the bus until 80 ns into the first $\overline{INTA}$ cycle. The first MCE can be inhibited by gating MCE with $\overline{LOCK}$. The 8086 $\overline{LOCK}$ output is activated during T2 of the first cycle and disabled during T2 of the second cycle. The overlap of $\overline{LOCK}$ with MCE allows the first MCE to be masked and the second MCE to gate the cascade address onto the local bus. Since the 8259A will not provide a cascade address until the second cycle, no information is lost. As with ALE, MCE is guaranteed valid within 58 ns of the start of T1 to allow 75 ns CAS address setup to the trailing edge of ALE. MCE remains active TCHCLmin – TCHLLmax + TCLMCLmin = 52 ns after ALE to provide data hold time to the latches.

If the 8288 is strapped in the IOB mode, the MCE output becomes $\overline{PDEN}$ and all I/O references are assumed to be devices on the local bus rather than the demultiplexed system bus. Since $\overline{INTA}$ cycles are considered I/O cycles, all interrupts are assumed to come from the local system and cascade addresses are not gated onto the system address bus. Additionally, the DEN signal is not enabled since no I/O transfers occur on the system bus. If the local I/O bus is also buffered by transceivers, the $\overline{PDEN}$ signal is used to enable those transceivers. $\overline{PDEN}$ A.C. characteristics are identical to DEN with $\overline{PDEN}$ enabled for I/O references and DEN enabled for instruction or memory data references.

5. Ready Timing

Ready timing based on address valid timing is the same for maximum and minimum mode systems. The delay from 8288 command valid to RDY valid at the 8284 is TCLCL – TCLMLmax – TRIVCLmin = 130 ns. This time is available for external circuits to determine the need to insert wait states and disable RDY or enable RDY to avoid wait states. $\overline{INTA}$, all read commands and advanced write commands provide this timing. The normal write command is not valid until after the RDY signal must be valid. Since both normal and advanced write commands are generated by the 8288 for all write cycles, the advanced write may be used to generate a RDY indication even though the selected device uses the normal write command.

Since separate commands are provided for memory and I/O, no M/$\overline{IO}$ signal is specifically available as in the minimum mode to allow an early 'wait state required' indication for I/O devices. The $\overline{S2}$ status line, however is logically equivalent to the M/$\overline{IO}$ signal and can be used for this purpose.

6. Other Considerations

The $\overline{RQ}$/$\overline{GT}$ timing is covered in the next section and will not be duplicated here. The only additional signals to be considered in the maximum mode are the queue status lines QS0, QS1. These signals are changed on the leading edge of each clock cycle (high to low transition) including idle and wait cycles (the queue status is independent of the bus activity). External logic may sample the lines on the low to high transition of each clock cycle. When sampled, the signals indicate the queue activity in the previous clock cycle and therefore lag the CPU's activity by one cycle. The $\overline{TEST}$ input requirements are identical to those stated for the minimum mode.

To inform the 8288 of HALT status when a HALT instruction is executed, the 8086 will initiate a status transition from passive to HALT status. The status change will cause the 8288 to emit an ALE pulse with an indeterminate address. Since no bus cycle is initiated (no command is issued), the results of this address will not affect CPU operation (i.e., no response such as READY is expected from the system). This allows external hardware to latch and decode all transitions in system status.

3G. Bus Control Transfer (HOLD/HLDA and $\overline{RQ}$/$\overline{GT}$)

The 8086 supports protocols for transferring control of the local bus between itself and other devices capable of acting as bus masters. The minimum mode configuration offers a signal level handshake similar to the 8080 and 8085 systems. The maximum mode provides an enhanced pulse sequence protocol designed to optimize utilization of CPU pins while extending the system configurations to two prioritized levels of alternate bus masters. These protocols are simply techniques for arbitration of control of the CPU's local bus and should not be confused with the need for arbitration of a system bus.

1. MINIMUM MODE

The minimum mode 8086 system uses a hold request input (HOLD) to the CPU and a hold acknowledge (HLDA) output from the CPU. To gain control of the bus, a device must assert HOLD to the CPU and wait for the HLDA before driving the bus. When the 8086 can relinquish the bus, it floats the $\overline{RD}$, $\overline{WR}$, $\overline{INTA}$ and M/$\overline{IO}$ command lines, the $\overline{DEN}$ and DT/$\overline{R}$ bus control lines and the multiplexed address/data/status lines. The ALE signal is not three-stated. The CPU acknowledges the request with HLDA to allow the requestor to take control of the bus. The requestor must maintain the HOLD request active until it no longer requires the bus. The HOLD request to the 8086 directly affects the bus interface unit and only indirectly affects the execution unit. The CPU will continue to execute from its internal queue until either more instructions are needed or an operand transfer is required. This allows a high degree of overlap between CPU and auxiliary bus master operation. When the requestor drops the HOLD signal, the 8086 will respond by dropping HLDA. The CPU will not re-drive the bus, command and control signals from three-state until it needs to perform a bus transfer. Since the 8086 may still be executing from its internal queue when HOLD drops, there may exist a period of time during which no device is driving the bus. To prevent the command lines from drifting below the minimum VIH level during the transition of bus control, 22K ohm pull up resistors should be connected to the bus command lines. The timing diagram in Figure 3G1 shows the handshake sequence and 8086 timing to sample HOLD, float the bus, and enable/disable HLDA relative to the CPU clock.

To guarantee valid system operation, the designer must assure that the requesting device does not assert con-

trol of the bus prior to the 8086 relinquishing control and that the device relinquishes control of the bus prior to the 8086 driving the bus. The HOLD request into the 8086 must be stable THVCH ns prior to the CPU's low to high clock transition. Since this input is not synchronized by the CPU, signals driving the HOLD input should be synchronized with the CPU clock to guarantee the setup time is not violated. Either clock edge may be used. The maximum delay between HLDA and the 8086 floating the bus is TCLAZmax – TCLHAVmin = 70 ns. If the system cannot tolerate the 70 ns overlap, HLDA active from the 8086 should be delayed to the device. The minimum delay for the CPU to drive the control bus from HOLD inactive is THVCHmin + 3TCLCL = 635 ns and THVCHmin + 3TCLCL + TCHCL = 701 ns to drive the multiplexed bus. If the device does not satisfy these requirements, HOLD inactive to the 8086 should be delayed. The delay from HLDA inactive to driving the busses is TCLCL + TCLCHmin – TCLHAVmax = 158 ns for the control bus and 2TCLCL – TCLHAVmax = 240 ns for the data bus.

1.1 Latency of HLDA to HOLD

The decision to respond to a HOLD request is made in the bus interface unit. The major factors that influence the decision are the current bus activity, the state of the $\overline{\text{LOCK}}$ signal internal to the CPU (activated by the software LOCK prefix) and interrupts.

If the $\overline{\text{LOCK}}$ is not active, an interrupt acknowledge cycle is not in progress and the BIU (Bus Interface Unit) is executing a T4 or TI when the HOLD request is received, the minimum latency to HLDA is:

35 ns	THVCH min (Hold setup)
65 ns	TCHCL min
200 ns	TCLCL (bus float delay)
10 ns	TCLHAV min (HLDA delay)
310 ns	@ 5 MHz

The maximum delay under these conditions is:

34 ns	(just missed setup time)
200 ns	delay to next sample
82 ns	TCHCL max
200 ns	TCLCL (bus float delay)
160 ns	TCLHAV max (HLDA delay)
677 ns	@ 5 MHz

If the BIU just initiated a bus cycle when the HOLD Request was received, the worst case response time is:

34 ns	THVCH (just missed)
82 ns	TCHCL max
7*200	bus cycle execution
N*200	N wait states/bus cycle
160 ns	TCLHAV max (HLDA delay)
1.676 μs	@ 5 MHz, no wait states

Note, the 200 ns delay for just missing is included in the delay for bus cycle execution. If the operand transfer is a word transfer to an odd byte boundary, two bus cycles are executed to perform the transfer. The BIU will not acknowledge a HOLD request between the two bus cycles. This type of transfer would extend the above maximum latency by four additional clocks plus N additional wait states. With no wait states in the bus cycle, the maximum would be 2.476 microseconds.

Although the minimum mode 8086 does not have a hardware $\overline{\text{LOCK}}$ output, the software LOCK prefix may still be included in the instruction stream. The CPU internally reacts to the LOCK prefix as would the maximum mode 8086. Therefore, the LOCK does not allow a HOLD request to be honored until completion of the instruction following the prefix. This allows an instruction which performs more than one memory reference (ex. ADD [BX], CX; which adds CX to [BX]) to execute without another bus master gaining control of the bus between memory references. Since the LOCK signal is active for one clock longer than the instruction execution, the maximum latency to HLDA is:

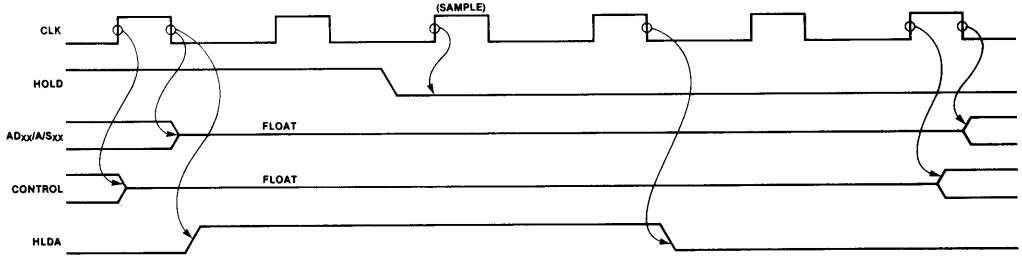


Figure 3G1. HOLD/HLDA Sequence

34 ns	THVCH (just miss)
200 ns	delay to next sample
82 ns	TCHCL max
(M + 1)*200 ns	LOCK instruction execution
200 ns	set up HLDA (internal)
160 ns	TCLHAV max (HLDA delay)
(M*200 ns) + 876 ns	@ 5 MHz

If the HOLD request is made at the beginning of an interrupt acknowledge sequence, the maximum latency to HLDA is:

34 ns	THVCH (just missed)
82 ns	TCHCL max
2600 ns	13 clock cycles for INTA
160 ns	TCLHAV max
2.876 μs	@ 5 MHz

1.2 Minimum Mode DMA Configuration

A typical use of the HOLD/HLDA signals in the minimum mode 8086 system is bus control exchange with DMA devices like the Intel 8257-5 or 8237 DMA controllers. Figure 3G2 gives a general interconnect for this type of configuration using the 8237-2. The DMA controller resides on the upper half of the 8086's local bus and shares the A8-A15 demultiplexing address latch of the 8086. All registers in the 8237-2 must be assigned odd addresses to allow initialization and interrogation by the CPU over the upper half of the data bus. The 8086 $\overline{RD}/\overline{WR}$ commands must be demultiplexed to provide separate I/O and memory commands which are compatible with the 8237-2 commands. The AEN control from the 8237-2 must disable the 8086 commands from the command bus, disable the address latches from the lower (A0-A7) and upper (A19-A16) address bus and select the 8237-2 address strobe (ADSTB) to the A8-A15 address latch. If the data bus is buffered, a pull-up resistor on the DEN line will keep the buffers disabled. The DMA controller will only transfer bytes between

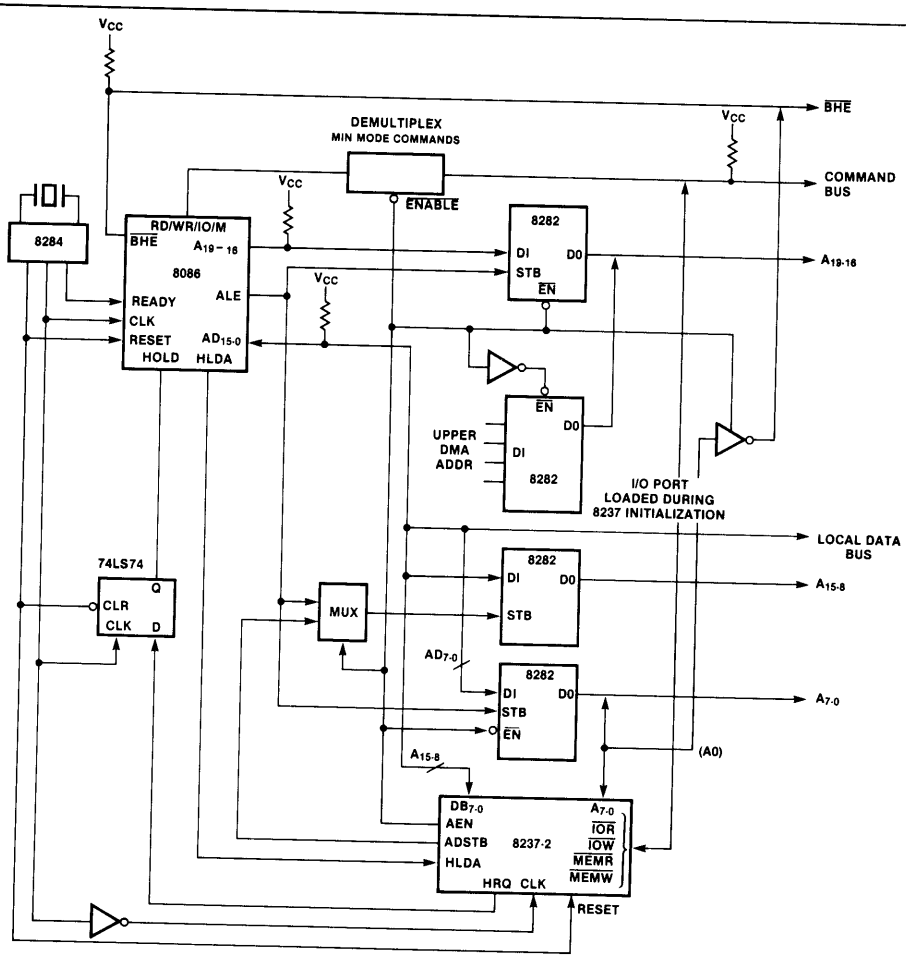


Figure 3G2. DMA Using the 8237-2

memory and I/O and requires the I/O devices to reside on an 8-bit bus derived from the 16-bit to 8-bit bus multiplex circuit given in Section 4. Address lines A7-A0 are driven directly by the 8237 and $\overline{BHE}$ is generated by inverting A0. If A19-A16 are used, they must be provided by an additional port with either a fixed value or initialized by software and enabled onto the address bus by AEN.

Figure 3G3 gives an interconnection for placing the 8257 on the system bus. By using a separate latch to hold the upper address from the 8257-5 and connecting the outputs to the address bus as shown, 16-bit DMA transfers are provided. In this configuration, AEN simultaneously enables A0 and $\overline{BHE}$ to allow word transfers. AEN still disables the CPU interface to the command and address busses.

2. MAXIMUM MODE ($\overline{RQ}/\overline{GT}$)

The maximum mode 8086 configuration supports a significantly different protocol for transferring bus control. When viewed with respect to the HOLD/HLDA sequence of the minimum mode, the protocol appears difficult to implement externally. However, it is necessary to understand the intent of the protocol and its purpose within the system architecture.

2.1 Shared System Bus ($\overline{RQ}/\overline{GT}$ Alternative)

The maximum mode $\overline{RQ}/\overline{GT}$ sequence is intended to transfer control of the CPU local bus between the CPU and alternate bus masters which reside totally on the local bus and share the complete CPU interface to the system bus. The complete interface includes the address latches, data transceivers, 8288 bus controller and 8289 multi master bus arbiter. If the alternate bus masters in the system do not reside directly on the 8086 local bus, system bus arbitration is required rather than local CPU bus arbitration. To satisfy the need for multimaster system bus arbitration at each CPU's system interface, the 8289 bus arbiter should be used rather than the CPU $\overline{RQ}/\overline{GT}$ logic.

To allow a device with a simple HOLD/HLDA protocol to gain control of a single CPU system bus, the circuit in Figure 3G4 could be used. The design is effectively a simple bus arbiter which isolates the CPU from the system bus when an alternate bus master issues a HOLD request. The output of the circuit, $\overline{AEN}$ (Address ENable), disables the 8288 and 8284 when the 8086 indicates idle status ($\overline{S0}$,$\overline{S1}$,$\overline{S2}$ = 1), $\overline{LOCK}$ is not active and a HOLD request is active. With $\overline{AEN}$ inactive, the 8288 three-states the command outputs and disables DEN

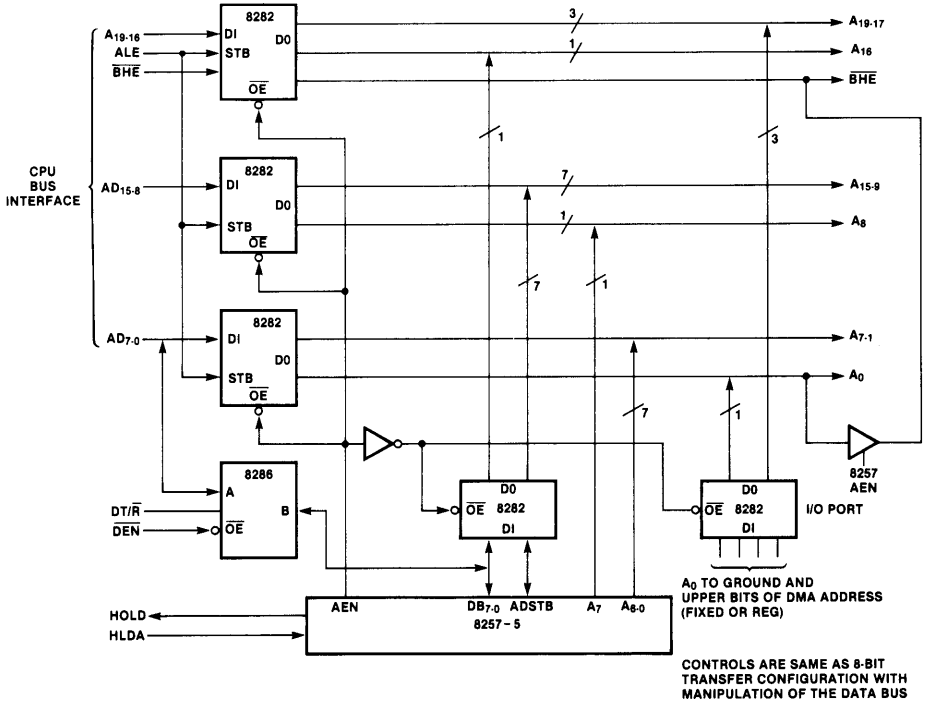


Figure 3G3. 8086 Min System, 8257 on System Bus 16-Bit Transfers

which three-states the data bus transceivers. $\overline{AEN}$ must also three-state the address latch (8282 or 8283) outputs. These actions remove the 8086 from the system bus and allow the requesting device to drive the system bus. The $\overline{AEN}$ signal to the 8284 disables the ready input and forces a bus cycle initiated by the 8086 to wait until the 8086 regains control of the system bus. The CPU may actively drive its local bus during this interval.

The requesting device will not gain control of the bus during an 8086 initiated bus cycle, a locked instruction or an interrupt acknowledge cycle. The $\overline{LOCK}$ signal from the 8086 is active between $\overline{INTA}$ cycles to guarantee the CPU maintains control of the bus. Unlike the minimum mode 8086 HOLD response, this arbitration circuit allows the requestor to gain control of the bus between consecutive bus cycles which transfer a word operand on an odd address boundary and are not locked. Depending on the characteristics of the requesting device, any of the 74LS74 outputs can be used to generate a HLDA to the device.

