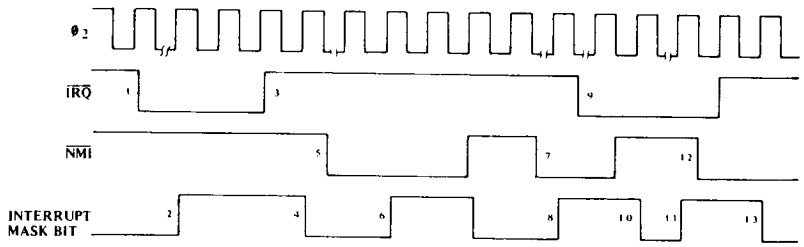


Examples of Interrupt Recognition by R6500

FIGURE 2-4

Each major event affecting the microprocessor is numbered in Figure 2-4, and the correspondingly numbered explanations for each are given below.

Event Number	System Activity
1.	Processor is executing from main program and $\overline{IRQ}$ goes to low state.
2.	Upon completion of current instruction, the processor recognizes the interrupt, stores the contents of PC and P onto the stack and then sets I during the fetch of the interrupt vector.
3.	After servicing the interrupt, $\overline{IRQ}$ should be reset before resetting the interrupt mask bit to avoid double interrupting.
4.	Before the processor resumes normal main program execution the interrupt mask bit will be reset low.
5.	$\overline{NMI}$ now goes low, signalling a non-maskable interrupt request.
6.	The $\overline{NMI}$ interrupt is recognized and serviced in the same manner as $\overline{IRQ}$.

Event Number	System Activity
7.	The processor has resumed normal operation when $\overline{NMI}$ again goes low requesting an interrupt.
8.	The interrupt mask bit is set high in response to the $\overline{NMI}$ request.
9.	Here $\overline{IRQ}$ has gone low to signal an interrupt request. This request is ignored since the $\overline{NMI}$ interrupt is being serviced and the interrupt mask is set.
10.	The interrupt mask bit is reset after servicing the $\overline{NMI}$ interrupt.
11.	The processor is now able to recognize the $\overline{IRQ}$ signal, which is still low, and does so by setting the interrupt mask bit.
12.	During the servicing of $\overline{IRQ}$, $\overline{NMI}$ goes from high to low. The processor then completes the current instruction and abandons the $\overline{IRQ}$ interrupt to service $\overline{NMI}$. $\overline{NMI}$ is serviced regardless of the state of the interrupt mask bit.
13.	After completing the $\overline{NMI}$ interrupt routine, the processor will resume execution of the $\overline{IRQ}$ routine, even though $\overline{IRQ}$ has subsequently gone high.

2.2.8 Reset (RES)

The $\overline{RES}$ line is used to initialize the microprocessor from a power-down condition. During the power-up time this line is held low, and writing from the microprocessor is inhibited. When the line goes high, the microprocessor will delay 6 cycles and then fetch the new program count vectors from specific locations in memory (PCL from location FFFC and PCH from location FFFD). This is the start of the user's code. It should be assumed that any time the reset line has been pulled low and then high, the internal states of the machine are unknown and all registers must be re-initialized during the restart sequence. Timing for the reset sequence is shown in Figure 2-3.

2.2.9 Synchronization Signal (SYNC)

In the R6502, a SYNC signal is provided to identify those cycles in which the processor is doing an OP CODE fetch. The SYNC line goes high

during Phase 1 of an OP CODE fetch and stays high for the remainder of that cycle. If the RDY line is pulled low during the Phase 1 clock pulse in which the SYNC line went high, the processor wi-l stop in its current state. It remains in that state until the RDY line goes high. In this manner, the SYNC signal can be used to control RDY to cause single-instruction execution. This application is discussed in detail in Section 3. Figure 2-5 contains a timing diagram for this signal.

2.2.10 Set Overflow (S.O.)

This pin sets the overflow flag on a negative transition from TTL one to TTL zero. This is designed to work with a future I/O part and should not be used in normal applications unless the user has programmed for the fact the arithmetic operations also affect the overflow flag.

2.2.11 Power Lines (V_{CC}, V_{SS})

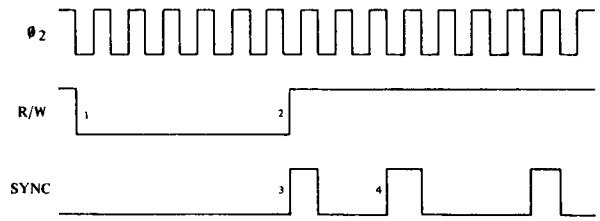
The V_{CC} and V_{SS} pins are the only power supply connections to the chip. The supply voltage is +5.0 V DC ± 5%. The absolute limit on the V_{CC} input is +7.0 V DC.

2.3 DEVICE TIMING– REQUIREMENTS AND GENERATION

The R6512 through R6515, requires a 5-volt, two-phase clock. The R6502 through R6507, however, can be used with an externally generated time base consisting of either a TTL-level single-phase clock, crystal oscillator, or RC network.

Figure 2-6 and 2-7 show the configuration for setting the frequency of oscillations with a crystal or with an RC network.

Figure 2-6 displays the crystal mode of operation in which the frequency of oscillation is set by the crystal operating in conjunction with the RC network. Figure 2-7 displays the same interconnects as in the crystal mode of time-base generation, with the crystal removed from the circuit. Values of the feedback resistor, R_F, and feedback capacitor, C_F, will be different for the crystal mode versus the RC mode. While the detail specifications for values of R_F and C_F are found in the data sheet for the R6502, clock timing can be generated by use of combinations of R_F in the range of 0 to 500K ohms and C_F in the range of 2 to 12 pf. The



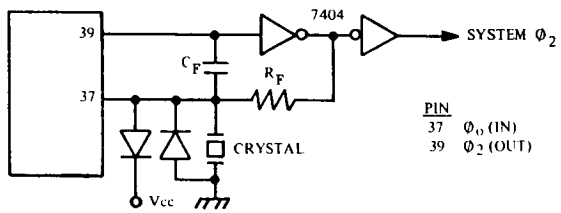
1. During a microprocessor write cycle, R/W signal low, the SYNC pulse does not occur.

2. The R/W signal goes high to signal the beginning of a microprocessor read cycle.

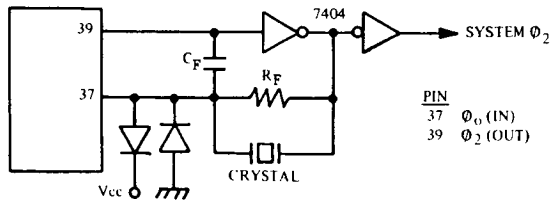
3. At the beginning of the read cycle a SYNC pulse will be be generated. This pulse will last for one cycle time. The SYNC pulse indicates that the microprocessor is reading an OP CODE from the memory field. In this case the SYNC pulse is high for one cycle as the processor reads the OP CODE.

4. The processor outputs another SYNC pulse indicating it has completed the previous instruction and is fetching another OP CODE. In this case three more cycles are needed to complete this instruction before the next SYNC pulse is generated. The SYNC pulse is aperiodic in that its generation is a function of the program and the resultant lengths of the instructions and addressing modes.

R6502 SYNC Signal
FIGURE 2-5

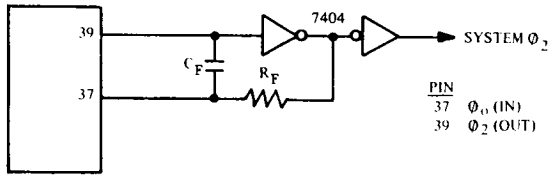


R6502 Parallel Mode Crystal-Controlled Oscillator



R6502 Series Mode Crystal-Controlled Oscillator

R6502 Time-Base Generation (Crystal-Controlled)
FIGURE 2-6



R6502 Time-Base Generator (RC Network)
FIGURE 2-7

reader is referred to the R6502 data sheet for a detailed description of the application of RC networks and crystal oscillators for generation of the time base in these modes of operation.

The R6500 bus discipline described in Section 1.2.2 is applicable wherever the oscillator is located. For data transfers to be properly carried out between the processor and the various support chips in the systems, the timing of the clocks controlling the internal processor operations must be very close to that of the Phase 2 clock out of pin 39 of the processor with no more than two TTL delays for clock buffering. It is important in systems which drive the clock generators with a TTL square wave that this input waveform not be employed to control the peripheral chips, unless care is taken to assure proper timing of the Phase 2 clock being used in these support chips.

SECTION 3

CONFIGURING THE MICROCOMPUTER SYSTEM

The first part of any microprocessor-based design effort is the microcomputer system configuration task. In fact, this probably requires more creativity from the designer than any other part of the design effort. The goal of the system configuration effort is the generation of a list of components which will make up the system, a detailed interconnect diagram, and a detailed description of the total system operation. This includes a definition of how the processor will control the peripheral devices as well as a definition of the internal operations to be performed. This does not include detailed implementation of the design such as laying out printed-circuit boards and writing programs, but does involve enough analysis of the total operation to ensure that the system will operate properly after all of the hardware and software has been assembled.

The technically based selection of components and the definition of the general operation of the system must be based on consideration of two factors:

1. System speed requirements
2. System input/output requirements

Both of these factors are interrelated. Accordingly, it will usually be necessary to define an I/O configuration, and then to verify that the processor can operate at the speed required by the peripheral devices. If there appears to be any difficulty with the I/O operation, the structure must be redefined and reanalyzed.

In addition to the speed requirements of the I/O devices, there are also general speed requirements for the internal processor operations (arithmetic operations, data manipulation, etc.). This speed requirement is usually somewhat more flexible than that associated with I/O but it should be defined along with any other system requirements. The ultimate test of system speed must wait for the generation of both the hardware and the program; however, the system requirements and capability must be analyzed very early in the system development process to ensure that no problems will arise during the last stages of the design.

3.1 INPUT/OUTPUT TECHNIQUES

3.1.1 The General-Purpose Input/Output (I/O) Port

Although the concept of the I/O port was introduced briefly in Section 1, and the operation of two R6500 system devices which provide general-purpose I/O capability has been discussed in Sections 1.5 and 1.6, little has been said about what factors must be considered when configuring I/O structure using these devices.

