

I/O events are usually asynchronous, i.e., not timed with the microprocessor internal instructions, therefore, it would be possible for the event to occur shortly after the programmer has stopped to look at I/O events which would mean that the event would not be sampled until the programmer took the time to stop his coding and sample again.

Because the sampling of I/O devices normally takes several byte counts or cycles to accomplish, the frequent insertion of checking routines into straight line code results in significant delays to the entire program. In trying to use this technique, there has to be a trade-off between the fact that the program wastes a significant amount of time checking events which have not yet occurred versus delaying checking of an event which has occurred and if not timely serviced the data may be lost.

In order to solve this dichotomy, the concept of interrupt is used to signal the microprocessor that an external event has occurred and the microprocessor should devote attention to it immediately. This technique accomplishes processing in which the microprocessor's program is interrupted and the event that caused the interrupt is serviced.

Transferring most of data and control to I/O devices in an interrupt driven environment will usually result in maximum program and/or programmer efficiency. Each event is serviced when it occurs which means there is a minimum amount of delaying in servicing events, also a minimum amount of coding because of elimination of the need to determine occurrence of several events simultaneously; each interrupting event is handled as a unique combination. It is possible to interrupt an interrupt processing routine and, therefore, all the interrupt logic uses the stack which allows processing of successive interrupts without any penalty other than increasing the stack length.

A real world example of an event which should interrupt is when the user is given a panic button indicating to the microcomputer some event has occurred which requires total immediate attention of the microprocessor to solving that problem.

The action and events are as follows: The microprocessor user pushes the panic button; the panic switch sensor causes an external device to indicate to the microprocessor an interrupt is desired; the microprocessor checks the status of the internal interrupt inhibit signal; if the internal inhibit is set, then the interrupt is ignored. However, if it is reset or when it becomes reset through some program reaction, the following set of operations occur:

Example 9.2: Interrupt Sequence

Cycles	Address Bus	Data Bus	External Operation	Internal Operation
1	PC	OP CODE	Fetch OP CODE	Hold Program Counter, Finish Previous Operation
2	PC	OP CODE	Fetch OP CODE	Force a BRK Instruction, Hold P-Counter
3	01FF	PCH	Store PCH on Stack	Decrement Stack Pointer to 01FE
4	01FE	PCL	Store PCL on Stack	Decrement Stack Pointer to 01FD
5	01FD	P	Store P on Stack	Decrement Stack Pointer to 01FC
6	FFFE	New PCL	Fetch Vector Low	Put Away Stack
7	FFFF	New PCH	Fetch Vector High	Vector Low → PCL and Set I
8	Vector PCH PCL	OP CODE	Fetch Interrupt Program	Increment PC to PC + 1

As can be seen in Example 9.2, the microprocessor uses the stack to save the reentrant or recovery code and then uses the interrupt vectors FFFE and FFFF, (or FFFA and FFFB), depending on whether or not an interrupt request or a non maskable interrupt request had occurred. It should be noted that the interrupt disable is turned on at this point by the microprocessor automatically.

Because the interrupt disable had to be off for an interrupt request to have been honored, the return from interrupt which loads the processor status from before the interrupt occurred has the effect of clearing the interrupt disable bit. After the interrupt has been acknowledged by the microprocessor by transferring to the proper vector location, there are a variety of operations which the user can perform to service the interrupt; however, all operations should end with a single instruction which reinitializes the microprocessor back to the point at which the interrupt occurred. This instruction is called the RTI instruction.

9.6 RTI - RETURN FROM INTERRUPT

This instruction transfers from the stack into the microprocessor the processor status and the program counter location for the instruction which was interrupted. By virtue of the interrupt having stored this data before executing the instruction and the fact that the RTI reinitializes the microprocessor to the same state as when it was interrupted, the combination of interrupt plus RTI allows truly reentrant coding.

The symbolic notation for RTI is $\uparrow P \uparrow PC$.

The RTI instruction reinitializes all flags to the position to the point they were at the time the interrupt was taken and sets the program counter back to its pre-interrupt state. It affects no other registers in the microprocessor.

RTI is a single byte instruction and its addressing mode is Implied.

In the following example, we can see the internal operation of the RTI which restores the microprocessor:

Example 9.3: Return from Interrupt

Cycles	Address Bus	Data Bus	External Operation	Internal Operation
1	0300	RTI	Fetch OP CODE	Finish Previous Operation, Increment PC to 0301
2	0301	?	Fetch Next OP CODE	Decode RTI
3	01FC	?	Discarded Stack	Increment Stack
4	01FD	P	Fetch P Register	Pointer to 01FD
5	01FE	PCL	Fetch PCL	Increment Stack Pointer to 01FE
6	01FF	PCH	Fetch PCH	Increment Stack Pointer to 01FF, Hold PCL
7	PCH PCL	OP CODE	Fetch OP CODE	M>PCL, Store Stack Pointer
				Increment New PC

Note the effects of the extra cycle (3) necessary to read data from stack which causes the RTI to take six cycles. The RTI has restored the stack, program counter and status register to the point they were at before the interrupt was acknowledged.

There is no automatic save of any of the other registers in the microprocessor. Because the interrupt occurred to allow data to be transferred using the microprocessor, the programmer must save the various internal registers at the time the interrupt is taken and restore them prior to returning from the interrupt. Saving of the registers is best done on the stack as this allows as many consecutive interrupts as the programming will allow for. Therefore, the routines which save all registers and restore them are as follows:

Example 9.4: Illustration of Save and Restore for Interrupts

Cycle	Bytes			
3	1	SAVE	PHA	Save A
2	1		TXA	Save X
3	1		PHA	
2	1		TYA	Save Y
3	1		PHA	
<u>13</u>	<u>5</u>			
4	1	RESTORE	PLA	Restore Y
2	1		TAY	
4	1		PLA	Restore X
2	1		TAX	
4	1		PLA	Restore A
<u>16</u>	<u>5</u>			

The SAVE coding assumes that the programmer wants to save and to restore registers A, X and Y. It should be noted that for many interrupts, the amount of coding that has to be performed in the interrupt is fairly small.

In this type of operation, it is usually more desirable to shorten the interrupt processing time and not use all of the registers in the machine. Therefore, a more normal interrupt processing routine would consist of just saving registers A and X which means that the restore routine would be just restore registers X and A. This has the effect of shortening the interrupt routine by two bytes, and also shortens the restore routine by two bytes and will cut 5 cycles out of the interrupt routine and 6 cycles out of the restore routine.

This technique combined with automatic features of the interrupt and the RTI allows multiple interrupts to occur with successive interrupts interrupting the current interrupt. This is one of the advantages of the use of the stack so that as many interrupts can interrupt other interrupts as can be held in the stack. The stack contains six bytes for every interrupt if all registers are saved, so 42 sequences of interrupts can be stored in one page. However, in more practical situations, consecutive interrupts hardly ever get more than about three deep.

The advantage of allowing an interrupt to interrupt an interrupt is that the whole concept behind the interrupt is that asynchronous events can be responded to as rapidly as possible; therefore, it is desirable to allow the processing to service one interrupt to be interrupted to service the second, as long as the first interrupt has been properly serviced.

To review how this is accomplished using the normal interrupt capability of the MCS650X, it is important that we review the bus concept which is inherent in the MCS6500 family and which is compatible with the M6800.

As has already been discussed, all I/O operations on this type of microprocessor are accomplished by reading and writing registers which

actually represent connections to physical devices or to physical pins which connect to physical devices.

Up until this point, this discussion has addressed itself to transferring of data into and out of the microprocessor. However, there is a concept that is inherent in the bus discipline that says that whenever an interrupt device capable of generating an interrupt desires to accomplish an interrupt, it performs two acts; first, it sets a bit, usually bit 7, in a register whose primary purpose is to communicate to the microprocessor the status of the device. The interrupting device causes one of perhaps many output lines to be brought low. These collector-or'd outputs are connected together to the $\overline{\text{IRQ}}$ pin on the MCS650X microprocessor.

The interrupt request to the MCS650X is the $\overline{\text{IRQ}}$ pin being at a TTL zero. In order to minimize the handshaking necessary to accomplish an interrupt, all interrupting devices obey a rule that says that once an interrupt has been requested by setting the bit and pulling interrupt low, the interrupt will be held by the device until the condition that caused the interrupt has been satisfied. This allows several devices to interrupt simultaneously and also allows the microprocessor to ignore an interrupt until it is ready to service it. This ignoring is done by the interrupt disable bit which can be set on by the programmer and is initialized on by the interrupt sequence or by the start sequence.

Once the interrupt line is low and interrupt disable is off, the microprocessor takes an interrupt which sets on the interrupt disable. The interrupt disable then keeps the input low line from causing more than one interrupt until an interrupt has been serviced. There is no other handshaking between the microprocessor and the interrupting device other than the collector-or'd line. This means that the microprocessor must use the normal addressing registers to determine which of several collector-or'd devices caused the line to go low and to process the interrupt which has been requested.

Once the processor has found the interrupting device by means of analyzing status bits which indicates an interrupt has been requested, the microprocessor then clears the status by reading or writing data as indicated by the status register.

It should be noted that a significant difference between status registers and data registers in I/O devices is that status registers are never cleared by being read, only by being written into or by the microprocessor transferring data from a data register which corresponds to some status in the status register. Detailed examples of this interaction are discussed in Chapter 11. The clearing of the status register also releases the collector-or'd output thereby releasing the interrupt pin request.

The basic interaction between the microprocessor and interrupting device is when interrupting device sets the status bit and brings its output \overline{IRQ} line low. If its output \overline{IRQ} line is connected to the microprocessor interrupt request line, the microprocessor waits until the interrupt disable is cleared, takes the interrupt vector, and sets the interrupt disable which inhibits further interrupts in the \overline{IRQ} line. The microprocessor determines which interrupting device is causing an interrupt and transfers data from that device.

Transferring of data clears the interrupt status and the \overline{IRQ} pin. At this point, the programmer could decide that he was ready to accept another interrupt even though the data may have been read but not yet operated on. Allowing interrupts at this point, gives the most efficient operation of the microprocessor in most applications.

There are also times when a programmer may be working on some coding the timing of which is so important that he cannot afford to allow an interrupt to occur. During these times, he needs to be able to turn on the interrupt disable. To accomplish this, the microprocessor has a set and clear interrupt disable capability.