Upon completion of its bus operations, the alternate bus master must relinquish control of the system bus and drop the HOLD request. After $\overline{AEN}$ goes inactive, the address latches and data transceivers are enabled but, if a CPU initiated bus cycle is pending, the 8288 will not drive the command bus until a minimum of 105 ns or maximum of 275 ns later. If the system is normally not ready, the 8284 $\overline{AEN}$ input may immediately be enabled with ready returning to the CPU when the selected device completes the transfer. If the system is normally ready, the 8284 $\overline{AEN}$ input must be delayed long enough to provide access time equivalent to a normal bus cycle. The 74LS74 latches in the design provide a minimum of TCLCHmin for the alternate device to float the system bus after releasing HOLD. They also provide 2TCLCL ns address access and 2TCLCL – TAEVCHmax ns (8288 command enable delay) command access prior to enabling 8284 ready detection. If HLDA is generated as shown in Figure 3G4, TCLCL ns are available for the 8086 to release the bus prior to issuing HLDA while HLDA is dropped almost immediately upon loss of HOLD.

A circuit configuration for an 8257-5 using this technique to interface with a maximum mode 8086 can be derived from Figure 3G3. The 8257-5 has its own address latch for buffering the address lines A15-A8 and uses its $\overline{AEN}$ output to enable the latch onto the address bus. The maximum latency from HOLD to HLDA for this circuit is dependent on the state of the system when the HOLD is issued. For an idle system the maximum delay is the propagation delay through the nand gate and R/S flip-flop (TD1) plus 2TCLCL plus TCLCHmax plus propagation delay of the 74LS74 and 74LS02 (TD2). For a locked instruction it becomes: TD1 + TD2 + (M + 2) *TCLCL + TCLCHmax where M is the number of clocks required for execution of the locked instruction. For the interrupt acknowledge cycle the latency is TD1 + TD2 + 9 *TCLCL + TCLCHmax.

2.2 Shared Local Bus ($\overline{RQ}/\overline{GT}$ Usage)

The $\overline{RQ}/\overline{GT}$ protocol was developed to allow up to two instruction set extension processors (co-processors) or other special function processors (like the 8089 I/O processor in local mode) to reside directly on the 8086 local bus. Each $\overline{RQ}/\overline{GT}$ pin of the 8086 supports the full protocol for exchange of bus control (Fig. 3G5). The sequence consists of a request from the alternate bus master to gain control of the system bus, a grant from the CPU to indicate the bus has been relinquished and a release pulse from the alternate master when done. The two $\overline{RQ}/\overline{GT}$ pins ($\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$) are prioritized with $\overline{RQ}/\overline{GT0}$ having the highest priority. The prioritization only occurs if requests have been received on both pins before a response has been given to either. For example, if a request is received on $\overline{RQ}/\overline{GT1}$ followed by a request on $\overline{RQ}/\overline{GT0}$ prior to a grant on $\overline{RQ}/\overline{GT1}$, $\overline{RQ}/\overline{GT0}$ will gain priority over $\overline{RQ}/\overline{GT1}$. However, if $\overline{RQ}/\overline{GT1}$ had already received a grant, a request on $\overline{RQ}/\overline{GT0}$ must wait until a release pulse is received on $\overline{RQ}/\overline{GT1}$.

The request/grant sequence interaction with the bus interface unit is similar to HOLD/HLDA. The CPU continues to execute until a bus transfer for additional instructions or data is required. If the release pulse is

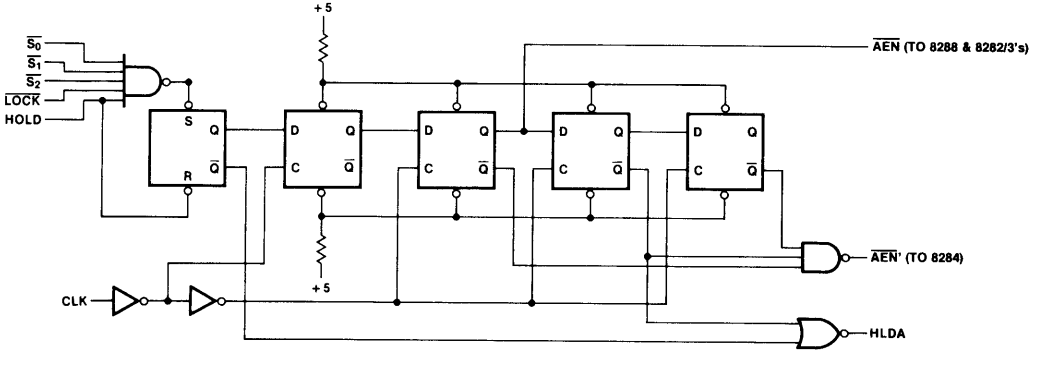


Figure 3G4. Circuit to Translate HOLD into AEN Disable for Max Mode 8086

received before the CPU needs the bus, it will not drive the bus until a transfer is required.

Upon receipt of a request pulse, the 8086 floats the multiplexed address, data and status bus, the $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ status lines, the $\overline{LOCK}$ pin and $\overline{RD}$. This action does not disable the 8288 command outputs from driving the command bus and does not disable the address latches from driving the address bus. The 8288 contains internal pull-up resistors on the $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$ status lines to maintain the passive state while the 8086 outputs are three-state. The passive state prevents the 8288 from initiating any commands or activating DEN to enable the transceivers buffering the data bus. If the device issuing the $\overline{RQ}$ does not use the 8288, it must disable the 8288 command outputs by disabling the 8288 $\overline{AEN}$ input. Also, address latches not used by the requesting device must be disabled.

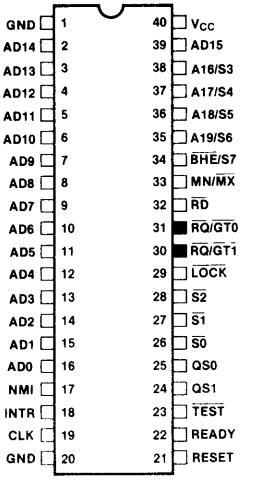


Figure 3G5. 8086 RQ/GT Connections

2.3 $\overline{RQ}/\overline{GT}$ Operation

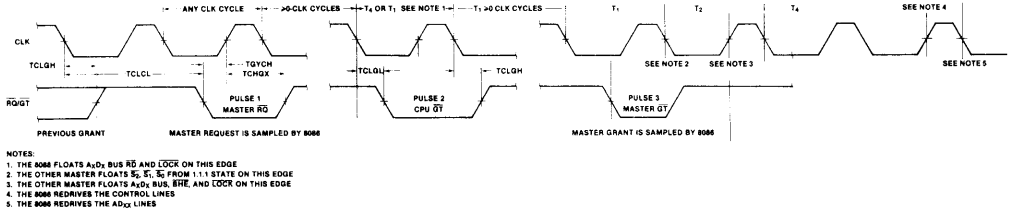
Detailed timing of the $\overline{RQ}/\overline{GT}$ sequence is given in Figure 3G6. To request a transfer of bus control via the $\overline{RQ}/\overline{GT}$ lines, the device must drive the line low for no more than one CPU clock interval to generate a request pulse. The pulse must be synchronized with the CPU clock to guarantee the appropriate setup and hold times to the clock edge which samples the $\overline{RQ}/\overline{GT}$ lines in the CPU. After issuing a request pulse, the device must begin sampling for a grant pulse with the next low to high clock edge. Since the 8086 can respond with a grant pulse in the clock cycle immediately following the request, the $\overline{RQ}/\overline{GT}$ line may not return to the positive level between the request and grant pulses. Therefore edge triggered logic is not valid for capturing a grant pulse. It also implies the circuitry which generates the request pulse must guarantee the request is removed in time to detect a grant from the CPU. After receiving the grant pulse, the requesting device may drive the local bus. Since the 8086 does not float the address and data bus, $\overline{LOCK}$ or $\overline{RD}$ until the high to low clock transition following the low to high clock transition the requestor uses to sample for the grant, the requestor should wait the float delay of the 8086 (TCLAZ) before driving the local bus. This precaution prevents bus contention during the access of bus control by the requestor.

To return control of the bus to the 8086, the alternate bus master relinquishes bus control and issues a release pulse on the same $\overline{RQ}/\overline{GT}$ line. The 8086 may drive the $\overline{S0}$-$\overline{S2}$ status lines, $\overline{RD}$ and $\overline{LOCK}$, three clock cycles after detecting the release pulse and the address/data bus TCHCLmin ns (clock high time) after the status lines. The alternate bus master should be three-stated off the local bus and have other 8086 interface circuits (8288 and address latches) re-enabled within the 8086 delay to regain control of the bus.

2.4 $\overline{RQ}/\overline{GT}$ Latency

The $\overline{RQ}$ to $\overline{GT}$ latency for a single $\overline{RQ}/\overline{GT}$ line is similar to the HOLD to HLDA latency. The cases given for the minimum mode 8086 also apply to the maximum mode. For each case the delay from $\overline{RQ}$ detection by the CPU to $\overline{GT}$ detection by the requestor is:

(HOLD to HLDA delay) – (THVCH + TCHCL + TCLHAV)



NOTES:
1. THE 8086 FLOATS A_XD_X BUS $\overline{RD}$ AND $\overline{LOCK}$ ON THIS EDGE
2. THE OTHER MASTER FLOATS $\overline{S_X}$, $\overline{S_Y}$, $\overline{S_Z}$ FROM 1 1 1 STATE ON THIS EDGE
3. THE OTHER MASTER FLOATS A_XD_X BUS, $\overline{BHE}$, AND $\overline{LOCK}$ ON THIS EDGE
4. THE 8086 REDRIVES THE CONTROL LINES
5. THE 8086 REDRIVES THE AD_{XX} LINES

Figure 3G6. Request/Grant Sequence

This gives a clock cycle maximum delay for an idle bus interface. All other cases are the minimum mode result minus 476 ns. If the 8086 has previously issued a grant on one of the $\overline{RQ}/\overline{GT}$ lines, a request on the other $\overline{RQ}/\overline{GT}$ line will not receive a grant until the first device releases the interface with a release pulse on its $\overline{RQ}/\overline{GT}$ line. The delay from release on one $\overline{RQ}/\overline{GT}$ line to a grant on the other is typically one clock period as shown in Figure 3G7. Occasionally the delay from a release on $\overline{RQ}/\overline{GT1}$ to a grant on $\overline{RQ}/\overline{GT0}$ will take two clock cycles and is a function of a pending request for transfer of control from the execution unit. The latency from request to grant when the interface is under control of a bus master on the other $\overline{RQ}/\overline{GT}$ line is a function of the other bus master. The protocol embodies no mechanism for the CPU to force an alternate bus master off the bus. A watchdog timer should be used to prevent an errant alternate bus master from 'hanging' the system.

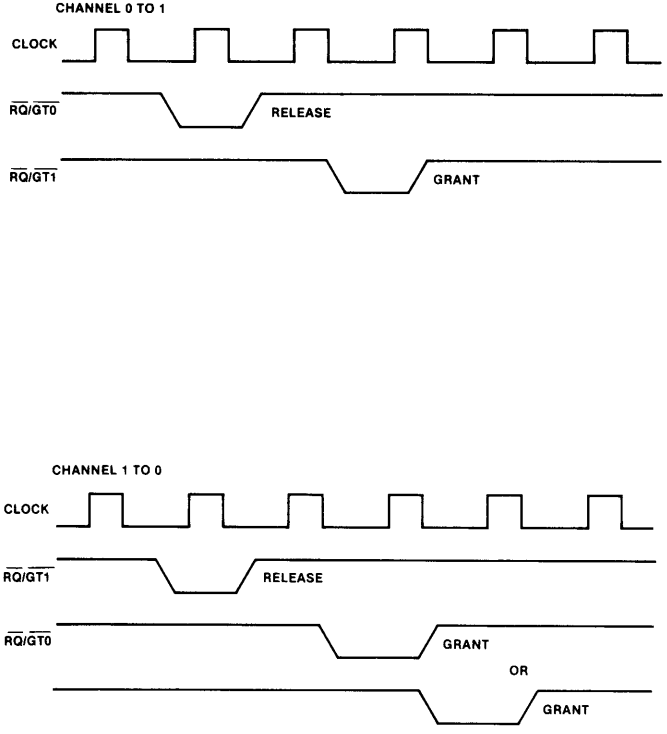


Figure 3G7. Channel Transfer Delay

2.5 RQ/GT to HOLD/HLDA Conversion

A circuit for translating a HOLD/HLDA hand-shake sequence into a $\overline{RQ}/\overline{GT}$ pulse sequence is given in Figure 3G8. After receiving the grant pulse, the HLDA is enabled TCHCLmin ns before the CPU has three-stated the bus. If the requesting circuit drives the bus within 20 ns of HLDA, it may be desirable to delay the acknowledge one clock period. The HLDA is dropped no later than one clock period after HOLD is disabled. The HLDA also drops at the beginning of the release pulse to provide 2TCLCL + TCLCH for the requestor to relinquish control of the status lines and 3TCLCL to float the remaining signals.

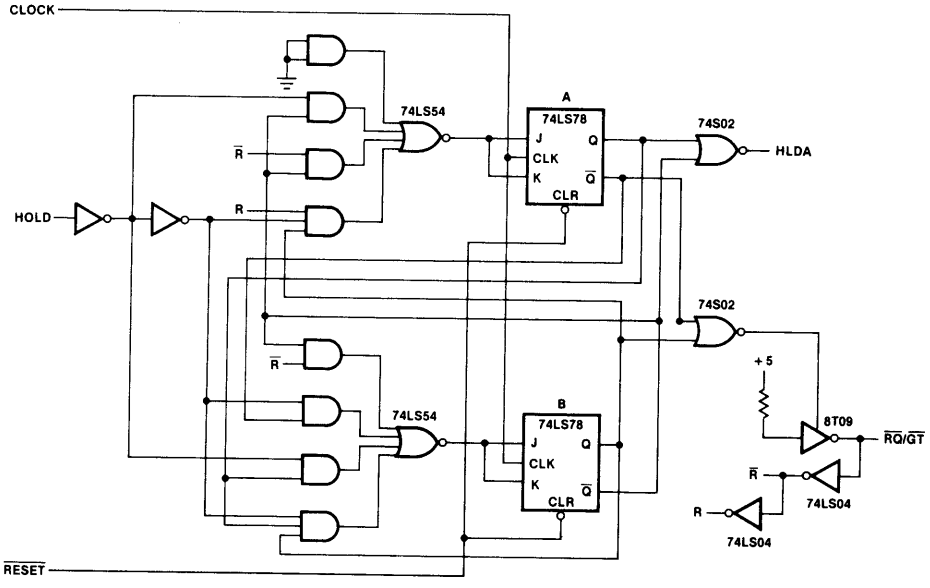


Figure 3G8a. HOLD/HLDA◄►$\overline{RQ}/\overline{GT}$ Conversion Circuit

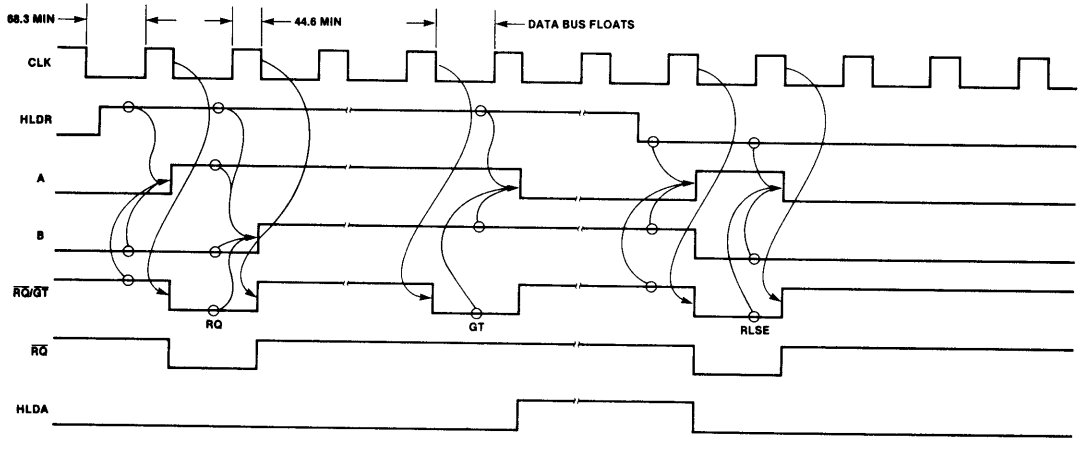


Figure 3G8b. HOLD/HLDA◄►$\overline{RQ}/\overline{GT}$ Conversion Timing

4. INTERFACING WITH I/O

The 8086 is capable of interfacing with 8- and 16-bit I/O devices using either I/O instructions or memory mapped I/O. The I/O instructions allow the I/O devices to reside in a separate I/O address space while memory mapped I/O allows the full power of the instruction set to be used for I/O operations. Up to 64K bytes of I/O mapped I/O may be defined in an 8086 system. To the programmer, the separate I/O address space is only accessible with INPUT and OUTPUT commands which transfer data between I/O devices and the AX (for 16-bit data transfers) or AL (for 8-bit data transfers) register. The first 256 bytes of the I/O space (0 to 255) are directly addressable by the I/O instructions while the entire 64K is accessible via register indirect addressing through the DX register. The later technique is particularly desirable for service procedures that handle more than one device by allowing the desired device address to be passed to the procedure as a parameter. I/O devices may be connected to the local CPU bus or the buffered system bus.

4A. Eight-Bit I/O

Eight-bit I/O devices may be connected to either the upper or lower half of the data bus. Assigning an equal number of devices to the upper and lower halves of the bus will distribute the bus loading. If a device is connected to the upper half of the data bus, all I/O addresses assigned to the device must be odd (A0 = 1). If the device is on the lower half of the bus, its addresses must be even (A0 = 0). The address assignment directs the eight-bit transfer to the upper (odd byte address) or lower (even byte address) half of the sixteen-bit data bus. Since A0 will always be a one or zero for a specific device, A0 cannot be used as an address input to select registers within a specific device. If a device on the upper half of the bus and one on the lower half are assigned addresses that differ only in A0 (adjacent odd and even addresses), A0 and $\overline{BHE}$ must be conditions of chip select decode to prevent a write to one device from erroneously performing a write to the other. Several techniques for generating I/O device chip selects are given in Figure 4A1.

The first technique (a) uses separate 8205's to generate chip selects for odd and even addressed byte peripherals. If a word transfer is performed to an even addressed device, the adjacent odd addressed I/O device is also selected. This allows accessing the devices individually with byte transfers or simultaneously as a 16-bit device with word transfers. Figure 4A1(b) restricts the chip selects to byte transfers, however a word transfer to an odd address will cause the 8086 to run two byte transfers that the decode technique will not detect. The third technique simply uses a single 8205 to generate odd and even device selects for byte transfers and will only select the even addressed eight-bit device on a word transfer to an even address.

If greater than 256 bytes of the I/O space or memory mapped I/O is used, additional decoding beyond what is shown in the examples may be necessary. This can be done with additional TTL, 8205's or bipolar PROMs (Intel's 3605A). The bipolar PROMs are slightly slower than multiple levels of TTL (50 ns vs 30 to 40 ns for TTL) but provide full decoding in a single package and allow inserting a new PROM to reconfigure the system I/O map without circuit board or wiring modifications (Fig. 4A2).

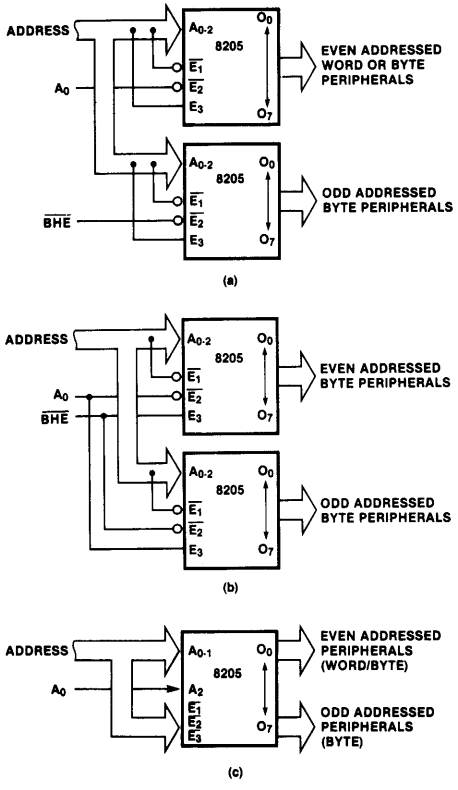


Figure 4A1. Techniques for I/O Device Chip Selects

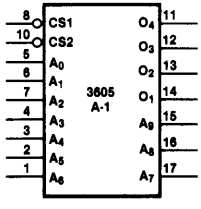


Figure 4A2. Bipolar PROM Decoder

One last technique for interfacing with eight-bit peripherals is considered in Figure 4A3. The sixteen-bit data bus is multiplexed onto an eight-bit bus to accommodate byte oriented DMA or block transfers to memory mapped eight-bit I/O. Devices connected to this interface may be assigned a sequence of odd and even addresses rather than all odd or even.

