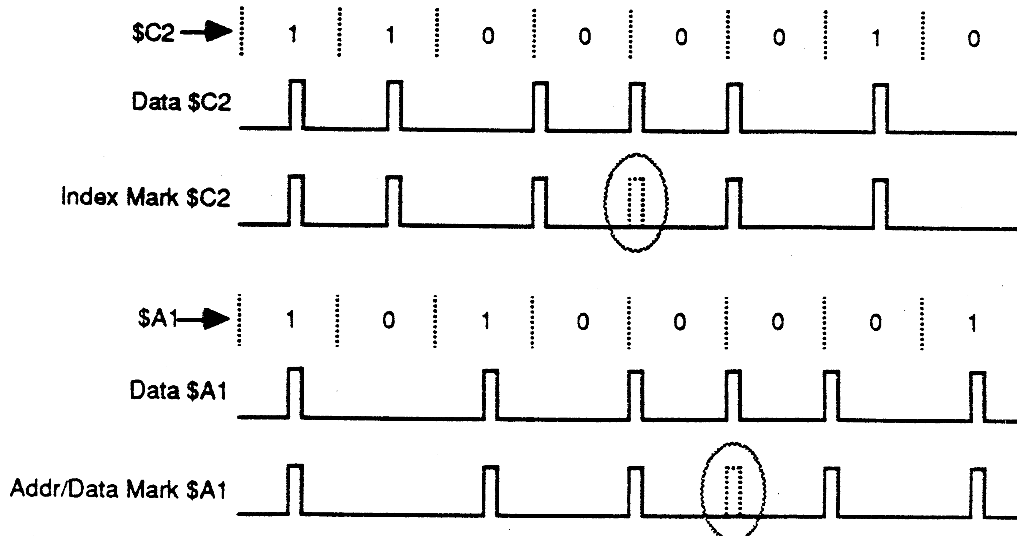
Since each bit cell is 2µsec long, this gives pulse-to-pulse times of 2µsec, 3µsec or 4µsec for the bit combinations of 000/11, 01/10 or 101 respectively, as shown:



A phenomenon that occurs with both encoding formats, but is much more pronounced in MFM is *peak shift*. What happens is that closer-spaced pulses (mainly 2µsec) tend to push apart, and in the process shrink the spacing between the farther-spaced pulses (3µsec and especially 4µsec). The effect is similar to putting two like magnets together: the closer you put them, the more they tend to push apart.



Like GCR, the MFM track format requires some reserved mark bytes to denote the start of sectors. However, since all byte values can be used as data, another method has to be used to identify the mark bytes. This is done by writing (and reading) an illegal MFM byte. The illegal combination comes about as a result of dropping one of those clock pulse that normally show up between adjacent "0" bits (in this case the middle clock pulse in a run of four zeroes). This creates a 4µsec cell time beginning with a "0" instead of the normal "101" combination. Two types of mark bytes are needed: an index mark (usually a $C2) and an address/data mark byte (usually a $A1). The difference between normal and mark bytes is shown:

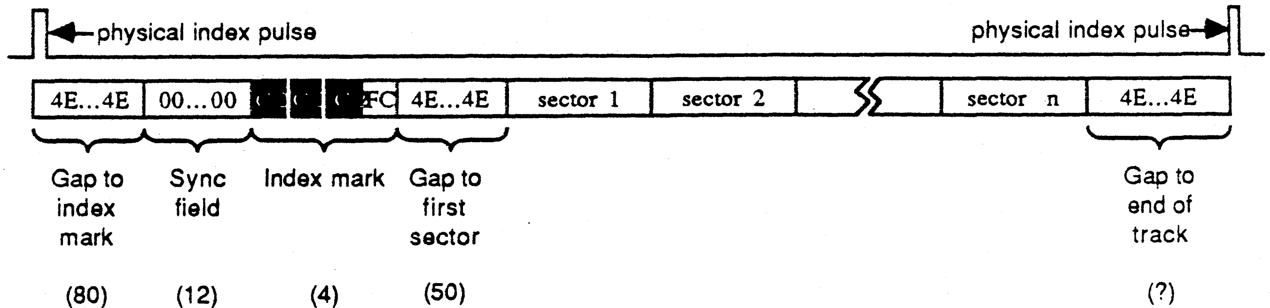


 Copyright © 1987-1988 by Apple Computer, Inc.

MFM Track Format

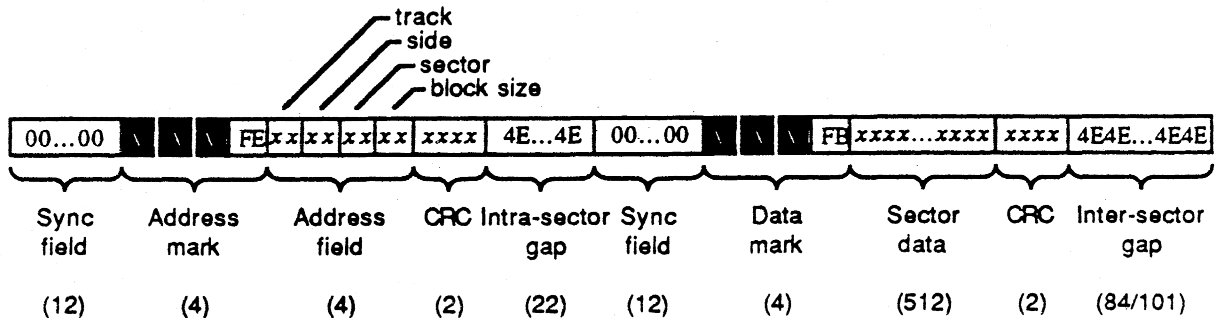
MFM disks are different from Apple GCR disks in that they have the same number of sectors on every track since the disk spins at a constant rate. This is not to say that MFM disks *can't* have variable numbers of sectors per track, but in in practice the disk controllers used to read and write MFM usually don't support it.

The start of a track is aligned with a physical index pulse that is built into the drive. This index pulse occurs once per revolution.



Right after the index pulse is a field called the *index mark* that marks the start of the track. It is composed of three "mark byte" $C2s followed by a normal $FC byte. Notice that the first of a set of mark bytes must be preceeded by a set of twelve zeroes. These zeros synchronize the hardware to the bits being read from the disk. The hardware then looks for the first non-2µsec pulse-to-pulse time (remember that sets of zeroes have a clock pulse between the bit cells), which will hopefully be the start of a mark byte.

The sectors are separated from each other by a fixed number of *gap* bytes. This gap is 84 bytes long on single-density disks and 101 bytes long on double-density disks. After the last sector there is a large gap that fills the empty space from there to the start of the next index pulse. Currently two disk formats are being used: the single-density format uses the same double-sided media as the 800K GCR disks, has 9 sectors per track side and can hold 720K bytes. The double-density format requires a different kind of media (marked with "HD"), has 18 sectors per track side and can hold 1440K bytes.



Each of the sectors is made up of an address field and a data field. Again, note that before the first of a set of mark bytes is a 12-byte sync field of all zeroes which synchronizes the hardware with the data being read from the disk.

The address field consists of a 4-byte address mark, the track, side and sector numbers, and a byte that tells how big the sector is. This byte should be $02 for both 720K and 1440K disks. Finally, there is a 2-byte CRC checksum which is calculated on all of the just-mentioned bytes (see below).

The data field contains a 4-byte data mark, 512 bytes of data, and a 2-byte CRC. The 12 bytes of tag information included in GCR sectors is not supported on MFM disks (see the *File Manager* and *Disk Driver* chapters of Inside Macintosh for more information on file tags).

 Copyright © 1987-1988 by Apple Computer, Inc.

CRC Generation/Detection

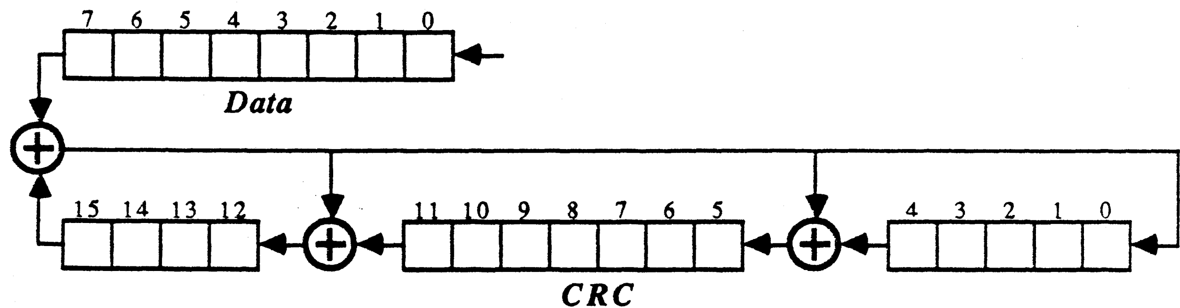
The ISM hardware uses a two-byte CRC (Cyclic Redundancy Check) that is appended to both the address and data fields. Running the data bytes through the CRC generator will produce a 16-bit CRC value that can be appended to the data that is written to disk. To determine if the data (and CRC) are correct when reading it back, run both the data *and* the calculated CRC through the generator. The resulting CRC will be zero if the data is correct. The CRC generator uses the CCITT-16 polynimial

$$G(x) = x^{16} + x^{12} + x^5 + 1.$$

A pseudo-code algorithm for calculating the CRC is shown below:

```
Initialize the CRC to all ones
repeat
    repeat
        xorBit = CRC[15] XOR DATA[7]          {XOR MSBs of data and CRC}
        CRC[4] = CRC[4] XOR xorBit            {then XOR that into the CRC}
        CRC[11] = CRC[11] XOR xorBit
        rotate CRC left one bit               {CRC[i] -> CRC[i+1], xorBit -> CRC[0]}
        shift DATA left one bit               {DATA[i] -> DATA[i+1]}
    for each bit in DATA
for each DATA byte to be included in CRC
```

And for those of you who want the pictures too:



 Copyright © 1987-1988 by Apple Computer, Inc.

Using the IWM Hardware

The IWM is accessed through 16 addresses which control the state of 8 latches, as shown.