The general-purpose I/O port consists of eight lines, each of which can act as either an input or an output. As an input, each line can detect the state of one switch or can detect one bit of data. As an output, each line can control one light, solenoid, etc. or can provide one bit of data to a peripheral device. If this technique is used in peripheral control, the operation of each line is totally defined in the system program.

For most systems, the general-purpose interface device provides more than adequate speed and flexibility to solve the entire peripheral interface problem. Usually, cost savings can be realized because of the reduced component cost and the necessity to stock only one type of interface device. In addition, use of the general-purpose peripheral interface device allows the designer to tailor the operation of the interface device to fit the problem at hand.

The ultimate component selection must be preceded by a study of each section of the system I/O structure and a study of the overall system performance. Ultimately, the set of general-purpose and special-purpose peripheral interface devices selected for a system must be chosen to minimize total cost, while ensuring satisfactory system performance.

Processor speed is a function of two factors: (1) the number of instructions required to perform the desired operations, and (2) the percentage of processor time required to service interrupts. The typical microcomputer system may employ several interrupt signals which occur at fixed intervals. At times, these may be combined with other interrupts being generated by a peripheral device. It is important that the total service time for these interrupts does not exceed that which is allowable, and that the time available to the processor for executing the main program is sufficient to allow the system to operate at its required speed.

During the system configuration process, detailed system programs need not be generated. However, it will be necessary to write small portions of the software to verify the speed of execution and to ensure proper operation of the total system.

This chapter will discuss special techniques for the control of the various components which may be included in a microcomputer system, as well as techniques for controlling peripheral devices which are attached to the system. A discussion of programming techniques which can be used to optimize the total system performance is contained in the Programming Manual.

3.1.2 The Special-Purpose Peripheral Interface Device

The special-purpose, dedicated I/O device must also be considered in any microcomputer design. These devices are designed to completely handle a single well-defined problem -- for example, driving a particular printer, handling a particular type of communications line or driving a scanned display. These special-purpose devices are designed to totally handle their particular tasks with very little help from the processor.

The primary advantage of this type of interface device is that it requires an absolute minimum amount of attention from the processor. The major disadvantage of special purpose I/O is increased component cost. The total production volume for these devices is less than that of the more universal I/O chips and also the total chip size is usually greater.

The use of special-purpose peripheral control devices will not be discussed in this manual. Instead, a detailed study will be made of the more general problem of configuring the 8-bit bidirectional peripheral port. In addition, this chapter will cover some special techniques which can greatly enhance the power of this type of interface device.

3.1.3 Configuring the General-Purpose I/O Port

The 8-bit peripheral control port included on the R6520 and the R6530 permits each line to be programmed to act as an input or an output. This is accomplished when the processor writes a pattern of 1's and 0's into the data direction register. Writing a 1 causes the pin to become an output, and writing a 0 causes it to act as an input. Although this operation is normally performed only during system initialization, the ability to do so under program control allows some very important peripheral control techniques. An example of this is described below.

The process of configuring the general-purpose I/O port involves, first, examining the peripheral devices to analyze the various control inputs, switches, sensors, data signals, etc. which must be handled by the microprocessor to properly control the device. Each function must then be assigned to a line on the I/O port. The ultimate goal of this process is the creation of a list of I/O pins, the function of each pin, and an indication of whether each pin is to be an input or an output.

Since each line is capable of operating as an input or an output, and since there is very little to differentiate one line from any other, the actual assignment can be made fairly late in the system development cycle after consideration of software techniques and printed-circuit board layout. In fact, software considerations may be the only thing which dictates that a signal be connected to one pin or another.

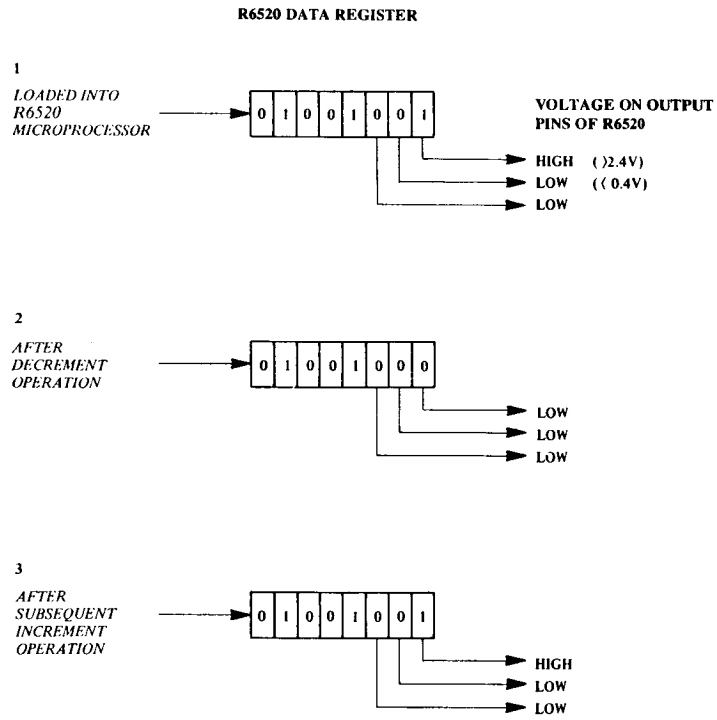
Developing a thorough understanding of the software in the R6500 systems will require a detail study of the Programming Manual. However, several operations which can be performed by the processor and which affect the assignment of inputs and outputs will be discussed briefly here.

ASSIGNMENT OF OUTPUTS

A major factor in the assignment of output pins can be the ability of the R650X processor to increment and decrement memory. Since the I/O port is treated as a location in memory, this incrementing and decrementing can be used to rapidly set and clear the low-order bit in this memory location. This is illustrated in Figure 3-1.

Note that this does not affect anything but the low-order bit if it is used properly as shown. This operation can be performed more rapidly than several other software techniques which can be employed to affect a single bit. Therefore, control of a single indicator, data line, etc. can be greatly enhanced by putting it on the low-order bit of an I/O port. This is the reason the low-order bit of both the R6530 peripheral ports (PA and PB0) provide the ability to drive transistors directly. In many applications, a simple transistor attached to one of those pins would provide very convenient control of a motor, lamp, etc.

The ability of the microprocessor to shift data in memory can be another very important factor in the assignment of outputs. Operations



Control of Low Order Bit of R6520 Output Register

FIGURE 3-1

which require sequential strobe signals can be controlled conveniently by shifting a single high (or low) signal from pin to pin under software control. The specific choice of pins can greatly enhance the ease with which this signal is controlled.

ASSIGNMENT OF INPUTS

In general, the processor deals with the input data from switches, keyboards, etc. by reading the data on the I/O port into the internal registers of the processor (usually the accumulator) and using the resulting condition of flags in the Processor Status Register to control the program which is executed. During this transfer process, the N flag in the Processor Status Register is set equal to the high-order bit (bit 7) of the word read from the I/O port. This N flag can then be used to cause the processor to execute different sections of the program (See the Programming Manual, Chapter 4, for a detailed discussion of Branching). Likewise, by performing certain instructions, the V flag in the Processor Status Register can be set equal to bit 6 on the I/O port. This flag can then be utilized to affect the program which is executed.

This operation of setting the internal flags from bits 6 and 7 of the memory word means that making these two lines inputs on an I/O port will permit very convenient testing of the condition of the switches, sensors, etc. attached to these inputs. If more than two input signals are to be attached to a port, the additional inputs should be placed on bit 5, then bit 4, and so on. The processor can then perform operations which shift the lower-order bits into bit 7 one at a time and sets the N flag equal to this bit. After each shift the N flag can be used to determine the actual program which is to be executed. (See the Programming Manual for a discussion of the Shift instructions.)

From the above example, one should conclude that the assignments which the designer makes will be very much a function of the software techniques which will be employed in controlling each line. It is very important that the designer be familiar with these techniques, and that he document the techniques which he has in mind when making the assignments. This is particularly important when the system program is to be written by someone else. Also, it is important that those persons doing the system development

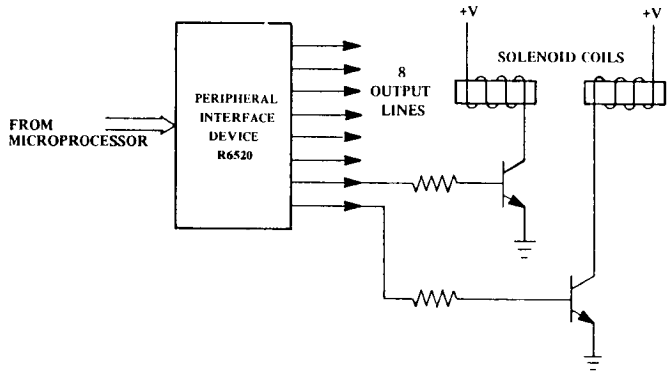
work constantly review the I/O structure to optimize the software involved as the system program is written.

3.1.4 Power-On Considerations

Section 1.2.4 discusses the operation of the system RESET function. Reference is made to the fact that this can be used to assure that all I/O lines come up in a known state when power is applied to the chip. Although this is a very important function, the designer must assure himself that this RESET state does not adversely affect the peripheral devices. This section describes some of the problems which can be encountered when the system is reset and discusses several techniques which can be used to guarantee smooth power-up operation.

The I/O lines of the R6530 and R6520 all enter the input state when the reset line goes to GND (< 0.4V). For the R6530 I/O lines, and for the Peripheral A port on the R6520, these pins will go to +5V DC (Vdd). This is due to the output structure on these pins. When these lines are in the input state, the output switch becomes an open circuit but the pull-up device continues to supply current to the pin.