9.7 SOFTWARE POLLING FOR INTERRUPT CAUSES

As was indicated above, any one of several devices are collector-or'd to cause an IRQ. The effect of any one of the devices or combination of them having polled the IRQ line low is always the same. The interrupt stores the current status of the program counter and processor on the stack and transfers to a fixed vector address. In servicing the interrupt, it is important to save those registers which will be used in the analysis of the interrupt and during the interrupt processing, so the normal first steps of the interrupt routine are to do the SAVE procedures.

The next operation is to determine which of the various potential interrupting devices caused the interrupt. To accomplish this, the programmer should make use of the fact that all interrupting devices signal the interrupt by a bit in the status register. All currently implemented 6800 and 6500 peripherals always have interrupt indicators; either bit 7 or bit 6 in their status register. Therefore, the basic loop that a user will use to verify the existence of an interrupt on one of five devices is as follows:

Example 9.5: Interrupt Polling

<u>No.of Bytes</u>	<u>Cycles</u>		
3	4	LDA	Status 1
2	2	BMI	FIRST
3	4	LDA	Status 2
2	2	BMI	SECOND
3	4	LDA	Status 3
2	2	BMI	THIRD
3	4	LDA	Status 4
2	2	BMI	FOURTH
3	4	LDA	Status 5
2	2	BMI	FIFTH
		RES1 JMP	to RESTORE
FIRST	LDA		DATA 1
		CLI	
		Process 1	
		etc.	

In this example, the simplest case where the potential interrupts are indicated by bit 7 being on, has been assumed. This allows advantage to be taken of the free N-bit test by following the load of the first status register with a branch on result minus. If the first device has an active interrupt request, the BMI will be taken to FIRST where the data is transferred. This automatically clears the interrupt for the first device. To allow multiple interrupts, the load A is followed by the CLI instruction which allows the program to accept another interrupt. As a result of the CLI, one of two things can occur; there is not another interrupt currently active, in which case, the microprocessor will continue to process the first interrupt down to the point where the interrupt is complete and the first subroutine does a jump to RESTORE, which is the routine that unsaves the registers that were used in the process of servicing the interrupt. If another device has an active interrupt which occurred either prior to the first interrupt or subsequent to it but before the microprocessor has reached the point where the CLI occurs, then the microprocessor will immediately interrupt again following the CLI, go back and save registers as defined before and come back into the polling loop. Therefore, multiple interrupts are serviced in the order in which they are looked at in polling sequence. Polling means that the program is asking each device individually whether or not it is the one that requested an interrupt.

It should be noted that polling has the effect of giving perfect priority in the sense that no matter which two interrupts occur before the microprocessor gets to service one, the polling sequence always gives priority to the highest priority device first, then the second, then the third, etc. In light of the fact that this polling sequence requires no additional hardware to implement other than is available in the interrupting devices themselves, this is the least expensive form of interrupt and the one that should be used whenever possible because of its independence from external hardware.

Although it would appear that the last interrupting device in a sequence pays a significant time penalty based on the amount of instructions to be executed before the last device is serviced, the amount of time to perform polls is only six cycles per device and, therefore, the extra penalty that the last device has to pay over the first device is 24 cycles. This is in comparison to a minimum time to cause an interrupt (eight cycles), plus store time for registers (in the range of another 8 to 13 cycles) which means that the delay to the last devices is roughly twice what it would be for the first device.

This timing just described represents a most interesting part of the analysis of interrupts for a microprocessor. There is a significant amount of fixed overhead which must be paid for the interrupt. This overhead includes the fact that the interrupts can only occur at the end of an instruction so, therefore, if an interrupt occurs prior to the end of an instruction, the microprocessor delays until the end of the instruction to service it. Therefore, in doing the worst case analysis, one has to consider the fact that the interrupt might be occurring in the middle of a seven cycle, read/modify/write instruction which means that the worst case time to process the first instruction in an interrupt sequence is 14 cycles (7 cycles plus the 7 cycles for the interrupt).

In light of the fact that saving of additional registers is often required (at least the accumulator A must be saved), at least twice the number of cycles will be required. Consequently the absolute minimum worse case time for an interrupt is 17 cycles plus the time to transfer data which is another 4 cycles. Therefore interrupt driven systems must be capable of handling a delay of at least 20 cycles and more realistically, 20 to 50 cycles before the first interrupt is serviced. This means that devices which are running totally interrupt driven must not require successive bytes of data to be transferred to the microprocessor in less than 30 or 40 cycles and on a given system, only one device is capable of operating at that rate at one time. This limits the interrupt driven frequency of data transfer to 40KHZ at a one megahertz clock system and 80KHZ on a two megahertz clock system.

An even more serious problem is the timing delay when an interrupt has just started to be serviced. The interrupt mask is on and higher priority interrupts are blocked from service. In this case, the delay to the service can easily stretch out to 100 cycles before the interrupt mask is cleared. This is one of the reasons for clearing up the interrupt mask as soon as data is transferred. (The non-maskable interrupt which will be discussed later is one solution to this problem.) A second is to only use interrupts for systems that have adequate buffering and/or slower transfer rates. This does not imply that most microprocessor applications should not be primarily interrupt driven. The MCS650X interrupt system is designed to be very economical and easy to apply. It should be used for almost all control applications, other than when the throughput described is not sufficient to handle the particular problem. It should be remembered that at one megahertz the fast MCS650X is not really capable of handling problems with more than 50KHZ byte throughput for a sustained period of operation. It is also true that in most control applications, many of the signals occur at much slower rates or are bufferable so that the response time to a request for service is significantly longer than the 20 to 50 cycles that can normally be expected with a polling system. Because of this, it is expected that most applications will be quite satisfied using the polling technique described above.

9.8 FULLY VECTORED INTERRUPTS

However, there are occasions where several high speed peripherals can be managed by the microprocessor if the user is willing to make the investment to attain a truly vectored interrupt. There is a second level of interrupt vectoring possible by just putting one high priority device on the non-maskable interrupt line. However, the case when multiple inputs are desired with both priority encoding and true vectoring, the MCS650X when combined with appropriate hardware has the ability in the first polling instruction to transfer control to appropriate interrupting device service software.

The MCS6520 contains, in its two bytes of memory, an indirect pointer to the address of the subroutine in which resides the interrupt processing for the devices, which the priority encoder has selected. This gives an effective service time of approximately 24 cycles to a prioritized interrupt and is one of the primary applications of the jump indirect capability.

9.8.1 JMP Indirect

This instruction establishes a new value for the program counter.

It affects only the program counter in the microprocessor and affects no flags in the status register.

JMP Indirect is a three byte instruction.

In the JMP Indirect instruction, the second and third bytes of the instruction represent the indirect low and high bytes respectively of the memory location containing ADL. Once ADL is fetched, the program counter is incremented with the next memory location containing ADH.

Example 9.6: Illustration of JMP Indirect

<u>Cycle</u>	<u>Address Bus</u>	<u>Data Bus</u>	<u>External Operation</u>	<u>Internal Operation</u>
1	0100	OP CODE	Fetch OP CODE	Finish Previous Operation. Increment PC to 0101
2	0101	IAL	Fetch IAL	Interpret Instructions Increment PC to 102
3	0102	IAH	Fetch IAH	Store IAL
4	IAH, IAL	ADL	Fetch ADL	Add 1 to IAL
5	IAH, IAL+1	ADH	Fetch ADH	Store ADL
6	ADH, ADL	Next OP CODE	Fetch Next OP CODE	

9.9 INTERRUPT SUMMARY

There is an interrupt request line (IRQ) that, when low, indicates one of the devices which are connected to the interrupt request line requires service. At the beginning of the interrupt service routine, the user should save, on the stack, whatever registers will be used in his interrupt processing routine. His program then goes through a polling sequence to determine the interrupting device by analyzing the status registers in the order of priority of service for the I/O devices. On finding a device which requires service, the data for that device should be read or written as soon as possible and the interrupt disable cleared so that the microprocessor can interrupt again to service lower priority devices. Devices with over 40KHz byte transfer, etc., and mixed devices with over 20KHz should not normally be run interrupt driven. All others should be run interrupt driven as it minimizes the service time and programming for interrupt I/O operations.

9.10 NON-MASKABLE INTERRUPT

As is discussed, it is often desirable to have the ability to interrupt an interrupt with a high priority device which cannot afford to wait during the time interrupts are disabled. For this reason, the MCS650X has a second interrupt line, called a Non-Maskable Interrupt. The input characteristics of this line are different than the interrupt request line which senses it needs service when it remains low. The non-maskable input is an edge sensitive input which means that when the collector-or'd input transitions from high to low, the microprocessor sets an internal flag such that at the beginning of the next instruction, no matter what the status of the interrupt disable, the microprocessor performs the interrupt sequence shown in Example 9.2 except that the vector pointer put out in cycle 6 and 7 is FFFA and FFFB.

This gives two effects of a non-maskable interrupt. First, no matter what the status of the interrupt disable, the non-maskable interrupt will interrupt at the beginning of the next instruction, therefore, the maximum response time to the vector point is 14 cycles. Secondly, the internal logic of the MCS650X is such that if an interrupt request and non-maskable interrupt occur simultaneously or if the non-maskable interrupt occurs prior to the time that the vectors are selected, the microprocessor

always assigns highest priority to the non-maskable interrupt. Therefore, the FFFA and FFFB vector are always taken if both interrupts are active at the time the vector is selected. Thus the non-maskable interrupt is always a higher priority fast response line, and can, in any given system be used to give priority to the high speed device.

It is possible to connect multiple devices to the non-maskable interrupt line except for the fact that the non-maskable interrupt is edge sensitive. Therefore, the same logic that allows the IRQ to stay low until the status has been checked and the data transferred will keep the non-maskable interrupt line in a low state until such time as the first interrupt is serviced. If subsequent to the first interrupt of a non-maskable interrupt line occurring, a second device which is collector-ored would have turned on its status and collector-or'd output, the clearing of the first interrupt request would not cause the line to re-initialize itself to the high state and the microprocessor would ignore the second interrupt. Therefore, multiple lines connected to the non-maskable interrupt must be carefully serviced.