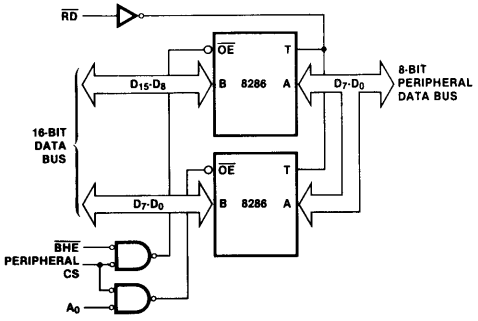
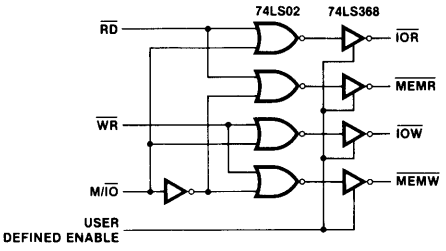


Figure 4A3. 16- to 8-Bit Bus Conversion



NOTE: IF IT IS NOT NECESSARY TO THREE-STATE THE COMMAND LINES, A DECODER (8205 OR 74S138) COULD BE USED. THE 74LS257 IS NOT RECOMMENDED SINCE THE OUTPUTS MAY EXPERIENCE VOLTAGE SPIKES WHEN ENTERING OR LEAVING THREE-STATE.

Figure 4C1. Decoding Memory and I/O RD and WR Commands for Minimum Mode 8086 Systems

4B. Sixteen-Bit I/O

For obvious reasons of efficient bus utilization and simplicity of device selection, sixteen-bit I/O devices should be assigned even addresses. To guarantee the device is selected only for word operations, A0 and BHE should be conditions of chip select code (Fig. 4B1).

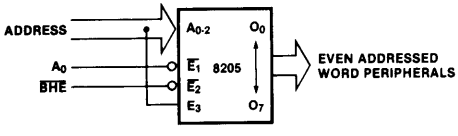


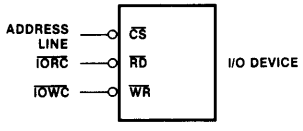
Figure 4B1. Sixteen-Bit I/O Decode

4C. General Design Considerations

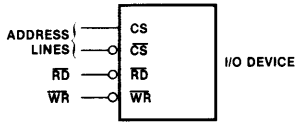
MIN/MAX, MEMORY I/O MAPPED AND LINEAR SELECT

Since the minimum mode 8086 has common read and write commands for memory and I/O, if the memory and I/O address spaces overlap, the chip selects must be qualified by M/IO to determine which address space the devices are assigned to. This restriction on chip select decoding can be removed if the I/O and memory addresses in the system do not overlap and are properly decoded; all I/O is memory mapped; or RD, WR and M/IO are decoded to provide separate memory and I/O read/write commands (Fig. 4C1). The 8288 bus controller in the maximum mode 8086 system generates separate I/O and memory commands in place of a M/IO signal. An I/O device is assigned to the I/O space or memory space (memory mapped I/O) by connection of either I/O or memory command lines to the command inputs of the device. To allow overlap of the memory and I/O address space, the device must not respond to chip select alone but must require a combination of chip select and a read or write command.

Linear select techniques (Fig. 4C2) for I/O devices can only be used with devices that either reside in the I/O address space or require more than one active chip select (at least one low active and one high active). Devices with a single chip select input cannot use linear select if they are memory mapped. This is due to the assignment of memory address space FFFFF0H-FFFFFFH to reset startup and memory space 00000H-003FFH to interrupt vectors.



(a) SEPARATE I/O COMMANDS



(b) MULTIPLE CHIP SELECTS

Figure 4C2. Linear Select for I/O

4D. Determining I/O Device Compatibility

This section presents a set of A.C. characteristics which represent the timing of the asynchronous bus interface of the 8086. The equations are expressed in terms of the CPU clock (when applicable) and are derived for minimum and maximum modes of the 8086. They represent the bus characteristics at the CPU.

The results can be used to determine I/O device requirements for operation on a single CPU local bus or buffered system bus. These values are not applicable to

a Multibus system bus interface. The requirements for a Multibus system bus are available in the Multibus interface specification.

A list of bus parameters, their definition and how they relate to the A.C. characteristics of Intel peripherals are given in Table 4D1. Cycle dependent values of the parameters are given in Table 4D2. For each equation, if more than one signal path is involved, the equation reflects the worst case path.

ex. TAVRL(address valid before read active) =
(1) Address from CPU to $\overline{RD}$ active
(or)
(2) ALE (to enable the address through the address latches) to $\overline{RD}$ active

The worst case delay path is (1).

For the maximum mode 8086 configurations, TAVWLA, TWLWHA and TWLCLA are relative to the advanced write signal while TAVWL, TWLWH and TWLCL are relative to the normal write signal.

TABLE 4D1. PARAMETERS FOR PERIPHERAL COMPATIBILITY

TAVRL — Address stable before RD leading edge	(TAR)
TRHAX — Address hold after RD trailing edge	(TRA)
TRLRH — Read pulse width	(TRR)
TRLDV — Read to data valid delay	(TRD)
TRHDZ — Read trailing edge to data floating	(TDF)
TAVDV — Address to valid data delay	(TAD)
TRLRL — Read cycle time	(TRCYC)
TAVWL — Address valid before write leading edge	(TAW)
TAVWLA — Address valid before advanced write	(TAW)
TWHAX — Address hold after write trailing edge	(TWA)
TWLWH — Write pulse width	(TWW)
TWLWHA — Advanced write pulse width	(TWW)
TDVWH — Data set up to write trailing edge	(TDW)
TWHDX — Data hold from write trailing edge	(TWD)
TWLCL — Write recovery time	(TRV)
TWLCLA — Advanced write recovery time	(TRV)
TSVRL — Chip select stable before RD leading edge	(TAR)
TRHSX — Chip select hold after RD trailing edge	(TRA)
TSLDV — Chip select to data valid delay	(TRD)
TSVWL — Chip select stable before WR leading edge	(TAW)
TWHSX — Chip select hold after WR trailing edge	(TWA)
TSVWLA — Chip select stable before advanced write	(TAW)

Symbols in parentheses are equivalent parameters specified for Intel peripherals.

In the given list of equations, TWHDXB is the data hold time from the trailing edge of write for the minimum mode with a buffered data bus. For this equation, TCVCTX cannot be a minimum for data hold and a maximum for write inactive. The maximum difference is 50 ns giving the result TCLCH-50. If the reader wishes to verify the equations or derive others, refer to Section 3F for assistance with interpreting the 8086 bus timing diagrams.

Figure 4D1 shows four representative configurations and the compatible Intel peripherals (including wait states if required) for each configuration are given in Table 4D3. Configuration 1 and 2 are minimum mode demultiplexed bus 8086 systems without (1) and with (2) data bus transceivers. Configurations 3 and 4 are maximum mode systems with one (3) and two (4) levels of address and data buffering. The last configuration is characteristic of a multi-board system with bus buffers on each board. The 5 MHz parameter values for these configurations are given in Table 4D4 and demonstrate the relaxed device requirements for even a large complex configuration. The analysis assumes all components are exhibiting the specified worst case parameter values and are under the corresponding temperature, voltage and capacitive load conditions. If the capacitive loading on the 8282/83 or 8286/87 is less than the maximum, graphs of delay vs. capacitive loading in the respective data sheets should be used to determine the appropriate delay values.

TABLE 4D2. CYCLE DEPENDENT PARAMETER REQUIREMENTS FOR PERIPHERALS

(a) Minimum Mode

TAVRL = TCLCL + TCLRLmin – TCLAVmax = TCLCL – 100
TRHAX = TCLCL – TCLRHmax + TCLLHmin = TCLCL – 150
TRLRH = 2TCLCL – 60 = 2TCLCL – 60
TRLDV = 2TCLCL – TCLRLmax – TDVCLmin = 2TCLCL – 195
TRHDZ = TRHAVmin = 155 ns
TAVDV = 3TCLCL – TDVCLmin – TCLAVmax = 3TCLCL – 140
TRLRL = 4TCLCL = 4TCLCL
TAVWL = TCLCL + TCVCTVmin – TCLAVmax = TCLCL – 100
TWHAX = TCLCL + TCLLHmin – TCVCTXmax = TCLCL – 110
TWLWH = 2TCLCL – 40 = 2TCLCL – 40
TDVWH = 2TCLCL + TCVCTXmin – TCLDVmax = 2TCLCL – 100
TWHDX = TWHDZmin = 89
TWLCL = 4TCLCL = 4TCLCL
TWHDXB = TCLCHmin + (– TCVCTXmax + TCVCTXmin) = TCLCHmin – 50

Note: Delays relative to chip select are a function of the chip select decode technique used and are equal to: equivalent delay from address – chip select decode delay.

(b) Maximum Mode

TAVRL = TCLCL + TCLMLmin – TCLAVmax = TCLCL – 100
TRHAX = TCLCL – TCLMHmax + TCLLHmin = TCLCL – 40
TRLRH = 2TCLCL – TCLMLmax + TCLMHmin = 2TCLCL – 25
TRLDV = 2TCLCL – TCLMLmax – TDVCLmin = 2TCLCL – 65
TRHDZ = TRHAVmin = 155
TAVDV = 3TCLCL – TDVCLmin – TCLAVmax = 3TCLCL – 140
TRLRL = 4TCLCL = 4TCLCL
TAVWLA = TAVRL = TCLCL – 100
TAVWL = TAVRL + TCLCL = 2TCLCL – 100
TWHAX = TRHAX = TCLCL – 40
TWLWHA = TRLRH = 2TCLCL – 25
TWLWH = TRLRH – TCLCL = TCLCL – 25
TDVWH = 2TCLCL + TCLMHmin – TCLDVmax = 2TCLCL – 100
TWHDX = TCLCHmin – TCLMHmax + TCHDZmin = TCLCHmin – 30
TWLCL = 3TCLCL = 3TCLCL
TWLCLA = 4TCLCL = 4TCLCL

TABLE 4D3. COMPATIBLE PERIPHERALS (5 MHz 8086)

	Configuration			
	Minimum Mode		Maximum Mode	
	Unbuffered	Buffered	Buffered	Fully Buffered
8251A	✓	1W	✓	✓
8253-5	✓	1W	✓	✓
8255A-5	✓	1W	✓	✓
8257-5	✓	1W	✓	✓
8259A	✓	✓	✓	✓
8271	✓	1W	✓	✓
8273	✓	1W	✓	✓
8275	✓	1W	✓	✓
8279-5	✓	1W	✓	✓
8041A*	✓	1W	✓	✓
8741A	✓	1W	✓	✓
8291	✓	✓	✓	✓

*Includes other Intel peripherals based on the 8041A (i.e., 8292, 8294, 8295).

✓ implies full operation with no wait states.

W implies the number of wait states required.

TABLE 4D4. PERIPHERAL REQUIREMENTS FOR FULL SPEED OPERATION WITH 5 MHz 8086

	Configuration			
	Minimum Mode		Maximum Mode	
	Unbuffered	Buffered	Buffered	Fully Buffered
TAVRL	70	72	70	58
TRHAX	57	27	169	141
TRLRH	340	320	375	347
TRLDV	205	150	305	261
TRHDZ	155	158	382	360
TAVDV	430	400	400	372
TRLRL	800	770	800	772
TAVWL	70	72	270	258
TAVWLA	—	—	70	58
TWHAX	97	67	169	141
TWLWH	360	340	175	147
TWLWHA	—	—	375	347
TDVWH	300	339	270	258
TWHDX	88	15	95	13
TWLCL	800	772	600	572
TWLCLA	—	—	800	772
TSVRL	52	54	52	40
TRHSX	50	50	171	143
TSLDV	412	382	382	354
TSVWL	52	54	252	240
TWHSX	90	90	171	143
TSVWLA	—	—	52	40

— Not applicable.

Peripheral compatibility is determined from the equations given for the CPU by modifying them to account for additional delays from address latches and data transceivers in the configuration. Once the system configuration is selected, the system requirements can be determined at the peripheral interface and used to evaluate compatibility of the peripheral to the system. During this process, two areas must be considered. First, can the device operate at maximum bus bandwidth and if not, how many wait states are required. Second, are there any problems that cannot be resolved by wait states.

Examples of the first are TRLRH (read pulse width) and TRLDV (read access or $\overline{RD}$ active to output data valid). Consider address access time (valid address to valid data) for the maximum mode fully buffered configuration.

TAVDV = 3TCYC – 140 ns — address latch delay — address buffer delay — chip select decode delay — 2 transceiver delays

Assuming inverting latches, buffers and transceivers with 22 ns max delays (8283, 8287) and a bipolar PROM decode with 50 ns delay, the result is:

TAVDV = 322 ns @ 5 MHz

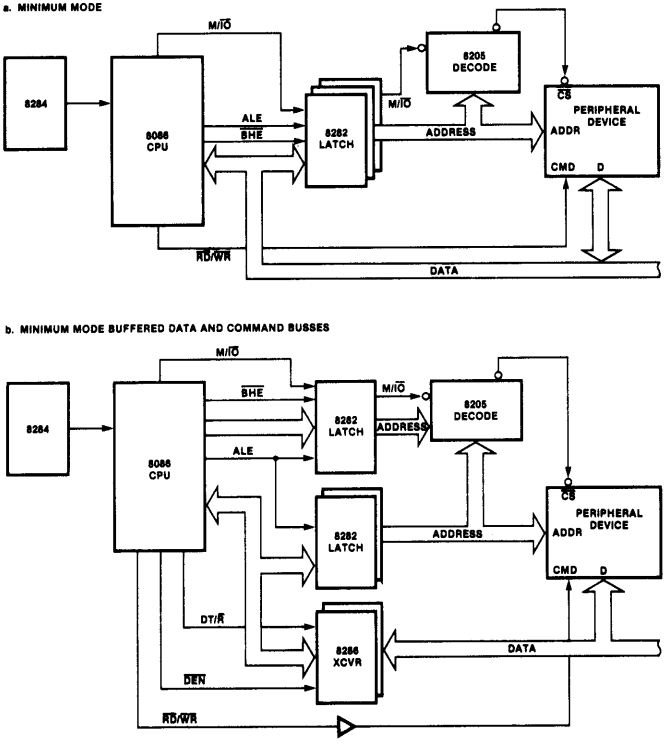
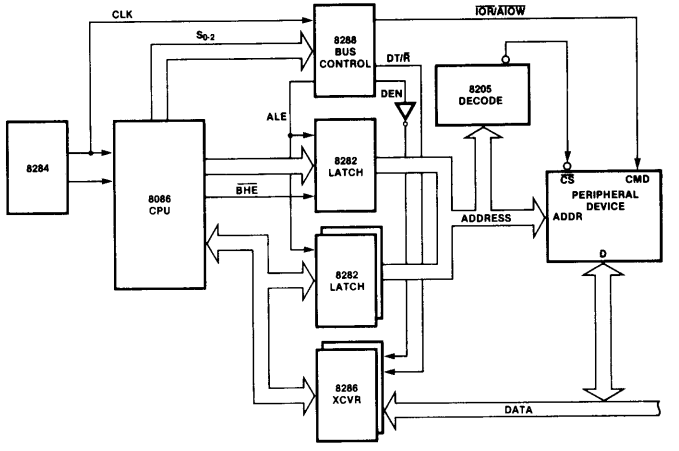


Figure 4D1. 8086 System Configurations

c. MAXIMUM MODE BUFFERED DATA BUS



NOTE: FOR OPTIMUM PERFORMANCE WITH INTEL PERIPHERALS, $\overline{AIOW}$ (ADVANCED WRITE) SHOULD BE USED.

d. MAXIMUM MODE DOUBLE BUFFERED SYSTEM

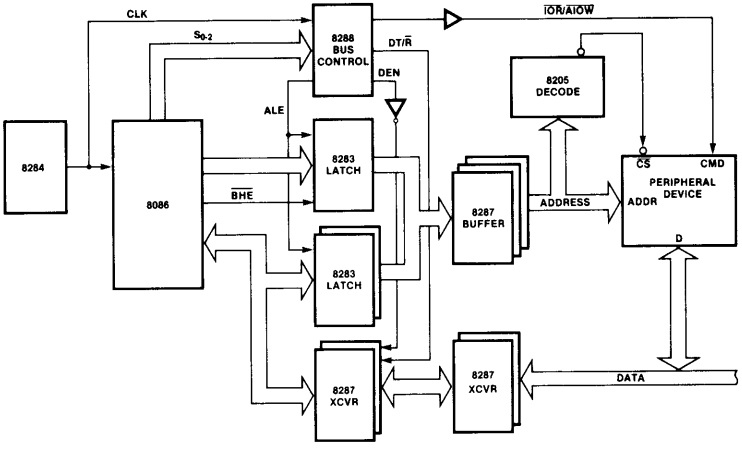


Figure 4D1. 8086 System Configurations (Con't)

The result gives the address to data valid delay required at the peripheral (in this configuration) to satisfy zero wait state CPU access time. If the maximum delay specified for the peripheral is less than the result, this parameter is compatible with zero wait state CPU operation. If not, wait states must be inserted until TAVDV + n * TCYC (n is the number of wait states) is greater than the peripherals maximum delay. If several parameters require wait states, either the largest number required should always be used or different transfer cycles can insert the maximum number required for that cycle.

The second area of concern includes TAVRL (address set up to read) and TWHDX (data hold after write). Incompatibilities in this area cannot be resolved by the insertion of wait states and may require either additional hardware, slowing down the CPU (if the parameter is related to the clock) or not using the device.

As an example consider address valid prior to advanced write low (TAVWLA) for the maximum mode fully buffered system.

TAVWLA = TCYC – 100 ns — address latch delay — address buffer delay — chip select decode delay + write buffer delay (minimum)

Assuming inverting latches and buffers with 22 ns delay (8283, 8287) and an 8205 address decoder with 18 ns delay

TAVWLA = 38 ns which is the time a 5 MHz 8086 system provides

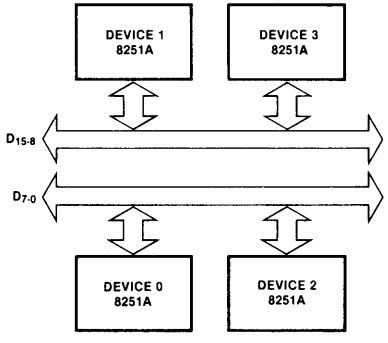
4E. I/O Examples

1. Consider an interrupt driven procedure for handling multiple communication lines. On receiving an interrupt from one of the lines, the invoked procedure polls the lines (reading the status of each) to determine which line to service. The procedure does not enable lines but simply services input and output requests until the associated output buffer is empty (for output requests) or until an input line is terminated (for the example only EOT is considered). On detection of the terminate condition, the routine will disable the line. It is assumed that other routines will fill a lines output buffer and enable the device to request output or empty the input buffer and enable the device to input additional characters.