Figure 3-2 shows a peripheral port which is configured to drive two solenoids. These solenoids can be controlled properly after the system is initialized; however, when the manual reset switch is activated, both I/O lines enter the input state, the transistors saturate (close) and the solenoids are activated. This can be catastrophic in most mechanical subsystems, so it is important that this potential condition be understood and prevented. Figure 3-3 shows two satisfactory solutions to this problem. The first, Figure 3-3a, requires that a "0" be written into the output line by the processor to actuate the solenoids. This ensures that the solenoids will not be powered simultaneously when the manual reset switch is pressed; however, it does introduce another potential problem. When the reset line on the peripheral interface device goes low (< 0.4V), the contents of both the Peripheral Data Register and the Data Direction register are cleared to zeros. If the Data Direction Register is set to 1's, both solenoids will immediately actuate due to the 0 stored in the Peripheral Data Register. This can be avoided completely if the system software first sets the bits



R6520 Control of Transistor Driven Solenoids
FIGURE 3-2

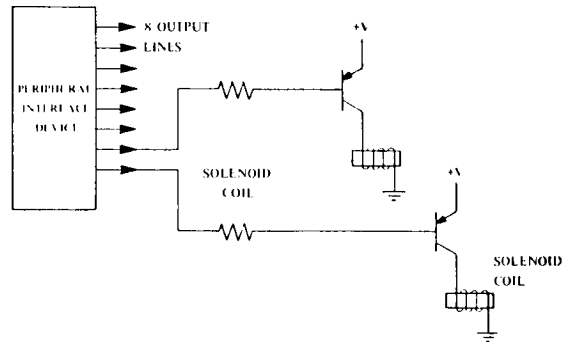
in the Peripheral Data Register to a 1 and then sets the Data Direction Register to a 1. The I/O pin will go high when the reset switch is actuated and will simply stay high through the initialization routine.

Figure 3-3b illustrates a solution which may be more applicable to a large system or a complex peripheral. In this approach, a separate output line is used to apply power to the peripheral device. The power to the entire peripheral, or to only the critical elements, is kept "off" until the entire system is initialized and is ready to run the system program.

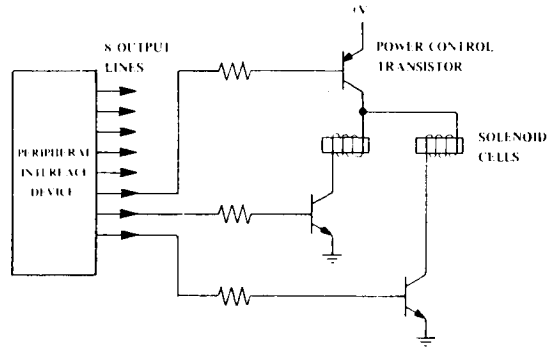
On the R6520 Peripheral B port, the I/O lines are open-circuit (high-impedance) in the input state. As a result, the configuration in Figure 3-2 will not cause the same problem on the R6520 Peripheral B port as would be expected on the R6530. In the input state, the I/O pin is incapable of sourcing any more than a few microamps.

However, if one were to use a solenoid driver as shown in Figure 3-4, the TTL input structure on the drivers would interpret the high-impedance state as a logic 1 and would actuate the solenoids; both the solutions in Figure 3-3 would be satisfactory in this case. However, the transistors are connected to the TTL buffer. In addition, the extra output shown in Figure 3-3b, controlling power to the peripheral device, could actually be utilized to enable the solenoid drivers if an enable input is available to these devices. This configuration is illustrated in Figure 2.5.

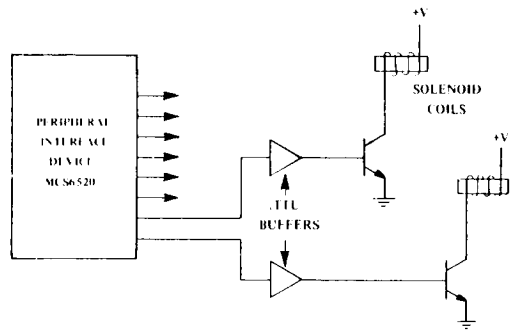
R6520 Control of PNP Transistor Driving Solenoid Coil
FIGURE 3-3a

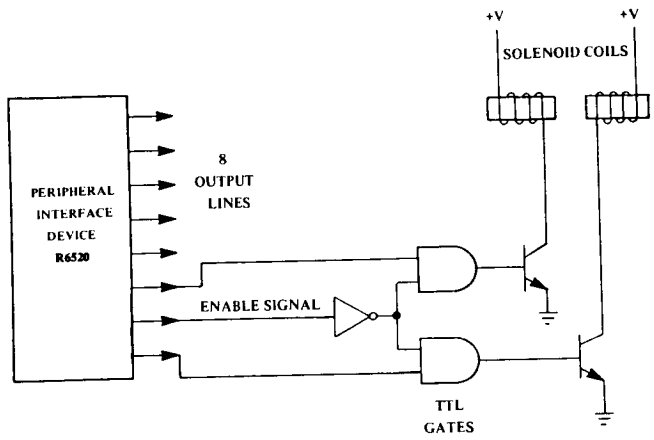


R6520 Controlling Both Power and Drivers of Solenoid Cell
FIGURE 3-3b



R6520 Driving TTL Buffers
FIGURE 3-4





R6520 Controlling Solenoids with Enable Signal and TTL Interface
FIGURE 3-5

3.1.5 Handshaking

The R6520 provides both interrupt control and data transfer control capability. The technique for controlling the transfer of data between the processor and a peripheral device is referred to as "handshaking." In this procedure, each device (the processor or peripheral) is capable of signalling the other that its operation is complete. The sequence differs somewhat for transfers into or out of the processor, so they will be discussed separately below.

HANDSHAKING ON DATA TRANSFERS FROM THE PROCESSOR

The transfer of data out of the processor into a peripheral device is performed by first writing the data into the data register within the R6520. These data then appear on the peripheral output lines where they can be read by the peripheral device for storage, display, etc.

Control of this data-transfer by handshaking requires first that the processor signal the peripheral device that data are available on the I/O port. The peripheral device then reads these data and signals to the processor that the data have been taken and that new data can be made available. The processor then makes new data available and the cycle is repeated.

As described in Section 1, the Peripheral B Interface Port on the R6520 is designed to perform handshaking on WRITE operations. The CB2 peripheral control line can be programmed to act as an output which goes low each time the processor writes data onto the Peripheral B I/O port. This is the signal which tells the peripheral device that data is available on these output lines.

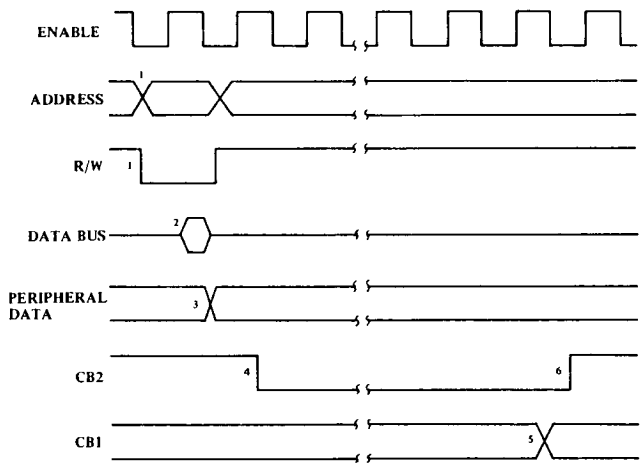
The CB2 output line will stay low until the peripheral device signals the processor that the data is taken. This is accomplished by interrupting the processor through the CB1 interrupt input.

The sequence which takes place during the "WRITE" handshaking operation described above is shown in Figure 3-6.

HANDSHAKING ON DATA TRANSFERS INTO THE PROCESSOR

The Peripheral A I/O port on the R6520 is designed to handshake on data transfers from the peripheral device into the processor. In this sequence, the peripheral device must signal the processor that data are available and the processor must signal back that data was taken. This is basically the same sequence as that performed in the previous operation. The CA1 interrupt input is used to interrupt the processor to indicate that there are data available on the Peripheral A I/O port. The peripheral device must then hold the data there until the processor reads them into its internal registers. When the processor reads the Peripheral A I/O port, the CA2 peripheral control line goes low to signal to the peripheral device that the data have been taken and new data can be made available. This entire sequence is shown on Figure 3-7.

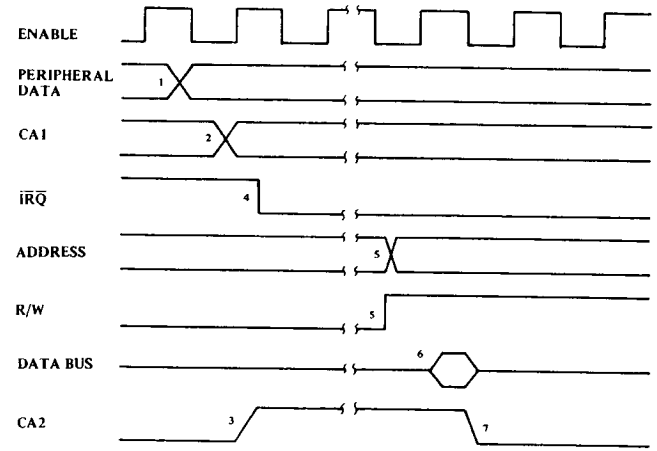
The handshaking operations described above can be an extremely powerful technique for interfacing data storage devices or, in general, any device which must transfer blocks of data and which has a variable response time. If the processor cannot predict the speed with which the



1. Processor puts out address of peripheral device and changes R/W signal to write enable (low).
2. During Phase 2 processor puts out data on Data Bus.
3. Data from the processor is accepted by the R6520 on the falling edge of the enable clock.
4. Peripheral Interface device now begins the handshake by signaling the peripheral device that data are available to read on the output port.
5. When the external peripheral device reads the data on the output port it will respond by a change in CB1.
6. The change in CB1 is followed by a positive transition of CB2 signalling the processor that data were accepted.

Write Handshake Sequence

FIGURE 3-6



1. New data are put out by peripheral device.
2. The peripheral interface device is signaled by CA1 that the new data are ready to read at the input port.
3. CA2 is put into the high state.
4. The processor is signalled that new data are ready to be read by a low level on the $\overline{IRQ}$ line.
5. The processor begins servicing the Interrupt request, and during the routine the processor will put out the read signal and the Address of the Peripheral Interface device.
6. The Peripheral interface will transfer the new data from the peripheral device to the microprocessor through the data bus.
7. When data have been transferred, the peripheral device will be signaled by CA2 going low.

Read Handshake Sequence

FIGURE 3-7

peripheral takes data, for instance, it must rely on the peripheral to signal that it has done so.

Initiating the data-transfer sequence is usually accomplished through a set of I/O lines separate from the port which is transferring the data. However, once the sequence is under way, the processor must deal with the peripheral device only when an interrupt has occurred. This allows the processor to execute the primary system program while still servicing these peripheral devices.