In any case, NMI is always one free high priority vectored interrupt. By virtue of the fact that it goes to a different vector pointer, the microprocessor programmer can be guaranteed that in 17 cycles he can transfer data from the interrupting device on the non-maskable interrupt input.

The IRQ and NMI are lines which, externally to the microprocessor, control the action to the microprocessor through an interrupt sequence. As is mentioned during the discussion on the start command, the restart cycle is a pseudo interrupt operation with a different vector being selected for reset which has priority over both interrupt and non-maskable interrupt. Non-maskable interrupt has priority over interrupt. There is also a software technique which allows the user to simulate an interrupt with a microprocessor command, BRK. It is primarily used for causing the microprocessor to go to a halt condition or stop condition during program debugging.

9.11 BRK - BREAK COMMAND

The break command causes the microprocessor to go through an interrupt sequence under program control. This means that the program counter of the second byte after the BRK is automatically stored on the stack along with the processor status at the beginning of the break instruction. The microprocessor then transfers control to the interrupt vector.

Symbolic notation for break is PC + 2↓ (FFFE)→PCL (FFFF)→PCH.

Other than changing the program counter, the break instruction changes no values in either the registers or the flags.

The BRK is a single byte instruction and its addressing mode is implied.

As is indicated, the most typical use for the break instruction is during program debugging. When the user decides that the particular program is not operating correctly, he may decide to patch in the break instruction over some code that already exists and halt the program when it gets to that point. In order to minimize the hardware cost of the break which is applicable only for debugging, the microprocessor makes use of the interrupt vector point to allow the user to trap out that a break has occurred. In order to know whether the vector was fetched in response to an interrupt or in response to a BRK instruction, the B flag is stored on the stack, at stack pointer plus 1, containing a one in the break bit position, indicating the interrupt was caused by a BRK instruction. The B bit in the stack contains 0 if it was caused by a normal IRQ. Therefore, the coding to analyze for this is as follows in Example 9.6.

Example 9.7: Break-Interrupt Processing

Cycles	Bytes	Check for A BRK	Flag
4	1	PLA	Load status register
3	1	PHA	Restore onto Stack
2	2	AND # \$ 10	Isolate B Flag
2	2	BNE BRKP	Branch to Break Programming
11	6		

↓

Normal Interrupt Processing

This coding can be inserted any place in the interrupt processing routine. During debugging, if the user can afford the execution time, it should be placed immediately after the save routine. If not, it can be put at the end of the polling routine which gives a priority to the polling devices as far as servicing the interrupts. However, it should be noted that in order not to lose the break, the returns from all interrupts during debugging should go through an equivalent routine.

Once the user has determined that the break is on, a second analysis and correction must be made. It does not operate in a normal manner of holding the program counter pointing at the next location in memory during the BRK instruction. Because of this, the value on the stack for the program counter is at the break instruction plus two. If the break had been patched over an instruction, this is usually of no significant consequence to the user. However, if it is desired to process the next byte after the break instruction, the use of decrement memory instructions in the stack must be used.

It is recommended that the user normally takes care of patching programs with break by processing a full instruction prior to returning and then use jump returns.

An interesting characteristic about the break instruction is that it's OP CODE is all zero's (0), therefore, BRK coding can be used to patch fusible link PROMS through a break to an E-ROM routine which inserts patch coding.

An example of using the break to patch with is shown below:

Example 9.8: Patching with a break utilizing PROMs

Old Code	FC21	LDA
	FC22	05
	FC23	21
	FC24	Next OP CODE
Patched Code	FC21	BRK 00
	FC22	05
	FC23	21
	FC24	Next OP CODE

The interrupt vector routine points to:

Patch	LDA
06	06
21	21
JMP	JMP
24	24
FC	FC

This coding substitutes:

LDA 2106
for the
LDA 2105
coding at
FC21

by use of the BRK and a break processing routine.

9.12 MEMORY MAP

A series of requirements were discussed to this point for the memory organization which can be illustrated by the following memory map:

Hex Address

0000-0OFF	RAM used for zero page and indirect memory addressing operation.
0100-01FF	RAM used for stack processing and for absolute addressing.
0200-3FFF	Normally RAM.
4000-7FFF	Normally I/O
8000-FFF9	Program storage normally ROM.
FFFA	Vector low address for NMI.
FFFB	Vector high address for NMI.
FFFC	Vector low address for RESET.
FFFD	Vector high address for RESET.
FFFE	Vector low address for IRQ + BRK.
FFFF	Vector high address for IRQ + BRK.

The addressing schemes for I/O control between locations 4000 and 8000 Hex, have not been fully developed. This is described in detail in the Hardware Manual, Chapter 2. The Zero Page addressing requires that RAM should be located starting in location 00. If more than one RAM page is necessary, RAM location 0100 through 01FF should be reserved for the stack or at least a portion of parts should be reserved for the stack with the rest of it being available to the user to use as normal RAM. Locations from 0200 up to 4000 are normally reserved for RAM expansion.

In small memory configurations such as are inherent in a MCS6530 class device, in order to minimize the addressing lines, page two (02XX) will be normally used for input/output as opposed to using the 40XX page which is used for devices which require significant amount of outboard RAM, ROM and I/O.

Because of the fact that the MCS650X has three very important vector points selected in highest order memory, it is usually more useful to write programs with the memory storage located at a starting address which allows the programmer to make sure that the last address in his ROM contains the start and interrupt vectors. Because of these allocations, the user finds himself working in three directions. RAM is assigned in location 0000 working up. I/O devices are started at location 4000 starting up and ROM starts at location FFFF and works down. Although this seems like an unusual concept, one must remember that the hardware really only gives performance to either end of memory and, therefore, data located in the middle has no priority one over the other. So starting at either end is just as useful a technique as starting at one end and working up.

In order to take maximum advantage of the capability of the microprocessor, particularly when using a symbolic assembler, working data should be located starting in the location 0, and stack addresses should be reserved until after analysis of the working storage requirements have been completed. Program storage should start in high order memory with some guess as to the amount of memory required being taken and that being taken as a start address. However, care should be taken to assign the three fixed vectors almost immediately at least symbolically as they are all necessary for correct operation of the microprocessor.

CHAPTER 10

SHIFT AND MEMORY MODIFY INSTRUCTIONS

10.0 DEFINITION OF SHIFT AND ROTATE

In many cases operations of the control systems must operate a bit at a time. Data is often available only bit-serial and sometimes sequential bit operations are the only way to solve a particular problem. In addition to that, in order to combine bits into a field, shift and rotate instructions are necessary. Multiply and divide routines all require the ability to move bits relative to one another in a full multiple byte field.

The shift instruction is one that takes a register such as the accumulator and moves all of the bits in the accumulator 1 bit to the right or 1 bit to the left. Examples of the shift and rotate instructions in the MCS650X are shown below:

Example 10.1: General shift and rotate

Shift Right		Before	B7	B6	B5	B4	B3	B2	B1	B0	Carry	C	BO							
		After	0	B7	B6	B5	B4	B3	B2	B1										
Shift Left		Before	<table border="1"><tr><td>B7</td><td>B6</td><td>B5</td><td>B4</td><td>B3</td><td>B2</td><td>B1</td><td>B0</td></tr></table>								B7	B6	B5	B4	B3	B2	B1	B0	C	BO
B7	B6	B5	B4	B3	B2	B1	B0													
		After	<table border="1"><tr><td>B6</td><td>B5</td><td>B4</td><td>B3</td><td>B2</td><td>B1</td><td>B0</td><td>0</td></tr></table>								B6	B5	B4	B3	B2	B1	B0	0	B7	
B6	B5	B4	B3	B2	B1	B0	0													
Rotate Left		Before	<table border="1"><tr><td>B7</td><td>B6</td><td>B5</td><td>B4</td><td>B3</td><td>B2</td><td>B1</td><td>B0</td></tr></table>								B7	B6	B5	B4	B3	B2	B1	B0	C	
B7	B6	B5	B4	B3	B2	B1	B0													
		After	<table border="1"><tr><td>B6</td><td>B5</td><td>B4</td><td>B3</td><td>B2</td><td>B1</td><td>B0</td><td>C</td></tr></table>								B6	B5	B4	B3	B2	B1	B0	C	B7	
B6	B5	B4	B3	B2	B1	B0	C													

As you can see from our example, moving data 1 bit to the right is called shift right. The natural consequence of the shift right is that the input bit or high order bit in this case is set to 0. Moving the data in the register 1 bit to the left is called shift left. In this case, the 0 is inserted in the low order position. These are the 2 shift capabilities that exist in the MCS650X microprocessor.

It should be noted that in both cases, the bit that is shifted from the register, the low order bit in shift right, and the high order bit in shift left, is stored in the carry flag. This is to allow the programmer to test the bit by means of the carry branches that are available and also to allow the rotate capability to transfer bits in multiple precision shifts.

The second part of the multiple precision shift instruction is the rotate which is shown in Example 10.1, in which the value of the carry bit becomes the low order bit of the register, and the output bit from the shift is stored in carry.

10.1 LSR – LOGICAL SHIFT RIGHT

This instruction shifts either the accumulator or a specified memory location 1 bit to the right, with the higher bit of the result always being set to 0, and the low bit which is shifted out of the field being stored in the carry flag.

The symbolic notation for LSR is

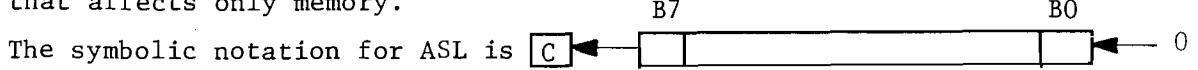


The shift right instruction either affects the accumulator by shifting it right 1 or is a read/modify/write instruction which changes a specified memory location but does not affect any internal registers. The shift right does not affect the overflow flag. The N flag is always reset. The Z flag is set if the result of the shift is 0 and reset otherwise. The carry is set equal to bit 0 of the input.

LSR is a read/write/modify instruction and has the following addressing modes: Accumulator; Zero Page; Zero Page,X; Absolute; Absolute,X.

10.2 ASL – ARITHMETIC SHIFT LEFT

The shift left instruction shifts either the accumulator or the address memory location 1 bit to the left, with the bit 0 always being set to 0 and the bit 7 output always being contained in the carry flag. ASL either shifts the accumulator left 1 bit or is a read/modify/write instruction that affects only memory.