The routine begins operation by loading CX with a count of the number of lines in the system and DX with the I/O address of the first line. The I/O addresses are assigned as shown in Figure 4E1 with 8251A's as the I/O devices. The status of each line is read to determine if it needs service. If yes, the appropriate routine is called to input or output a character. After servicing the line or if no service is needed, CX is decremented and DX is incremented to test the next line. After all lines have been tested and serviced, the routine terminates. If all interrupts from the lines are OR'd together, only one interrupt is used for all lines. If the interrupt is input to the CPU through an 8259A interrupt controller, the 8259A should be programmed in the level triggered mode to guarantee all line interrupts are serviced.

To service either an input or output request, the called routine transfers DX to BX, and shifts BX to form the offset for this device into the table of input or output buffers. The first entry in the buffer is an index to the next character position in the buffer and is loaded into the SI register. By specifying the base address of the table of buffers as a displacement into the data segment, the base + index + displacement addressing mode allows direct access to the appropriate memory location. 8086 code for part of this example is shown in Figure 4E2.

2. As a second example, consider using memory mapped I/O and the 8086 string primative instructions to perform block transfers between memory and I/O. By assigning a block of the memory address space (equivalent in size to the maximum block to be transferred to the I/O device) and decoding this address space to generate the I/O device's chip select, the block transfer capability is easily implemented. Figure 4E3 gives an interconnect for 16-bit I/O devices while Figure 4E4 incorporates the 16-bit bus to 8-bit bus multiplexing scheme to support 8-bit I/O devices. A code example to perform such a transfer is shown in Figure 4E5.



DEVICES ARE CONNECTED TO THE UPPER AND LOWER HALVES OF THE DATA BUS.

ADDRESS		
0	DEVICE 0	DATA
1	DEVICE 1	DATA
2	DEVICE 0	CONTROL/STATUS
3	DEVICE 1	CONTROL/STATUS
4	DEVICE 2	DATA
5	DEVICE 3	DATA
6	DEVICE 2	CONTROL/STATUS
7	DEVICE 3	CONTROL/STATUS
ETC.	"	"

Figure 4E1. Device Assignment

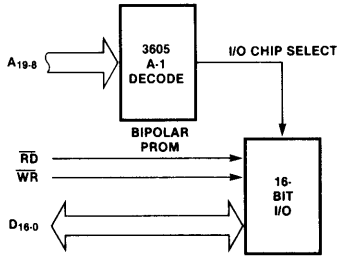
```
; THIS CODE DEMONSTRATES TESTING DEVICE
; STATUS FOR SERVICE, CONSTRUCTING THE
; APPROPRIATE LINE BUFFER ADDRESS FOR INPUT
; AND OUTPUT AND SERVICING AN INPUT
; REQUEST

                MASK EQU OFFFDH
CHECK_STATUS:   INPUT  AL, DX           ; GET 8251A STATUS.
                MOV    AH, AL
                TEST   AH, READ_OR_WRITE_STATUS
                JZ     NEXT_IO
                CALL   ADDRESS
                TEST   AH, READ STATUS
                JZ     WRITE_SERVICE
                CALL   READ
                TEST   AH, WRITE STATUS
                JZ     NEXT_IO
WRITE_SERVICE:  CALL   WRITE
NEXT_IO:        DEC    CX               ; TEST IF DONE.
                JNC    EXIT             ; YES, RESTORE & RETURN.
                AND    DX, MASK         ; REMOVE A1 AND
                ADD    DX, 3            ; INCREMENT ADDRESS.
                OR     DX, 2            ; SELECT STATUS FOR
                JMP    CHECK_STATUS     ; NEXT INPUT.

ADDRESS:        AND    DX, MASK         ; SELECT DATA.
                MOV    BH, DL           ; CONSTRUCT BUFFER
                INC    BH               ; DISPLACEMENT FOR
                SHR    BH               ; THIS DEVICE.
                XOR    BL, BL           ; BX IS THE DISPLACEMENT.
                RET

READ:           INPUT AL, DX                        ; READ CHARACTER.
                MOV SI, READ_BUFFERS [BX]           ; GET CHARACTER POINTER.
                MOV READ_BUFFERS [BX + SI], AL      ; STORE CHARACTER.
                INC READ_BUFFERS [BX]               ; INCR CHARACTER POINTER.
                CMP AL, EOT                         ; END OF TRANSMISSION?
                JNZ CONT_READ
                CALL DISABLE READ                   ; YES, DISABLE RECEIVER.
                CONT_READ: RET                      ; SEND MESSAGE THAT INPUT
                                                    ; IS READY.
```

Figure 4E2.



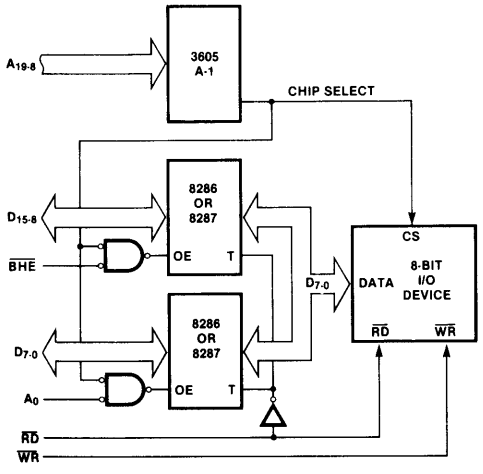
TRANSFER 256 BYTE BLOCKS TO THE I/O DEVICE

THE ADDRESS SPACE ASSIGNED TO THE I/O DEVICE IS

	A_{19}	A_8 | A_7	A_0
FROM	← BASE	ADDRESS →	← 0's →
THRU	← BASE	ADDRESS →	← 1's →

MEMORY DATA NEED NOT BE ALIGNED TO EVEN ADDRESS BOUNDARIES
I/O TRANSFERS MUST BE WORD TRANSFERS TO EVEN ADDRESS BOUNDARIES

Figure 4E3. Block Transfer to 16-Bit I/O Using 8086 String Primatives



ADDRESS ASSIGNMENT SAME AS PREVIOUS EXAMPLE. 16-BIT BUS IS MULTIPLEXED ONTO AN 8-BIT PERIPHERAL BUS.

Figure 4E4. Block Transfer to 8-Bit I/O Using 8086 String Primatives

```
                ; DEFINE THE I/O ADDRESS SPACE
                  I/O SEGMENT
                  ORG BLOCK_ADDRESS
I/O_BLOCK:        DW 128 DUP (?)
                  I/O ENDS

                ; ASSUME THE DATA IS FROM THE CURRENT
                ; DATA SEGMENT
                  CLD                          ; DF = FORWARD
                  LES DI, I/O_BLOCK_ADDRESS    ; I/O BLOCK ADDRESS
                                               ; CONTAINS THE ADDRESS
                                               ; OF I/O BLOCK
                  MOV CX, BLOCK_LENGTH
                  MOV SI, SOURCE_ADDRESS
                  MOVS I/O BLOCK               ; PERFORM WORD TRANSFERS

                ; END CODE EXAMPLE
```

NOTE THE CODE IS CAPABLE OF PERFORMING BYTE TRANSFERS BY CHANGING THE I/O BLOCK DEFINITION FROM 128 WORD TO 256 BYTES

Figure 4E5. Code for Block Transfers

5. INTERFACING WITH MEMORIES

Figure 5.1 is a general block diagram of an 8086 memory. The basic characteristics of the diagram are the partitioning of the 16-bit word memory into high and low 8-bit banks on the upper and lower halves of the data bus and inclusion of $\overline{BHE}$ and A0 in the selection of the banks. Specific implementations depend on the type of memory and the system configuration.

5A. ROM and EPROM

The easiest devices to interface to the system are ROM and EPROM. Their byte format provides a simple bus interface and since they are read only devices, A0 and $\overline{BHE}$ need not be included in their chip enable/select decoding (chip enable is similar to chip select but additionally determines if the device is in active or standby power mode). The address lines connected to the devices start with A1 and continue up to the maximum number the device can accept, leaving the remaining address lines for chip enable/select decoding. To connect the devices directly to the multiplexed bus, they must have output enables. The output enable is also necessary to avoid bus contention in other configurations. Figure 5A1 shows the bus connections for ROM and EPROM memories. No special decode techniques are required for generating chip enables/selects. Each valid decode selects one device on the upper and lower halves of bus to allow byte and word access. Byte access is achieved by reading the full word onto the bus with the 8086 only accepting the desired byte. For the minimum mode 8086, if $\overline{RD}$, $\overline{WR}$ and M/$\overline{IO}$ are not decoded to form separate commands for memory and I/O, and the I/O space overlaps the memory space assigned to the EPROM/ROM then M/$\overline{IO}$ (high active) must be a condition of chip enable/select decode. The output enable is controlled by the system memory read signal.

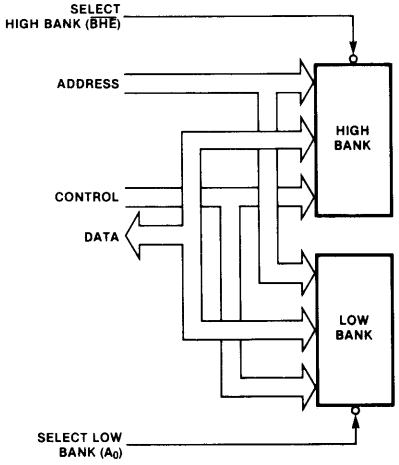
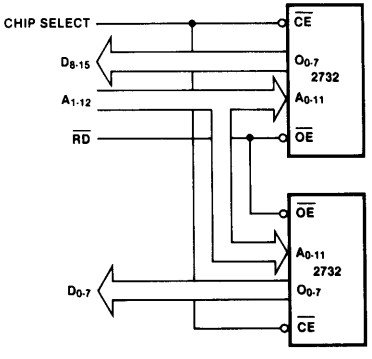


Figure 5.1. 8086 Memory Array



NOTE A_0 AND $\overline{BHE}$ ARE NOT USED.

Figure 5A1. EPROM/ROM Bus Interface

Static ROM's and EPROM's have only four parameters to evaluate when determining their compatibility to the system. The parameters, equations and evaluation techniques given in the I/O section are also applicable to these devices. The relationship of parameters is given in Table 5A1. TACC and TCE are related to the same equation and differ only by the delay associated with the chip enable/select decoder. As an example, consider a 2716 EPROM memory residing on the multiplexed bus of a minimum mode configuration:

TACC = 3TCLCL – 140 – address buffer delay = 430 ns
(8282 = 30 ns max delay)

TCE = TACC – decoder delay = 412 ns
(8205 decoder delay = 18 ns)

TOE = 2TCLCL – 195 = 205 ns

TDF = = 155 ns

TABLE 5A1. EPROM/ROM PARAMETERS

TOE — Output Enable to Valid Data ≡ TRLDV
TACC — Address to Valid Data ≡ TAVDV
TCE — Chip Enable to Valid Data ≡ TSLDV
TDF — Output Enable High to Output Float ≡ TRHDZ

The results are the times the system configuration requires of the component for full speed compatibility with the system. Comparing these times with 2716 parameter limits indicates the 2716-2 will work with no wait states while the 2716 will require one wait state. Table 5A2 demonstrates EPROM/ROM compatibility for the configurations presented in the I/O section. Before designing a ROM or EPROM memory system, refer to AP-30 for additional information on design techniques that give the system an upgrade path from 16K to 32K and 64K devices.

TABLE 5A2. COMPATIBLE EPROM/ROM (5 MHz 8086)

	Configuration			
	Minimum Mode		Maximum Mode	
	Unbuffered	Buffered	Buffered	Fully Buffered
2716-1	✓	✓	✓	✓
2716-2	✓	1W	1W	1W
2732	1W	1W	1W	1W
2332	✓	✓	✓	✓
2364	✓	✓	✓	✓

5B. Static RAM

Interfacing static RAM to the system introduces several new requirements to the memory design. A0 and $\overline{BHE}$ must be included in the chip select/chip enable decoding of the devices and write timing must be considered in the compatibility analysis.

For each device, the data bus connections must be restricted to either the upper or lower half of the data bus. Devices like the 2114 or 2142 must not straddle the upper and lower halves of the data bus (Fig. 5B1). To allow selecting either the upper byte, lower byte or full 16-bit word for a write operation, $\overline{BHE}$ must be a condition of decode for selecting the upper byte and A0 must be a condition of decode for selecting the lower byte. Figure 5B2 gives several selection techniques for devices with single chip selects and no output enables (2114, 2141, 2147). Figure 5B3 gives selection techniques for devices with chip selects and output enables.

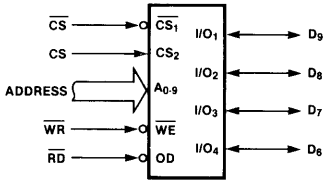


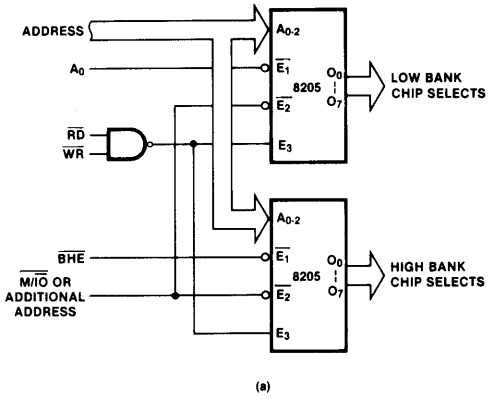
Figure 5B1. Incorrect Connection of 2142 Across Byte Boundaries

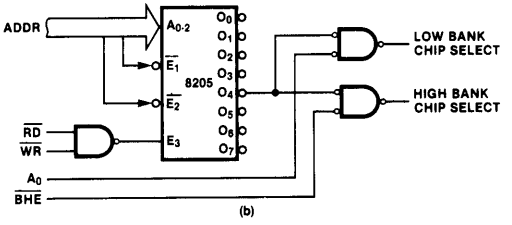
The first group requires inclusion of A0 and $\overline{BHE}$ to decode or enable the chip selects. Since these memories do not have output enables, read and write are used as enables for chip select generation to prevent bus contention. If read and write are not used to enable the chip selects, devices with common input/output pins (like the 2114) will be subjected to severe bus contention between chip select and write active. For devices with separate input/output lines (like 2141, 2147), the outputs can be externally buffered with the buffer enable controlled by read. This solution will only allow bus contention between memory devices in the array during chip select transition periods. These techniques are considered in more detail in Section 2C.

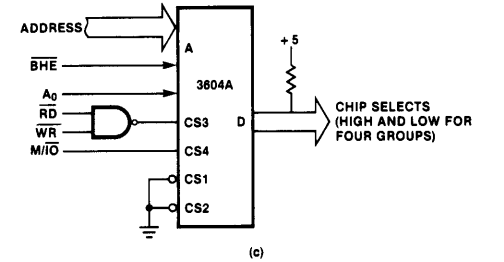
For devices with output enables (2142), write may be gated with $\overline{BHE}$ and A0 to provide upper and lower bank write strobes. This simplifies chip select decoding by eliminating $\overline{BHE}$ and A0 as a condition of decode. Although both devices are selected during a byte write operation, only one will receive a write strobe. No bus contention will exist during the write since a read command must be issued to enable the memory output drivers.

If multiple chip selects are available at the device, $\overline{BHE}$ and A0 may directly control device selection. This allows normal chip select decoding of the address space and direct connection of the read and write commands to the devices. Alternately, the multiple chip select inputs of the device could directly decode the address space (linear select) and be combined with the separate write strobe technique to minimize the control circuitry needed to generate chip selects.

As with the EPROM's and ROM's, if separate commands are not provided for memory and I/O in the minimum mode 8086 and the address spaces overlap, M/$\overline{IO}$ (high active) must be a condition of chip select decode. Also, the address lines connected to the memory devices must start with A1 rather than A0.







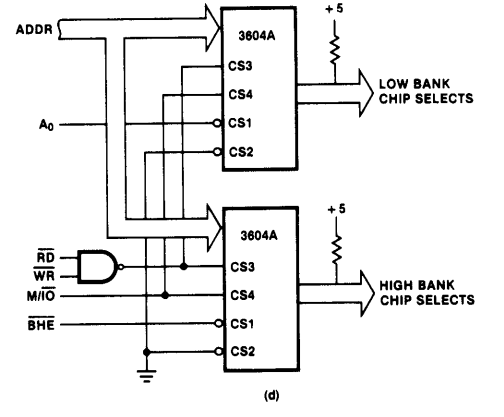
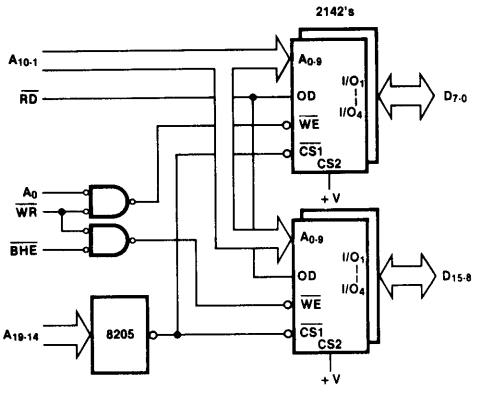
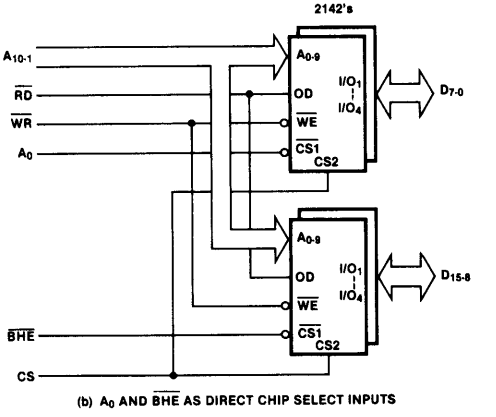


Figure 5B2. Generating Chip Selects for Devices without Output Enables





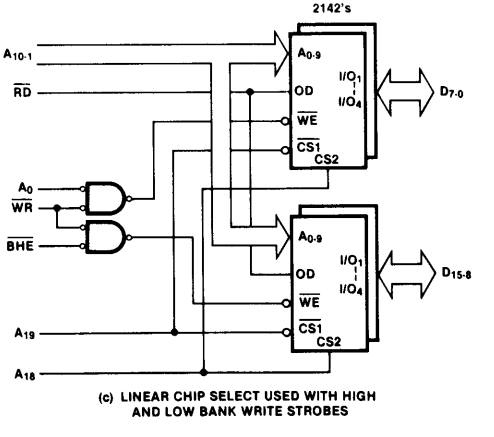


Figure 5B3. Chip Selection for Devices with Output Enables

For analysis of RAM compatibility, the write timing parameters listed in Table 5B1 may also need to be considered (depending on the RAM device being considered). The CPU clock relative timing is given in Table 5B2. The equations specify the device requirements at the CPU and provide a base for determining device requirements in other configurations. As an example consider the write timing requirements of a 2142 in a maximum mode buffered 8086 system (Figure 5B4). The 2142 write parameters that must be analyzed are TWA advanced write pulse width, TWR write release time, TDWA data to write time overlap and TDH data hold from write time.

TWA = 2TCLCL – TCLMLmax + TCLMHmin = 375 ns.
TWR = 2TCLCL – TCLMHmax + TCLLHmin + TSHOVmin = 170 ns.
TDWA = 2TCLCL – TCLDVmax + TCLMHmin – TIVOVmax = 265 ns.
TDH = TCLCH – TCLMHmax + TCHDXmin + TIVOVmin = 95 ns.