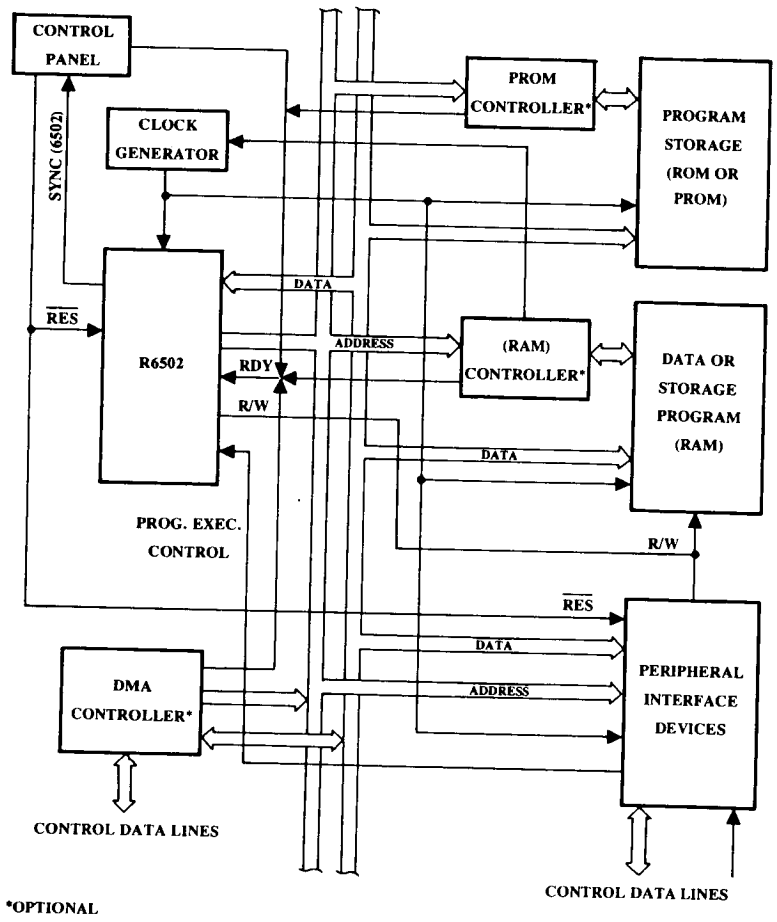
3.2 THE MICROPROCESSOR/SUPPORT CHIPS INTERFACE

The system block diagram (Figure 3-8) shows the basic data paths which allow the R6500 system to operate. Data Bus, Address Bus, R/W signal, etc. are shown as simple connections between the various chips in the system. These data paths will exist in any system, no matter how complex. Nevertheless each element of the microprocessor interface must be examined to ensure that each chip is properly driven with signals which meet all specifications for the device, that the inter-chip timing is proper, and that the overall system is operating as required.

3.2.1 Assigning Addresses in the R6500 System

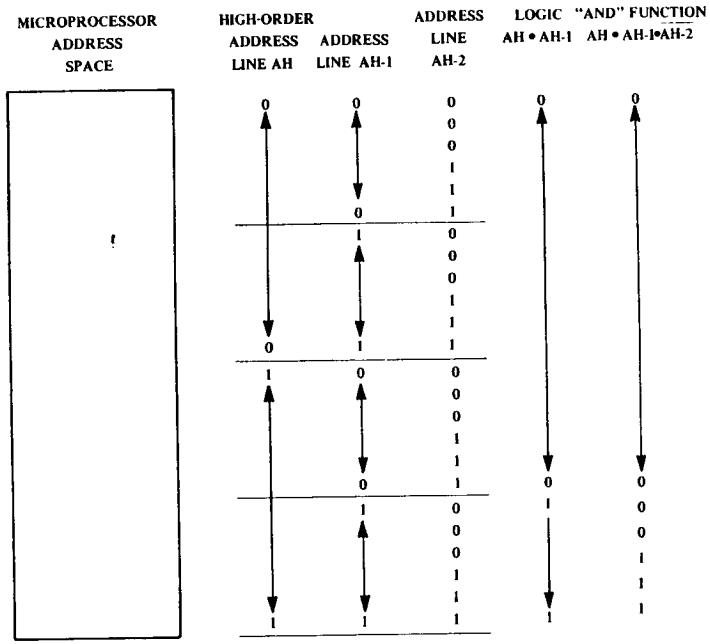
The only method which the microprocessor has for selecting between the various RAMs, ROMs, etc. in a system is through the address output lines. For this reason, the designer must use these lines very carefully to achieve minimum system cost and to ensure satisfactory system performance.

Before looking at how the address lines can be configured to minimize total system cost or program execution time, the designer should understand how the binary value associated with each address line is related to the total address space available to the microprocessor and how the AND function of various address lines can be employed to select large blocks of addresses. Figure 3-9 illustrates the state of the three high-order address lines for the entire address space available to the R650X. Note that the highest-order address line is a logic 1 for exactly half of the available address. The AND function of the two highest-order address lines is a logic 1 for one-fourth of the available addresses, and so forth. Figure 3-9 also illustrates several AND functions derived from the three highest-order address lines; each is true for a different block of the available addresses.



Organization of Microcomputer System

FIGURE 3-8



Example of "AND" Function Using High-Order Address Lines
FIGURE 3-9

Generation of the AND function of various high-order address lines is extremely important because of the chip select techniques employed on the processor support chips. As described in Section 5.1.4, the R6520 has three chip-select lines. The entire chip is selected for reading or writing data when CS1 and CS2 are high (> 2.4V) and CS3 is low (< 0.4V). Selection of the address lines which enable the various chips in the system is a very basic but very important part of the system configuration task.

It is important to note here that very few microprocessor-based systems actually require that the processors be able to access a full 65,536 words. In fact, most systems can be programmed in less than 2,000 words for program and data memory. The full address space is made available primarily because it permits the configuration of systems with an absolute minimum of separate decoding chips between the processor and the support chips. It is possible to assign any block of address to each type of chip (RAM, ROM, peripheral interface chips, etc.) in the system. However, each of the assigned addresses must be mutually exclusive. Only one of the support chips should be selected for every address used in the system program.

ROM ADDRESS ASSIGNMENT

The assignment of ROM addresses is dictated by the fact that the interrupt and RESET vectors must be located in the six high-order words in memory. These are fixed vectors and must be stored permanently in these locations. For this reason, the program memory (usually ROM) is usually assigned the high-order addresses. In fact, the recommended procedure is to use A15 (A12 for R6504 and R6507 and A11 for R6503, R6505 and R6506) to select program ROM.

RAM ADDRESS ASSIGNMENT

There are several factors which determine the location of the RAM in an R650X-based system. Data stored in memory under control of the internal processor Stack Pointer will always go into Page One (ADH = 01). Also, the entire set of Page Zero addressing modes relies on there being data storage RAM in Page Zero. For this reason, the RAM in a R650X-based system should be placed in the low-order addresses in memory.

With the RAM in low-order memory and the ROM in high-order memory, the peripheral interface devices must go somewhere in between. This is

accomplished in Figure 2.10 by using A15 · A14 to select ROMs, $\overline{A15}$ to select RAM, and A15 · $\overline{A14}$ to select all peripheral interface devices. This allows differentiation between the types of support chips. The addressing structure can be completed by allowing for selection of each chip in the groups.

The addresses which select the various registers, peripheral ports, etc. within the peripheral interface devices that are normally employed will not be sequential. For this reason, it is usually recommended that the technique shown in Figure 3-10 be employed to differentiate between the peripheral interface chips. This permits selection of 12 devices with no decoding in a R6502-based system, up to nine R6520 devices in a R6504 or R6507 based system, and up to eight devices in a R6503, R6505 or R6506-based system.

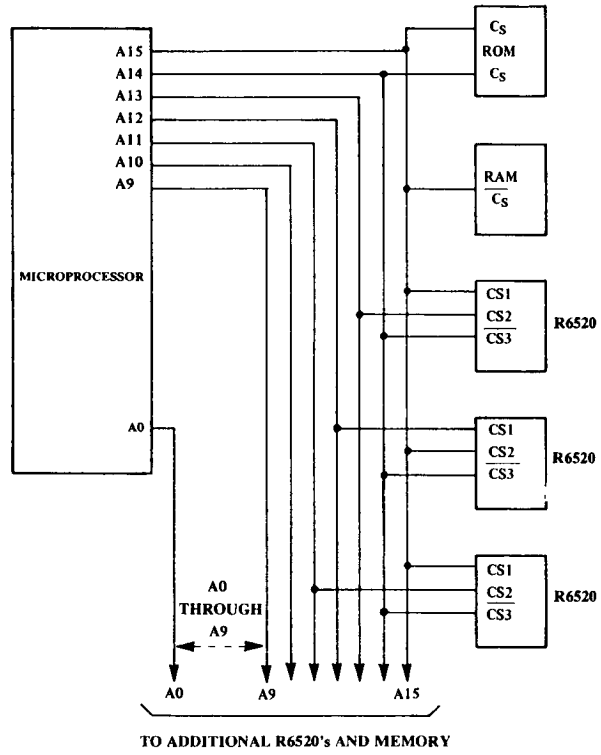
ADDITIONAL ADDRESS ASSIGNMENT TECHNIQUES

In many systems, the techniques illustrated above may not represent the best solution to the system problem. This is particularly true if program execution speed is a primary consideration. The time required to access the peripheral devices can be reduced by putting these devices in Page Zero. The entire set of Page Zero addressing modes can then be used to access these devices. In addition, the polling of the R6520 control registers during interrupt servicing can be facilitated greatly by putting the control registers in sequential addresses. These registers can then be accessed, making use of the Page Zero, Indexed addressing mode described in the Programming Manual. The address interconnect which allows this is shown in Figure 3-11. Note that this implementation requires external address decoding chips, but, for the system requiring it, this incremental cost will result in high operating speeds.