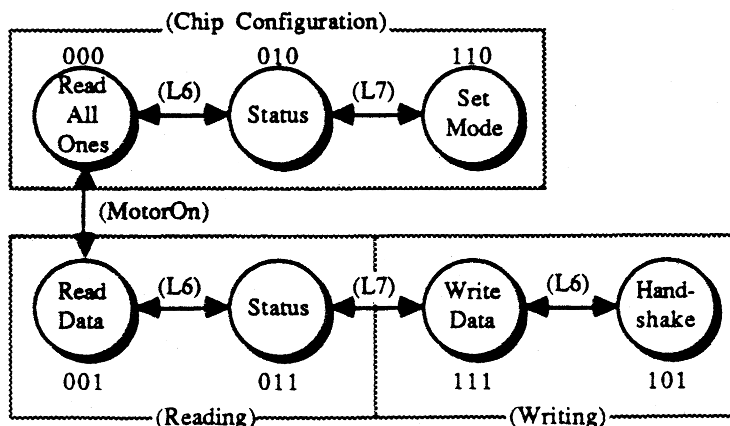
[0] set PHASE0 output level
[1] set PHASE1 output level
[2] set PHASE2 output level
[3] set PHASE3 output level
[4] set MotorOn state bit
[5] drive select (0=internal, 1=external)
[6] set L6 state bit
[7] set L7 state bit

The upper three address bits (A1-A3) select a latch, and A0 becomes the contents of the latch. The PHASE*x* latches are directly connected to the PHASE*x* pins on the IWM. The drive select latch determines which /ENBL*x* pin will be active.

The L7, L6 and MotorOn state bits decode to select one of six registers that are used for reading and writing. Reading from a register must be done from a "0" state (L7=0, L6=0 or MotorOn=0). Writing to a register must be done from a "1" state (L7=1, L6=1 or MotorOn=1), although the first write to a write-register should be made to L7=1. If an operation occurs that changes the state of one of these bits, the new state will select the register to be accessed and whether that operation will be a read or a write.

L7	L6	MotorOn	Register (State Name)
0	0	0	Read All Ones
0	0	1	Read Data
0	1	X	Read Status
1	0	X	Read Write-Handshake
1	1	0	Set Mode
1	1	1	Write Data

An important note is that you can't just switch indiscriminately from one register to another by any path you choose (remember you can only change one bit at a time), since it's possible to write garbage to one of the other registers – often with disasterous results. So the best way to cycle between the various registers as shown below.

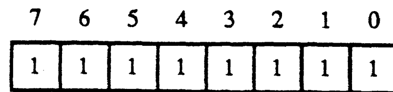


 Copyright © 1987-1988 by Apple Computer, Inc.

IWM Register Set

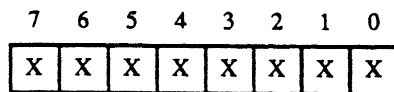
Each register description includes the register name, whether it's a read or write register, the state address [L7-L6-MotorOn] as three binary digits, and a description of each bit. *X* means don't care, that is, either a 0 or 1 may be substituted.

Read All Ones **R** **[000]**



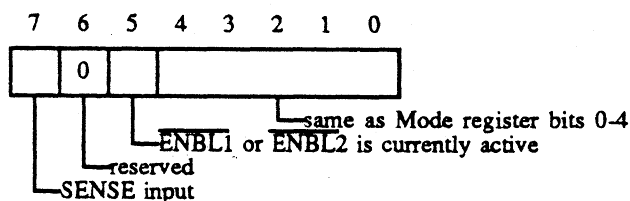
7 - 0 Reading from this address always returns the value $FF.

Read Data **R** **[001]**



Reading from either L7=0 or L6=0 while in this state will return the current contents of the read buffer.

Read Status **R** **[01*X*]**



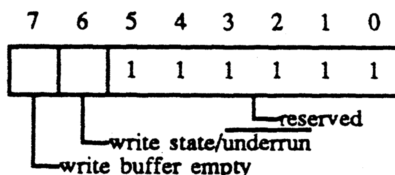
4 - 0 These bits echo the lowest five bits of the mode register.

5 The /ENBLx active bit will be set to 1 whenever either of the drive enable lines is active (low).

6 This bit is reserved for future expansion and will always read as a "0".

7 This bit returns the current state of the SENSE input.

Read Write-Handshake **R** **[10*X*]**



5 - 0 These bits are reserved for future expansion and will always read as "1"s.

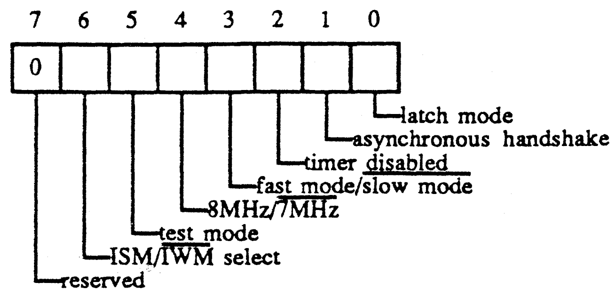
6 This bit will be a "1" while writing data. However if a byte hasn't been loaded into the Write Data register before the IWM hardware goes looking for it (called an underrun), then this bit will be reset to "0" until either the chip is reset or taken out of write mode.

7 The write buffer empty bit will be set to "1" whenever the chip is ready to accept another byte from the processor.

 Copyright © 1987-1988 by Apple Computer, Inc.

Set Mode **W [110]** **Reset to 00000000**

The mode register is not accessable for up to one second when the timer is enabled and counting down. This is because the mode register is selected by clearing the internal *delayed* MotorOn line, so that while the timer is counting down, this line will be held high even when MotorOn has been cleared.



0 In latch mode, the MSB of the read data is latched internally during /DEV low (this internally latched MSB is then used for the determination of a valid data read). This bit should be set in asynchronous mode.

1 If the *asynchronous* bit is zero (synchronous mode) each data byte must be read or written at exact intervals (on each byte boundary). If the asynchronous bit is set, each data byte is read when it's available or written when the write buffer is empty. It is highly recommended that asynchronous mode be used whenever possible since the timimg is more flexible.

2 If this bit is "0", then /ENBL*x* will be held active for 2^{23}±100 FCLK periods (about 1 second) after the MotorOn state bit is reset to "0". If the latch mode bit is set the timer is not guaranteed to count up to 2^{23}. MotorOn is active, either /ENBL1 or /ENBL2 will be active.

3 *Fast mode* selects a bit cell time of 2µsec instead of 4µsec.

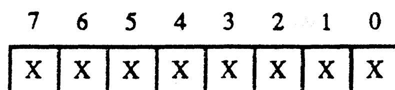
4 This bit indicates whether the input clock (FCLK) is to be divided by 7 (0) or 8 (1) to provide 1µsec internal timings.

5 When this bit is set, device operation is unspecified, except that status register bit 5 can always be read and that the mode register can always be set. Also the 1-second timer will count down in 100ms instead of 1 second.

6 The *ISM/IWM* bit selects which register set will be used. To select the ISM set, you must write to the GCR mode register **four times in a row** with this bit set to "1", "0", "1","1", respectively. This somewhat torturous route is set up to prevent unintentional intrusions into the ISM world by existing software. After the switch, all further accesses to the SWIM will then be routed to the ISM register set until you clear bit 6 in the ISM mode register.

7 This bit is reserved for future expansion and should always be set to "0".

Write Data **W [111]**



Writing to either L7=, L6=1 or MotorOn=1 while in this state will write a byte of data to the write buffer. This byte will in turn be loaded into the shifting hardware and sent out serially to the disk. This buffer must be kept full, else an *underrun* will occur and further bytes will not be written to the disk. The *buffer empty* bit in the Handshake register becomes a "1" whenever the chip can accept another byte. Writing to the chip faster than this won't hurt anything, but the last byte written to the chip when the buffer is emptied is the one that will be used.

 Copyright © 1987-1988 by Apple Computer, Inc.

Using The ISM

The ISM is designed to read and write both the current Apple GCR disks as well as those using MFM encoding (it is also possible to read and write other formats). It has highly flexible read and write timing made possible by user-programmable parameters; error correction logic that tries to interpret the bit times out of the muddle of raw serial data; and write pre-compensation to correct for peak shift. The ISM uses a 2-byte read/write FIFO, so the software can "slip" out a byte from time to time without causing an overrun (reading too quickly) or underrun (writing too slowly).

Unlike the IWM, the four address lines directly select one of 16 registers. The address bit A3 acts as the read/write line for the registers; registers are read when A3=1 and written when A3=0. Since there are no "forbidden" paths for moving from state to state, registers may be accessed in any order. However there is one rule that can't be ignored: **the time between successive chip accesses must be no less than 4 FCLKs (if the Setup register's *FCLK/2* bit is "0") or 8 FCLKs (if *FCLK/2* is "1"). This includes accesses which cause a mode switch to the IWM.**

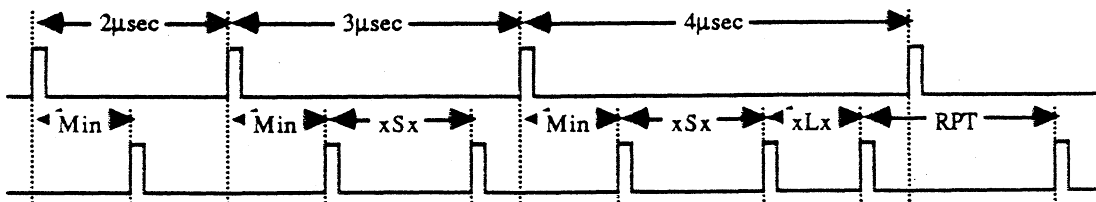
The following sections describe the parameter RAM, Trans-Space machine, and Error Correction — weird terms now, but (hopefully) clearer later. Read on.

Parameter RAM

The parameter RAM is one of the ISM's registers, and actually consists of 16 bytes which control the read and write timing that give the hardware so much of its flexibility. This section discusses the individual parameters and how they fit into the greater scheme of things. Note that in the timing descriptions, cell times are given to be either 2-, 3- or 4µsec. This is just because the main use of the chip will be for reading and writing MFM at 16MHz (sort of a standard, I guess). HOWEVER, it's just as easy to substitute in 2-, 4- and 6µsec cell times if you're working with Apple GCR, for example.