TABLE 5B1. TYPICAL WRITE TIMING PARAMETERS

TW — Write Pulse Width
TWR — Write Release (Address Hold From End of Write)
TDW — Data and Write Pulse Overlap
TDH — Data Hold From End of Write
TAW — Address Valid to End of Write
TCW — Chip Select to End of Write
TASW — Address Valid to Beginning of Write

TABLE 5B2. CYCLE DEPENDENT WRITE PARAMETERS FOR RAM MEMORIES

(a) Minimum Mode
TW = TWLWH = 2TCLCL – 60 = 340 ns
TWR = TCLCL – TCVCTXmax + TCLLHmin = 90 ns
TDW = 2TCLCL – TCLDVmax + TCVCTXmin = 300 ns
TDH = TWHDX = 88 ns
TAW = 3TCLCL – TCLAVmax + TCVCTXmin = 500 ns
TCW = TAW – Chip Select Decode
TASW = TCLCL – TCLAVmax + TCVCTXmin = 100 ns
(b) Maximum Mode
TW = TCLCL – TCLMLmax + TCLMHmin = 175 ns
TWR = TCLCL – TCLMHmax + TCLLHmin = 165 ns
TDW = TW = 175 ns
TDH = TCLCHmin – TCLMHmax + TCHDXmin = 93 ns
TAW = 3TCLCL – TCLAVmax + TCLMHmin = 500 ns
TCW = TAW – Chip Select Decode
TASW = 2TCLCL – TCLAVmax + TCLMLmin = 300 ns
TWA* = TW + TCLCL = 375 ns
TDWA* = 2TCLCL – TCLDVmax + TCLMHmin = 300 ns
TASWA* = TASW – TCLCL = 100 ns
*Relative to Advanced Write.

Comparing these results with the 2142 family indicates the standard 2142 write timing is fully compatible with this 8086 configuration. Read timing analysis is also necessary to completely determine compatibility of the devices.

5C. Dynamic RAM

Dynamic RAM is perhaps the most complex device to design into a system. To relieve the engineer of most of this burden, Intel provides the 8202 dynamic RAM controller as part of the 8086 family of peripheral devices. This section will discuss using the 8202 with the 8086 to build a dynamic memory system for an 8086 system. For additional information on the 8202, refer to the 8202 data sheet (9800873) and application note AP-45 Using the 8202 Dynamic RAM Controller (9800809A).

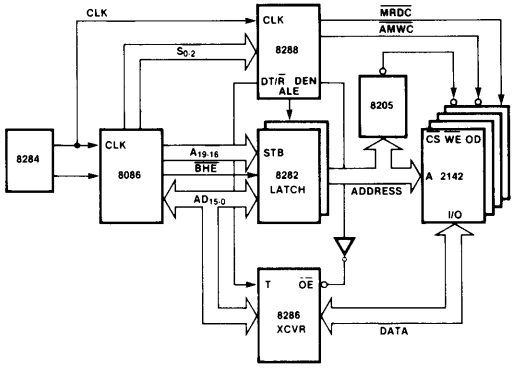


Figure 5B4. Sample Configuration for Compatibility Analysis Example

5.C.1 Standard 8086-8202 Interconnect

Figure 5.C.1.1 shows a standard interconnection for an 8202 into an 8086 system. The configuration accommodates 64K words (128K bytes) of dynamic RAM addressable as words or bytes. To access the RAM, the 8086 initiates a bus cycle with an address that selects the 8202 (via $\overline{PCS}$) and the appropriate transfer command ($\overline{MRDC}$ or $\overline{MWTC}$). If the 8202 is not performing a refresh cycle, the access starts immediately, otherwise, the 8086 must wait for completion of the refresh. $\overline{XACK}$ from the 8202 is connected to the 8284 RDY input to force the CPU to wait until the RAM cycle is completed before the CPU can terminate the bus cycle. This effectively synchronizes the asynchronous events of refresh and CPU bus cycles. The normal write command ($\overline{MWTC}$) is used rather than the advanced command ($\overline{AMWC}$) to guarantee the data is valid at the dynamic RAMS before the write command is issued. The gating of $\overline{WE}$ with A0 and $\overline{BHE}$ provides selective write strobes to the upper and lower banks of memory to allow byte and word write operations. The logic which generates the strobe for the data latches allows read data to propagate to the system as soon as the data is available and latches the data on the trailing edge of $\overline{CAS}$.

DETAILED TIMING

Read Cycle

For no wait state operation, the 8086 requires data to be valid from $\overline{MRDC}$ in:

2TCLCL – TCLML – TDVCL – buffer delays = 291 ns.

Since the 8202 is $\overline{CAS}$ access limited, we need only examine $\overline{CAS}$ access time. The 8202/2118 guarantees data valid from 8202 $\overline{RD}$ low to be:

(tph + 3tp + 100 ns) 8202 TCC delay + TCAC for the 2118

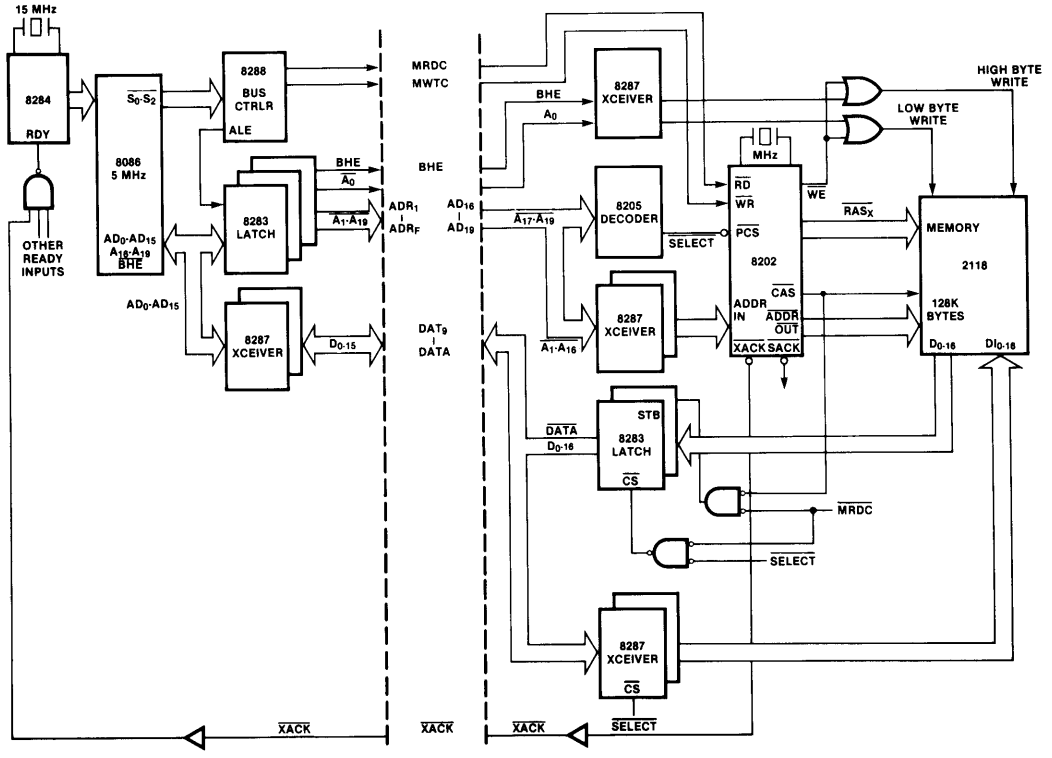


Figure 5C1.1. 5 MHz 8086/8202/128K Byte System — Double Data, Control and Address Buffering (Note: Bus driver on 8202 is not needed if less than 64K bytes are used)

For a 25 MHz 8202 and 2118-3, we get 297 ns which is insufficient for no wait state operation. If only 64K bytes are accessed, the 8202 requires only (tph + 3tp + 85 ns) giving 282 ns access and no wait states required. Refer to Figure 5.C.1.2 and 5C.1.3 for timing information on the 8202 and 2118.

Write Cycle

An important consideration for dynamic RAM write cycles is to guarantee data to the RAM is valid when both $\overline{CAS}$ and $\overline{WE}$ are active. For the 2118, if $\overline{WE}$ is valid prior to $\overline{CAS}$, the data setup is to $\overline{CAS}$ and if $\overline{CAS}$ is valid before $\overline{WE}$ (as would occur during a read modify write cycle) the data setup time is to $\overline{WE}$. For the 8202, the $\overline{WR}$ to $\overline{CAS}$ delay is analyzed to determine the data setup time to $\overline{CAS}$ inherently provided by the 8202 command to $\overline{RAS}/\overline{CAS}$ timing. The minimum delay from $\overline{WR}$ to $\overline{CAS}$ is:

TCCmin = tph + 2tp + 25 = 127 ns @ 25 MHz

Subtracting buffer delays and data setup at the 2118, we have 83 ns to generate valid data after the write command is issued by the CPU (in this case the 8288). Since the 8086 will not guarantee valid data until TCLAVmax – TCLMLmin = 100 ns from the advanced write signal, the normal write signal is used. The normal write $\overline{MWTC}$ guarantees data is valid 100 ns before it is active. The worst case write pulse width is approximately 175 ns which is sufficient for all 2118's.

Synchronization

To force the 8086 to wait during refresh the $\overline{XACK}$ or $\overline{SACK}$ lines must be returned to the 8284 ready input. The maximum delay from $\overline{RD}$ to $\overline{SACK}$ (if the 8202 is not performing refresh) is TAC = tp + 40 = 80 ns. To prevent a wait state at the 8086, RDY must be valid at the 8284 TCLCHmin – TCLMLmax – TR1VCLmax = 48 ns after the command is active. This implies that under worst case conditions, one wait state will be inserted for every read cycle. Since $\overline{MWTC}$ does not occur until one clock later, two wait states may be inserted for writes.

The $\overline{XACK}$ from command delay will assert RDY TCC + TCX = (tph + 3tp + 100) + (5tp + 20) = 460 ns after the command. This will typically insert one or two wait states.

Unless 2118-3's are used in 64K byte or less memories, $\overline{SACK}$ must not be used since it does not guarantee a wait state. From the previous access time analysis we saw that other configurations required a wait state.

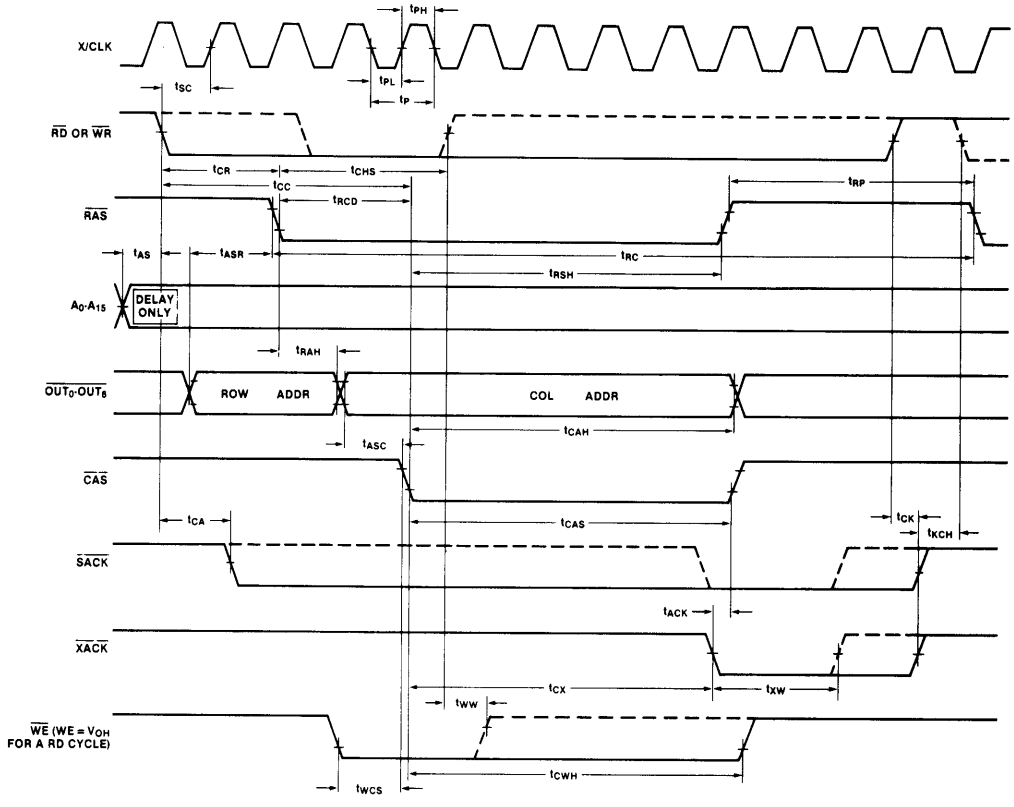


Figure 5C1.2. 8202 Timing Information

A.C. CHARACTERISTICS

$T_A = 0°C$ to $70°C$, $V_{CC} = 5V \pm 10\%$

Measurements made with respect to $RAS_1 - RAS_4$, CAS, WE, $OUT_0 - OUT_6$ are at 2.4V and 0.8V. All other pins are measured at 1.5V.

Loading:
64 Devices

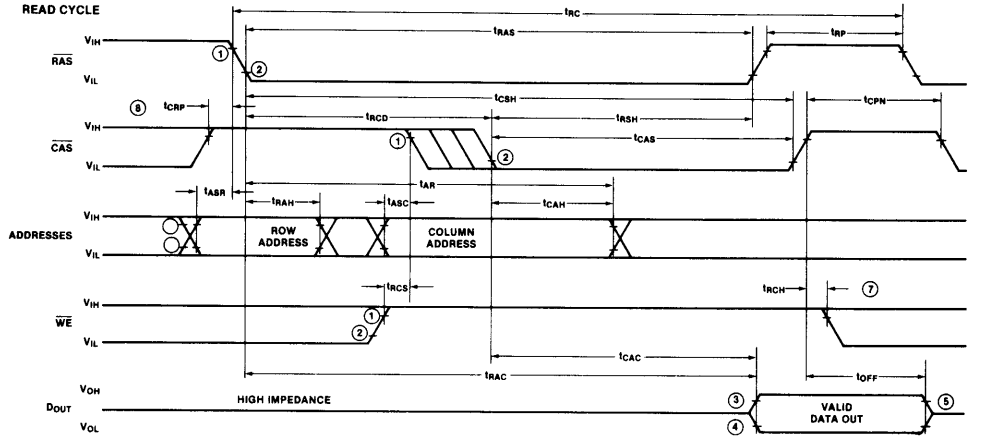
$\overline{SACK}$,$\overline{XACK}$	CL = 30 pF
$\overline{OUT_0} - \overline{OUT_6}$	CL = 320 pF
$\overline{RAS_1} - \overline{RAS_4}$	CL = 230 pF
$\overline{WE}$	CL = 450 pF
$\overline{CAS}$	CL = 640 pF

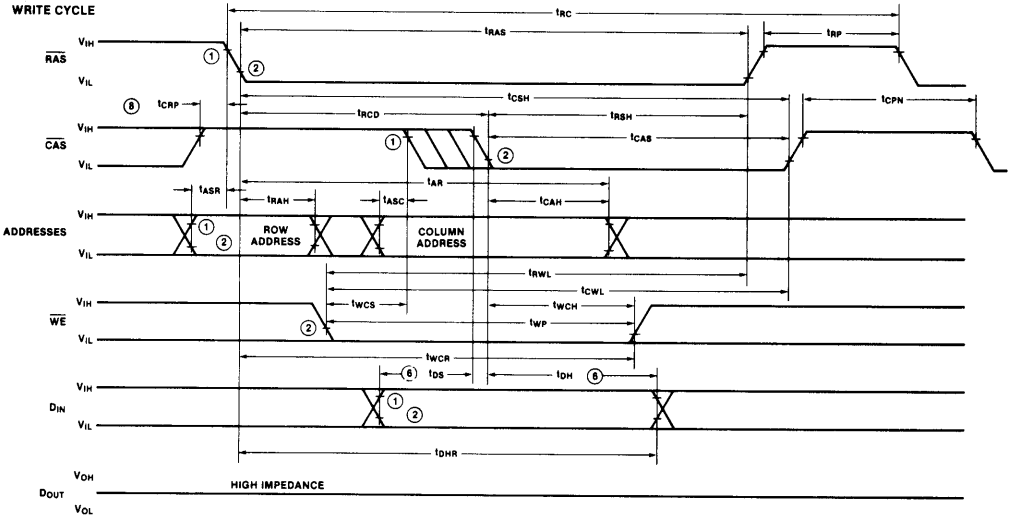
Symbol	Parameter	Min	Max	Units
t_P	Clock (Internal/External) Period (See Note 1)	40	54	ns
t_{RC}	Memory Cycle Time	$10\ t_P - 30$	$12\ t_P$	ns
t_{RAH}	Row Address Hold Time	$t_P - 10$		ns
t_{ASR}	Row Address Setup Time	t_{PH}		ns
t_{CAH}	Column Address Hold Time	$5\ t_P$		ns
t_{ASC}	Column Address Setup Time	$t_P - 35$		ns
t_{RCD}	$\overline{RAS}$ to $\overline{CAS}$ Delay Time	$2\ t_P - 10$	$2\ t_P + 45$	ns
t_{WCS}	$\overline{WE}$ Setup to $\overline{CAS}$	$t_P - 40$		ns
t_{RSH}	$\overline{RAS}$ Hold Time	$5\ t_P - 30$		ns
t_{CAS}	$\overline{CAS}$ Pulse Width	$5\ t_P - 30$		ns
t_{RP}	$\overline{RAS}$ Precharge Time (See Note 2)	$4\ t_P - 30$		ns
t_{WCH}	$\overline{WE}$ Hold Time to $\overline{CAS}$	$5\ t_P - 35$		ns
t_{REF}	Internally Generated Refresh to Refresh Time 64 Cycle 128 Cycle	 $548\ t_P$ $264\ t_P$	 $576\ t_P$ $288\ t_P$	 ns ns
t_{CR}	$\overline{RD}$, $\overline{WR}$ to $\overline{RAS}$ Delay	$t_{PH} + 30$	$t_{PH} + t_P + 75$	ns
t_{CC}	$\overline{RD}$, $\overline{WR}$ to $\overline{CAS}$ Delay	$t_{PH} + 2\ t_P + 25$	$t_{PH} + 3\ t_P + 100$	ns
t_{RFR}	REFRQ to $\overline{RAS}$ Delay	$1.5\ t_P + 30$	$2.5\ t_P + 100$	ns
t_{AS}	A_0—A_{15} to $\overline{RD}$, $\overline{WR}$ Setup Time (See Note 4)	0		ns
t_{CA}	$\overline{RD}$, $\overline{WR}$ to $\overline{SACK}$ Leading Edge		$t_P + 40$	ns
t_{CK}	$\overline{RD}$, $\overline{WR}$ to $\overline{XACK}$, $\overline{SACK}$ Trailing Edge Delay		30	ns
t_{KCH}	$\overline{RD}$, $\overline{WR}$ Inactive Hold to $\overline{SACK}$ Trailing Edge	10		ns
t_{SC}	$\overline{RD}$, $\overline{WR}$, $\overline{PCS}$ to X/CLK Setup Time (See Note 3)	15		ns
t_{CX}	$\overline{CAS}$ to $\overline{XACK}$ Time	$5\ t_P - 40$	$5\ t_P + 20$	ns
t_{ACK}	$\overline{XACK}$ Leading Edge to $\overline{CAS}$ Trailing Edge Time	10		ns
t_{XW}	$\overline{XACK}$ Pulse Width	$2\ t_P - 25$		ns
t_{LL}	REFRQ Pulse Width	20		ns
t_{CHS}	$\overline{RD}$, $\overline{WR}$, $\overline{PCS}$ Active Hold to $\overline{RAS}$	0		ns
t_{WW}	$\overline{WR}$ to $\overline{WE}$ Propagation Delay	8	50	ns
t_{AL}	S_1 to ALE Setup Time	40		ns
t_{LA}	S_1 to ALE Hold Time	$2\ t_P + 40$		ns
t_{PL}	External Clock Low Time	15		ns
t_{PH}	External Clock High Time	22		ns
t_{PH}	External Clock High Time for $V_{CC} = 5V \pm 5\%$	18		ns

Notes:

1. t_P minimum determines maximum oscillator frequency.
   t_P maximum determines minimum frequency to maintain 2 ms refresh rate and t_{RP} minimum.
2. To achieve the minimum time between the $\overline{RAS}$ of a memory cycle and the $\overline{RAS}$ of a refresh cycle, such as a transparent refresh, REFRQ should be pulsed in the previous memory cycle.
3. t_{SC} is not required for proper operation which is in agreement with the other specs, but can be used to synchronize external signals with X/CLK if it is desired.
4. If t_{AS} is less than 0 then the only impact is that t_{ASR} decreases by a corresponding amount.