The instruction does not affect the overflow bit, sets N equal to the result bit 7 (bit 6 in the input), sets Z flag if the result is equal to 0, otherwise resets Z and stores the input bit 7 in the carry flag.

ASL is a read/modify/write instruction and has the following addressing modes: Accumulator; Zero Page; Zero Page,X; Absolute; Absolute,X

10.3 ROL – ROTATE LEFT

The rotate left instruction shifts either the accumulator or addressed memory left 1 bit, with the input carry being stored in bit 0 and with the input bit 7 being stored in the carry flags.



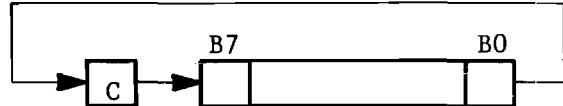
The ROL instruction either shifts the accumulator left 1 bit and stores the carry in accumulator bit 0 or does not affect the internal registers at all. The ROL instruction sets carry equal to the input bit 7, sets N equal to the input bit 6, sets the Z flag if the result of the rotate is 0, otherwise it resets Z and does not affect the overflow flag at all.

ROL is a read/modify/write instruction and it has the following addressing modes: Accumulator; Zero Page; Zero Page,X; Absolute; Absolute,X.

10.4 ROR - ROTATE RIGHT (Available on Microprocessors after June, 1976)

The rotate right instruction shifts either the accumulator or addressed memory right 1 bit with bit 0 shifted into the carry and carry shifted into bit 7.

The symbolic notation for ROR is



The ROR instruction either shifts the accumulator right 1 bit and stores the carry in accumulator bit 7 or does not affect the internal registers at all. The ROR instruction sets carry equal to input bit 0, sets N equal to the input carry and sets the Z flag if the result of the rotate is 0; otherwise it resets Z and does not affect the overflow flag at all.

ROR is a read/modify/write instruction and it has the following addressing modes: Accumulator; Zero Page; Absolute; Zero Page,X; Absolute,X.

10.5 ACCUMULATOR MODE ADDRESSING

As indicated, all of the shift instructions can operate on the accumulator. This is a special addressing mode that is unique to the shift instructions and operates with the following set of operations:

Example 10.2: Rotate accumulator left

Cycles	Address Bus	Data Bus	External Operation	Internal Operation
1	100	OP CODE	Fetch Next OP CODE	Finish Previous Operation; Increment PC to 101
2	101	Next OP CODE	Fetch Discarded OP CODE	Decode Current Instruction; Hold P-Counter
3	101	Next OP CODE	Fetch Next OP CODE	Shift Through the Adder
4	102	?	Fetch Second Byte	Store Results into A; Interpret Next OP CODE

As we can see, the accumulator instructions have the same effect as the single-byte non-stack instructions in the sense that the instruction contains both the OP CODE and the register in which the operations are going to be performed; therefore, in cycle 2, the microprocessor holds the pro-

gram counter and in cycle 3, fetches the same program counter location and starts the next instruction operation. At the same time, it is transferring the results from the adder into the accumulator; this is because of the look-ahead and pipelining characteristics of the MCS650X. . The accumulator shift and rotate operations take only 2 cycles and 1 byte of memory.

10.6 READ/MODIFY/WRITE INSTRUCTIONS

The MCS650X has a series of instructions which allow the user to change the contents of memory directly with a single instruction. These instructions include all of the shift, rotate, increment and decrement memory instructions. The operation of each of these instructions is the same in that the addressing mode that is defined for the instruction is implemented the same way as if for normal instructions. After the address has been calculated, the effective address is used to read the memory location into the microprocessor arithmetic unit (ALU). The ALU performs the operation and then the same effective address is used to write the results back into memory. The most difficult operation is the addressing mode Absolute Indexed which is illustrated in Example 10.3 for the rotate left instruction, ROL:

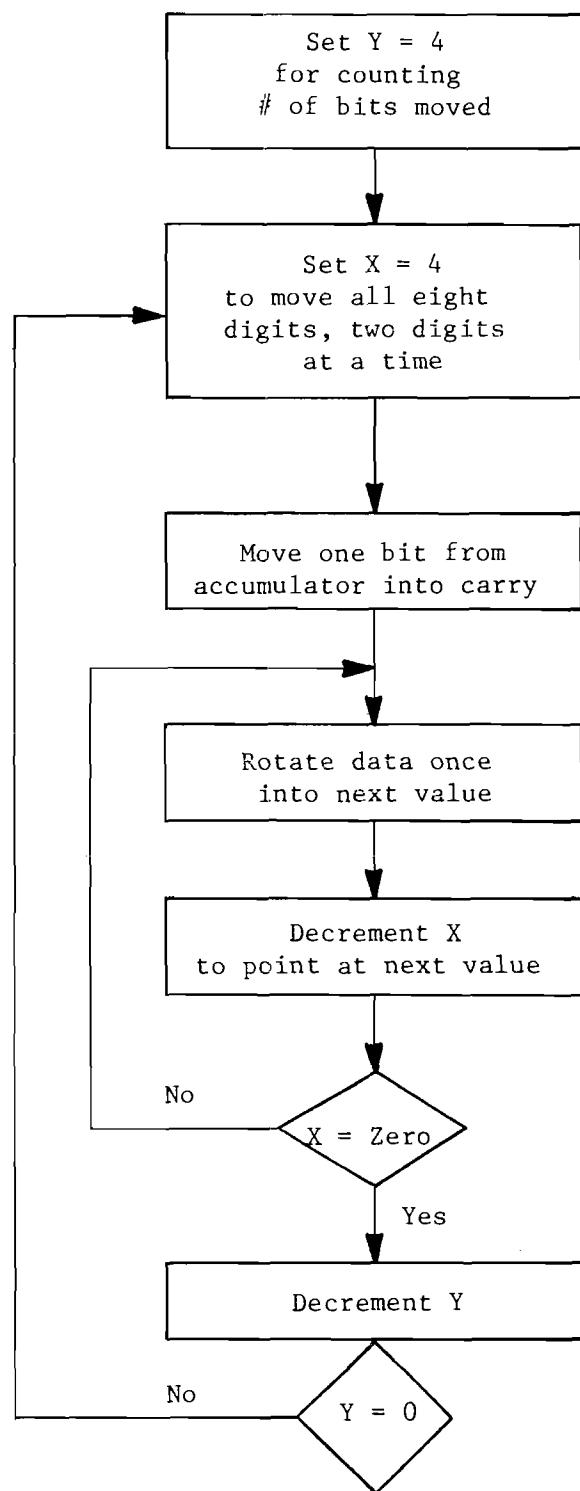
Example 10.3: Rotate memory left Absolute,X

<u>Cycles</u>	<u>Address Bus</u>	<u>Data Bus</u>	<u>External Operation</u>	<u>Internal Operation</u>
1	100	OP CODE	Fetch OP CODE	Finish Previous Operation, Increment PC to 101
2	101	ADL	Fetch ADL	Decode Current Instruction, Increment PC to 102
3	102	ADH	Fetch ADH	Add ADL + X, Increment PC to 103
4	ADH, ADL + X	?	False Read	Add Carry from Previous Add to ADH
5	ADH + C, ADL + X	Data	Fetch Value	
6	ADH + C, ADL + X	?	Destroy Memory	Perform Rotate, Turn on Write
7	ADH + C, ADL + X	Shifted Data	Store Results	Set Flags
8	103	OP CODE	Fetch Next OP CODE	Increment PC to 104

Cycle 4 is a wasted cycle because read/modify/write instruction should wait until the carry had been added to the address high in order to avoid writing a false memory location. This is the same logic that is used in the store instruction in which the look-ahead or the short cut addressing mode is not taken advantage of. Cycle 4 is an intermediate read, and cycle 5 is when the actual data that is going to be operated on is read.

The address lines now hold at that address for cycles 5, 6 and 7. The microprocessor signals both itself and the outside world those operations during which it will not recognize the ready line. It does this by pulling the Write line. The Write line is pulled in cycle 6 because data is written into the memory location that is going to be written into again in cycle 7 with correct data.

Because data bits read from memory have to be modified and returned, there is no pipelining effect other than the overlap of the adding in the address low and index register. The 7 cycles it takes to perform read/modify/write Absolute Indexed,X instruction is the worst case in timing for any section of the machine except for interrupt. This unique ability to modify memory directly is perhaps best illustrated by the coding in Example 10.4 which is used to shift a 4-bit BCD number, which has been accumulated in the high 4 bits of the accumulator as part of the decoding operation, from the accumulator into a memory field. Figure 10.1 is a flow chart of this example. Examples such as this often occur in point-of-sale terminals and other machines in which BCD data is entered sequentially. This example assumes that the value is keyboard entered, through which data is entered into the accumulator from left to right but has to be shifted into memory from right to left. The value in the field before the shift is a 1729 which after the shift will be a 17,295.



Flow Chart for Moving in a New BCD Number

FIGURE 10.1

Example 10.4: Move a new BCD number into field

	<u>Before</u>	<u>After</u>
Field	00 00 17 29	00 01 72 95
Accumulator	50	00

Coding

<u>Bytes</u>	<u>Instruction</u>	
2	LDY 4	
2	LOOP-2 LDX 4	set up for 4 moves
1	ASLA	
3	LOOP-1 ROL Price -1, X	
1	DEX	shift the field 1 bit
2	BNE LOOP-1	
1	DEY	shifts four times.
<u>2</u> <u>14 bytes</u>	BNE LOOP-2	

There are several new concepts introduced in this example; the first is the use of index register Y as just a counter to count the number of times the character has been bit-shifted. It is a common approach to use bit shifts, as is implemented in the MCS650X family, to shift data into memory. The power of being able to communicate directly in memory is shown by shifting bits from one byte to the next byte using a single ROL indexed instruction. This example uses a loop within a loop and it should be noted that LOOP 1 occurs 4 times for every time LOOP 2 occurs. The internal loop is very important in the sense that this loop executes 16 times for the problem; therefore, its execution time should be optimized.

In addition to having the ability to shift and rotate memory, the MCS650X has the ability to increment and decrement memory locations.

10.7 INC – INCREMENT MEMORY BY ONE

This instruction adds 1 to the contents of the addressed memory location.

The symbolic notation is $M + 1 \rightarrow M$.