	7 6 5 4 3 2 1 0		7 6 5 4 3 2 1 0
0	MIN Cell Time	8	LSL
1	Correction Multiplier	9	LSS
2	SSL	A	LLL
3	SSS	B	LLS
4	SLL	C	LATE | NORMAL
5	SLS	D	TIME 0
6	RPT	E	EARLY | NORMAL
7	CSLS	F	TIME 1

The first twelve bytes are used in read timing, and the remaining four are used for write timing. The relationships between the read parameters is shown below:



The read parameters are basically bounds for determining what the actual pulse-to-pulse cell time is. The *MIN Cell Time* is the lower limit for the smallest possible legal cell time; *xSx* parameters define the 2µsec/3µsec boundary; the *xLx* parameters define the 3µsec/4µsec boundary; and *RPT* defines the upper limit for the largest possible legal cell time.

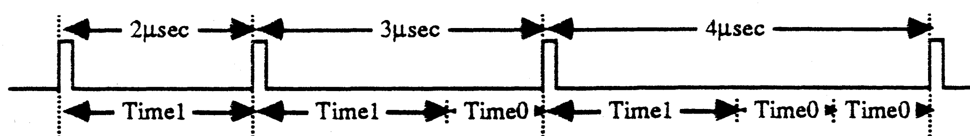
 Copyright © 1987-1988 by Apple Computer, Inc.

The "S"s and "L"s in the *xSx* and *xLx* parameter names refer to the cell times for the *previous*, *current* and *next* cells. A short cell is 2µsec long, and a long cell is 3 or 4µsec long. The various "S"s and "L"s are differentiated so that the bounds can be adjusted for all possible cell combinations. So for the 3µsec cell shown above, *SSL* would define the lower bounds and *SLL* the upper bounds.

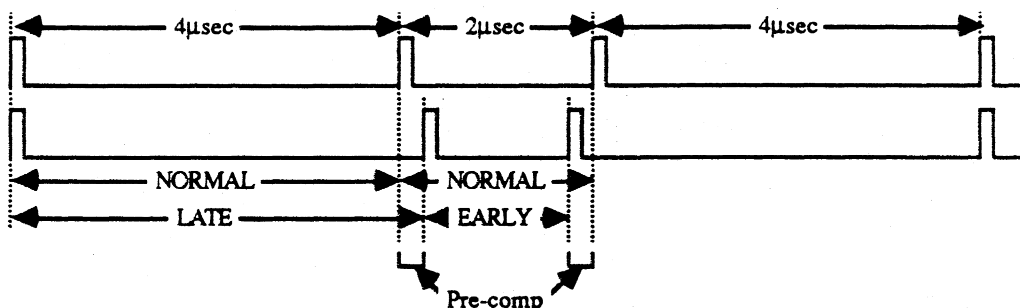
The CSLS (Compensating SLS) parameter is used when the previous cell time was assumed to be a long, but is resolved to actually be a short. CSLS is shorter than the LLS since a previous short cell would tend to scrunch the current longer cell.

The Correction Multiplier (MULT) is not a measure of cell boundaries, but rather a constant used in error correction. It is defined as the value which causes an 8-bit counter to count from 0 to 255 over the period of 32 samples if every cell is exactly the length that it theoretically should be. The 32 samples are taken when reading the sync field that preceeds mark bytes. If you recall, the sync field is made up of all $00s, which is a gaggle of 2µsec cells. So if the 2µsec cells aren't, then the counter won't be 255 and since the actual value can be read back (see Error Correction), you can determine how far off the cell times are.

The write parameters are used for determining where the pulses should be put when writing to the disk. The *TIME 0* and *TIME 1* values are used to build your basic 2µsec, 3µsec and 4µsec pulse spacings, as shown:



The *EARLY*, *NORMAL* and *LATE* values are used for setting the amount of *write pre-compensation.* Pre-comp is an adjustment made to the pulse placement in the opposite direction that peak shift would push in an effort to make the pulses end up where you wanted them to in the first place:



To calculate the parameters, we first need a few variables to work with: t_2, t_3 and t_4 are the number of FCLKs (the SWIM chip's clock) in 2µsec, 3µsec and 4µsec, respectively. t_{ps} is the typical amount of peak shift in FCLKs. t_{pc} is the number of FCLKs of pre-comp. A bit of notation also: "[x]" means to truncate x, that is, get rid of the fractional part of x. Most of the RAM parameters are based on these values:

$t_{MIN} = [t_2 / 2]$	$t_{SSL} = (t_2 + t_3) / 2$	$t_{LSL} = (t_2 + t_3) / 2 + t_{ps}$
$t_{RPT} = 3/4 * t_2$	$t_{SSS} = (t_2 + t_3) / 2 - t_{ps}$	$t_{LSS} = (t_2 + t_3) / 2$
$t_{TIME0} = t_3 - t_2$	$t_{SLL} = (t_3 + t_4) / 2 - t_{ps}$	$t_{LLL} = (t_3 + t_4) / 2$
$t_{TIME1} = t_2$	$t_{SLS} = (t_3 + t_4) / 2 - (2 * t_{ps})$	$t_{LLS} = (t_3 + t_4) / 2 - t_{ps}$
$t_{NORMAL} = [\text{greater of } t_2 / 4 \text{ and } 7]$		

 Copyright © 1987-1988 by Apple Computer, Inc.

Once we have those values, we can then calculate the parameters (remember that the *xSx* parameters are relative to *MIN* and the *xLx* parameters are relative to *xSx*).

$MIN = t_{MIN}$	$LSL = (t_{LSL} - t_{MIN})$
$MULT = 65536 / (32 * t_2)$	$LSS = (t_{LSS} - t_{MIN})$
$SSL = (t_{SSL} - t_{MIN})$	$LLL = (t_{LLL} - [t_{LSL}])$
$SSS = (t_{SSS} - t_{MIN})$	$LLS = (t_{LLS} - [t_{LSS}])$
$SLL = (t_{SLL} - [t_{SSL}])$	$LATE/NORMAL = (16 * [t_{NORMAL} + t_{pc}]) + t_{NORMAL}$
$SLS = (t_{SLS} - [t_{SSS}])$	$TIME0 = t_{TIME0}$
$RPT = t_{RPT}$	$EARLY/NORMAL = (16 * [t_{NORMAL} - t_{pc}]) + t_{NORMAL}$
$CSLS = SLS - [t_{LSS} - t_{SSS}])$	$TIME1 = t_{TIME1}$

Finally, *MIN* is reduced by 3 clocks, and *xSx*, *xLx* and *TIMEx* are reduced by 2 clocks to take into account internal delays in the ISM hardware.

MIN = MIN - 3	LSL = LSL - 2
SSL = SSL - 2	LSS = LSS - 2
SSS = SSS - 2	LLL = LLL - 2
SLL = SLL - 2	LLS = LLS - 2
SLS = SLS - 2	TIME0 = TIME0 - 2
RPT = RPT - 2	TIME1 = TIME1 - 2
CSLS = CSLS - 2	

All of the parameters except for *MULT*, *EARLY*, *NORMAL* and *LATE* are initial counter values in *half-clock* units. A half-clock is one half of the clock period, which means that the counters are decremented on every rising or falling edge of the clock. Since the above values are in full clock units they must be converted to the half-clock units used by the chip. If the value is represented by "*i.f*" where *i* is the integer part and *f* is the fractional part, the value that will be stuffed into the register will be:

$2 * i + 0$ if $.00 \le f < .25$ (rounded down)
$2 * i + 1$ if $.25 \le f < .75$ (rounded to .50)
$2 * (i+1)$ if $.75 \le f \le .99$ (rounded up)

Examples:
13.24 rounds down to 13.0
9.25 rounds up to 9.5
22.75 rounds up to 23.0

NOTE: these are "theoretical" values calculated for ideal conditions. We've found that they work very well in the nominal case (no drive speed variation, etc), but that performance falls off as you move away from nominal. Usually you would start with these values and experiment until you get the ones that work best for your situation.

In the case of a Macintosh system with Sony SuperDrives, we get better results if we ignore the peak shift component (t_{ps}) of the equations to calculate the *xSx* and *xLx* parameters:

$$t_{xSx} = (t_2 + t_3) / 2$$
$$t_{xLx} = (t_3 + t_4) / 2$$

and then factor peak shift back into the *Sxx* parameters only at the end. This is because if the previous cell was a "S", it must have been 2µsec, whereas if it was a "L", it could have been either a 3- or 4µsec cell so the peak shift is less certain. We use about 1 clock of peak shift is for the SSx and 1/2 clock for the SLx parameters.

 Copyright © 1987-1988 by Apple Computer, Inc.

As an example, let's compare the theoretical and experimental parameters for a 15.6672MHz FCLK; 2-, 3- and 4µsec cells; 125nsec of write pre-compensation; and a typical peak shift of 100nsec. First we need the basic building blocks:

$t_2 = (2.00\mu sec)(15.6672MHz) = 31.33$ clocks
$t_3 = (3.00\mu sec)(15.6672MHz) = 47.00$ clocks
$t_4 = (4.00\mu sec)(15.6672MHz) = 62.67$ clocks

$t_{ps} = (100.00nsec)(15.6672MHz) = 1.57$ clocks
$t_{pc} = (125.00nsec)(15.6672MHz) = 1.96$ clocks —> 2.0 clocks

Then we calculate the parameters using the equations on page 14 (except for the *xSx* and *xLx* parameters which are calculated both ways):

Theoretical

$t_{MIN} = [t_2 / 2] = [31.33 / 2] = 15.00$ clocks
$t_{SSL} = (t_2 + t_3) / 2 = 39.17$ clocks
$t_{SSS} = (t_2 + t_3) / 2 - t_{ps} = 37.60$ clocks
$t_{SLL} = (t_3 + t_4) / 2 - t_{ps} = 53.27$ clocks
$t_{SLS} = (t_3 + t_4) / 2 - (2 * t_{ps}) = 51.70$ clocks
$t_{RPT} = 3/4 * t_2 = 3/4 * 31.33 = 23.50$ clocks
$t_{LSL} = (t_2 + t_3) / 2 + t_{ps} = 40.74$ clocks
$t_{LSS} = (t_2 + t_3) / 2 = 39.17$ clocks
$t_{LLL} = (t_3 + t_4) / 2 = 54.84$ clocks
$t_{LLS} = (t_3 + t_4) / 2 - t_{ps} = 53.27$ clocks
$t_{TIME0} = t_3 - t_2 = 15.67$ clocks
$t_{TIME1} = t_2 = 31.33$ clocks
$t_{NORMAL} =$ [greater of $t_2 / 4$ and 7] = 7.0 clocks

"Real-World"

$t_{MIN} = [t_2 / 2] = [31.33 / 2] = 15.00$ clocks
$t_{SSL} = (t_2 + t_3) / 2 = 39.17$ clocks
$t_{SSS} = (t_2 + t_3) / 2 = 39.17$ clocks
$t_{SLL} = (t_3 + t_4) / 2 = 54.84$ clocks
$t_{SLS} = (t_3 + t_4) / 2 = 54.84$ clocks
$t_{RPT} = 3/4 * t_2 = 3/4 * 31.33 = 23.50$ clocks
$t_{LSL} = (t_2 + t_3) / 2 = 39.17$ clocks
$t_{LSS} = (t_2 + t_3) / 2 = 39.17$ clocks
$t_{LLL} = (t_3 + t_4) / 2 = 54.84$ clocks
$t_{LLS} = (t_3 + t_4) / 2 = 54.84$ clocks
$t_{TIME0} = t_3 - t_2 = 15.67$ clocks
$t_{TIME1} = t_2 = 31.33$ clocks
$t_{NORMAL} =$ [greater of $t_2 / 4$ and 7] = 7.0 clocks

Calculate the actual parameter values (and round appropriately)...