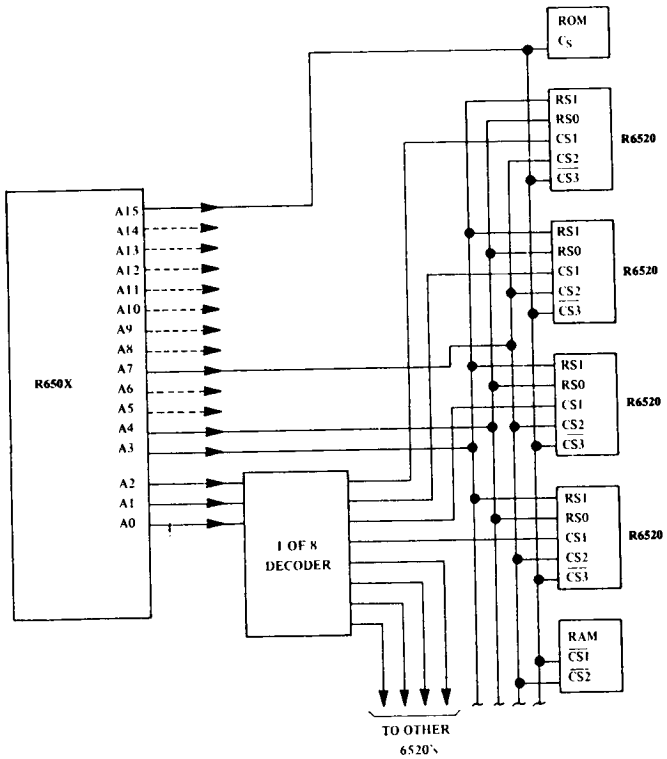
The system designer must become familiar with the addressing lines and their effect on the address space available to the processor. Even more importantly, there is a significant relationship between software and hardware in microprocessor systems and a full understanding of both can facilitate optimization of the trade-off between speed and cost for the system under design.

3.2.2 Interrupts

The basic concept of interrupts is introduced in Section 1.2.3 of this manual. However, little is said there about the hardware and software



Typical Address Assignments
FIGURE 3-10



Page Zero Chip Select Addressing Scheme
FIGURE 3-11

techniques which are required to assure proper implementation of the interrupt system. This section is designed to introduce the designer to the details of interrupts and interrupt servicing techniques.

INTERRUPT PRIORITIZING

Chapter 1 makes reference to various techniques for hardware prioritizing of interrupts to allow more rapid servicing of interrupts. The goal of this hardware is to allow the processor to go directly to the program which services the highest-priority active interrupt without taking the time to poll each interrupting device.

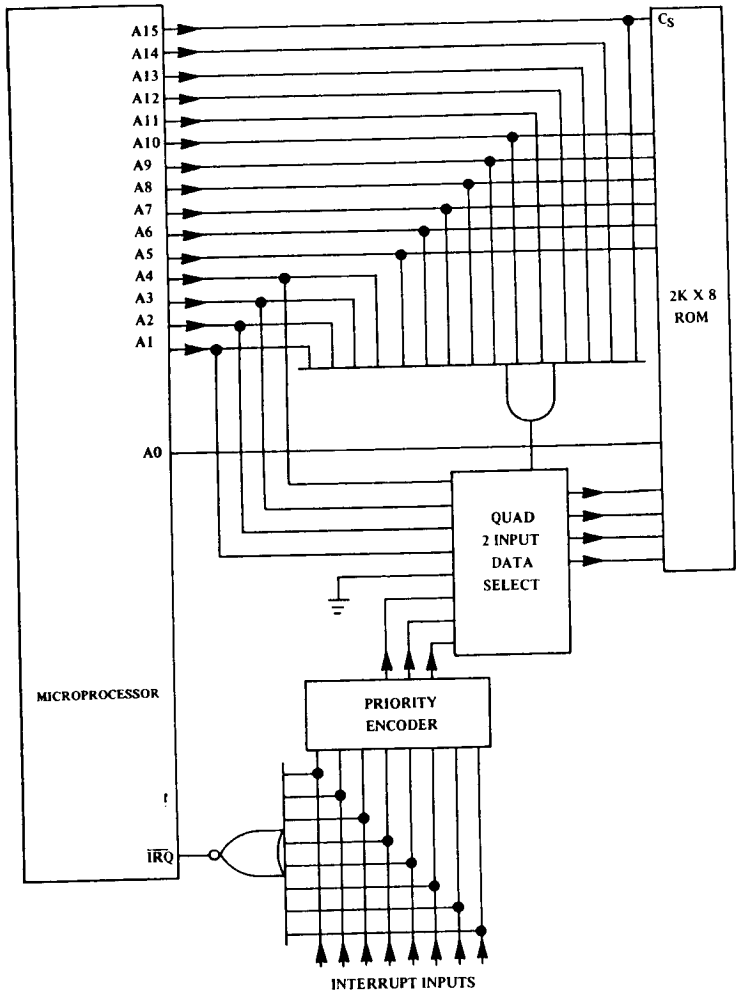
All hardware prioritizing techniques are based on the "priority encoder" shown in Figure 3-12. This device has eight inputs which are assigned a priority level from one to eight and generates a three-bit binary code corresponding to the highest-priority active input signal.

The generation of this three-bit code is in reality a trivial task for the designer. However, relating this code to the address of the corresponding interrupt service routine is much more difficult and represents an opportunity for creativity on the part of the designer. Several solutions will be illustrated here to demonstrate what can be done. These are certainly not assumed to be the only solutions. Each system must be considered separately to assure that the implementation chosen is as close to optimum as possible.

EXAMPLE 1: SELECTING THE INTERRUPT VECTOR

The final step of interrupt response within the processor is the fetching of an interrupt vector from two fixed addresses in memory. The interrupt vector located in these fixed addresses identifies the address of the software which the processor executes to poll the interrupting devices. Instead of pointing to the polling routine, it would be much faster to go directly to the software which actually services the interrupt. This requires a unique vector for each interrupt.

The technique illustrated in Figure 3-12 assumes that the interrupt vectors are located in ROM at addresses below that normally assigned to the interrupt vector. The decoder detects the fact that the processor is reading FFFE or FFFF. At this time the address inputs AD1, AD2 and AD3 into the ROM are driven from the priority encoder. Instead of accessing FFFE or FFFF, the



Selecting the Interrupt Vector

FIGURE 3-12

interrupt vector will come from two addresses selected by the priority encoder. The actual hardware involved is quite simple and the interrupt response time is an absolute minimum.

EXAMPLE 2: USING THE PROCESSOR SOFTWARE POWER

These several solutions to the vectored interrupt problem take advantage of certain instructions which can be performed by the processor. The first of these employs an instruction called the "Jump Indirect." This instruction causes the processor to begin executing the program located at that address contained in two sequential memory locations.

As in Example 1, the three-bit output from the priority encoder becomes part of the address of the interrupt software. If the output of the priority encoder is connected to the inputs of a peripheral interface device, the processor can then perform a Jump Indirect operation using the address on the two peripheral I/O ports. This is illustrated in Figure 2.13.

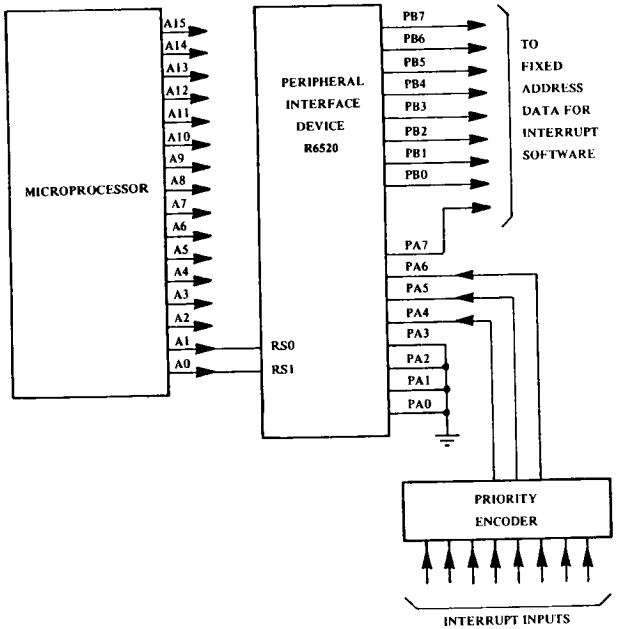
Another solution which takes advantage of the processor software is shown in Figure 3-14. Once again, the output of the priority encoder is connected to the inputs of a peripheral I/O port. However, in this approach, the priority encoder is connected to the low-order bits and the other bits can serve as control or input lines for other functions.

In this method, the three bits from the priority encoder will become part of an address established in memory. This address will then be used in a Jump Indirect instruction as before. This operation is detailed in Figure 3-15.

3.2.3 Memory Interface Control Using RDY

The ability to stop the microprocessor can be extremely important when using memory devices that are not directly compatible with the R650X family.

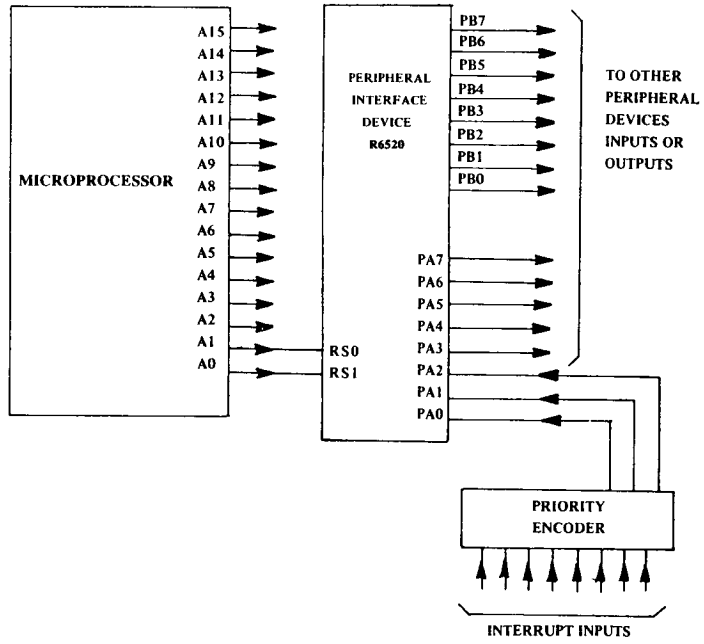
The RDY line can be used to stop the processor in any "non-write" cycle -- i.e., any cycle in which the processor is not attempting to write data into memory. The processor can be stopped for any number of clock cycles -- from one cycle for interfacing with slow memories to many cycles for DMA applications and for single cycle execution.



NOTE: CONNECTING THE ADDRESS LINES AS SHOWN PUTS THE TWO R6520 I/O PORTS IN SEQUENTIAL ADDRESSES.