The increment memory instruction does not affect any internal registers and does not affect the carry or overflow flags. If bit 7 is on as the result of the increment, N is set, otherwise it is reset; if the increment causes the result to become 0, the Z flag is set on, otherwise it is reset.

The addressing modes for increment are: Zero Page; Zero Page,X; Absolute; Absolute,X.

10.8 DEC – DECREMENT MEMORY BY ONE

This instruction subtracts 1, in two's complement, from the contents of the addressed memory location.

Symbolic notation for this instruction is $M - 1 \rightarrow M$.

The decrement instruction does not affect any internal register in the microprocessor. It does not affect the carry or overflow flags. If bit 7 is on as a result of the decrement, then the N flag is set, otherwise it is reset. If the result of the decrement is 0, the Z flag is set, otherwise it is reset.

The addressing modes for decrement are: Zero Page; Zero Page,X; Absolute; Absolute,X.

In many examples through the report, we have used the ability to increment and decrement registers in the microprocessors. The advantages of incrementing and decrementing in memory are that it is possible to keep external counters or to directly influence a bit value by means of these instructions. It is sometimes useful during I/O instructions.

10.9 GENERAL NOTE ON READ/MODIFY/WRITE INSTRUCTIONS

The ability to read, modify and write memory is unique to MCS6500 class microprocessors. The usefulness of the instructions is limited only by the user's approach to organizing memory. Even though the instructions are fairly long in execution, they are significantly shorter than having to load and save other registers to perform the same function. Experience in organizing programs to take advantage of this manipulation of memory will allow the user to fully appreciate the power of these instructions.

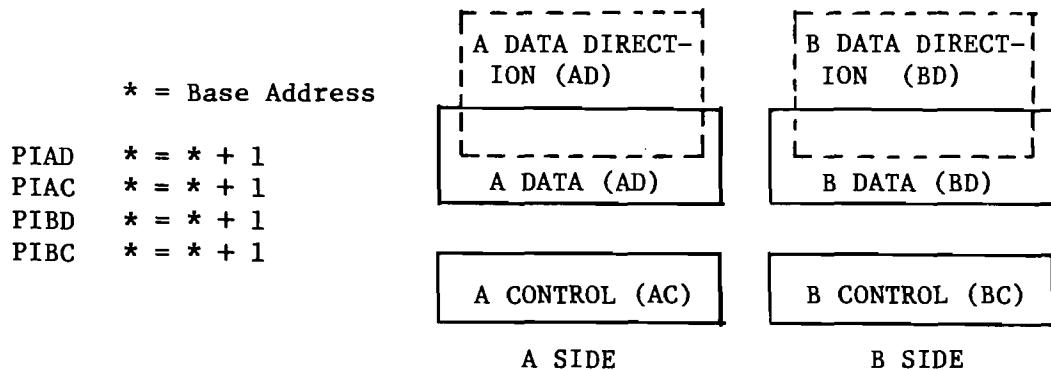
CHAPTER 11

PERIPHERAL PROGRAMMING

11.0 REVIEW OF MCS6520 FOR I/O OPERATIONS

It should be noted that in the following discussions, the major difference between the MCS6530 I/O and the main register of the MCS6520 is that the extra bit in the control register need not be used in the MCS6530. All registers in the MCS6530 are directly addressable.

Example 11.1: The MCS6520 Register Map



In Example 11.1 a programming form to describe the PIA is shown. The programming form is used in the Cross-Assembler and Resident Assembler with the MCS650X product family. The notation $*$ = is used to define any location. The notation means that the assembler instruction counter is set equal to the value following the equal sign. The expression $* = * + 1$ causes the assembler to recognize that there is one byte of memory associated with the term; therefore, we can see that the definition of the four registers PIAD, PIAC, PIBD and PIBC are consecutive memory locations starting at some base address, with

the first byte addressed as PIAD, the second byte addressed as PIAC, the third byte addressed as PIBD, and the fourth byte as PIBC. This is a normal way a MCS6520 would be organized and this is the way the programming form should be set up. The base address is picked up by an algorithm described in the hardware manual but normally it is a value between 4004 and 4080 Hex. Each MCS6520 is given a base address which works progressively up from 4004 Hex.

In Example 11.1 two registers are shown in dotted lines. This is because each of the A DATA (AD) and B DATA (BD) parts of the MCS6520 are actually two registers having the same address, one which specifies the direction of each of the input/output paths (the Data Direction Register), the second one which is actually the connection to the input/output paths (the Data Register). Because of pin limitations on the MCS6520, the microprocessor can only directly address one of the registers at a time. Differentiation as to which register is being connected to the microprocessor is a function of bit 2 in the respective control register (AC and BC). If bit 2 is off, the Data Direction Register is being addressed; if it is on, the Data Register is being addressed.

During the initialization sequence, therefore, the MCS6520 starts out with all registers at zero. This means that the microprocessor is addressing the Data Direction Register. The PIA initialization is done by writing the direction of the pins into the Data Direction Register (AD, BD) and then setting on the control flag as described below. After that, the program will normally be dealing with the data registers.

Example 11.2: General PIA Initialization

LDA # DIRECT STA PIAD	Initialize Direction
LDA # CONTR STA PIAC	Initialize Control

Example 11.2 illustrates a general form of initialization and can be completed for as many PIA's as there are in the system.

11.1 MCS6520 INTERRUPT CONTROL

The MCS6520 has a basic interrupt capability which is under control of the programmer. Almost all MCS6500 I/O devices that allow interrupts have an interrupt control register that allows the user to disable the interrupt. This will keep inputs which are not necessarily active from causing spurious interrupts which must be handled by the microprocessor. Examples of this are open tape loops or other signals which have high impedance noise sensitive inputs except when connected to some kind of media. In this type of application, normally the interrupt is enabled by some physical action from the person using the device such as loading of the cassette, pushing the power-on switch, etc. In the case of the MCS6520, there are two interrupt causing conditions for each control register.

Each of these interrupts concern themselves with one input pin. The Control Register allows the programmer to decide whether or not the pin is sensitive to positive edge signals or negative edge signals and whether or not an interrupt shall occur when the selected transition has occurred.

It should be noted that, therefore, it is possible for a line to cause a status bit to be set without causing an interrupt. The comprehensive I/O Program in Section 11.5 uses this combination.

Example 11.3: Interrupt Mode Setup

<u>Bit 7 Status Bit:</u>	<u>Bits</u>	<u>1</u>	<u>0</u>	<u>Interrupt</u>
Set on Negative Edge		0	0	No
Set on Negative Edge		0	1	Yes
Set on Positive Edge		1	0	No
Set on Positive Edge		1	1	Yes

<u>Bit 6 Status Bit:</u>	<u>Bits</u>	<u>4</u>	<u>3*</u>	<u>Interrupt</u>
Set on Negative Edge		0	0	No
Set on Negative Edge		0	1	Yes
Set on Positive Edge		1	0	No
Set on Positive Edge		1	1	Yes

*if bit 5 equals zero

The proper combination of bits are usually determined during the design of the MCS6520 interconnection and form the constant which is loaded in the control register. The constant that is loaded in the control register should contain bit 2 on. For example, to allow bit 7 to be set on negative going signals with interrupt enable and bit 6 to be set on positive signals with interrupt disable, the control value would be Hex 15.

With bit 5 on, the pin that controls bit 6 can be set as an output pin. The output pin is either controllable by the microprocessor directly or acts as a handshake to reflect the status of reads and writes of the data register. The operation of the output pins CA2, CB2 depends on how bits 5, 4, and 3 are programmed, as shown in Example 11.4.

Example 11.4: CA2, CB2 Output Control

<u>CA2 Output With:</u>	<u>Bit 5 on</u>	
	<u>Bit 4</u>	<u>Bit 3</u>
Low on read or write until bit 7 on	0	0
Low on read or write for one cycle	0	1
Always 0	1	0
Always 1	1	1

The decision as to whether or not to use the one cycle low until bit 7 comes on is a hardware decision, depending on the device which is hooked to the pin.

It should be of interest to the programmer to note that bit 6 controls pins known as CA2 or CB2 which can be considered to be auxiliary outputs which are controlled by bit 3 assuming the processor is initialized so that bit 5 and bit 4 are ones.

Example 11.5 shows the use of controlling bit 3 using AND and OR instructions; however, it should be noted that this technique applies for any individual bit in the PIA data direction register also:

Example 11.5: Routine to Change CA2 or CB2 Using Bit 3 Control

Set CA2

LDA	PIAC
ORA	#\$08
STA	PIAC

Clear CA2

LDA	PIAC
AND	#\$F7
STA	PIAC

Note: \$ - Direction to Assembler for Hex Notation
- Direction to Assembler for Production Operator

By similar techniques, every pin in the microprocessors of the MCS6520 can be controlled. There are two particular notes to remember:

1. In the MCS6520, both bit 6 and bit 7 are cleared on either side by reading of the corresponding data register if bit 6 has been set up as an input. This means that polling sequences for I/O instructions should only read the status registers and then read the data registers after the status has been determined, otherwise false clearing of the status data may occur.
2. Even though the handshake for the CB2 pin is on write of B data, a read of B data must be done to clear bit 7.

11.2 IMPLEMENTATION TRICKS FOR USE OF THE MCS6520 PERIPHERAL INTERFACE DEVICES

11.2.1 Shortcut Polling Sequences

In section 9.7, the techniques for using a LOAD A to poll for interrupts was covered; however, the I/O devices on the MCS6520 can either set bit 6 or bit 7 on to cause an interrupt; therefore, a different technique needs to be used to analyze the MCS6520 to poll a series of 6520's each one of which could have caused the interrupt. It is for this purpose that the BIT instruction senses both bit 6 and bit 7. Coding for a full poll of a PIA is as shown:

Example 11.6: Polling the MCS6520

Interrupt Vector	JMP STORE	
	LDA #CO	Set up Mask for 6 and 7
	BIT PIAC	Check for neither 6 or 7
	BEQ NXT1	
	BMI SEVEN	If 7, go to save-- otherwise clear
	Process BIT	
	6 INTERRUPT	
NXTI	BIT PIABC	
	BEQ NXTZ	
	etc.	

This program takes full advantage of the BIT instruction by checking for both bit 7 and 6 clear. BMI to SEVEN just checks N is on and that N is a higher priority. If bit 6 is one, the overflow bit will also be set, allowing the finish of the process seven routine to test the overflow and jump back to the process bit 6 coding.