Theoretical:

MIN = t_{MIN} = 15.0 clocks
MULT = $[65536 / (32 * t_2)] = 65$
SSL = $t_{SSL} - t_{MIN} = 24.17$ -> 24.0 clocks
SSS = $t_{SSS} - t_{MIN} = 22.60$ -> 22.5 clocks
SLL = $t_{SLL} - [t_{SSL}] = 14.27$ -> 14.5 clocks
SLS = $t_{SLS} - [t_{SSS}] = 14.70$ -> 14.5 clocks
RPT = $t_{RPT} = 23.50$ -> 23.5 clocks
CSLS = SLS-$[t_{LSS}$-$t_{SSS}]$ = 13.70 -> 13.5 clocks
LSL = $t_{LSL} - t_{MIN} = 25.74$ -> 25.5 clocks
LSS = $t_{LSS} - t_{MIN} = 24.17$ -> 24.0 clocks
LLL = $t_{LLL} - [t_{LSL}] = 14.84$ -> 15.0 clocks
LLS = $t_{LLS} - [t_{LSS}] = 14.27$ -> 14.5 clocks
LATE/NORMAL = $97
TIME0 = t_{TIME0} = 15.67 -> 15.5 clocks
EARLY/NORMAL = $57
TIME1 = t_{TIME1} = 31.33 -> 31.5 clocks

"Real-World":

MIN = t_{MIN} = 15.0 clocks
MULT = $[65536 / (32 * t_2)] = 65$
SSL = $t_{SSL} - t_{MIN} = 24.17$ -> 24.0 clocks
SSS = $t_{SSS} - t_{MIN} = 24.17$ -> 24.0 clocks
SLL = $t_{SLL} - [t_{SSL}] = 15.84$ -> 16.0 clocks
SLS = $t_{SLS} - [t_{SSS}] = 15.84$ -> 16.0 clocks
RPT = $t_{RPT} = 23.50$ -> 23.5 clocks
CSLS = SLS-$[t_{LSS}$-$t_{SSS}]$ = 15.84 -> 16.0 clocks
LSL = $t_{LSL} - t_{MIN} = 24.17$ -> 24.0 clocks
LSS = $t_{LSS} - t_{MIN} = 24.17$ -> 24.0 clocks
LLL = $t_{LLL} - [t_{LSL}] = 15.84$ -> 16.0 clocks
LLS = $t_{LLS} - [t_{LSS}] = 15.84$ -> 16.0 clocks
LATE/NORMAL = $97
TIME0 = t_{TIME0} = 15.67 -> 15.5 clocks
EARLY/NORMAL = $57
TIME1 = t_{TIME1} = 31.33 -> 31.5 clocks

Copyright © 1987-1988 by Apple Computer, Inc.

Finally, add in the peak shift to the "real-world" Sxx parameters:

SSx = 24.0 - 1.0 = 23.0 clocks
SLx = 16.0 - 0.5 = 15.5 clocks
CSLS = 16.0 - 0.5 = 15.5 clocks

Here's how the theoretical values compare with the "real-world" values:

Parameter	*Theoretical*	*"Real-World"*
MIN	15.0 FCLKs	15.0 FCLKs
MULT	65	65
SSL	24.0 FCLKs	23.0 FCLKs
SSS	22.5 FCLKs	23.0 FCLKs
SLL	14.5 FCLKs	15.5 FCLKs
SLS	14.5 FCLKs	15.5 FCLKs
RPT	23.5 FCLKs	23.5 FCLKs
CSLS	13.5 FCLKs	15.5 FCLKs
LSL	25.5 FCLKs	24.0 FCLKs
LSS	24.0 FCLKs	24.0 FCLKs
LLL	15.0 FCLKs	16.0 FCLKs
LLS	14.5 FCLKs	16.0 FCLKs
Late/Normal	$97	$97
TIME0	15.5 FCLKs	15.5 FCLKs
Early/Normal	$57	$57
TIME1	31.5 FCLKs	31.5 FCLKs

 Copyright © 1987-1988 by Apple Computer, Inc.

Trans-Space Machine

The Trans-Space Machine (TSM) is a part of the ISM hardware that converts data bytes into MFM. Bit 6 of the Setup register (discussed later) controls whether the TSM is bypassed or not. If you are using the ISM for reading/writing GCR data, it <u>must</u> be bypassed since the data does not need any special encoding other than that done in software. The TSM uses the *previous two*, *current* and *next* data bits to determine how the data will be encoded:

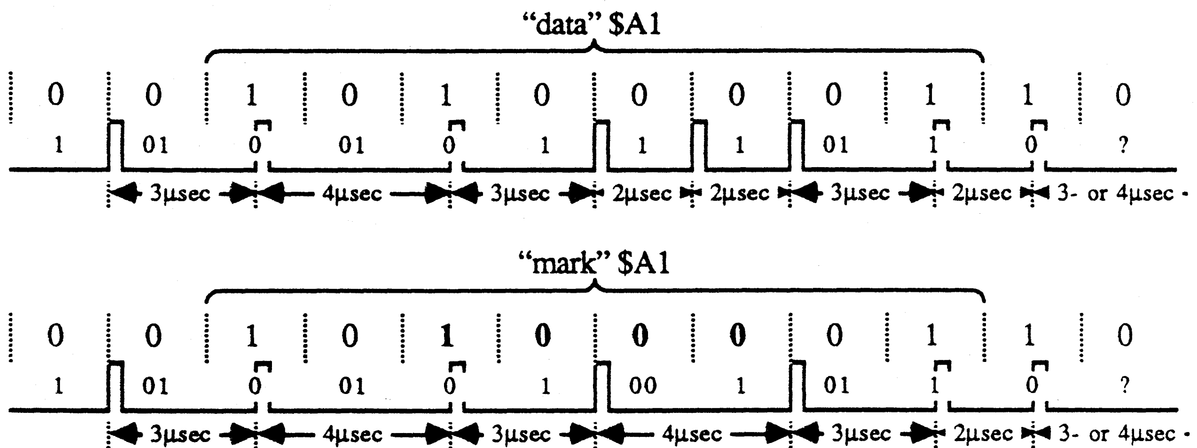
Current Bit	Next Bit	Trans-Space Data
0	0	1
0	1	01
1	0	0
1	1	1

"Where are the *previous two* data bits in the table," you say? Well, they're not there because the only time they're used is when writing a mark byte. In that case, more than two bits is needed to determine where to drop the clock bit (aha!). The TSM then looks for the pattern "1000" and drops a clock bit before the last zero.

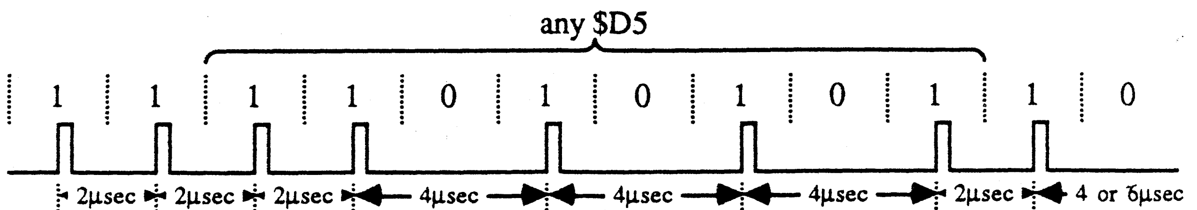
The zeroes and ones output from the TSM (or the actual data bits if the TSM is bypassed) correspond to the *TIME 0* and *TIME 1* RAM parameters. A "0" bit makes a delay *TIME 0* clocks long; a "1" bit makes a delay *TIME 1* clocks long and then causes a pulse (or transition) to occur on the WRDATA output to the disk. For the case of 2,3,4µsec MFM, *TIME 0* is 1µsec long and *TIME 1* is 2µsec long, so we get:

2µsec cells = *TIME 1* clocks
3µsec cells = *TIME 1* + *TIME 0* clocks
4µsec cells = *TIME 1* + *TIME 0* + *TIME 0* clocks

We can compare how the TSM generates the pattern written to the disk for both a data and mark $A1:



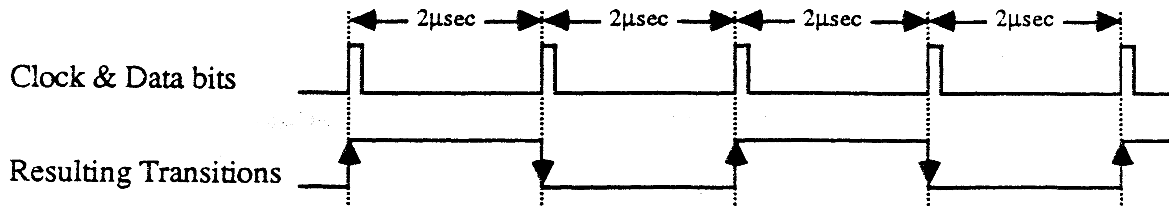
For GCR, there is no encoding of data bytes, so the output pulses are generated directly from the data:



 Copyright © 1987-1988 by Apple Computer, Inc.