Figure 5C1.2. 8202 Timing Information (Con't)





NOTES: 1,2. $V_{IH\ MIN}$ AND $V_{IL\ MAX}$ ARE REFERENCE LEVELS FOR MEASURING TIMING OF INPUT SIGNALS.
3,4. $V_{OH\ MIN}$ AND $V_{OL\ MAX}$ ARE REFERENCE LEVELS FOR MEASURING TIMING OF D_{OUT}.
5. t_{OFF} IS MEASURED TO $|I_{OUT}| < |I_{OL}|$.
6. t_{DS} AND t_{DH} ARE REFERENCED TO $\overline{CAS}$ OR $\overline{WE}$, WHICHEVER OCCURS LAST.
‡. t_{RCH} IS REFERENCED TO THE TRAILING EDGE OF $\overline{CAS}$ OR $\overline{RAS}$, WHICHEVER OCCURS FIRST.
8. t_{CRP} REQUIREMENT IS ONLY APPLICABLE FOR $\overline{RAS}/\overline{CAS}$ CYCLES PRECEDED BY A $\overline{CAS}$-ONLY CYCLE (i.e., FOR SYSTEMS WHERE $\overline{CAS}$ HAS NOT BEEN DECODED WITH $\overline{RAS}$).

Figure 5C1.3. 2118 Family Timing

A.C. CHARACTERISTICS[1,2,3]

$T_A = 0°C$ to $70°C$, $V_{DD} = 5V \pm 10\%$, $V_{SS} = 0V$, unless otherwise noted.

READ, WRITE, READ-MODIFY-WRITE AND REFRESH CYCLES

Symbol	Parameter	2118-3		2118-4		2118-7		Unit	Notes
		Min.	Max.	Min.	Max.	Min.	Max.		
t_{RAC}	Access Time From $\overline{RAS}$		100		120		150	ns	4,5
t_{CAC}	Access Time from $\overline{CAS}$		55		65		80	ns	4,5,6
t_{REF}	Time Between Refresh		2		2		2	ms	
t_{RP}	$\overline{RAS}$ Precharge Time	110		120		135		ns	
t_{CPN}	$\overline{CAS}$ Precharge Time (non-page cycles)	50		55		70		ns	
t_{CRP}	$\overline{CAS}$ to $\overline{RAS}$ Precharge Time	0		0		0		ns	
t_{RCD}	$\overline{RAS}$ to $\overline{CAS}$ Delay Time	25	45	25	55	25	70	ns	7
t_{RSH}	$\overline{RAS}$ Hold Time	70		85		105		ns	
t_{CSH}	$\overline{CAS}$ Hold Time	100		120		165		ns	
t_{ASR}	Row Address Set-Up Time	0		0		0		ns	
t_{RAH}	Row Address Hold Time	15		15		15		ns	
t_{ASC}	Column Address Set-Up Time	0		0		0		ns	
t_{CAH}	Column Address Hold Time	15		15		20		ns	
t_{AR}	Column Address Hold Time to $\overline{RAS}$	60		70		90		ns	
t_T	Transition Time (Rise and Fall)	3	50	3	50	3	50	ns	8
t_{OFF}	Output Buffer Turn Off Delay	0	45	0	50	0	60	ns	
READ AND REFRESH CYCLES									
T_{RC}	Random Read Cycle Time	235		270		320		ns	
t_{RAS}	$\overline{RAS}$ Pulse Width	115	10000	140	10000	175	10000	ns	
t_{CAS}	$\overline{CAS}$ Pulse Width	55	10000	65	10000	95	10000	ns	
t_{RCS}	Read Command Set-Up Time	0		0		0		ns	
t_{RCH}	Read Command Hold Time	0		0		0		ns	
WRITE CYCLE									
t_{RC}	Random Write Cycle Time	235		270		320		ns	
t_{RAS}	$\overline{RAS}$ Pulse Width	115	10000	140	10000	175	10000	ns	
t_{CAS}	$\overline{CAS}$ Pulse Width	55	10000	65	10000	95	10000	ns	
t_{WCS}	Write Command Set-Up Time	0		0		0		ns	9
t_{WCH}	Write Command Hold Time	25		30		45		ns	
t_{WCR}	Write Command Hold Time, to $\overline{RAS}$	70		85		115		ns	
t_{WP}	Write Command Pulse Width	25		30		50		ns	
t_{RWL}	Write Command to $\overline{RAS}$ Lead Time	60		65		110		ns	
t_{CWL}	Write Command to $\overline{CAS}$ Lead Time	45		50		100		ns	
t_{DS}	Data-In Set-Up Time	0		0		0		ns	
t_{DH}	Data-In Hold Time	25		30		45		ns	
t_{DHR}	Data-In Hold Time, to $\overline{RAS}$	70		85		115		ns	
READ-MODIFY-WRITE CYCLE									
t_{RWC}	Read-Modify-Write Cycle Time	285		320		410		ns	
t_{RRW}	RMW Cycle $\overline{RAS}$ Pulse Width	165	10000	190	10000	265	10000	ns	
t_{CRW}	RMW Cycle $\overline{CAS}$ Pulse Width	105	10000	120	10000	185	10000	ns	
t_{RWD}	$\overline{RAS}$ to $\overline{WE}$ Delay	100		120		150		ns	9
t_{CWD}	$\overline{CAS}$ to $\overline{WE}$ Delay	55		65		80		ns	9

NOTES:
1. All voltages referenced to V_{SS}.
2. Eight cycles are required after power-up or prolonged periods (greater than 2 ms) of $\overline{RAS}$ inactivity before proper device operation is achieved. Any 8 cycles which perform refresh are adequate for this purpose.
3. A.C. Characteristics assume $t_T = 5$ ns.
4. Assume that $t_{RCD} \leq t_{RCD}$ (max.). If t_{RCD} is greater than t_{RCD} (max.) then t_{RAC} will increase by the amount that t_{RCD} exceeds t_{RCD} (max.).
5. Load = 2 TTL loads and 100 pF.
6. Assumes $t_{RCD} \geq t_{RCD}$ (max.).
7. t_{RCD} (max.) is specified as a reference point only; if t_{RCD} is less than t_{RCD} (max.) access time is t_{RAC}, if t_{RCD} is greater than t_{RCD} (max.) access time is $t_{RCD} + t_{CAC}$.
8. t_T is measured between V_{IH} (min.) and V_{IL} (max.).
9. t_{WCS}, t_{CWD} and t_{RWD} are specified as reference points only. If $t_{WCS} \geq t_{WCS}$ (min.) the cycle is an early write cycle and the data out pin will remain high impedance throughout the entire cycle. If $t_{CWD} \geq t_{CWD}$ (min.) and $t_{RWD} \geq t_{RWD}$ (min.), the cycle is a read-modify-write cycle and the data out will contain the data read from the selected address. If neither of the above conditions is satisfied, the condition of the data out is indeterminate.

Figure 5C1.3. 2118 Family Timing (Con't)

5.C.2 Enhanced Operation

Two problems are evident from the previous investigation:

1) $\overline{SACK}$ timing from command will not allow reliable operation while $\overline{XACK}$ is not active early enough to prevent wait states.

2) The normal write command required to guarantee data setup is not enabled until the CPU has sampled READY thereby forcing multiple wait states during write operations.

The first problem could be resolved if an early command could be generated that would guarantee $\overline{SACK}$ was valid when READY was sampled and $\overline{SACK}$ to data valid satisfied the CPU requirements. Figure 5.C.2.1 is a circuit which provides an early read command derived from the maximum mode status. The early command is enabled from the trailing edge of ALE and disabled on the trailing edge of the normal command. The command provides an additional TCHCLmin – TCHLLmax + TCLMLmax – circuit delays = 53 ns of access time and time to generate RDY from the early command. If we go back to our previous equations, early command to valid data at the CPU is now:

TCHCLmin – TCHLLmax + 2TCLCL – TDVCLmax – buffer and circuit delays = 333 ns

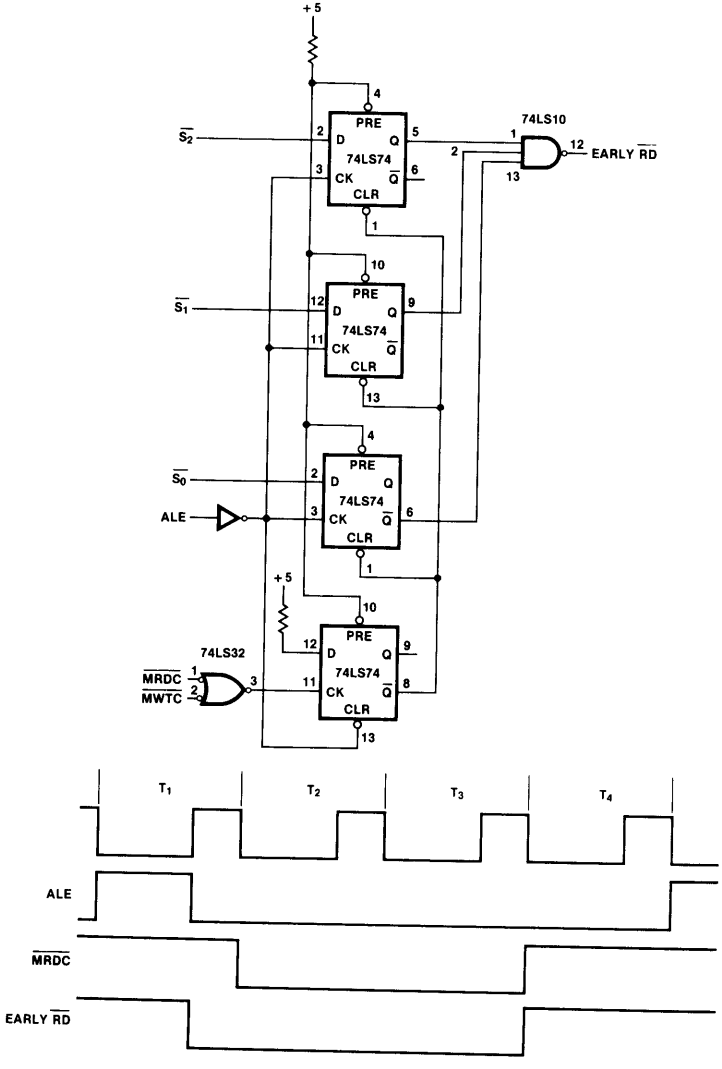


Figure 5C2.1. Early Read and Write Command Generation

We can now use the slowest 2118 which gives 8202 and 2118 access of 320 ns. Early command to RDY timing is TCLCL – TCHLLmax – circuit delays – TR1VCLmax = 115 ns and provides 35 ns of margin beyond the 8202 command to $\overline{SACK}$ delay.

The write timing of the 8202 and write data valid timing of the 8086 do not allow use of an early write command. However, if the 8202 clock is reduced from 25 MHz to 20 MHz and $\overline{WE}$ to the RAM's is gated with $\overline{CAS}$, the advanced write command ($\overline{AMWC}$) may be used. At 20 MHz the minimum command to $\overline{CAS}$ delay is 148 ns while the maximum data valid delay is 144 ns.

The reduced 8202 clock frequency still satisfies no wait state read operation from early read and will insert no more than one wait state for write (assuming no conflict with refresh). 20 MHz 8202 operation will however require using the 2118-4 to satisfy read access time.

Note that slowing the 8202 to 22.2 MHz guarantees valid data within 10 ns after $\overline{CAS}$ and allows using the 2118-7. Since this analysis is totally based on worst case minimum and maximum delays, the designer should evaluate the timing requirements of his specific implementation.

It should be noted that the 8202 $\overline{SACK}$ is equivalent to $\overline{XACK}$ timing if the cycle being executed was delayed by refresh. Delaying $\overline{SACK}$ until $\overline{XACK}$ time causes the CPU to enter wait states until the cycle is completed. If the cycle is a read cycle, the $\overline{XACK}$ timing guarantees data is valid at the CPU before RDY is issued to the CPU.

The use of the early command signals also solves a problem not mentioned previously. The cycle rate of the 8202 @ 20 MHz requires that commands (from leading edge to leading edge) be separated by a minimum of 695 ns. The maximum mode 8086 however may issue a read command 600 ns after the normal write command. For the early read command and advanced write command, 725 ns are guaranteed between commands.

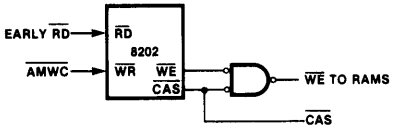


Figure 5C2.2. Delayed Write to Dynamic RAMs

APPENDIX I
BUS CONTENTION AND ITS EFFECT ON SYSTEM INTEGRITY

SYSTEM ARCHITECTURE

As higher performance microprocessors have become available, the architecture of microprocessor systems has been evolving, again placing demands on memory. For many years, system designers have been plagued with the problem of bus contention when connecting multiple memories to a common data bus. There have been various schemes for avoiding the problem, but device manufacturers have been unable to design internal circuits that would guarantee that one memory device would be "off" the bus before another device was selected. With small memories (512x8 and 1Kx8), it has been traditional to connect all the system address lines together and utilize the difference between t_{ACC} and t_{CO} to perform a decode to select the correct device (as shown in Figure 1).

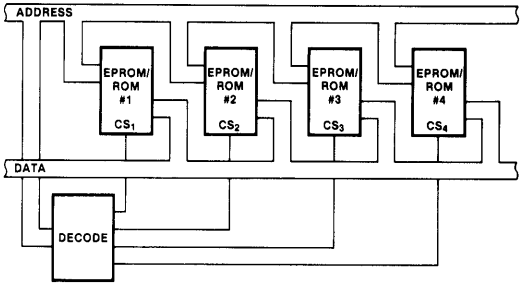


Figure 1. Single Control Line Architecture

With the 1702A, the chip select to output delay was only 100 ns shorter than the address access time; or to state it another way, the t_{ACC} time was 1000 ns while the t_{CO} time was 900 ns. The 1702A t_{ACC} performance of 1000 ns was suitable for the 4004 series microprocessors, but the 8080 processor required that the corresponding numbers be reduced to $t_{ACC} = 450$ ns and $t_{CO} = 120$ ns. This allowed a substantial improvement in performance over the 4004 series of microprocessors, but placed a substantial burden on the memory. The 2708 was developed to be compatible with the 8080 both in access time and power supply requirements. A portion of each 8080 machine cycle time had to be devoted to the architecture of the system decoding scheme used. This devoted portion of the machine cycle included the time required for the system controller (8224) to perform its function before the actual decode process could begin.

Let's pause here and examine the actual decode scheme that was used so we can understand how the control functions that a memory device requires are related to system architecture.

The 2708 can be used to illustrate the problem of having a single control line. The 2708 has only one read control function, chip select ($\overline{CS}$), which is very fast ($t_{CO} = 120$ ns) with respect to the overall access time ($t_{ACC} = 450$ ns) of the 2708. It is this time difference (330 ns) that is used to perform the decode function, as illustrated in Figure 2. The scheme works well and does not limit system performance, but it does lead to the possibility of bus contention.

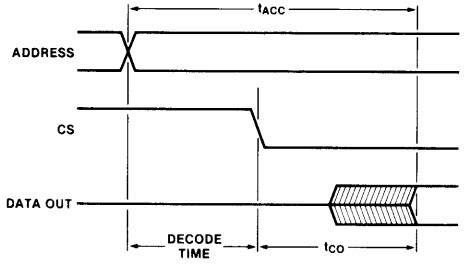


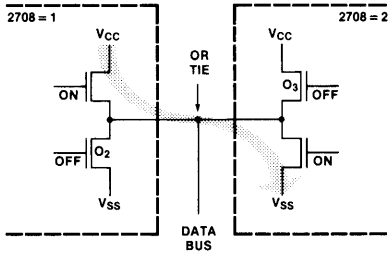
Figure 2. Single Line Control Architecture

BUS CONTENTION

There are actually two problems with the scheme described in the previous section. First, if one device in a multiple memory system has a relatively long deselect time, and a relatively fast decoder is used, it would be possible to have another device selected at the same time. If the two devices thus selected were reading opposite data; that is, device number one reading a HIGH and device number two reading a LOW, the output transistors of the two memory devices would effectively produce a short circuit, as Figure 3 illustrates. In this case, the current path is from V_{CC} on device number one to GND on device number two. This current is limited only by the "on" impedance of the MOS output transistors and can reach levels in excess of 200 mA per device. If the MOS transistors have a lot of "extra" margin, the current is usually not destructive; however, an instantaneous load of 400 mA can produce "glitches" on the V_{CC} supply—glitches large enough to cause standard TTL devices to drop bits or otherwise malfunction, thus causing incorrect address decode or generation.

The second problem with a single control line scheme is more subtle. As previously mentioned, there is only one control function available on the 2708 and any decoding scheme must use it out of necessity. In addition, any inadvertent changes in the state of the high order address lines that are inputs to the decoder will cause a change in the device that is selected. The result is the same as before—bus contention, only from a different source. The deselected device cannot get "off" the bus before the selected one is "on" the bus as the addresses rapidly change state. One approach to solving this problem would be to design (and specify as a maximum) devices

with t_{DF} time less than t_{CO} time, thereby assuring that if one device is selected while another is simultaneously being deselected, there would be some small (20 ns) margin. Even with this solution, the user would not be protected from devices which have very fast t_{CO} times (t_{CO} is specified as a maximum).



RESULTS OF IMPROPER TIMING WHEN OR TYING MULTIPLE MEMORIES.

Figure 3. Results of Improper Timing when OR Tying Multiple Memories

The only sure solution appears to be the use of an external bus driver/transceiver that has an independent enable function. Then that function, not the "device selecting function," or addresses, could control the flow of data "on" and "off" the bus, and any contention problems would be confined to a particular card or area of a large card. In fact, many systems are implemented that way—the use of bus drivers is not at all uncommon in large systems where the drive requirements of long, highly capacitive interconnecting lines must be taken into consideration—it also may be the reason why more system designers were not aware of the bus contention problem until they took a previously large (multicard) system and, using an advanced micorprocessor and higher density memory devices, combined them all on one card, thereby eliminating the requirement for the bus drivers, but experiencing the problem of bus contention as described above.

THE MICROPROCESSOR/MEMORY INTERFACE

From the foregoing discussion, it becomes clear that some new concepts, both with regard to architecture and performance are required. A new generation of two control line devices is called for with general requirements as listed below:

1. Capability to control the data "on" and "off" the system bus, independent of the device selecting function identified above.

2. Access time compatible with the high performance microprocessors that are currently available.

Now let's examine the system architecture that is required to implement the two line control and prevent bus contention. This is shown in the form of a timing diagram (Figure 4). As before, addresses are used to generate the unique device selecting function, but a separate and independent Output Enable (OE) control is now used to gate data "on" and "off" the system data bus. With this scheme, bus contention is completely eliminated as the processor determines the time during which data must be present on the bus and then releases the bus by way of the Output Enable line, thus freeing the bus for use by other devices, either memories or peripheral devices. This type of architecture can be easily accomplished if the memory devices have two control functions, and the system is implemented according to the block diagram shown in Figure 5. It differs from the previous block diagram (shown in Figure 1) in that the control bus, which is connected to all memory Output Enable pins, provides separate and independent control over the data bus. In this way, the microprocessor is always in control of the system; while in the previous system, the microprocessor passed control to the particular memory device and then waited for data to become available. Another way to look at it is, with a single control line the sytem is always asynchronous with respect to microprocessor/memory communications. By using two control lines, the memory is synchronized to the processor.