Bit 6 and bit 7 were sampled by the single BIT instruction. Speed was accomplished by loading the mask for just bit 6 and 7 into the register which allows the BEQ instruction to determine that neither of the two flags is on.

This routine depends on the fact that in the MCS6520, if CA2 or CB2 is an output, bit 6 is always zero.

11.2.2 Bit Organization on MCS6520's

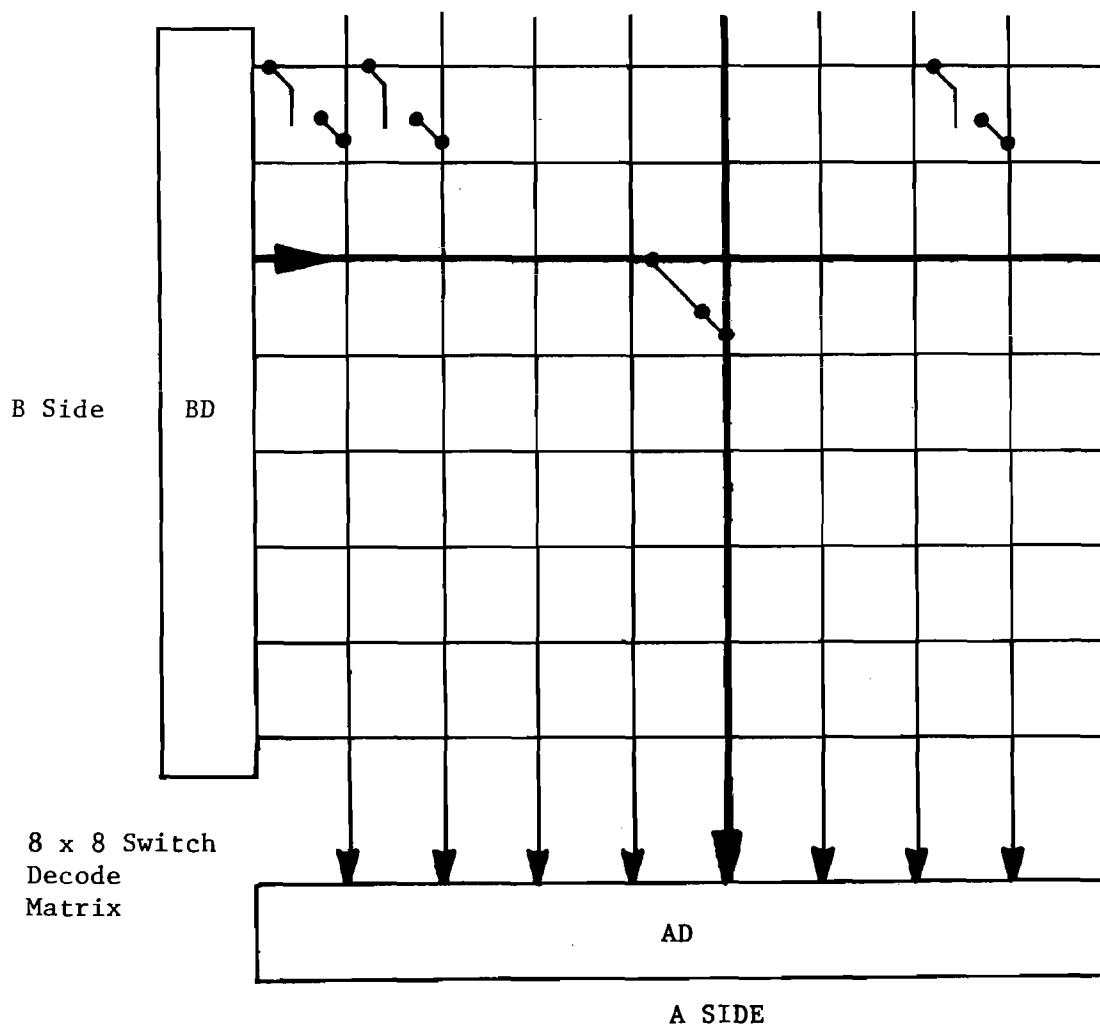
In the microprocessor, there is a definite positional preference for the testing of single bits. In the MCS6520 Data Direction Register, it is possible to select any combinations of input/output pins by the pattern that is loaded in the Data Direction Register. A one bit corresponds to an output and a zero bit corresponds to an input. The natural tendency would be to use MCS6520s with all eight bits organized into a byte. There is relatively little advantage to organizing this way unless the eight bits are to be treated as a single byte by the program. This is often not the case, more often the bits are a collection of switches, coils, lights, etc.

On such combinations, advantage should be taken of the fact that bit 7 is directly testable so that a more useful combination of eight pins on one MCS6520 register would be seven outputs and a single input with the single input on bit 7. This organization allows the programmer to load and branch on that location without ever having to perform a bit or shift instruction to isolate a particular bit.

A similar capability for setting a single bit involves the organization of data with seven inputs and a single output with a single output located in bit 0. This bit may be set or cleared by an INC or DEC instruction without affecting the rest of the bits in the register because the input pins ignore signals written from the microprocessor. Therefore, the more skilled MCS6500 programmer will often mix single outputs on bit 0 and a single input on bit 7 with bits of the corresponding opposite type.

11.2.3 Use of READ/MODIFY/WRITE Instruction For Keyboard Encoding

A rather unique use of the memory with a READ/MODIFY/WRITE operation involves setting the data register at all zeros, then using the three state output of the B side to sample a keyboard. The following Figure 11.1 shows the connection for a 64 key keyboard organized 8 x 8:



Keyboard Encoding Matrix Diagram

FIGURE 11.1

The B side is set up to act as a strobe so that each of the output lines will have a ground on it during one scan cycle. The eight A side data inputs are then sampled and decoded by the microprocessor giving a 64 key keyboard which is directly translatable into code.

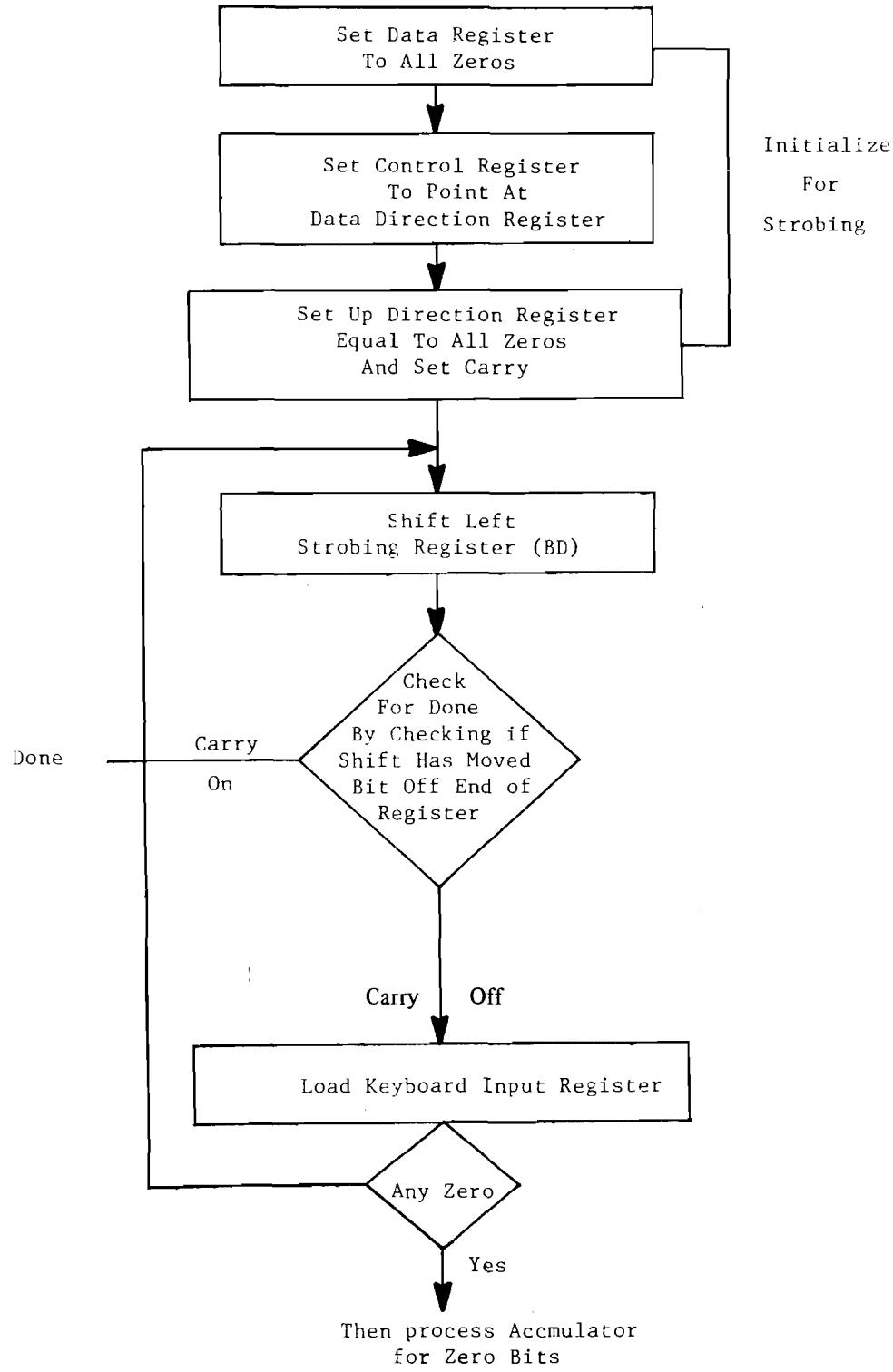
Figure 11.1 and Example 11.7 make use of the capability of the microprocessor to move a bit through the MCS6520 register location. This program also uses the compare instruction and the ability to detect a carry during a shift.

Example 11.7: Coding for Strobing an 8 x 8 Keyboard

Output Strobe is indicated by a one in Data Director Register. Any connection is indicated by a zero in register bit.

LDX #0	Initialize B Data Register
STX PIABD	
LDA PIABC	
AND #FB	Initialize Control Register to
STA PIABC	Address Data Direction Register
STX PIABC	
SEC Set low end bit on	
LOOP ROL PIABD	Shift for Strobe
BCS DONE	if all sampled, exit
LDA PIAAD	
CMP #FF	Check for no zeros
BEQ LOOP	
DONE -----	If any zeros, then process them.