Error Correction

The ISM hardware can correct for errors due to assymetry and speed variation using the famous Correction State Machine (CSM). The process starts when the *ACTION* bit is set in the mode register and reading begins. The CSM looks for 32 pairs of minimum cells which coincidently show up in a run of zero bytes, such as a sync field. After that it looks to see if the first non-minimum cell belongs to a mark byte. If not, it starts looking for minimum cells again. It keeps track of the amount of error for both cells in a pair. 32 pairs are used so that there is a large enough sample to accurately represent the amount of error that is actually occurring. The reason why it looks for pairs is that when the data was written to a disk, each of the data or clock bits causes a reversal in the direction of the magnetic field, so we get alternating positive and negative transitions.



It's important to make a distinction between the positive and negative transitions. Due to properties of the media, there is a certain amount of error in determining the exact location of a transition. However, the error for positive transitions tends to be in one direction while the error for negative transitions tends to be in the other direction. The CSM then corrects one way for every other transition and the other way for the other transitions. The difference in the error between the positive and negative transitions represents the assymetry error.

The other correction that can be made is for variations in the drive motor's rotational speed. If the motor speeds up, the cells become shorter; if it slows down, they become longer. Both of these cases make it difficult to read the data because the parameters are based on a particular drive speed.

HOWEVER, the ISM hardware does provide a way to dynamically adjust the parameters to the current drive speed. The Correction Register (see the register set description in the next section) returns two bytes that correspond to the amount of error for the two bytes in a pair over the above-mentioned 32-byte sampling. The first byte is the cumulative error for "even" transitions and the second byte is for "odd" transitions.

If the value is in the range 0 to 192, then cell times were too long and this value is the amount of error. If the value is in the range 193 to 255, then cell times were too short and the amount of error is 256-value. In either case, these values represent the number of clocks of error per 256 clocks. The average of the two values is the speed error, and this can be used to create a new set of parameters, as shown:

corr1 is the first correction byte
corr2 is the second correction byte

corrAvg = (corr1 + corr2) / 2

FCLK' = FCLK * [1 + (corrAvg / 256)]

So to calculate a new set of parameters, we substitute the new FCLK' value into the equations for t_2, t_3, t_4, t_{ps} and t_{pc}, and then use these to calculate the actual parameters.

Example:
FCLK = 15.6672MHz
corr1 = 241 –> 241-256 = -15
corr2 = 37

corrAvg = (-15 + 37) / 2 = 11

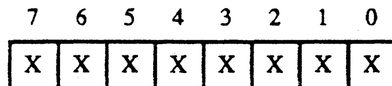
FCLK' = 15.6672MHz * [1 + (11 / 256)] = 16.34MHz

 Copyright © 1987-1988 by Apple Computer, Inc.

ISM Register Set

This section describes the function of all bits in the 16 ISM registers. Each register description includes the register name, whether it's a read or write register, and the register's address in binary (A3...A0). Some mention is made of the term *ACTION*. This is just a bit in the Mode register that is used to start up a read or write operation.

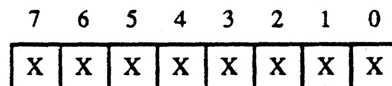
DATA Register **R/W [x000]** **(*ACTION*=1)**
CORRECTION Register **R [1000]** **(*ACTION*=0)**



When *ACTION* is set, this register reads data from and writes data to the FIFO. If a mark byte is read from this location, an error will occur (see Error register, bit 1). If there is still valid data to be read when ACTION is not set, it can be read from the Mark register.

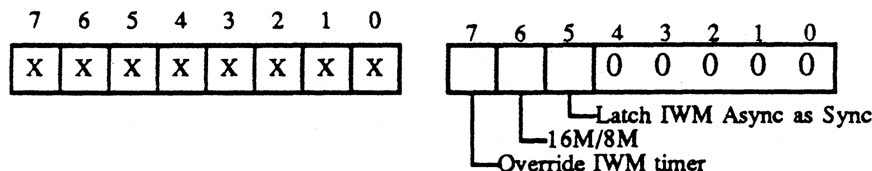
When *ACTION* is not set, two consecutive reads from this location will provide error correction information (see the section on error correction).

MARK Register **R/W [x001]**



Reading from this register will allow a mark byte to be read without causing an error. Writing to this register will cause a byte to be written that has a transition missing between two adjacent zero-bits. This is then interpreted as a mark byte.

CRC Register **W [0010]** **(*ACTION*=1)**
IWM Configuration Register **W [0010]** **(*ACTION*=0)**



This location performs two separate functions depending on the state of the *ACTION* bit. When *ACTION*=1, writing any value to this location will cause*two* bytes from the internal CRC generator to be written out instead of a regular data byte. When *ACTION*=0, the uppermost three bits modify some of the IWM-mode operations. This feature is not supported in the standard ISM.

4 - 0 These bits are reserved for future expansion and should always be set to zero.

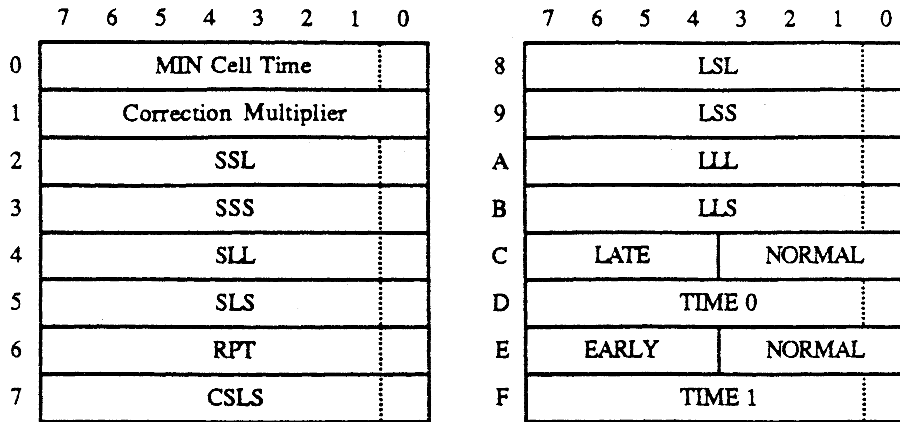
5 Setting this bit to "1" causes the most-significant data bit (D7) to be latched in asynchronous mode (IWM mode register bit 1) as if the IWM was operating in synchronous mode.

6 Setting the bit to "1" causes the IWM timer to take twice as long to time out as usual.

7 If this bit="1", the IWM timer can be killed before it times out. To actually kill the timer, set MotorOn=0, and then toggle the drive select either low-to-high or high-to-low.

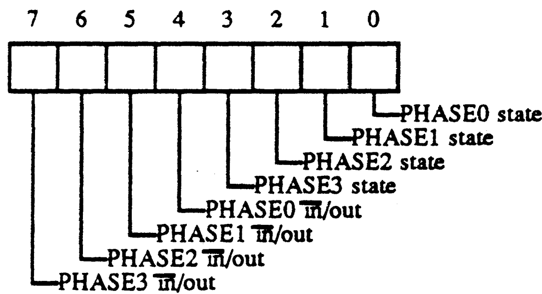
Copyright © 1987-1988 by Apple Computer, Inc.

PARAMETER RAM R/W [x011]



This location consists of 16 bytes of parameter data used to control the read/write timing. An auto-increment counter accesses consecutive RAM addresses every time that a read or write is made to this register. The counter is set to zero after any access is made to the Mode 0 register (register 6) or the chip is reset. See the section on parameter RAM calculations for a more complete description of what each parameter does.

PHASE Register R/W [x100] Reset to 11110000

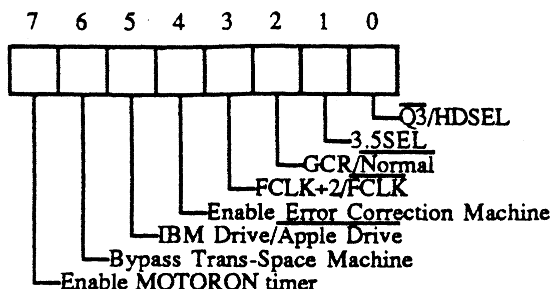


The Phase register controls the direction and state of the four phase lines. Bits 4-7 control the direction of each phase line. Clearing a bit causes the line to be an input, while setting a bit makes the line an output. Bits 0-3 reflect the state of the individual phase lines. If a phase line is configured as an output, setting its corresponding state bit high or low sets the output level on that pin high or low; if it's configured as an input, reading the bit shows the current level of the signal connected to that pin.

NOTE: when the SWIM switches between the IWM and ISM register sets, the current levels of the phase lines are carried over so that no glitches occur.

 Copyright © 1987-1988 by Apple Computer, Inc.

The Setup register is used to configure the ISM hardware.



0 "0" makes the Q3*/HDSEL pin an input to support the Q3 clock ; "1" makes the pin an output to use as a drive head select line.

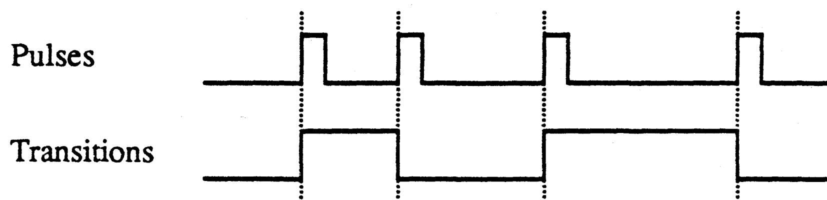
1 Sets the state of the 3.5SEL* pin (note the output state is the inverse of the bit value).

2 Setting the bit selects GCR mode; clearing it selects the normal operating mode.

3 Setting the bit causes the FCLK clock frequency to be divided by 2; otherwise the clock is passed on unmodified.

4 Enables the Error Correction Machine (see the section on error correction).

5 Sets up the RDDATA and WRDATA signals to be either pulses (1) or transitions (0):



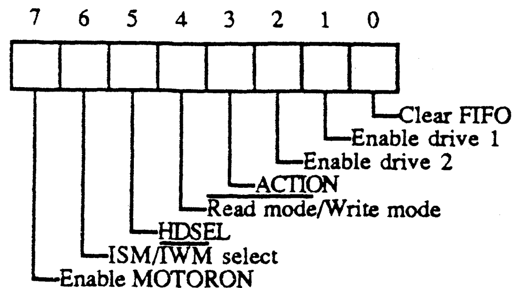
6 Causes the Trans-Space logic to be bypassed. This bit must be set for GCR operation.