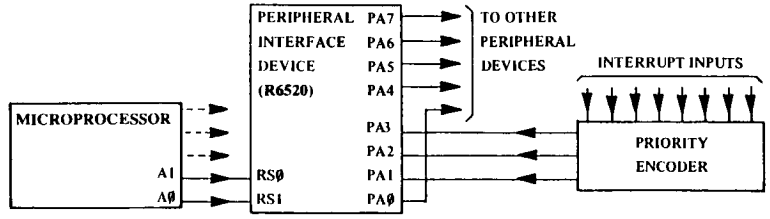
Using R6520 for Jump Indirect Interrupt Routines

FIGURE 3-13



Priority Encoder Connected to Low-Order Bits of R6520

FIGURE 3-14 a



Priority Encoder to Peripheral Interface Scheme

FIGURE 3-14 b

Priority Encoder Connection Schemes

FIGURE 3.14

```
INTVEC --->  PHA                    Receive Interrupt Vector
             TXA
             PHA
             LDA    IPA AO          Read PIA Port
             AND    #OE             Clear PIA
             TAX                    Transfer Acc. to X index reg.
             LDA    VEC TAB, X      Load Acc. from Interrupt Vector
                                    Table stored in memory
             STA    JMP1            Set Low-Order Address Byte
                                    of Interrupt Vector
             INX                    Increment X Index Register
             LDA    VEC TAB, X      Load Acc. from Interrupt Vector
                                    Table
             STA    JMP1+1          Set high-order Address Byte
                                    of Interrupt Vector
             JMP    (JMP1)          Go to Interrupt Service
                                    Software
```

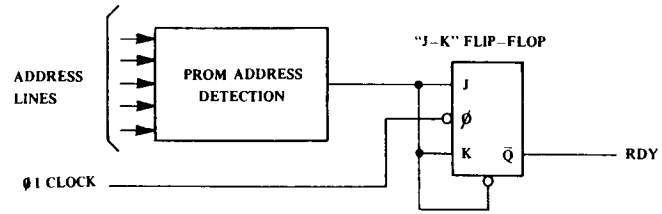
Software Program to Implement Interrupt from above Hardware Configuration

FIGURE 3-15

INTERFACING SLOW PROMS

One of the principal applications of RDY is in the control of light-erasable PROMs or EAROMs. These devices generally have longer access times than that required by the microprocessor when operation at 1 MHz clock frequency and are incapable of making data available on the data bus within 100 nanoseconds of the end of the Phase 2 clock pulse. The Phase 2 clock pulse is used to latch data or instructions on the data bus; therefore, if the data are not available at the correct time, the processor must be held up for one full cycle. The instruction will then be latched on the following Phase 2 pulse. Execution of the instruction will then proceed during the next cycle. Suggested logic for performing this function is shown in Figure 3-16.

Note that the data present on the data bus during the Phase 2 clock pulse after RDY goes high are the data that will be used in the instruction execution which takes place during the following cycle.



Interfacing Scheme for Slow PROM's

FIGURE 3-16

DIRECT MEMORY ADDRESS (DMA) TECHNIQUES

Transfer of data from peripheral storage devices into the microcomputer data memory (RAM) can normally be handled one byte at a time under control of the microprocessor. However, in large data terminals, control systems, etc. the primary data storage device may be a high-speed tape or disk. In systems such as these, the data transfer from the storage device into memory must be performed at speeds greater than the processor can handle. The control of the transfer must be performed outside the processor in a separate controller, and the peripheral device must gain direct access to the system RAM.

Direct Memory Access requires primarily that the processor have no need to access the memory involved. This is generally ensured by stopping the processor completely. The DMA controller must then gain access to the R/W line and both the address and data busses on the memory unit.

Provision for stopping the processor is available on the R6502, R6505 and R6507. This is accomplished by pulling the RDY line on the processor to GND (< 0.4V). The processor will stop in the first non-write cycle with the data bus in the high-impedance state. After the processor has stopped, the DMA controller must provide the address and data for the memory and must control R/W if data are being transferred into memory.

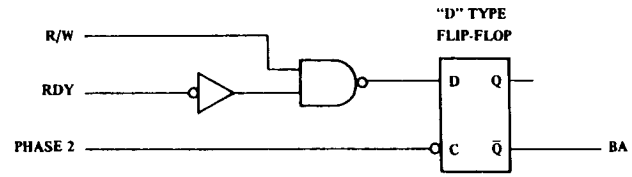
Providing addresses for the memories can be accomplished by gating addresses from either the DMA controller or the microprocessor into the memories. This can be accomplished very easily with a Quad 2-input data selector. During the DMA operation, the addresses fed to the memories are those generated by the DMA controller. After the DMA operation is complete, the input-select signal to the data selector is inverted, and the addresses generated by the processor once again determine which memory word is being accessed. The R/W line to the memories can be controlled in the same way as the address lines.

The data bus must be controlled in a somewhat different manner. This is necessitated by the fact that these lines are "bidirectional," with the data bus pins on the processor and the support chips serving for both input and output. The output buffers in each of these chips are capable of entering a high-impedance state to allow any of the devices to drive the bus during data and instruction transfers. For this reason, a bidirectional, "three-state" bus extender is required to interface the DMA controller to the system data bus. The logic necessary to provide full address bus and data bus control for DMA applications is shown in Figure 3-17.

The R6502, R6503, R6504, R6505, and R6507 do not make provision for the Bus Available signal. However, these processors still stop in the first non-write cycle. For this reason, the logic shown in Figure 3-17 should be used to generate a Bus Available signal for the DMA controller.

CONTROL OF DYNAMIC RAMS IN THE R6500 SYSTEM

For systems which must contain a large quantity of Read/Write memory (RAM), the 4096-bit dynamic RAMs can provide the required storage with a minimum number of parts. However, there is one major drawback to these devices -- they must be refreshed periodically. For most devices currently available, this refresh period is about 2 milliseconds for the entire chip. Refreshing the entire chip requires 32 Read operations which can be performed all at once every 2 milliseconds, or performed one-at-a-time approximately every 64 microseconds.



Logic Used to Generate Bus Available Signal for DMA Applications
FIGURE 3-17

Unless a separate controller is used to perform this refresh operation, the use of dynamic memories can be very detrimental to system performance.

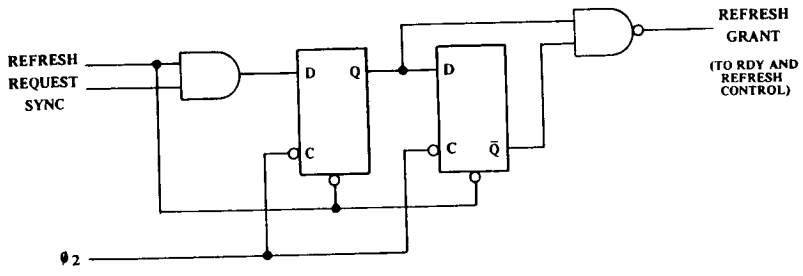
As with any Direct Memory Access, the processor must be stopped to assure that the processor and the DMA controller are not attempting to access the memories concurrently. The RDY input provides this capability. A counter operating directly from the system clock will provide a very convenient refresh signal. Each time the counter goes through a count of 63, a "refresh request" pulse is generated. The actual memory refresh operation must take place during a Read operation with the processor stopped for 1 cycle. Determining when the processor has stopped presents exactly the same problem as in DMA operations. The controller must pull the RDY line low and must then examine the R/W line to determine when the processor is in a Read cycle.

The specific operation performed during the refresh cycle is a function of the devices being used. However, it should be noted that the time available for refreshing the memory is "N - 1/2" cycles, where N is the number of cycles that the processor is stopped. Control of the memory address lines must be returned to the processor at the beginning of Phase 1 if the memories are to have a full cycle to make valid data available on the data bus. This leaves one-half cycle available to perform

the refresh operation if the processor is stopped for one cycle. A full 1-1/2 cycles can be made available by stopping the processor for two cycles. This latter implementation is more compatible with most dynamic RAMs currently available.

As described above, a primary problem in the implementation of dynamic RAM systems is related to knowing when the processor has stopped. A full one-half cycle is required in the implementations described above. The R6502, however, provides a signal -- the SYNC signal -- which can be utilized to predict that the processor will stop in the very next cycle. It is impossible for a Write operation to immediately follow an instruction fetch cycle. This allows the memory refresh controller to assume control of the address lines at the beginning of that cycle instead of after the trailing edge of Phase 1.

The RDY line is pulled low on Phase 1, and the processor is guaranteed to stop. Control of the address lines is returned to the processor on the next Phase 1 and RDY is set high at the same time. The result is the refresh logic has a full cycle to refresh the memories and the processor loss only one cycle of execution time. A suggested configuration for this control logic is illustrated in Figure 3-18.



Control Logic for Refresh Signal for Dynamic RAMS
FIGURE 3-18

3.3 ADDITIONAL SYSTEM CONSIDERATIONS

After the basic system configuration is complete, extensive breadboarding and testing are usually required before the design is finalized. However, this breadboarding and evaluation must be preceded by a complete evaluation of the cost and performance of the proposed design to guarantee that the various goals of the project will be met.

The first step in evaluating the design is to estimate the amount of ROM and RAM that will be required, as well as the number and type of interface devices required to control the peripherals

3.3.1 Peripheral Interface Devices

The number and type of peripheral devices can generally be estimated very accurately. However, it is important to keep in mind that these estimates must be subject to review after a full analysis of system performance is completed. The designer may find it necessary to employ a special-purpose interface part or to redesign the I/O structure if the evaluation of system performance reveals that the system cannot operate at the required speed. Use of special-purpose peripheral interface parts will reduce the number of tasks which must be handled by the processor and consequently can increase the overall system speed, but this generally involves additional component cost.

Similarly, the use of a fully vectored interrupt can lead to increased performance at increased cost. The goal of any design program must be to meet all the system performance at the minimum possible cost.

After the various peripheral devices in the system have been evaluated to determine the number of inputs and outputs required, the total required by all peripherals can be divided by 16 to determine the number of devices required. This is a good first approximation which will be re-evaluated as the system development progresses.

3.3.2 RAM

The evaluation of the amount of RAM required by the system is a somewhat more difficult problem than estimation of peripheral devices. This is due primarily to the fact that much of the RAM is required by the system software as working storage, such as storage of immediate results in

arithmetic operations. Since the system program will probably not be written when these estimates are first attempted, the probability of error in this portion of the estimate may be fairly high.