A and PIABD can now be used to find out just what key is depressed.



Keyboard Strobe Sequence

FIGURE 11.2

11.3 MCS6530 PROGRAMMING

Although they have separate addressing, the Data Direction and Input/Output Registers operate the same as on the MCS6520.

Programming of the Interval Timer has some special problems. First of all, the time is effectively located in all addresses from XXX4-XXXF. By picking the proper address, the programmer is able to control the P scale for the timeout. Initialization of the Interval Timer is done by a LOAD A followed by STORE A into the timing count. The value stored in the timing counter represents the number of states which the counter will count through. The address used to load will determine how many additional divisions of the basic clock cycle will be counted.

When the counter finally counts to zero, it continues to count past zero at the one cycle clock rate in order to give the user an opportunity to sample the Status Register, then come back and read the Count Register to determine how long it has been since an interrupt occurred..

Servicing an interrupt is the same for this Control Register as for any other interrupting register. Bit 7 is set on in the Status Register to indicate that the Interval Timer is in the interrupt state and bit 7 is reset by the reading of the Counter.

11.3.1 Reading of the Counter Register

Because of the nature of counting past zero, the number in the Count Register is in two's complement form. It can be added directly to and used to correct the next count in a sequential string of counts or for correction for one cycle accuracy.

11.4 HOW TO ORGANIZE TO IMPLEMENT CODING

The specific details of organizing to get coding assembled is a function of the software that is used to implement the coding. Two software programs are currently available for the MCS650X family.

The Cross Assembler is available on various time share systems or for batch use on the user's system. Its documentation is covered in

the Cross-Assembler Manual, publication number 6500-60. The Resident Assembler is available in the Microcomputer Development Terminal, as well as for sale in ROMs. The documentation for this is covered in the Resident Assembler Manual, publication number 6500-65.

The major advantages of using an assembler are that the assembler takes mnemonics and labels and calculates the fixed code. Reference to the OP CODE tables in the appendix shows that coding in Hex is quite difficult because there is no ordered pattern to the instruction Hex codes.

The Cross Assembler or Resident Assembler allows one to specify all inputs and outputs in symbolic form on a documented listing. Symbolic addressing is a technique which has the following advantages over numerical addressing:

1. It allows the user to postpone until the last minute actual memory allocation in a program which is being developed. In a microprocessor that has memory-oriented features such as Zero Page, memory management is important. It is desirable to have as many as possible of the read/write values in the Zero Page. However, until the coding is complete, the organization of Zero Page may be in doubt. Values which are originally assigned in Zero Page may not be as valuable thereafter some analysis of the coding either indicates that the applications of these values use indirect references or indexing by Y which does not allow the program to really take advantage of Zero Page locations whereas some other code which may not be as frequently used might still result in a code reduction by use of Zero Page. This allocation, if all the fields are defined symbolically, can be done on the final assembly without any changing in the user's codes.
2. Use of symbolic addresses for programming branches leads to a better documented program and as one soon determines calculation of relative branches is difficult and subject

to change any time a coding change is made. For example, if one has organized a program with a loop in which three or four branches all return to the same point and then discovers a programming error which requires a single instruction to be added between the return point and various branches, each branch would have to be edited and recalculated. The symbolic assembler accomplishes this automatically on the next assembly pass.

11.4.1 Label Standards

The MCS650X assemblers have been done on a reserve word basis in which the various mnemonics which have been described are always considered to be OP CODE mnemonics. If any three character fields exactly match a mnemonic then the assembler assumes that the field is an OP CODE and proceeds to evaluate the addressing. Any other label may be located in free form anywhere in the coding. This means that one should organize one's labels such that he never has a three character label which inadvertently might be considered an OP CODE. The easiest way to accomplish this is to always follow a pattern on labels.

Good programming practice requires that the user develop a systems flow chart for his own basic program and individual flow charts for subroutines before starting the coding. From the time the routine is flow charted, it is very easy for the user to then assign a mnemonic label to the basic subroutine.

In this text, notations like LOOP, LOOP 1, etc. are used. In an ADD, loop would be ADLP.

The MCS650X assembler allows six spaces for labels. It is good practice to use two characters to generally identify the subroutine, two more characters for mnemonic purposes and then a numbering system which allows correlation between various addresses within a LOOP within a subroutine. By strictly numbering such that ADLP1 is different from ADLP3, each can be addresses within the same LOOP.

It is assumed that the PIA's are connected in the normal manner of Status Register Address equal to Data Register Address + 1.

The following table and flow chart defines the program implemented in the example.

Table #1 contains the address of all of the MCS6520 Status Registers.

Table #2 contains the address of the put-away location for the respective data.

Table #1

PIA #1

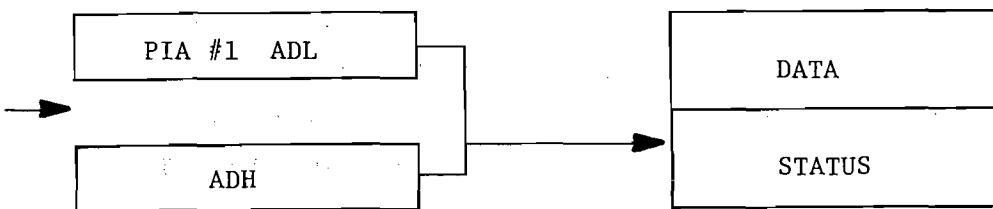
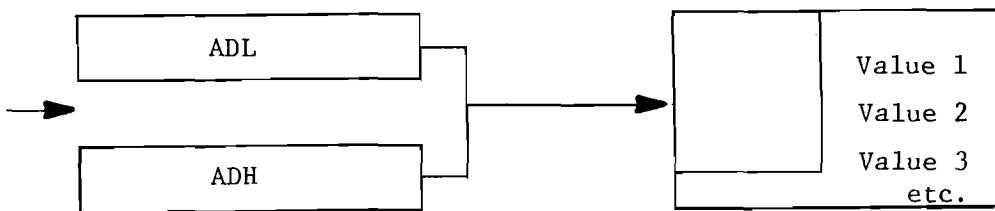


Table #2

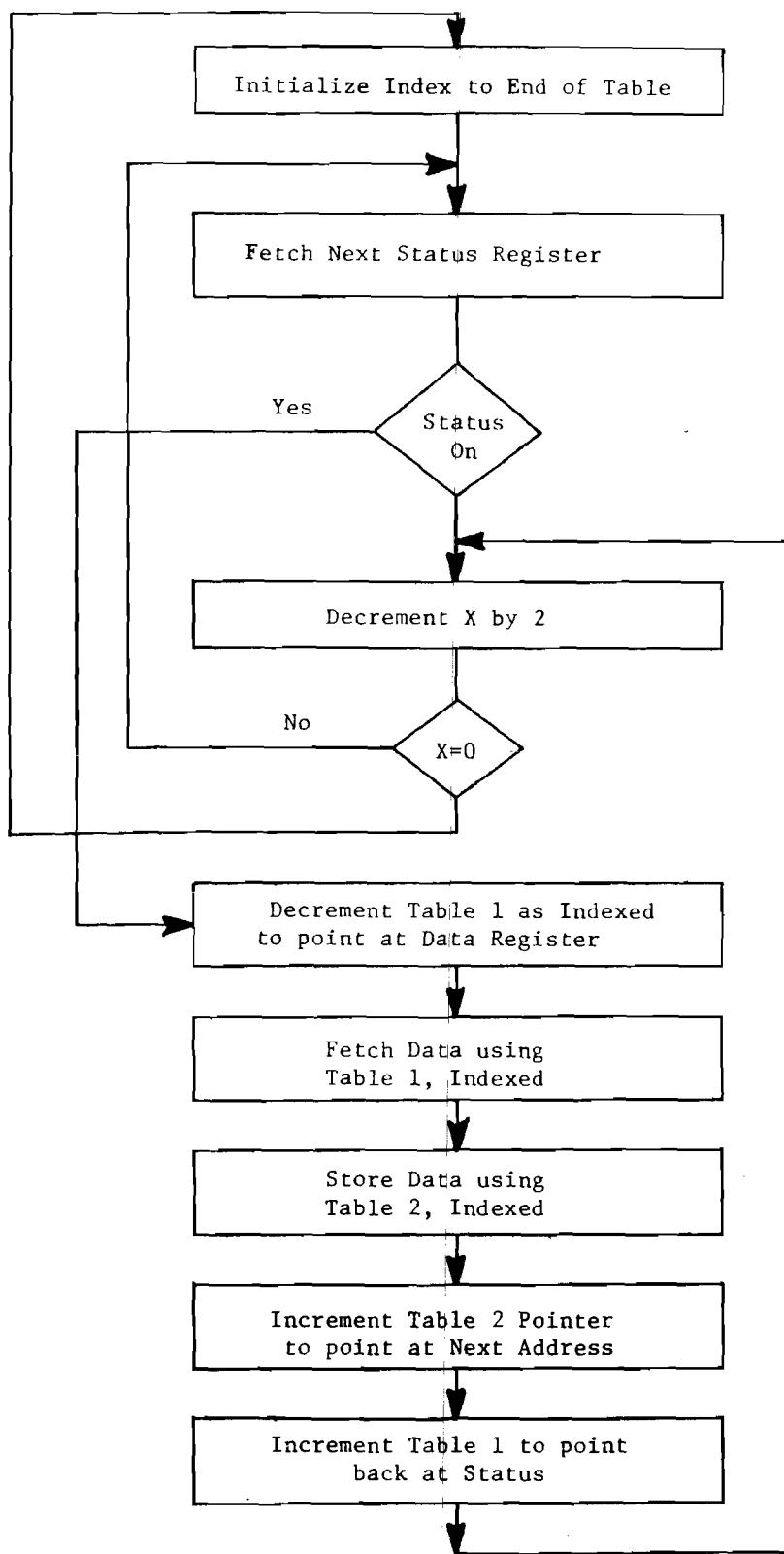
List



Using six character labels, there are a hundred combinations of code which could be used in a given routine or loop without the user having to think through the rest of mnemonic notation. The use of characters plus a numeric for all references is sound programming practice. The advantage of using this technique allows one to use three character mnemonics without ever interfering with the reserve word of the microprocessor OP CODE mnemonics because they never have a numeric in the mnemonic.