7 This bit is used to enable/disable the MotorOn timer which causes the drive enables to remain active for about 1/2 second (at 16MHz) after the MotorOn bit is disabled. After setting this bit, it is necessary to toggle the ACTION bit on then off to enable the timer.

 Copyright © 1987-1988 by Apple Computer, Inc.

MODE Register **W** [011x] Reset to 00000000
STATUS Register **R** [1110]

The Mode register is used to set the various modes of the chip. One or more bits can be set to "0" by writing a byte with those bit(s) set to the "zeroes" location (0110); the bit(s) can be set to "1" by writing to the "ones" location (0111). This scheme is used to make it possible to modify individual bits without affecting other bits in the register that don't need to change. The Status register is used to read back the current value of the mode register.



0 Toggling the *clear FIFO* bit high then low clears the FIFO to begin a read or write operation, and initializes the CRC generator with its starting value. Since this value is different for reading or writing, the read*/write mode bit must be set to the appropriate state before toggling the Clear FIFO bit.

1 Setting this bit along with bit 7 (MotorOn) will enable drive 1.

2 Setting this bit along with bit 7 (MotorOn) will enable drive 2.

3 Setting the *ACTION* bit to "1" starts a read or write operation. It should be set only after everything else has been set up. When writing, at least one byte of data should be written to the FIFO before this bit is set to prevent an underrun when the chip goes to fetch a byte from the [empty] FIFO. This bit will be cleared if an error occurs while in write mode, but not in read mode.

4 This bit determines whether an operation will be a read (0) or write (1) operation.

5 Sets the state of the HDSEL pin if the Q3*/HDSEL bit in the Setup register is set to "1".

6 Clearing this bit switches to the IWM register set. As long as this bit remains a "1" the ISM register set will stay selected.

7 Enables/disables the /ENBL1 and /ENBL2 drive enables (assuming bit 1 or 2 is set). This bit must be set prior to setting ACTION and must not be cleared until after ACTION is cleared.

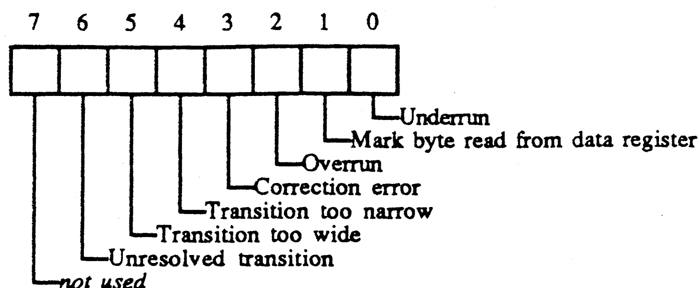
NOTE: MotorOn should be disabled before switching back to the IWM register set. The first thing to do after switching is to clear L7 (and optionally L6) to get out of write mode.

NOTE: after setting ACTION on a read operation, the first byte that will be returned will be a mark byte (as defined in the section on MFM encoding). The search for the mark byte is invisible to the software since it is handled entirely by the SWIM chip.

 Copyright © 1987-1988 by Apple Computer, Inc.

ERROR Register **R** **[1010]** **Reset to 00000000**

This register shows what kind of error has occurred. When any of the bits is set, the Error bit in the Handshake register will also be set. Once one error bit is set, no other bits can be set until the register is cleared. The register is cleared by either reading it or resetting the chip. This register must be cleared before beginning a read or write operation.



0 The processor is not reading/writing fast enough to keep up with the chip.

1 A mark byte (missing transition) was read from the Data register.

2 The processor is reading faster than bytes are available or writing faster than the FIFO is requesting bytes.

3 The correction number obtained in the Error Correction Machine is so large that the error cannot be corrected.

4 A transition occurred before the *MIN* cell time, making the cell too narrow to be legal.

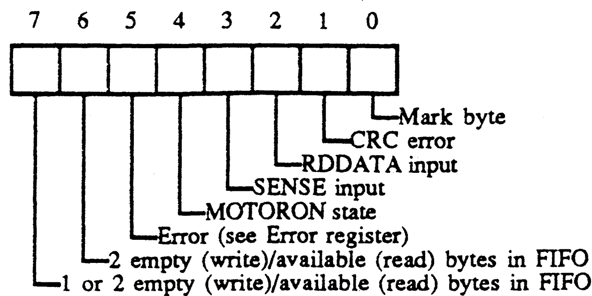
5 A transition didn't occur before *MIN+xSx+xLx+RPT* clocks, making the cell too wide to be legal.

6 There were three marginal transitions in a row which implies that the transitions cannot be resolved.

7 This bit is reserved for future expansion and will always read as a "0".

 Copyright © 1987-1988 by Apple Computer, Inc.

HANDSHAKE Register   R   [1111]



0 If set to "1", it indicates that the next byte to be read is a mark byte (i.e., has a dropped clock pulse).

1 The *CRC error* bit is cleared to zero if the CRC generated on the bytes up to and including the byte about to be read is zero (meaning all the bytes are correct). It's set to "1" if the internal CRC is currently non-zero. The bit is usually checked when the second CRC byte is about to be read from the FIFO.

2 This bit returns the current state of the RDDATA input from the drive.

3 This bit returns the current state of the SENSE input.

4 This bit is set to "1" if either the MotorOn bit in the mode register is a "1" or the timer is timing out.

5 If this bit is set, it indicates that one of the bits in the Error register is set. The bit is cleared by reading the Error register or when the chip is reset.

6 In read mode, this bit indicates that the FIFO contains 2 bytes to be read. In write mode, it indicates that 2 bytes can be written to the FIFO.

7 In read mode, this bit indicates that the FIFO contains *at least* 1 byte to be read. In write mode, it indicates that *at least* 1 byte can be written to the FIFO.

    Copyright © 1987-1988 by Apple Computer, Inc.

IWM State/ISM Register Mapping

This table shows how the IWM's state bits and the ISM's registers are mapped into the SWIM's address space:

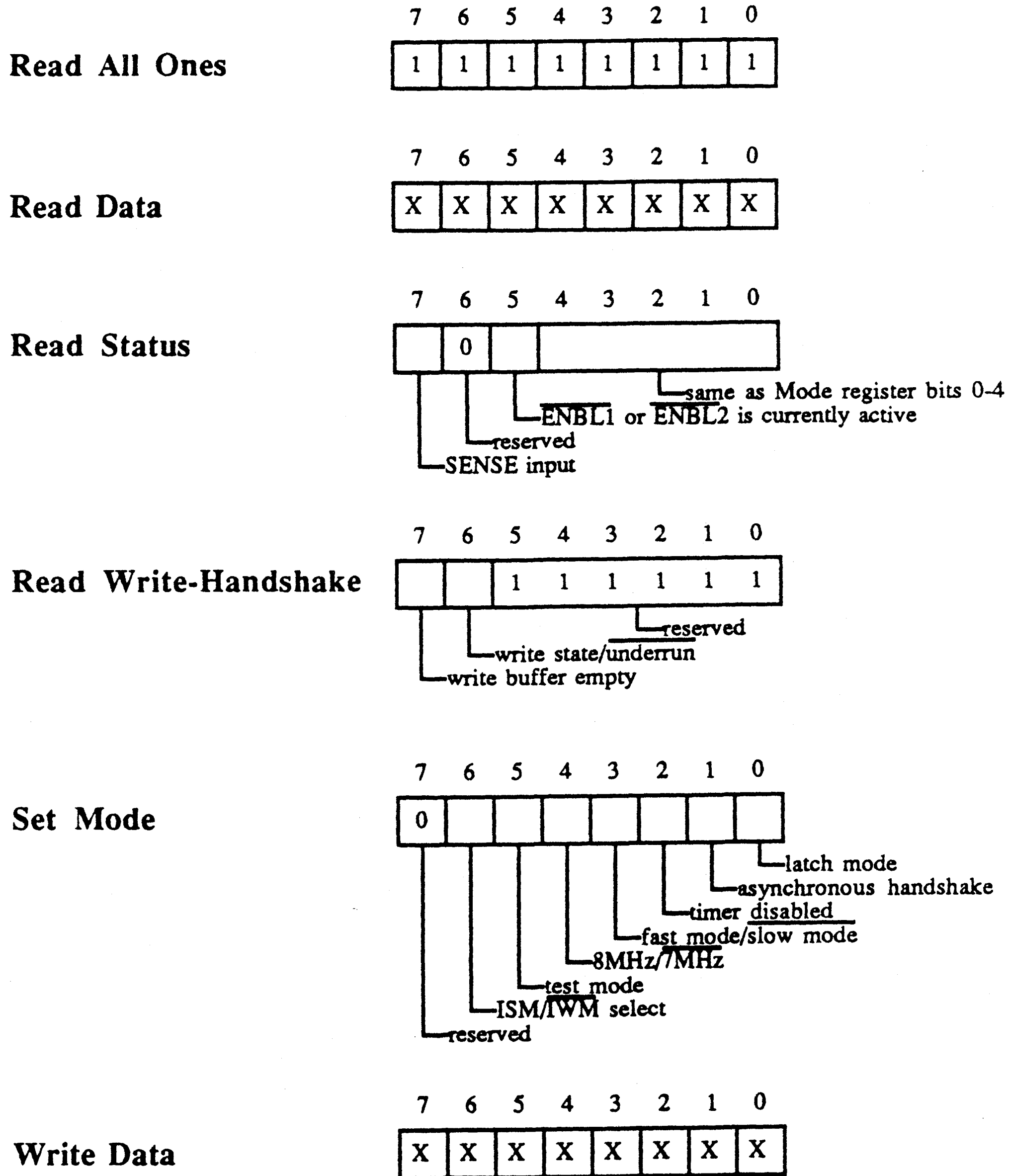
Address	*IWM State*	*ISM Register*
0	PHASE0 = 0	Write Data
1	PHASE0 = 1	Write Mark
2	PHASE1 = 0	Write CRC/IWM Config
3	PHASE1 = 1	Write Parameter RAM
4	PHASE2 = 0	Write Phases
5	PHASE2 = 1	Write Setup
6	PHASE3 = 0	Write Mode (0's)
7	PHASE3 = 1	Write Mode (1's)
8	MOTORON = 0	Read Data
9	MOTORON = 1	Read Mark
10	DRIVESEL = 0	Read CRC
11	DRIVESEL = 1	Read Parameter RAM
12	L6 = 0	Read Phases
13	L6 = 1	Read Setup
14	L7 = 0	Read Status
15	L7 = 1	Read Handshake

 Copyright © 1987-1988 by Apple Computer, Inc.