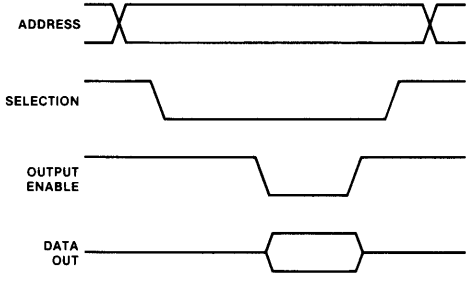


Figure 4. Two Control Line Architecture

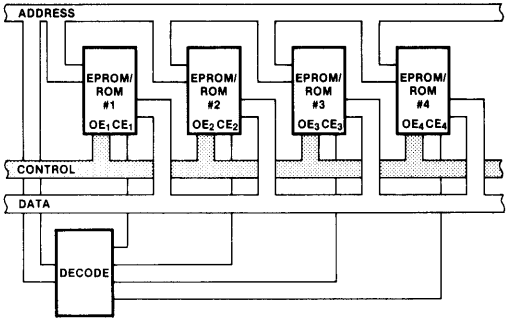


Figure 5. Two Control Line Architecture

intel®

APPLICATION NOTE

AP-61

July 1979

Multitasking for the 8086

Cecil Moore
Applications Engineer
Microprocessor Products

Multitasking For the 8086

Contents

INTRODUCTION

Real-time software systems differ markedly from batch processing systems. An external signal indicating that it is time for an hourly log or an interrupt caused by an emergency condition is an event usually not encountered in batch processing. Because real-time control systems of all types share a number of characteristics, it is possible to develop flexible operating systems which will meet the needs of a great majority of real-time applications. Intel Corporation has developed such a system, the RMX/80™ system, for the iSBC™ line of 8080/85 based single board computers. Thus, the user is released from the chore of designing an operating system and is free to concentrate his efforts on the applications software for the individual tasks and merely integrate them into a pre-existing system.

But what if a user does not need all the capabilities of an RMX/80™ system or wants a different hardware configuration than an iSBC™ computer? This application note contains a set of PL/M-86 procedures designed to be used in medium-complexity 8086 real-time systems.

A normal control system can be broken down into a number of concurrently executable tasks. The CPU can be running only one task at any instant of time but the speed of the processor often makes concurrent tasks appear to be running simultaneously. Breaking the software functions into separate concurrent tasks is the job of the designer/programmer. Once this is done there remains the problem of integrating these tasks with a supervisory program which acts as a traffic cop in the scheduling and execution of the separate tasks. This note discusses a set of PL/M-86 procedures to implement the supervisory program function.

A minimum operating system might (like its batch processing cousin) have only a queue for ready tasks (tasks waiting to be executed). Any task that becomes ready is put on the bottom of the queue and when a running task is finished, the task on the top of the queue is started. Any interrupt causes the state of the system to be saved, an interrupt routine to be executed, the state of the system to be restored, and execution of the interrupted program to continue. The interrupt routine might (or might not) put a new task on the ready queue. This approach has worked well for many simple control systems, especially in the single-chip computer area. But what features are lacking in this approach that are necessary (or at least nice)?

1. A system of priorities is often needed. All waiting ready tasks must be executed sooner or later but some tasks need immediate attention while others can be run when there is nothing else to do. If a midnight monthly report, due for completion by 8 a.m. the next day, is in the process of printing at 1 a.m. and a fire alarm occurs, it is reasonable to assume that the fire alarm has higher priority since the fire could conceivably render the monthly report irrelevant.

There are a number of ways in which to assign priorities. Tasks are usually numbered and may be assigned priorities according to their ascending (or descending) numbers. They could instead be grouped into a number of priority levels, with tasks on the same level having equal priorities. The latter approach is taken in this application note.

Assume that a monthly report is being printed and an alarm occurs in the external world that, because of its importance, must be attended to immediately. The interrupt routine, executed as a result of the alarm input, should not automatically return to the interrupted logging routine but instead should call a preempt routine which checks to see if a higher priority task is ready for execution. The reason for this is that the monthly report routine, if returned to, has no way of "knowing" that a higher priority task is waiting to be executed. The alarm output task has been readied by the interrupt routine and since it is known to be higher priority than the logging task, it is executed first, thereby immediately signaling the system operator that there has been an alarm. It then returns to the logging task provided that there are no further high priority tasks waiting to be executed. The logging printer may not have even paused during the alarm output task. The computer appears to human beings to be executing concurrent tasks simultaneously.

Of course, the alarm output function could be performed inside the interrupt procedure. But sooner or later, the designer will encounter a worst case situation in which there is not enough time to execute all required tasks between interrupts, and the system will fall behind in real-time. It is much cleaner to make the interrupt procedures as short as possible and stack up tasks to be executed than to stack up interrupt procedures.

2. Another feature that might be necessary is a capability to put a task to sleep for a known period of real time. Assume a relay output must remain closed for one second. Most real-time systems cannot tolerate the dedication of the CPU to such a trivial task for that length of time so a system of programmable dynamic delays could be implemented. This application note implements such a system.

Although the PL/M-86 procedures here have been debugged and tested, it is assumed that the user will want to change, add, or delete features as needed. This application note is intended to present ideas for a logical structure of procedures that, because they are written in PL/M-86, can be easily modified to user requirements. Each procedure will be discussed in detail and integration and optional features will be presented.

PL/M-86

PLM-86 is a block structured high level language that allows direct design of software modules. Using PL/M-86, designers can forget their assembly level

coding problems and design directly in a subset of the English language. The 8086 architecture was designed to accommodate highly structured languages and the PLM-86 compiler is quite efficient in the generation of machine code.

PLM-86 STRUCTURE

PL/M-86 automatically keeps track of the level of the different software blocks. (See Chapter 10, "PL/M-86 Programming Manual"). There are methods of writing PL/M-86 which contribute to the understandability of the source code without adding to the amount of object code generated. For instance, the following three IF/THEN/ELSE blocks generate identical object code but are compiled from different source statements.

Line	Level	Statement
3	1	IF A = B THEN C = D; ELSE E = F; G = H;
7	1	IF A = B THEN
8	1	C = D;
		ELSE
9	1	E = F;
10	1	G = H;
11	1	IF A = B THEN DO;
13	2	C = D ;
14	2	END;
15	1	ELSE DO;
16	2	E = F;
17	2	END;
18	1	G = H;

It is not instantly apparent from the code on line 3 or the code starting at line 7 which statements will be executed. However, adding the DO; and END; statements (starting at line 11) remove any doubt. Either the statements starting at line 11 or the statements starting at line 15 will be executed and the statement on line 18 will be executed in either case. Why? Because all these lines are at level 1 in the block structure. The other lines are at level 2 because of the DO;/END; combinations. When one refers to the relatively complex structures of the task multiplexer procedures, the usefulness of such an approach is obvious, as the procedures have been indented according to the level numbers generated by PL/M-86. In particular, if the designer is not careful, nested IF/THEN/ELSE statements can generate improper results. Using a proper number of DO;/END; combinations avoids the possible ambiguity in nested IF/THEN/ELSE statements as can be seen in the ACTIVATE$TASK procedure listed in the PL/M-86 source code later in this note. The DO;/END; construct naturally must be used when multiple statements are required within the IF/THEN/ELSE blocks. Following are examples of the possible primary structures of PL/M-86:

```
DO;
    A = B;
    C = D;
    END;

DO WHILE A = B;
    C = D;
    E = F;
    END;

DO I = 1 TO 5;
    A = I;
    C = D + I;
    END;

DO CASE A;
    A = B;
    A = C;
    A = D;
    END;

IF  A = B THEN DO;
    C = D;
    END;
ELSE DO;
    E = F;
    END;

IF  A = B THEN DO;
    C = D;
    END;
ELSE IF A = C THEN DO;
    D = E;
    END;
ELSE IF A = D THEN DO;
    E = F;
    END;
ELSE DO;
    F = G;
    END;
```

A complete tutorial on structured programming is beyond the scope and intent of this application note and the reader is referred to the appropriate references appearing in the bibliography.

ANATOMY OF THE TASK MULTIPLEXER

Once a decision is made on the details of the kind of data structure that is needed to implement the task multiplexer, the procedures that manipulate the structure are relatively simple to write. The following characteristics are assumed for the task multiplexer appearing in this application note.

There are two levels of priority, high and low. All high priority tasks that are ready to run will be dispatched, executed, and completed, on a FIFO basis, before any low priority task is dispatched.

Any task can be interrupted. No task multiplexer procedure can be interrupted.

If a high priority task is interrupted, it will be completed before any other task is dispatched. If a low priority task is interrupted, all ready high priority tasks will be dispatched, executed, and completed before program control is returned to the low priority task.

There are two ready queues, one for high priority tasks and one for low priority tasks. Each queue has a head (top) pointer and a tail (bottom) pointer and tasks on any queue are link-listed from head to tail. Tasks are "dispatched" (taken off the queue) at the head and "activated" (put on the queue) at the tail on a FIFO basis.

Link-listed queues are chosen for simplicity. All dispatch and activate information is contained in the head and tail pointers. Tasks located in the middle of these link-lists are of no concern for activating and dispatching. This means, of course, that tasks are executed in the order that they appear on the queue, i.e., first-in, first-out.

There is a pointer byte associated with each task. If a task is on either the low priority or high priority ready queue, its associated pointer byte will point to the next task number on the list. These pointer bytes enable the task ready lists to be linked. Note that the pointer byte is 0 for the last task on a list.

There is a status (flag) byte associated with each task. If a task is on a ready list or a delay list, bit 7 will be a "1" indicating that that particular task is busy. If a task is on either high priority or low priority ready queues, bit 6 will be a "1" indicating that the task is on one of the ready queues. If the task is listed on the delay list, (see next item), bit 5 will be a "1" indicating that this particular task has a delay in progress. If a task is unlisted, bits 5-7 will be "0." Bits 0-4 are not used by the task multiplexer procedures and are available to the user, giving 5 user defined flags per task.

There is a delay byte associated with each task. This feature allows tasks to be "put to sleep" for a variable length of time, from 1 to 255 "ticks" of the interrupt clock. If a task does not need an associated delay then this byte is available to the user as a utility byte to be used for any purpose. These delays will be discussed in detail later in the application note.

The following diagram is a representation of the task multiplexer data structure:

TASK NUMBER	POINTER BYTE	STATUS BYTE	DELAY BYTE
0	n	n+1	n+2
1	n+3	n+4	n+5
2	n+6	n+7	n+8
3	n+9	n+10	n+11
4	n+12	n+13	n+14
5	n+15	n+16	n+17
m-1	n+3m-6	n+3m-5	n+3m-4
m	n+3m-2	n+3m-1	n+3m

3m+3 TOTAL RAM BYTES
n = FIRST RAM ADDRESS OF ARRAY

Following is a chart of what a task multiplexer data structure might look like at a given moment in time:

HIGH$PRIORITY$HEAD = 5
HIGH$PRIORITY$TAIL = 3
LOW$PRIORITY$HEAD = 8
LOW$PRIORITY$TAIL = 10
DELAY$HEAD = 4

TASK NUMBER	TASK(n).PNTR	TASK(n).STATUS	TASK(n).DELAY
0	*	*	*
1	3	1100 0000	0
2	0	1010 0000	3
3	0	1100 0000	0
4	7	1010 0000	4
5	1	1100 0000	0
6	0	0000 0000	0
7	2	1010 0000	6
8	10	1100 0000	0
9	0	0000 0000	0
10	0	1100 0000	0

*See text.

What information can one ascertain from observation of the above chart? The ready-to-run high priority tasks, in order, are 5,1,3. This can be seen by following the high priority ready linked list from head to tail. The ready-to-run low priority tasks, in order are 8, 10. The TASK(n).PNTR byte = 0 for the last listed task. Tasks 4, 7, 2 are listed, in order, on the delay list and have associated delays of 4, 10, 13 ticks respectively. Tasks 6 and 9 are not listed and therefore idle. The * for the TASK (0) bytes indicate a special condition. There is no TASK00 allowed and a zero condition is treated as an error condition. TASK(0).PNTR byte is used for the DELAY$HEAD byte to minimize code in the ACTIVATE$DELAY procedure. TASK(0).STATUS and TASK(0).DELAY are unused bytes.

DEFINITIONS

NEW$TASK is the number of the task that will be installed on a ready list or the delay list when ACTIVATE$TASK or ACTIVATE$DELAY is called.

NEW$DELAY is the value of the delay that will be installed on the delay list when ACTIVATE$DELAY is called.

A task is defined as RUNNING if it is in the act of execution or if an interrupt routine is executing which interrupted a RUNNING task.

A task is defined as PREEMPTED if it has been interrupted and a higher priority task is being executed.

A task is defined as READY if it is contained within one of the ready queues.

A task is defined as IDLE if its BUSY$BIT (bit 7) is not set, i.e., it is not listed anywhere else. Note that it is possible to completely disable an IDLE task simply by setting its BUSY$BIT. In that case, it is not and cannot be listed anywhere else. This feature is useful during system integration.

STATE DIAGRAM

The state diagram indicates the relationships among the possible task states and the procedures involved in changing states.

The state diagram looks somewhat complicated and a discussion of the possible change of states is in order. Assuming a certain existing state, future possible states will be discussed including the procedures which can cause the change of state.

From the unlisted (idle) state, the ACTIVATE$TASK procedure will put the NEW$TASK on either the high priority ready queue or the low priority ready queue at the tail end of the queue. The number of the task automatically assigns the priority and therefore the proper queue. All task numbers below FIRSTLOWPRIORITY$TASK are assumed to be high priority tasks. Also, from the unlisted state the ACTIVATE$DELAY procedure will put the NEW$TASK and NEW$DELAY at the proper position on the delay list.

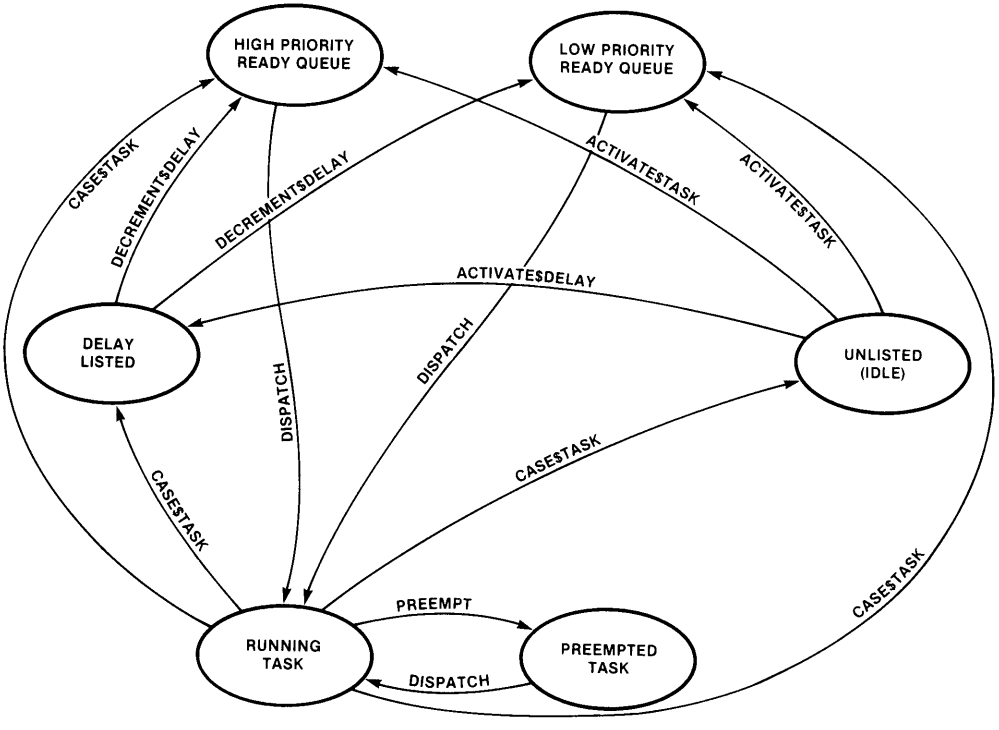
After a task has been put on either high priority ready queue or low priority ready queue it eventually will go to the RUNNING$TASK state. The DISPATCH procedure accomplishes this action.

From the delay list a task can only go to one of the ready queues. When a task's associated delay goes to zero the DECREMENT$DELAY procedure calls the ACTIVATE$TASK procedure and installs the NEW$TASK on the proper ready queue.

From the RUNNING$TASK state a task may use the CASE$TASK procedure to put itself on the ready list tail by setting NEW$TASK = RUNNING$TASK. It may instead put itself on the delay list by setting NEW$TASK = RUNNING$TASK and also setting NEW$DELAY equal to something other than zero. Otherwise, it will progress to the unlisted state upon completion.

The CASE$TASK procedure unlists tasks when they have completed execution. A low priority RUNNING$TASK will go to the preempted state if a high priority task is on the ready list following an interrupt during execution of the low priority task if the PREEMPT procedure is called.

And finally, a PREEMPTED$TASK will return to a RUNNING$TASK state when all high priority ready tasks have completed execution. This is accomplished by the DISPATCH procedure which then returns to the PREEMPT procedure.



STATE DIAGRAM

Some lockouts are necessary to avoid chaos in the task multiplexer. These are as follows:

The BUSY$BIT = 1 in the TASK(n).STATUS byte will abort the ACTIVATE$TASK and the ACTIVATE$DELAY procedures and return an indication of the aborting by setting the STATUS byte equal zero. A task must be unlisted to be able to be installed on a list.

A RUNNING$TASK may put itself on a list after it has executed but it is not allowed to re-list any listed tasks (i.e., no task may ever be listed twice at the same time!). A task that tries to activate another task that is already busy can wait (via the delay feature) for the required task to complete execution, become idle, and therefore be available to be activated. A PREEMPTED$TASK may not be listed. If the ACTIVATE$TASK or ACTIVATE$DELAY procedure is called and NEW$TASK = PREEMPTED$TASK, the procedure will be aborted and return with STATUS = 0. Otherwise, the STATUS byte is returned with the new task status.

Only one task may be preempted as there are only two levels of priority. The user may desire to implement many levels of priority in which case a linked-list of preempted tasks could be declared in a structure which includes the number of the first task in each priority level group of tasks. This obviously complicates the PREEMPT and DISPATCH procedures.

The tasks themselves are made into reentrant procedures because of the necessary forward references of the CASE$TASK procedure.

PL/M-86 allows structures and arrays of structures. The structure needed for the task multiplexer is a link-list pointer byte, a task status byte, and a task delay byte. Each task has an associated pointer byte, status byte, and delay byte. These are combined into an array of up to 255 tasks. For purposes of this discussion, the number of tasks is chosen as an arbitrary 10, leading to the following array declaration.

```
DECLARE TASK(10)STRUCTURE
(PNTR BYTE,STATUS BYTE,DELAY BYTE);
```

Thus the delay byte associated with task number 7 can be accessed by using the variable TASK(7).DELAY and the status of task number 5 can be examined through the use of TASK(5).STATUS. The TASK(n).PNTR byte contains the task number of the next listed task on the same list as TASK(n), i.e., if TASK(n) is on the delay list, then TASK(n).PNTR will contain the number of the next task on the delay list or 0 indicating the end of the list.

TASK(n).STATUS is a byte with the following reserved flags:

```
BIT 7   BUSY$BIT, "1" IF TASK IS BUSY
BIT 6   READY$BIT, "1" IF ON READY LIST
BIT 5   DELAY$BIT, "1" IF ON DELAY LIST
BIT 4 — BIT 0     UNUSED
```

The unused bits in the STATUS byte are available to the user.

The TASK(n).DELAY byte is a number which can put TASK(n) to sleep for up to 255 system clock ticks. The system clock tick is interrupt driven from the user's timer and its period is chosen for the particular application. A one millisecond timer is popular and assuming such a time, delays of up to 255 ms are available in the task multiplexer as it is written. If this delay range is not wide enough, the user may want to define his TASK(n).DELAY as a word instead of a byte in the PL/M-86 declare statement, giving delays of up to 65 seconds from the basic one millisecond clock tick.

LINKED LISTS

Linked lists are useful for a number of reasons. However, a treatise on linked lists would defeat the purpose of this application note and the reader is referred to the references listed in the bibliography.