In addition to working storage, the RAM must provide storage for:

1. The "stack" (described in the Programming Manual)
2. Peripheral input data storage
3. Peripheral output data storage

Items 2 and 3 above can be evaluated quite accurately, since a detailed analysis of the peripheral devices has usually been completed when these estimates are first attempted. In general, a block of RAM must be made available for each peripheral device. The amount of RAM required for each is a function of the type of peripheral device being interfaced and just how the device is to be controlled.

The amount of RAM required by the stack is a function of both the interrupt structure and the system software. As a result, an estimate of this requirement must be based on the system programmer's best estimates of his requirements. This should be combined with an estimate of the required working storage and the peripheral data storage requirements to obtain an estimate of the total system RAM.

3.3.3 ROM

The amount of ROM required in a system cannot be determined accurately until the system program is completed. However, by partitioning the system program into definable pieces, an estimate can be made of each task and the total can be obtained of the ROM required by each section.

Most programs consist of easily defined sections such as the software for each peripheral device, arithmetic routines, etc. These are the pieces which should be examined separately to estimate the ROM required by each.

3.4 EVALUATING SYSTEM PERFORMANCE

As discussed in the previous section, the peripheral interface structure for a system is fairly easy to configure if one assumes that R6520-type devices are used. However, before going too far into hardware construction, it is important that the total system performance be evaluated to minimize the probability that major problems will arise in the later stages of the design.

Evaluating system performance involves first determining whether or not the processor is capable of processing all interrupts with the speed required,

and then determining that the processor has sufficient time to perform non-interrupt operations.

The prioritized interrupt structure assumes that, at times, more than one interrupt will occur and that there will be delays encountered in servicing some interrupts caused by the presence of other interrupts. This structure will perform satisfactorily if these delays are not too great.

The interrupt processing time should be evaluated starting with the highest-priority interrupt, then going to the next-highest priority, each time keeping in mind the total time which can be lost due to concurrent higher-priority interrupts. Each time an interrupt is examined, the worst microprocessor response time which can be encountered should be estimated. If this time is still adequate for the function being handled by the interrupt, then that aspect of the system operation can be expected to perform satisfactorily.

The ability of the R650X microprocessors to handle interrupts quickly and conveniently represents one of the real strengths of this microprocessor family. However, in any system being developed, it is important that the percentage of processor time spent servicing interrupts not be so large that the internal data handling, arithmetic operations, etc. cannot be executed properly.

Since the interrupts are usually asynchronous and are not related directly to the main line program, the time lost to interrupts can usually be viewed as an average percentage of the total time. The speed with which the main program can be executed will be reduced by this percentage.

The interrupt service routines are usually short and easy to evaluate. However, the main program is much more difficult to estimate. Fortunately, it is also usually much less critical. Those operations which must meet a particular speed requirement can be examined in detail by the programmer to determine the execution time. This estimated execution time must then be reduced to allow for the time lost to interrupts.

The final step in ensuring satisfactory system performance is a worst-case analysis. This is to determine if there are any places in the program where worst-case interrupts can cause excessive delays in the execution of other programs being executed. Although the effort involved in a complete worst-case analysis is usually excessive, this is one part of the system development task which can lead to significantly greater assurance of success for the entire development process.

SECTION 4

BRINGING UP THE R6500 MICROCOMPUTER SYSTEM

4.1 MICROCOMPUTER TESTING

After many hours of planning, hardware construction, and programming effort, the microcomputer system designer must face what can be his most difficult task: "bringing up" his system. The modern microcomputer with its minimum chip count, and its minimum number of control and data lines, represents a tremendous advance in system design when everything is working properly. However, it can also represent a testing nightmare to the designer who is attempting to troubleshoot the hardware and software which constitute the total design.

A microcomputer lacks many of the things which make testing of conventional logic relatively convenient. To begin with, one simply cannot see most of the control signals, data transfers, etc. which allow the system to operate. In addition, it is impossible to examine directly the contents of the registers and latches which store data within the processor. The data can be examined only indirectly, by looking at the signals on the inputs and outputs to the chip at the proper time.

This problem is compounded by the fact that many programs must be tested "dynamically" -- i.e., the system must be running at its full operating speed with non-recurring events or with a total lack of usable oscilloscope-triggering signals.

For these and many other reasons, it is important that the system designer build effective testing capability into both his hardware and his software. This is particularly true for the pre-production prototypes. When combined with the procedures discussed below, this will minimize both the time and the effort spent in producing that first operational system. After the program and the hardware are completely debugged, many of the testing tools discussed below can be removed from the prototype design without affecting system performance. This allows the designer to arrive at his final production design very shortly after he has proven that the prototypes are operating satisfactorily.

4.1.1 Static Testing

Static testing (i.e., execution of the program), one cycle or one instruction at a time, is the first step in the checkout of any system. In this way, the general flow of the program can be examined and for much of the program the validity of data transfers into and out of memory can be verified. As shown in Figure 4-1, the logic necessary to control RDY to allow Single-Cycle and Single-Instruction Execution is relatively simple. This hardware and its use in system testing are discussed below.

SINGLE-CYCLE EXECUTION

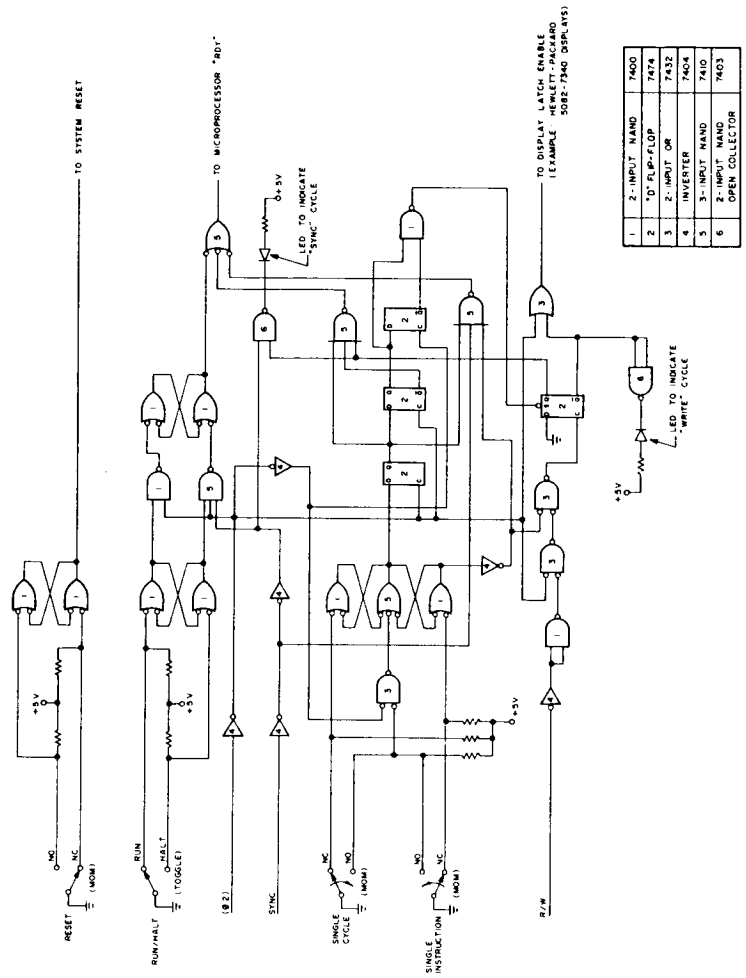
The timing required for Single-Cycle-Execution is shown in Figure 4-2. In this operation, the RDY line has been brought low (GND) to halt the processor. To allow execution of a single cycle, the RDY line goes high (+2.4V) for one cycle each time the Single Cycle switch is activated. Note that the RDY line goes high while the Phase 1 clock is high and the internal timing counter advances on the next Phase 1 clock pulse.

Single-cycle operation allows stopping the processor in any cycle except a WRITE cycle. This allows detailed examination of all cycles of the instruction fetch operation. In addition, it permits detailed examination of operand fetches. Thus, it is possible to verify the operation of most of the hardware involved in memory addressing and control. It is also possible to verify the operation of most of the peripheral interface hardware. This can greatly reduce the time required to test the full dynamic operation of the peripheral device.

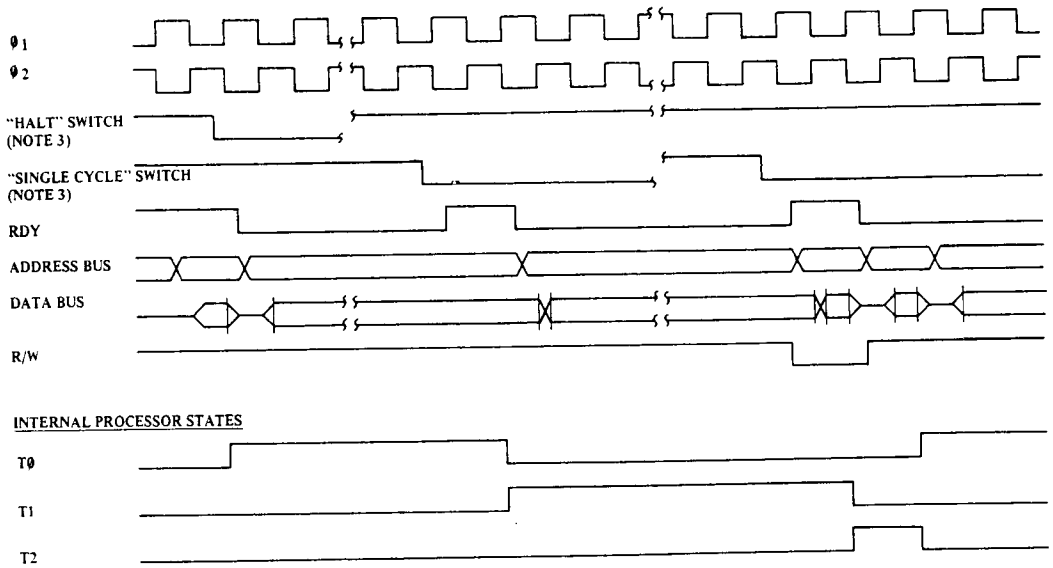
Note that if depressing the Single Cycle switch allows the processor to advance into a WRITE cycle, the processor will complete this cycle and will then stop in the first READ cycle (R/W = 1) which follows. This timing is shown in Figure 4-2.