SWIM 2

0 data xfer
1 mark xfer
10 → 2 error reg read
2 write crc
3 parameter ram
4 phase register
5 setup
6 clear mode
7 set mode
15 → 7 handshake
11 → 6 read status

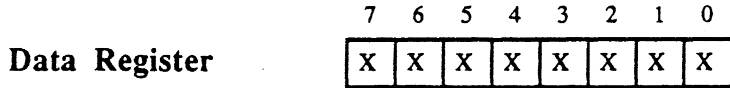
IWM Register Summary



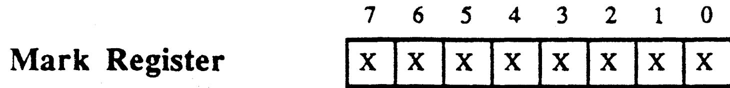
 Copyright © 1987-1988 by Apple Computer, Inc.

ISM Register Summary

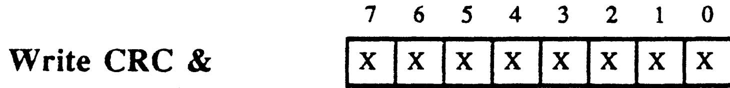
Data Register



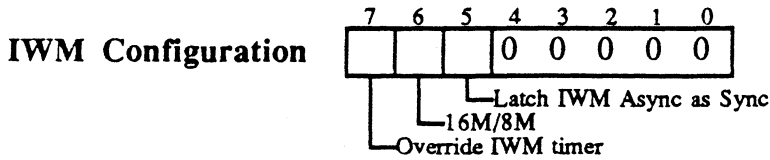
Mark Register



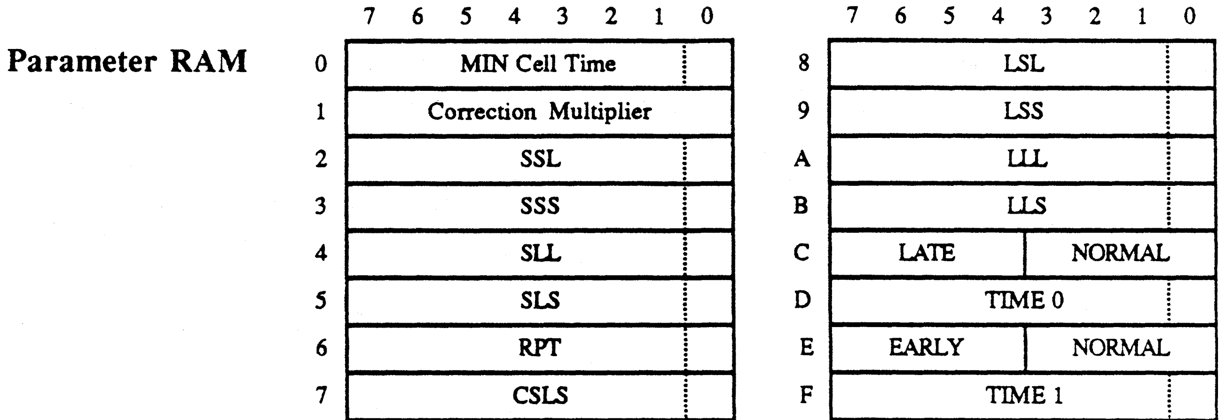
Write CRC &



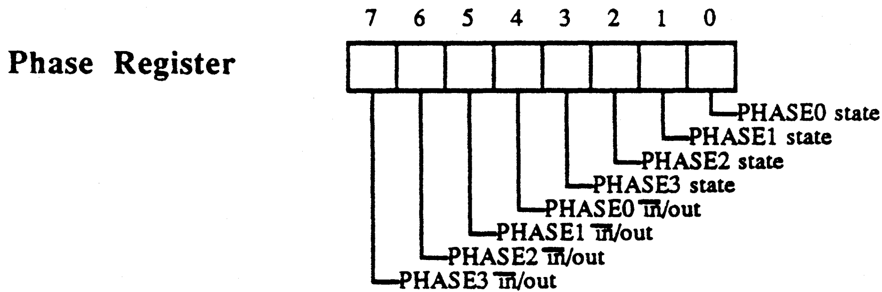
IWM Configuration



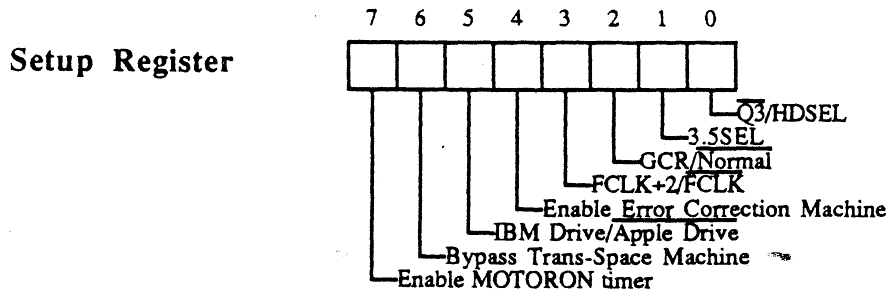
Parameter RAM

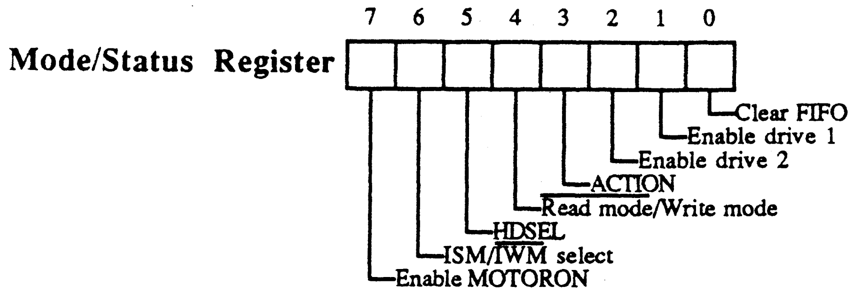


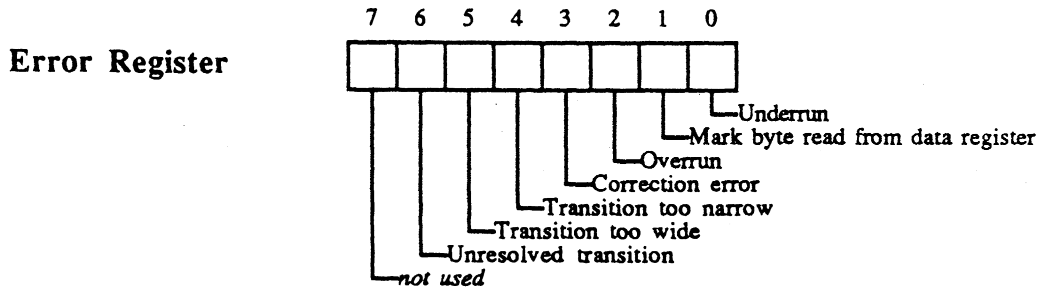
Phase Register

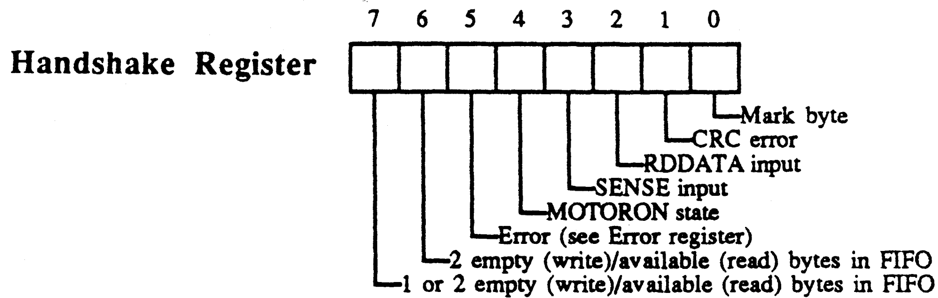


 Copyright © 1987-1988 by Apple Computer, Inc.









 Copyright © 1987-1988 by Apple Computer, Inc.

Code Examples

Here are a couple of short code fragments that illustrate how to read and write in both GCR and MFM. Each example will be written in both 68000 and 6502 assembly language so that everyone can see what's going on.