The linked lists used in this application note have a head byte associated with each list, i.e., the head byte contains the number of the first task on the list. The first task pointer byte points to the second task on the list, etc. The pointer of the last task on the list is set at zero to indicate that it is the last task. Two of the linked lists are ready queues and require a tail byte as well as a head byte. The tail byte points to the last entry on the list. Tasks are put on the bottom, or tail, of the ready lists and are taken off the top, or head, of the ready lists. The delay list has no tail but does have a head, called a DELAY$HEAD. The delay list is not a queue, as delays are installed on the list in order of delay magnitude for reasons to be explained later.

There are two ready lists, one for high priority tasks and one for low priority tasks. The head and tail pointers associated with these two lists are: HIGH$PRIORITY$HEAD, HIGH$PRIORITY$TAIL, LOW$PRIORITY$HEAD, and LOW$PRIORITY$TAIL. Obviously, the structure can be expanded to any number of priority levels by expanding the head and tail pointers and the historical record of the preempted tasks.

DELAY STRUCTURE

A task multiplexer can have a number of simultaneous delays active and it would be efficient if there were a way to keep from decrementing all delays on every clock tick, which is most time consuming. One way to accomplish this feat is to move the problem from the DECREMENT$DELAY routine to the ACTIVATE$DELAY routine. The delays are arranged in a linked-list of ascending sizes such that the value of each delay includes the sum of all previous delays. This allows the decrementing of only one delay during each clock tick interrupt routine. An example will further illuminate this approach. Suppose the following conditions exist:

Task 7 has a 5 millisecond delay

Task 3 has an 8 millisecond delay

Task 9 has a 14 millisecond delay

The delay structure is arranged so that:

DELAY$HEAD = 07
TASK(7).PNTR = 03
TASK(3).PNTR = 09
TASK(9).PNTR = 00
TASK(7).DELAY = 05 (FIRST DELAY = 5)
TASK(3).DELAY = 03 (5 + 3 = 8)
TASK(9).DELAY = 06 (5 + 3 + 6 = 14)

The linked-list is arranged so that the delays are in ascending order and each delay is equal to the sum of all previous delays up through that point. Since this is true, all delays are effectively decremented merely by decrementing the first delay. Of course, something for nothing is impossible and the speed gained by arranging the delays in the above manner is paid for by the complexity of the ACTIVATE$DELAY routine. But since the ACTIVATE$DELAY routine is executed less frequently than the DECREMENT$DELAY routine, the savings in real time is worth the added complexity.

Suppose a new delay is to be activated in the above scheme. Task 5 with a delay of 10 milliseconds is to be added. A before and after chart will indicate what the ACTIVATE$DELAY procedure must accomplish.

BEFORE

TASK NUMBER		07	03	09	
POINTER	07	03	09	00	
DELAY		05	03	06	

AFTER

TASK NUMBER		07	03	05	09
POINTER	07	03	05*	09@	00
DELAY		05	03	02@	04*

FIRST POINTER IS THE DELAY$HEAD
CHANGES ARE MARKED WITH AN *
ADDITIONS ARE MARKED WITH AN @

Note that the pointer before the added task has changed and the delay after the added task has changed. The function of the ACTIVATE$DELAY procedure is to accomplish these changes and additions.

PROCEDURES

The following procedure explanations reference the PL/M-86 source code listing which follows the application note text.

ACTIVATE$TASK Procedure

This procedure is initiated by a call instruction with the byte NEW$TASK containing the number of the task to be put on the proper ready queue.

Interrupts must be disabled whenever the link-lists are being changed. If interrupts are enabled when this procedure is called, they should be re-enabled upon returning.

The assignment of priority is a simple matter. A declare statement, DECLARE FIRSTLOWPRIORITY$TASK LITERALLY 'N,' (where N is the actual number of the first low priority task) indicates to the procedures that tasks 1 to N are high priority tasks and tasks N or higher are low priority tasks.

This procedure checks the busy bit in the status byte to see if this particular task is already busy and if so, returns a STATUS of zero. Otherwise, it returns the new STATUS of the task. It then checks the priority to see if this particular task is a high or low priority. If it is high priority, then the task pointer pointed to by the HIGH$PRIORITY$TAIL pointer is changed from zero to the number of the NEW$TASK. The HIGH$PRIORITY$TAIL pointer is then changed to the number of the NEW$TASK and the pointer associated with NEW$TASK is made equal to zero. This completes the ACTIVATE$TASK functions. If the new task is a low priority task, then the same functions are performed using the LOW$PRIORITY$TAIL pointer.

ACTIVATE$DELAY Procedure

This procedure is initiated by a call with the byte NEW$TASK containing the number of the task to be put on the delay list and the byte NEW$DELAY containing the value of the associated delay.

Interrupts are disabled and the busy bit of this particular task is checked. If the busy bit is set the STATUS byte is set to zero and the procedure returns without activating the delay. If the busy bit is not set the integer value DIFFERENCE is set equal to the NEW$DELAY value. POINTER$0 is set equal to the DELAY$HEAD. POINTER$1 is set to zero. The DO WHILE loop executes until POINTER$0 equals zero or DIFFERENCE is less than zero. Remember that the proper place to insert the new delay is being searched for, and that will be either at the end of the list (POINTER$0 = 0) or when the sum of the previous delays do not exceed the new delay value. The DO WHILE loop has POINTER$0, POINTER$1, OLD$DIFFERENCE, and DIFFERENCE keeping track of where the procedure is in the loop, while searching for the proper place to insert the new delay. The existing delays are sequentially subtracted from the remains of NEW$DELAY according to the link-listed order until the end of the list or a negative result is encountered indicating that the proper delay insertion point has been reached. At this point POINTER$0 contains the task number to be assigned to TASK(NEW$TASK).PNTR. POINTER$1 contains the task number immediately preceding the NEW$TASK such that TASK(POINTER$1). PNTR = NEW$TASK and our link list is fully updated, with the actual delays yet to go. If POINTER$0 = 0 it means that the new delay is larger than any of the other delays and therefore should go on the end of the list so TASK(NEW$TASK).DELAY is set equal to the DIFFERENCE. If

POINTER$0 is not equal to zero then if POINTER$0 equals POINTER$1 (indicating that there were not any delays previously listed), then TASK(POINTER$1).PNTR is set equal to zero. TASK(NEW$TASK).DELAY is set equal to the OLD$DIFFERENCE and TASK (POINTER$0).DELAY is set equal to the negative of DIFFERENCE which at this point is negative, thereby resulting in a positive unsigned number. The reader is encouraged to implement an example (see Delay Structure section) to prove that the above approach is valid. Particular attention should be paid to the contents of the two pointers, as they are the key to the procedure. The final function of this procedure is to set the BUSY$BIT and DELAY$BIT in the TASK(NEW$ TASK).STATUS byte. The byte named STATUS which is returned by this procedure is set equal to the status of the new task. If it is desired to have interrupts enabled, they must be enabled after the procedure return instruction. The reason for such a complex method of activating a delay will become apparent in the following section.

DECREMENT$DELAY Procedure

The first delay on the linked-list is decremented and, if it is zero, the associated task is put on the appropriate ready queue. The next delay (if any) is checked to see if it is zero and if so, that task is put on the appropriate ready queue, etc. A loop is performed until either no delay or a non-zero delay is found. The procedure then returns.

It is assumed that this procedure is part of an interrupt routine and that the interrupts are disabled during its execution. Interrupts cannot be enabled during changes to any of the linked-lists or else recovery may not be possible.

This procedure begins by checking to see if there are any active delays. If DELAY$HEAD = 0 then this procedure returns immediately. Otherwise it decrements the first delay. If this delay goes to zero then the associated task number is passed to the ACTIVATE$ TASK procedure as the OFF$DELAY byte. A new DELAY$HEAD is chosen from the next link-listed delay and that delay checked for a value of zero which will happen if the first two or more delays are equal. This loop is accomplished by the DO WHILE DELAY$ HEAD <> 0 AND TASK(DELAY$HEAD).DELAY = 0; This procedure is designed to require very little CPU time unless a delay times out. The DO WHILE loop is bypassed if the resulting delay value is not zero. A certain amount of care should be exercised to insure that many delays do not all time out at the same time. One method would be to modify the ACTIVATE$DELAY procedure to insure that there are no zero entries in the delay bytes. The basic procedure, however, assumes that the clock "tick" timing will be chosen to minimize the above potential problem.

CASE$TASK Procedure

This procedure performs the function of calling the task indicated by the contents of the RUNNING$TASK byte. All listed tasks are called in this manner. The CASE$TASK procedure is called by the DISPATCH procedure. When a particular task has completed execution it returns to the CASE$TASK procedure which then resets the BUSY$BIT and the READY$BIT and returns to the DISPATCH procedure after setting RUNNING$TASK equal to zero. This procedure allows a task to relist itself immediately upon returning from execution.

PREEMPT PROCEDURE

The PREEMPT procedure is called whenever it is possible that a high priority task has been put on the ready queue while a low priority task was in the process of execution. An example will illustrate:

Assume that the control system is being interrupted by the 60 Hz line frequency and a register is being incremented each time this 16.67 ms edge occurs. When the register gets to 60 (indicating that one second has passed), the register is zeroed and the high priority timekeeping task is put on the ready queue. Assume also that a low priority data logging task was running when this interrupt occurred. The interrupt routine calls PREEMPT. If a high priority task is running, PREEMPT simply returns. But in our example, a low priority task is running so PREEMPT transfers RUNNING$TASK to PREEMPTED$TASK and calls DISPATCH, which calls CASE$TASK, which calls the time-keeping task. When the time-keeping task has completed, it returns to CASE$TASK which returns to DISPATCH which returns to the PREEMPT procedure which returns to the interrupt routine which returns to the interrupted low priority data logging task if no other high priority tasks are on the ready queue. If the high priority ready queue is not empty, any and all high priority tasks will be completed before the interrupted routine is returned to. PREEMPT refuses to return to the interrupt routine until HIGH$ PRIORITY$HEAD is equal to zero. It is important to note that a low priority task will not be preempted unless the PREEMPT procedure is called. As noted above, it is normally called from the interrupt routine which interrupted the low priority task, but there is nothing to prohibit PREEMPT from being called from inside a low priority task procedure.

DISPATCH PROCEDURE

This procedure calls a high priority task if HIGH$ PRIORITY$HEAD is not equal to zero, restores a preempted task if PREEMPTED$TASK is not equal to zero, calls a low priority task if LOW$PRIORITY$HEAD is not equal to zero, and simply returns if there is nothing to do, all in order of priority. The DISPATCH procedure is called from the main program loop which must enable interrupts as DISPATCH disables interrupts as soon as

it is called. It is also called by the PREEMPT procedure. RUNNING$TASK must be 0 when this procedure is called.

PL/M-86 PROCEDURES

Because the block structure and levels are so important to the understanding of the following procedures, they have been indented according to level. This was a simple task accomplished by no indenting for level one, indenting once for level two, etc. The resulting attractive, easy to follow format was worth the effort to increase the initial level of understanding for readers of this application note who are not intimately familiar with PL/M.

Everything except the very simple main program loop has been made into procedures. Interrupt routines and tasks are also procedures. Keeping track of interrupts, calls, and returns is easy for PL/M and a violation of the block structure through such devices as GOTO targets outside the procedure body is the best way the author knows to crash and burn. Honor the power of the structure, accept the limitations involved, and checkout and debugging will be a pleasure.

Since CASE$TASK references the individual tasks, the task procedure structure was included in the PL/M-86 compilation. All the user has to do is insert the particular task code in place of the /*TASKnn CODE*/ comment, define the interrupt procedures and the system should be ready to run. Obviously, the user will desire to change the total number of tasks and the number of the FIRSTLOWPRIORITY$TASK.

INITIALIZATION AND THE MAIN LOOP

The last entry in the PL/M-86 program is the initialization process which essentially zeros the task multiplexer data and the main loop which loops until TRUE = FALSE, i.e. forever, with interrupts enabled. The STATUS = STATUS instruction simply insures that the loop can be interrupted as the instruction following an ENABLE instruction is not interruptible.

These few instructions are included for information only and will need to be expanded considerably for use in a real-world system. The task multiplexer procedures were checked out on an iSBC 86/12™ computer running under random interrupt control and these instructions were the minimum necessary to cause the system to run. As was stated earlier, the following source code does not include any interrupt procedures and these will have to be generated following the format explained in the PL/M-86 programming manual.

ADDITIONAL IDEAS

Resource allocation is a feature that could be added to the task multiplexer. To keep it simple and yet avoid the deadlock problem (two tasks each grab a resource that the other needs), an extra array can be added to the TASK(n).XXX structure in which each bit in the byte (or word), represents a resource necessary for the execution of a task. A RESOURCES$STATUS byte can then keep the dynamic busy status of the system resources (printers, terminals, floating point math packages, etc.). When the CASE$TASK procedure is called, the resources required by the next RUNNING$ TASK can be compared to the RESOURCES$STATUS byte to see if the required resources are available. If they are, the following PL/M-86 statement will update the new status of the resources:

```
RESOURCES$STATUS = RESOURCES$STATUS OR
        TASK(RUNNING$TASK).RESOURCES;
```

However, if the resources are not available, the CASE$ TASK procedure can return the task to the ready or delay list and try again later. When the task has completed, the following PL/M-86 statement will update the resources status byte:

```
RESOURCES$STATUS = RESOURCES$STATUS AND NOT
        TASK(RUNNING$TASK).RESOURCES;
```

Message passing from task to task may also be necessary. Assuming that a task will have only one message at a time to deliver or receive, another byte could be added to the task structure such that TASK(RUNNING$TASK).MESSAGE could represent a byte containing the number of the task wishing to deliver a message to the RUNNING$TASK. Since a task can call CASE$TASK which in turn will call another task, message block parameters can be passed directly from one task to another. The task that calls CASE$TASK must handle the necessary housekeeping involved in recovering after the message has been passed. Of course, the data structure would have to be expanded to accommodate the message parameters and blocks. For further ideas involving message handling refer to the RMX/80™ user's guide.

Two additional relatively simple procedures could be added to obtain the SUSPEND and RESUME features of the RMX/80™ system. Remember that if the BUSY$BIT is set in a TASK(n).STATUS byte and the task is unlisted, then it cannot be listed. If it is desired to dynamically enable and disable a task, this bit could be set by a SUSPEND procedure and reset by the RESUME procedure.

SOURCE CODE

```
TM86:DO;

DECLARE TOTAL$TASKS LITERALLY '10';
DECLARE TRUE LITERALLY '0FFH';
DECLARE FALSE LITERALLY '0';
DECLARE BUSY$BIT LITERALLY '10000000B';
DECLARE READY$BIT LITERALLY '01000000B';
DECLARE DELAY$BIT LITERALLY '00100000B';
DECLARE FIRST$LOW$PRIORITY$TASK LITERALLY '6';

DECLARE TASK(TOTAL$TASKS) STRUCTURE(PNTR BYTE, STATUS BYTE, DELAY BYTE);
DECLARE HIGH$PRIORITY$HEAD BYTE, HIGH$PRIORITY$TAIL BYTE;
DECLARE LOW$PRIORITY$HEAD BYTE, LOW$PRIORITY$TAIL BYTE;
DECLARE RUNNING$TASK BYTE, PREEMPTED$TASK BYTE;
DECLARE STATUS BYTE, NEW$TASK BYTE, NEW$DELAY BYTE;
DECLARE DELAY$HEAD BYTE AT (@TASK(0).PNTR);

ACTIVATE$TASK: PROCEDURE; /* ASSUMES NEW$TASK<>0 */
   DISABLE;
   IF (TASK(NEW$TASK).STATUS AND BUSY$BIT)<>0 THEN STATUS=0;
   ELSE /* SINCE TASK IS NOT BUSY */ DO;
      IF NEW$TASK < FIRST$LOW$PRIORITY$TASK THEN DO;
         IF HIGH$PRIORITY$TAIL<>0 THEN DO;
            TASK(HIGH$PRIORITY$TAIL).PNTR=NEW$TASK;
            END;
         ELSE /* SINCE HIGH$PRIORITY$TAIL=0 THEN */ DO;
            HIGH$PRIORITY$HEAD=NEW$TASK;
            END;
         HIGH$PRIORITY$TAIL=NEW$TASK;
         END;
      ELSE /* SINCE TASK IS LOW PRIORITY THEN */ DO;
         IF LOW$PRIORITY$TAIL<>0 THEN DO;
            TASK(LOW$PRIORITY$TAIL).PNTR=NEW$TASK;
            END;
         ELSE /* SINCE LOW$PRIORITY$TAIL=0 THEN */ DO;
            LOW$PRIORITY$HEAD=NEW$TASK;
            END;
         LOW$PRIORITY$TAIL=NEW$TASK;
         END;
      TASK(NEW$TASK).PNTR=0;
      TASK(NEW$TASK).STATUS=TASK(NEW$TASK).STATUS OR
            BUSY$BIT OR READY$BIT;
      STATUS=TASK(NEW$TASK).STATUS;
      END;
   NEW$TASK=0;
   RETURN;
   END ACTIVATE$TASK;
```

```
ACTIVATE$DELAY: PROCEDURE;/*ASSUMES NEW$TASK, NEW$DELAY<>0*/
   DECLARE POINTER$0 BYTE, POINTER$1 BYTE;
   DECLARE OLD$DIFFERENCE INTEGER, DIFFERENCE INTEGER;
   DISABLE;
   IF (TASK(NEW$TASK).STATUS AND BUSY$BIT)<>0 THEN STATUS=0;
   ELSE /* SINCE TASK IS NOT BUSY */ DO;
      DIFFERENCE=INT(NEW$DELAY);
      POINTER$0=DELAY$HEAD;
      POINTER$1=0;
      DO WHILE POINTER$0<>0 AND DIFFERENCE>0;
         OLD$DIFFERENCE=DIFFERENCE;
         DIFFERENCE=DIFFERENCE-INT(TASK(POINTER$0).DELAY);
         IF DIFFERENCE>0 THEN DO;
            POINTER$1=POINTER$0;
            POINTER$0=TASK(POINTER$1).PNTR;
            END;
         END;
      TASK(NEW$TASK).PNTR=POINTER$0;
      TASK(POINTER$1).PNTR=NEW$TASK;
      IF POINTER$0=0 THEN TASK(NEW$TASK).DELAY=LOW(UNSIGN(DIFFERENCE));
      ELSE /* SINCE DIFFERENCE<0 THEN */ DO;
         IF POINTER$0=POINTER$1 THEN TASK(POINTER$1).PNTR=0;
         TASK(NEW$TASK).DELAY=LOW(UNSIGN(OLD$DIFFERENCE));
         TASK(POINTER$0).DELAY=LOW(UNSIGN(-DIFFERENCE));
         END;
      TASK(NEW$TASK).STATUS=TASK(NEW$TASK).STATUS OR
            BUSY$BIT OR DELAY$BIT;
      STATUS=TASK(NEW$TASK).STATUS;
      END;
   NEW$TASK=0;
   NEW$DELAY=0;
   RETURN;
   END ACTIVATE$DELAY;

DECREMENT$DELAY: PROCEDURE;     /* ASSUMES INTERRUPTS DISABLED */
   DECLARE OFF$DELAY BYTE;
   IF DELAY$HEAD<>0 THEN DO;
      TASK(DELAY$HEAD).DELAY=TASK(DELAY$HEAD).DELAY-1;
      DO WHILE DELAY$HEAD<>0 AND TASK(DELAY$HEAD).DELAY=0;
         OFF$DELAY=DELAY$HEAD;
         DELAY$HEAD=TASK(DELAY$HEAD).PNTR;
         TASK(OFF$DELAY).STATUS=TASK(OFF$DELAY).STATUS
             AND NOT(BUSY$BIT OR DELAY$BIT);
         NEW$TASK=OFF$DELAY;
         CALL ACTIVATE$TASK;
         END;
      END;
   RETURN;
   END DECREMENT$DELAY;
```