Appendix A contains a detailed summary of the data which should appear on the address and data lines during each cycle of the R6502 instructions.

Note that the processor often puts out an address and fetches data which it ignores. This is an inherent feature of the processor which uses a "look-ahead" approach to pipelining. Examination of the SYNC signal will allow the designer to keep track of exactly when the data fetched from memory is utilized within the processor and when it is ignored.



Suggested Static Test Control Logic
FIGURE 4-1



NOTES: 1. *INDICATES AN UNDETERMINED TIME PERIOD DURING WHICH THE SIGNAL WILL CHANGE.*

2. *THE DATA BUS ENTERS THE HIGH-IMPEDANCE STATE DURING EACH PHASE ONE PULSE. HOWEVER, WHILE THE PROCESSOR IS STOPPED THE DATA BUS WILL APPEAR TO REMAIN HIGH OR LOW AS SHOWN.*

3. *SWITCH ACTUATION IS INDICATED BY A LOW SIGNAL.*

Single-Cycle Timing
FIGURE 4-2

A very simple "data trap" can be built into prototype systems to allow examination of the address and data generated by the processor during WRITE cycles. This trap may latch the contents of both the address and data busses or it may latch only the address bus. The latter can be sufficient if a separate means of examining data in memory is provided (see Section 4.2). A suggested configuration for the "data trap" is shown in Figure 4-3. This circuit can be used to display the contents of the address and data busses for both READ and WRITE cycles. The WRITE data are latched and held during the next READ cycle. Depressing the Latch Reset switch then opens the inputs to the latches and allows monitoring of the subsequent READ cycles.

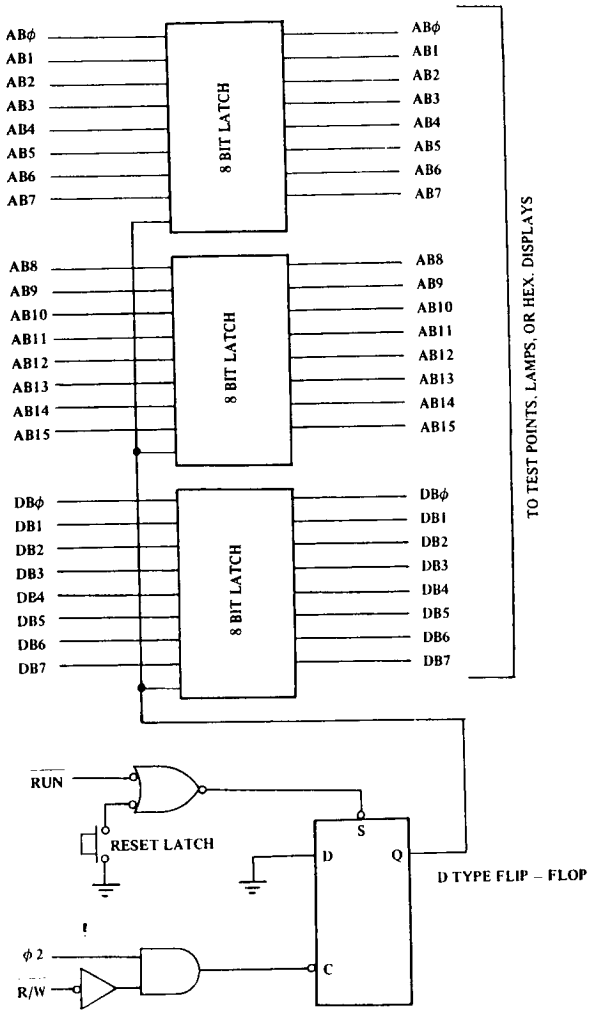
SINGLE INSTRUCTION EXECUTION

While it is extremely useful to be able to analyze the execution of each instruction in detail, it is often sufficient just to look at the general program flow. This is particularly useful when examining the operation of branches and jumps in a program. Single instruction execution is designed to allow this capability on the R6502 which outputs a SYNC signal.

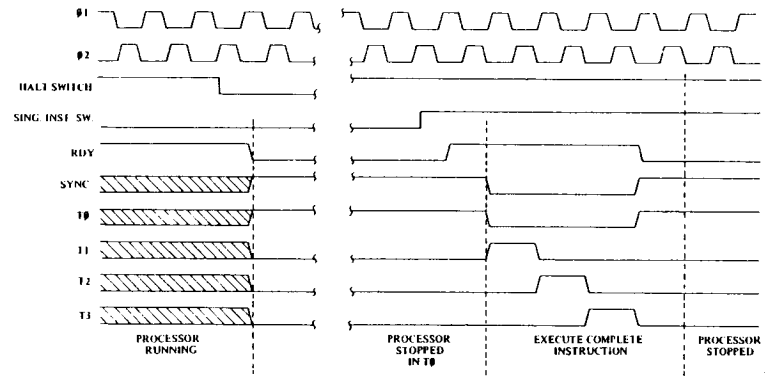
The operation of the single instruction execution logic is based on generation of a SYNC signal within the processor. This signal goes high (> +2.4V DC) during each OP CODE fetch cycle. Single instruction execution is implemented by using SYNC to force RDY low (< +0.4V DC). Under these conditions, the processor will always stop with an OP CODE address on the address bus and the OP CODE on the data bus. The timing for this operation is shown in Figure 4-4. Note that this diagram assumes that the processor is stopped in an OP CODE fetch cycle. Depressing the Single Instruction switch (Figure 4-1) allows execution of that instruction. The processor then stops when the next OP CODE is fetched.

4.1.2 Dynamic Testing

Through static testing techniques, the designer should be able to verify the operation of most of his processor interface hardware, such as the Bus Expanders and Address Decoders (for selecting ROMs, RAMs, etc.). However, this is only a first step to assuring proper system operation. Most peripheral devices cannot be properly tested unless the processor is operating at full speed. This necessitates full dynamic testing.



Microprocessor Single Cycle Data Trap
FIGURE 4-3



Single Instruction Execution
FIGURE 4-4

Dynamic testing generally involves causing the processor to execute a program loop, i.e., to execute a repetitive sequence of instructions. This allows the use of an oscilloscope in examining the processor operation. This repetitive operation can be externally induced through the $\overline{RES}$ or Interrupt ($\overline{IRQ}$ or $\overline{NMI}$) lines, or it can be a part of the program being executed. Both techniques play an important role in the system checkout process.

EXTERNALLY INDUCED LOOPS

The most direct means of causing the processor to execute a loop is to drive one of the direct inputs ($\overline{RES}$, $\overline{IRQ}$ or $\overline{NMI}$) with a signal generator. This technique can be employed to troubleshoot systems which are only partially operational, since it does not rely on proper execution of a particular set of instructions to cause looping to occur. However, this technique can be used only if an oscilloscope can be employed in examining system operation; to do so requires an effective scope-synchronizing signal. For this reason, the following section will discuss not only the signals to be tested and the waveforms which one should see, but also the techniques one may use to assure generation of an effective scope sync.

Probably the most basic operation performed within the processor is the RESET function. Without the RESET hardware and software operating properly, the system will never enter its normal operating mode. For this reason, the first major function to be tested, both statically and dynamically, is the $\overline{RES}$ input.

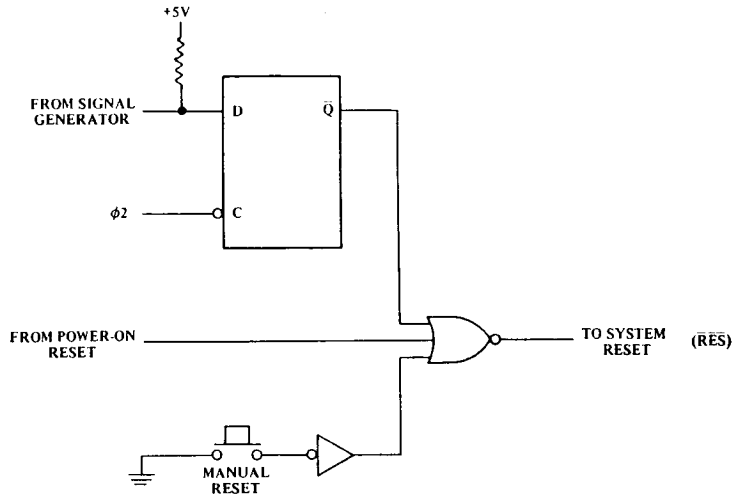
A suggested configuration for dynamically testing the RESET input is shown in Figure 4-5. In this diagram, the RESET input is being driven from a signal generator. Between the signal generator and the processor is a D-type flip-flop to synchronize the chip reset signal to the processor clocks. This synchronizing is extremely important because it stabilizes the data being displayed on the oscilloscope with respect to the scope sync.

The most effective procedure for testing the dynamic operation of the RESET function is to reset the system initially at a rate of approximately one-fifth of the clock rate. This will allow the processor to execute the first few instructions in the reset sequence before being recycled. The designer can then closely examine the timing of address, data and R/W signals. Use of the delayed sweep feature available on most modern oscilloscopes will allow examination of any part of the RESET operation.

When proper operation of the RESET input has been verified, the same technique can be applied to both the $\overline{IRQ}$ and the $\overline{NMI}$ inputs. Driving either of these inputs with a signal generator synchronized to the processor clocks will permit a close examination of the dynamic operation of the interrupt polling sequence. This provides a very important look at the Peripheral Interface selection logic to ensure that all peripheral devices are responding to the proper address.

SOFTWARE LOOPS

During system checkout, the designer must verify the operation of many simple functions which must all operate properly before the entire system is operational. The use of simple software loops will allow a detailed examination of one function at a time. Most importantly, it allows the designer to use an oscilloscope to examine events which may occur very infrequently and which are normally very difficult to see.



Suggested Configuration For Dynamic Reset Testing
FIGURE 4-5

The execution of software loop requires the writing of a program which ends in a JMP back to the beginning of the program. Once the processor enters the loop it will continue to execute the same sequence of instructions until the RESET switch is pushed.

To utilize software loops effectively there must be an event which happens only once each time the processor executes the loop. This signal can be used to trigger the oscilloscope. Including a single WRITE operation in the program allows the R/W signal to be used to trigger the scope. Likewise, careful selection of address in the program will allow use of an address line as a scope sync. Finally, lacking anything else, setting and resetting a peripheral interface device output pin at the beginning of the program provides a very effective sync signal.