Read GCR Address Field (68000):

```
addrMarks   DC.B     $D5,$AA,$96,$DE,$AA,$FF
DNiblTbl    DC.B     $00,$01,...         ;De-nibblized values (00xxxxxx) for all 64 combinations

RdAddr      LEA      DNiblTbl-$96,A3 ;Point to de-nibblizing table
            MOVEA.L  IWM,A4
            LEA      Q6L(A4),A4      ;Point to L6=0 for speed

RdAddrMark  LEA      addrMarks,A0
            MOVEQ    #3-1,D1
RdNextMark  MOVE.B   (A4),D0         ;Read a byte
            BPL.S    RdNextMark      ;(not valid until bit 7=1)
            CMP.B    (A0)+,D0        ;Is it the correct mark byte?
            BNE.S    RdAddrMark      ;Start over if not
            DBRA     D1,RdNextMark   ;Loop until all mark bytes have been read

            MOVEQ    #0,D1
RdTrack     MOVE.B   (A4),D1         ;Read the track number
            BPL.S    RdTrack
            MOVE.B   0(A3,D1),D1     ; and de-nibblize it: [0][0][T5][T4][T3][T2][T1][T0]
            MOVE.B   D1,D4           ;Initialize the checksum
            ROR.W    #6,D1           ;Make space for 6 bits of side

            MOVEQ    #0,D2
RdSector    MOVE.B   (A4),D2         ;Read the sector number
            BPL.S    RdSector
            MOVE.B   0(A3,D2),D2     ; and de-nibblize it: [0][0][S5][S4][S3][S2][S1][S0]
            EOR.B    D2,D4           ;Update the checksum

            MOVEQ    #0,D3
RdSide      MOVE.B   (A4),D3         ;Read the side number + upper track number bits
            BPL.S    RdSide
            MOVE.B   0(A3,D3),D1     ; and de-nibblize it: [0][0][S0][T10][T9][T8][T7][T6]
            EOR.B    D1,D4           ;Update the checksum
            ROL.W    #6,D1           ;D1.W=[0][0][0][0][S0][T10][T9][T8][T7][T6][T5][T4][T3][T2][T1][T0]

RdFormat    MOVE.B   (A4),D3         ;Read the format byte (number of sides)
            BPL.S    RdFormat
            MOVE.B   0(A3,D3),D3     ; and de-nibblize it: [0][0][F5][F4][F3][F2][F1][F0]
            EOR.B    D3,D4           ;Update the checksum

            MOVEQ    #0,D0
RdChecksum  MOVE.B   (A4),D0         ;Read the checksum
            BPL.S    RdChecksum
            MOVE.B   0(A3,D0),D0     ; and de-nibblize it:  [0][0][C5][C4][C3][C2][C1][C0]
            EOR.B    D0,D4           ;Is the checksum OK?
            BNE.S    BadChecksum     ;Exit if not

            MOVEQ    #2-1,D0
RdBitSlip   MOVE.B   (A4),D4         ;Read a couple of bit slip marks
            BPL.S    RdBitSlip
            CMP.B    (A0)+,D4        ;Is it the correct byte?
            BNE.S    NoBitSlip       ;Exit with an error if not
            DBRA     D0,RdBitSlip
```

 Copyright © 1987-1988 by Apple Computer, Inc.

Read GCR Address Field (6502):

```
addrMarks       DC.B  $96,$AA,$D5
bitSlipMarks    DC.B  $AA,$DE
DNiblTbl        DC.B  $00,$01,...             ;De-nibblized values (00xxxxxx) for all 64 combinations

RdAddr          LDX   slot16                  ;X=16*(slot+8)
                BIT   MotorOn                 ;Enable the drive

RdAddrMark      LDY   #3-1
RdNextMark      LDA   Q6L,X                   ;Read a byte
                BPL   RdNextMark              ;(not valid until bit 7=1)
                CMP   addrMarks,Y             ;Is it the correct mark byte?
                BNE   RdAddrMark              ;Start over if not
                DEY
                BPL   RdNextMark              ;Loop until all mark bytes have been read

RdTrack         LDY   Q6L,X                   ;Read the track number
                BPL   RdTrack
                LDA   DNiblTbl-$96,Y          ; and de-nibblize it: [0][0][T5][T4][T3][T2][T1][T0]
                STA   checksum                ;Initialize the checksum
                ASL   A                       ;A=[T5][T4][T3][T2][T1][T0][0][0]
                ASL   A
                STA   track

RdSector        LDY   Q6L,X                   ;Read the sector number
                BPL   RdSector
                LDA   DNiblTbl-$96,Y          ; and de-nibblize it: [0][0][S5][S4][S3][S2][S1][S0]
                STA   sector
                EOR   checksum                ;Update the checksum
                STA   checksum

RdSide          LDY   Q6L,X                   ;Read the side number + upper track number bits
                BPL   RdSide
                LDA   DNiblTbl-$96,Y          ; and de-nibblize it: [0][0][S0][T10][T9][T8][T7][T6]
                STA   track+1
                EOR   checksum                ;Update the checksum
                STA   checksum
                LSR   track+1
                ROR   track
                LSR   track+1                 ;track+1=[0][0][0][0][S0][T10][T9][T8]
                ROR   track                   ;track=[T7][T6][T5][T4][T3][T2][T1][T0]

RdFormat        LDY   Q6L,X                   ;Read the format byte (number of sides)
                BPL   RdFormat
                LDA   DNiblTbl-$96,Y          ; and de-nibblize it: [0][0][F5][F4][F3][F2][F1][F0]
                STA   format
                EOR   checksum                ;Update the checksum
                STA   checksum

RdChecksum      LDY   Q6L,X                   ;Read the checksum
                BPL   RdChecksum
                LDA   DNiblTbl-$96,Y          ; and de-nibblize it: [0][0][C5][C4][C3][C2][C1][C0]
                EOR   checksum                ;Is the checksum OK?
                BNE   BadChecksum             ;Exit if not

                LDY   #2-1
RdBitSlip       LDA   Q6L,X                   ;Read a couple of bit slip marks
                BPL   RdBitSlip
                CMP   bitSlipMarks,Y          ;Is it the correct byte?
                BNE   NoBitSlip               ;Exit with an error if not
                DEY
                BPL   RdBitSlip
```

 Copyright © 1987-1988 by Apple Computer, Inc.

Read MFM Address Field (68000):

```
addrMarks     DC.B      $A1,$A1,$A1,$FE

RdAddr        MOVEA.L IWM,A4
              LEA       rHandshake(A4),A3        ;Point to the Handshake register
              LEA       rMark(A4),A4             ; and the Read Data register for speed

RdAddrMark    TST.B     rError-rMark(A4)         ;Clear the error register
              MOVE.B    #$18,wZeroes-rMark(A4)   ;Clear the write and action bits
              MOVE.B    #$01,wOnes-rMark(A4)     ;Toggle to clear FIFO bit to clear out
              MOVE.B    #$01,wZeroes-rMark(A4)   ; any data in the FIFO
              MOVE.B    #$08,wOnes-rMark(A4)     ;Turn on the action bit: GO!
              LEA       addrMarks,A0
              MOVEQ     #4-1,D1

RdNextMark    TST.B     (A3)                     ;Wait for data valid
              BPL.S     RdNextMark
              MOVE.B    (A4),D0                  ; then read a mark byte
              CMP.B     (A0)+,D0                 ;Is it the correct mark byte?
              BNE.S     RdAddrMark               ;Start over if not
              DBRA      D1,RdNextMark            ;Loop until all mark bytes have been read

              MOVEQ     #0,D1
RdTrack       TST.B     (A3)                     ;Wait for data valid
              BPL.S     RdNextMark
              MOVE.B    (A4),D1                  ;Read the track number

RdSide        TST.B     (A3)                     ;Wait for data valid
              BPL.S     RdNextMark
              TST.B     (A4)                     ;Is it side 1?
              BEQ.S     Side0
              BSET      #11,D1                   ;Yes, set the side bit to "1"

Side0         MOVEQ     #0,D2
RdSector      TST.B     (A3)                     ;Wait for data valid
              BPL.S     RdNextMark
              MOVE.B    (A4),D2                  ;Read the sector number

              MOVEQ     #$20,D3
RdBlkSize     TST.B     (A3)                     ;Wait for data valid
              BPL.S     RdNextMark
              OR.B      (A4),D3                  ;Read the block size byte (should be $02)

RdCRC1        TST.B     (A3)                     ;Wait for data valid
              BPL.S     RdNextMark
              TST.B     (A4)                     ;Toss the first CRC byte

RdCRC2        MOVE.B    (A3),D5                  ;Wait for data valid (save the CRC error bit)
              BPL.S     RdNextMark
              TST.B     (A4)                     ;Toss the second CRC byte

              BTST      #1,D5                    ;CRC error?
              BNE.S     CRCError                 ;Exit with error if so
```

 Copyright © 1987-1988 by Apple Computer, Inc.

Read MFM Address Field (6502):

```
addrMarks    DC.B $FE,$A1,$A1,$A1        ;Address mark bytes (backwards)

RdAddr       LDX  slot16                 ;X = 16*(slot+8)

RdAddrMark   BIT  rError,X               ;Clear the error register
             LDA  #$18                   ;Clear the write and action bits
             STA  wZeroes,X
             LDA  #$01                   ;Toggle to clear FIFO bit to clear out
             STA  wOnes,X                ; any data in the FIFO
             STA  wZeroes,X
             LDA  #$08                   ;Turn on the action bit: GO!
             STA  wOnes,X
RdAddrMark   LDY  #4-1

RdNextMark   LDA  rHandshake,X           ;Wait for data valid
             BPL  RdNextMark
             LDA  rMark,X                ; then read a mark byte
             CMP  addrMarks,Y            ;Is it the correct mark byte?
             BNE  RdAddrMark             ;Start over if not
             DEY
             BPL  RdNextMark             ;Loop until all mark bytes have been read

RdTrack      LDA  rHandshake,X           ;Wait for data valid
             BPL  RdTrack
             LDA  rData,X
             STA  track                  ;Read the track number

RdSide       LDA  rHandshake,X           ;Wait for data valid
             BPL  RdSide
             LDA  rData,X                ;Read the side number
             STA  side

RdSector     LDA  rHandshake,X           ;Wait for data valid
             BPL  RdSector
             LDA  rData,X                ;Read the sector number
             STA  sector

RdBlkSize    LDA  rHandshake,X           ;Wait for data valid
             BPL  RdBlkSize
             LDA  rData,X                ;Read the block size byte (should be $02)
             STA  blockSize

RdCRC1       LDA  rHandshake,X           ;Wait for data valid
             BPL  RdCRC1
             LDA  rData,X                ;Toss the first CRC byte

RdCRC2       LDA  rHandshake,X           ;Wait for data valid (save the CRC error bit)
             BPL  RdCRC2
             LDY  rData,X                ;Toss the second CRC byte

             AND  #%00000010             ;CRC error?
             BNE  CRCError               ;Exit with error if so
```

 Copyright © 1987-1988 by Apple Computer, Inc.

Bibliography

"Integrated WOZ Machine (IWM) Device Specification," Apple Computer, Inc. 1978.

Macintosh .SONY floppy disk driver source, Apple Computer, Inc. 1982.

"Inside Macintosh, Volumes I, II, III", Apple Computer, Inc. 1985.

"ISM ASIC Specifications, revision 4.1", Apple Computer, Inc. July 2, 1987.

"SWIM Chip Specification", Apple Computer, Inc. September 29, 1987.

"Microcomputer Products Data Book, Volume 2 of 2", pp. 6-21, 6-24, 6-25 (µPD765A), NEC Electronics Inc. 1987.

 Copyright © 1987-1988 by Apple Computer, Inc.