11.5 COMPREHENSIVE I/O PROGRAM

Figure 11.3 demonstrates the program flow in support of the Cross-Assembler listing (Example 11.9) of a time-sharing routine of a program which illustrates the use of the indexed indirect to perform a search of eight devices which have active signals for servicing. The implementation of the eight devices is done in MCS6520's where the MCS6520 status is set up to be a flag in bit 7 of a Control Register.



Program Flow - Polling for Active Signal

FIGURE 11.3

Example 11.8: Polling for Active Signal

CARD = LOC CODE CARD

3
4 CARD SERIAL NUMBER
5 AH SYSTEMS BENCHMARK =5 - POLLING 8 PERIPHERALS
6
7 MEMORY LOCATION
8
9 SET TABLES AND STORAGE AREAS
10
11 0000 *=\$02
12 0002 05 40 TABLE1 .WORD PIA1AC
13 0004 07 40 .WORD PIA1BC
14 0006 09 40 .WORD PIA2AC
15 0008 0B 40 .WORD PIA2BC
16 000A 11 40 .WORD PIA3AC
17 000C 13 40 .WORD PIA3BC
18 000E 21 40 .WORD PIA4AC
19 0010 23 40 .WORD PIA4BC
20 0012 00 02 TABLE2 .WORD STORE1
21 0014 50 02 .WORD STORE2
22 0016 A0 02 .WORD STORE3
23 0018 F0 02 .WORD STORE4
24 001A 40 03 .WORD STORE5
25 001C 90 03 .WORD STORE6
26 001E E0 03 .WORD STORE7
27 0020 30 04 .WORD STORE8
28
29 0022 *=\$200
30 0200 STORE1 *==+80
31 0250 STORE2 ==+80
32 02A0 STORE3 ==+80
33 02F0 STORE4 ==+80
34 0340 STORE5 ==+80
35 0390 STORE6 ==+80
36 03E0 STORE7 ==+80
37 0430 STORE8 ==+80
38
39
40 MAIN PROGRAM
41
42 0480 *=\$FC00 PROGRAM LOCATION

COMMENT
TABLE OF PIA PERIPHERAL CONTROL

OPCODE LABEL VALUE ADDRESS MNEMONIC SYMBOLIC ADDRESS

43 FC00 A2 10 PLOP1 LDX >016 INITIALIZE INDEX REGISTER X WITH 16
44 FC02 A1 00 PLOP2 LDA (TABLE1-2,X) INDIRECT ADDRESSING OF PERIPHERAL CONTROL
45 FC04 30 06 BMI DOT IF FLAG SET BRANCH AND SERVICE THE DEVICE
46 FC06 CA DEX IF NOT SEARCH NEXT ONE
47 FC07 CA DEX
48 FC08 DO F8 BNE (PLOP2) START AGAIN TO POLL FROM THE BEGINNING
49 FC0A FO F4 BEQ PLOP1 WHEN DONE START FROM BEGINNING AGAIN
50
51 SERVICE ROUTINE
52
53 FC0C D6 00 DOIT DEC TABLE1-2,X MOVE THE POINTER TO PIA DATA REGISTER
54 FC0D A1 00 LDA (TABLE1-2,X) READ DATA IN
55 FC10 B1 10 STA (TABLE2-2,X) STORE THE DATA INTO THE BUFFER
56 FC12 F6 10 INC TABLE2-2,X SET BUFFER POINTER TO NEXT LOCATION
57 FC14 F6 00 INC TABLE1-2,X
58 FC16 DO E8 BNE PLOP1 WHEN DONE START FROM BEGINNING AGAIN
59
60
61 ASSIGN PIA LOCATION
62
63 FC18 *=\$4004
64 4004 PIA1AD ==+1 FIRST PERIPHERAL
65 4005 PIA1AC ==+1
66 4006 PIA1BD ==+1 SECOND
67 4007 PIA1BC ==+1
68 4008 *=\$4008
69 4008 PIA2AD ==+1 THIRD
70 4009 PIA2AC ==+1
71 400A PIA2BD ==+1 FOURTH
72 400B PIA2BC ==+1
73 400C *=\$4010
74 4010 PIA3AD ==+1 FIFTH
75 4011 PIA3AC ==+1
76 4012 PIA3BD ==+1 SIXTH
77 4013 PIA3BC ==+1
78 4014 *=\$4020
79 4020 PIA4AD ==+1 SEVENTH
80 4021 PIA4AC ==+1
81 4022 PIA4BD ==+1 EIGHTH
82 4023 PIA4BC ==+1
83 .END END OF PROGRAM

APPENDIX A

INSTRUCTION LIST

ALPHABETIC BY MNEMONIC

DEFINITION OF

INSTRUCTION GROUPS

MCS6501-MCS6505 MICROPROCESSOR INSTRUCTION SET – ALPHABETIC SEQUENCE

ADC	Add Memory to Accumulator with Carry	JSR	Jump to New Location Saving Return Address
AND	"AND" Memory with Accumulator	LDA	Load Accumulator with Memory
ASL	Shift Left One Bit (Memory or Accumulator)	LDX	Load Index X with Memory
BCC	Branch on Carry Clear	LDY	Load Index Y with Memory
BCS	Branch on Carry Set	LSR	Shift Right One Bit (Memory or Accumulator)
BEQ	Branch on Result Zero	NOP	No Operation
BIT	Test Bits in Memory with Accumulator	ORA	"OR" Memory with Accumulator
BMI	Branch on Result Minus	PHA	Push Accumulator on Stack
BNE	Branch on Result not Zero	PHP	Push Processor Status on Stack
BPL	Branch on Result Plus	PLA	Pull Accumulator from Stack
BRK	Force Break	PLP	Pull Processor Status from Stack
BVC	Branch on Overflow Clear	ROL	Rotate One Bit Left (Memory or Accumulator)
BVS	Branch on Overflow Set	ROR	Rotate One Bit Right (Memory or Accumulator)
CLC	Clear Carry Flag	RTI	Return from Interrupt
CLD	Clear Decimal Mode	RTS	Return from Subroutine
CLI	Clear Interrupt Disable Bit	SBC	Subtract Memory from Accumulator with Borrow
CLV	Clear Overflow Flag	SEC	Set Carry Flag
CMP	Compare Memory and Accumulator	SED	Set Decimal Mode ^a
CPX	Compare Memory and Index X	SEI	Set Interrupt Disable Status
CPY	Compare Memory and Index Y	STA	Store Accumulator in Memory
DEC	Decrement Memory by One	STX	Store Index X in Memory
DEX	Decrement Index X by One	STY	Store Index Y in Memory
DEY	Decrement Index Y by One		
EOR	"Exclusive-Or" Memory with Accumulator		
INC	Increment Memory by One	TAX	Transfer Accumulator to Index X
INX	Increment Index X by One	TAY	Transfer Accumulator to Index Y
INY	Increment Index Y by One	TSX	Transfer Stack Pointer to Index X
JMP	Jump to New Location	TXA	Transfer Index X to Accumulator
		TXS	Transfer Index X to Stack Pointer
		TYA	Transfer Index Y to Accumulator

A.1 INTRODUCTION

The microprocessor instruction set is divided into three basic groups. The first group has the greatest addressing flexibility and consists of the most general purpose instructions such as Load, Add, Store, etc. The second group includes the Read, Modify, Write instructions such as Shift, Increment, Decrement and the Register X movement instructions. The third group contains all the remaining instructions, including all stack operations, the register Y, compares for X and Y and instructions which do not fit naturally into Group One or Group Two.

There are eight Group One instructions, eight Group Two instructions, and all of the 39 remaining instructions are Group Three instructions.

The three groups are obtained by organizing the OP CODE pattern to give maximum addressing flexibility (16 addressing combinations) to Group One, to give eight combinations to Group Two instructions and the Group Three instructions are basically individually decoded.

A.2 GROUP ONE INSTRUCTIONS

These instructions are: Add With Carry (ADC), (AND), Compare (CMP), Exclusive Or (EOR), Load A (LDA), Or (ORA), Subtract With Carry (SBC), and Store A (STA). Each of these instructions has a potential for 16 addressing modes. However, in the MCS6501 through MCS6505, only eight of the available modes have been used.

Addressing modes for Group One are: Immediate, Zero Page, Zero Page Indexed by X, Absolute, Absolute Indexed by X, Absolute Indexed by Y, Indexed Indirect, Indirect Indexed. The unused eight addressing modes are to be used in future versions of the MCS650X product family to allow addressing of additional on-chip registers, of on-chip I/O ports, and to allow two byte word processing.

A.3 GROUP TWO INSTRUCTIONS

Group Two instructions are primarily Read, Modify, Write instructions. There are really two subcategories within the Group Two instructions. The components of the first group are shift and rotate instructions and are: Shift Right (LSR), Shift Left (ASL), Rotate Left (ROL), and Rotate Right (ROR).

The second subgroup includes the Increment (INC) and Decrement (DEC) instructions and the two index register X instructions, Load X (LDX) and Store X (STX). These instructions would normally have eight addressing modes available to them because of the bit pattern. However, to allow for upward expansion, only the following addressing modes have been defined: Zero Page, Zero Page Indexed by X, Absolute, Absolute Indexed by X, and a special Accumulator (or Register) mode. The four shift instructions all have register A operations; the incremented or decremented Load X and Store X instructions also have accumulator modes although the Increment and Decrement Accumulator has been reserved for other purposes. Load X from A has been assigned its own mnemonic, TAX. Also included in this group are the special functions of Decrement X which is one of the special cases of Store X. Included also in this group in the X decodes are the TXS and TSX instructions.

All Group One instructions have all addressing modes available to each instruction. In the case of Group Two instructions, another addressing mode has been added; that of the accumulator and the other special decodes have also been implemented in this basic group. However, the primary function of Group Two instructions is to perform some memory operation using the appropriate index.

It should be noted for documentation purposes that the X instructions have a special mode of addressing in which register Y is used for all indexing operations; thus, instead of Zero Page Indexed by X, X instructions have Zero Page Indexed by Y, and instead of having Absolute Indexed by X, X instructions have Absolute Indexed by Y.