September 1979

Debugging Strategies and Considerations for 8089 Systems

Debugging Stragegies and Considerations for 8089 Systems

Contents

Our thanks to John Atwood and Dave Ferguson, the authors of this note. Both John and Dave are members of Intel's 8089 design engineering group. Please direct any questions you may have to your local Intel FAE (field application engineer) or to MPO Marketing at Intel, Santa Clara.

INTRODUCTION

The Intel 8089 is the first integrated I/O processor available. This I/O processor (IOP) makes available the power of I/O channels, as used in mainframes and minicomputers, in a microcomputer form. Designed as part of the MCS-86™ family, the IOP can be interfaced with the MCS-80™ and MCS-85™ families as well.

An I/O channel is basically a processor remote from the main CPU, which independently runs I/O operations upon command of the CPU. To relate the 8089 to existing LSI components, it is similar to a microprocessor that is time-multiplexed with a DMA controller, but with two channels available. However, since the 8089 processor is optimized for I/O and multiprocessor operations, and the DMA has been made much more flexible than existing DMA controllers, a truly general purpose and powerful I/O control system is available on one chip.

Due to the uniqueness of the 8089, this application note was written to review debugging strategies and point out possible pitfalls when developing an IOP system. Debugging an IOP system is very similar to debugging microprocessor/DMA controller systems, and many of the techniques described here are standard microprocessor techniques. However, several factors are present which can complicate the debugging process:

1. Multiprocessor Operation

Although usable by itself, the IOP is designed to be used with other processors. All factors normally encountered with multiprocessor operation, including bus arbitration, processor communication, critical code sections, etc., must be addressed in the design and debug of an IOP system.

2. DMA Tie-In to IOP Program Execution

The relationship between IOP program execution and DMA transfers and termination is different from earlier DMA controllers and should be fully understood to properly run the system.

3. Dependency of Programs on Real-Time I/O Operations

Requirements by I/O devices for maximum data rates and minimum latency times force the software programmer to be aware of hardware timing constraints and can complicate program debugging.

4. Dual Channel Operation

Related to multiprocessor operation and real-time dependencies, the two independent channels available on the 8089 may have to be coordinated with each other to make the whole system function. Dependence of one channel on the other can also complicate debugging.

Due to the complexities of running in a real-time environment, as many steps as possible should be taken to facilitate debugging. A major help here is to make sure as much of the hardware and software as possible is working before running real-time tasks. This is a good practice anyway, but it should be reemphasized that a complex multichannel system can quickly get out of hand if more than a few things are not right.

An aid to debugging any system is a clean, well organized system design. The 8089 lends itself to structured, modular software interfaces to the host CPU, via the linked-list initialization structure, and parameter communication through the parameter block (PB) area. Some of the aspects of structured programming that aid debugging are:

- *Top Down Programming* — The functions done by low-level routines are well understood, and the number of program fixes, which can cause more errors, is minimized.
- *Program Modularity* — Small, easy to manage subprograms can be debugged independently, increasing the chance that the entire system will work the first time.
- *Modular Remoteness* — By having all program modules communicate only through a well-defined interface, one module's knowledge of the "inner workings" of another is minimized. System software complexity is reduced. Updates to program modules are more reliable, too.

Two major areas of debugging will be outlined here — static (or functional) debugging in which the hardware and software are not tested in a real-time environment, and real-time debugging. Applying a logic analyzer to IOP debugging will also be explained, and a review of IOP operation and potential problems will be done.

STATIC (OR FUNCTIONAL) DEBUGGING

The predominant errors in a system, when first tried out, are either errors in implementation (i.e., wrong hookups or coding errors), or an incorrect implementation (a wrong assumption somewhere). Most of these bugs can be found through static debugging techniques that are usually easier to work with than real-time testing.

Hardware Testing

Static hardware testing is done mainly to see if all individual parts of the system work, so the whole system will "play" when run. The level of testing can run from checking for continuity and shorts (which finds only hookup errors) to trying to move data around and running I/O devices from a monitor or special test programs (which can also find incorrect circuit design). In all but the simplest systems, the latter approach is recommended since it is a step towards software debugging.

Several approaches to hardware testing will be covered. Running diagnostic programs (such as a monitor) out of the IOP's host system, in both the LOCAL and REMOTE modes, will be covered. The case where the host system cannot support diagnostic software and must have an external processor to exercise the IOP and its peripherals will also be explained.

The case where the host system can run diagnostics or test programs that have interactive user I/O, such as a CRT terminal or teletype, provides the most straightforward way to test the IOP. Naturally, before these programs can be run, the basic hardware must be correct enough to run programs. When this point is reached, a monitor program can be used to exercise memory and I/O controllers on the system bus.

It should be mentioned that aids, other than just testing with software, are helpful for hardware debugging. While a necessity for real-time debugging, a logic analyzer is also a definite help for static hardware debugging. Its main use in hardware debugging is showing timing relationships between address or data paths and other signals. It is especially useful for functional software debugging, to be described shortly. The last debugging section outlines the use of an analyzer with the IOP. Of course, an oscilloscope, logic probes and pulsers, etc., can be used to trace out specific logic or timing problems.

LOCAL Mode

When the IOP is running in the LOCAL Mode, all I/O controllers and memory are accessible by the host or controlling CPU. Thus a standard monitor, such as the one supplied with the SDK-86 or available for the iSBC-86/12™ development kit, can exercise all hardware on the bus.* The breakpoint routines, however, will not work due to the different instruction set. The 8086 or 8088 is best suited for running the IOP in the LOCAL mode due to identical status lines and bus timing, as well as the Request/Grant line, which eliminates bus arbitration hardware. Figure 1 shows the general LOCAL mode configuration.

*The SDK-86 serial monitor is a good basis for a general 8086 monitor. The IOP cannot be used directly with the SDK-86, since the 8086 is running in the minimum mode. The SDK-86 can be converted to run in the maximum mode, if desired.

REMOTE Mode

From a system design standpoint, running the IOP in the REMOTE Mode is advantageous in that it removes the I/O bus cycles from the system bus. Normally, the remote I/O is not accessible to the host CPU. Until the IOP is able to run its own test programs to transfer data from the REMOTE bus to the system bus, I/O controllers and memory on the REMOTE bus will be invisible to the host. To get around this problem during prototyping, either an external processor interface can be used (see next section), or a temporary bypass can be made to access the REMOTE bus from the system bus.

Bypassing the normal REMOTE/SYSTEM interface is a handy technique for doing preliminary debugging on the REMOTE bus. This can be done by memory-mapping the IOP's I/O space into an unused portion of the host CPU's system memory space. When accessing this space, the IOP access to its own I/O space is disabled, and a separate set of address buffers, transceivers and bus control signal buffers are enabled. Reads and writes can then be done to the formerly inaccessible REMOTE bus by the host CPU.

A simple system (Figure 2) implements this bypassing scheme. It was designed for just forcing or examining devices on the REMOTE bus and may not read or write correctly if the IOP is simultaneously trying to do bus cycles. A more sophisticated arbitration system would permit reliable run-time checking also.

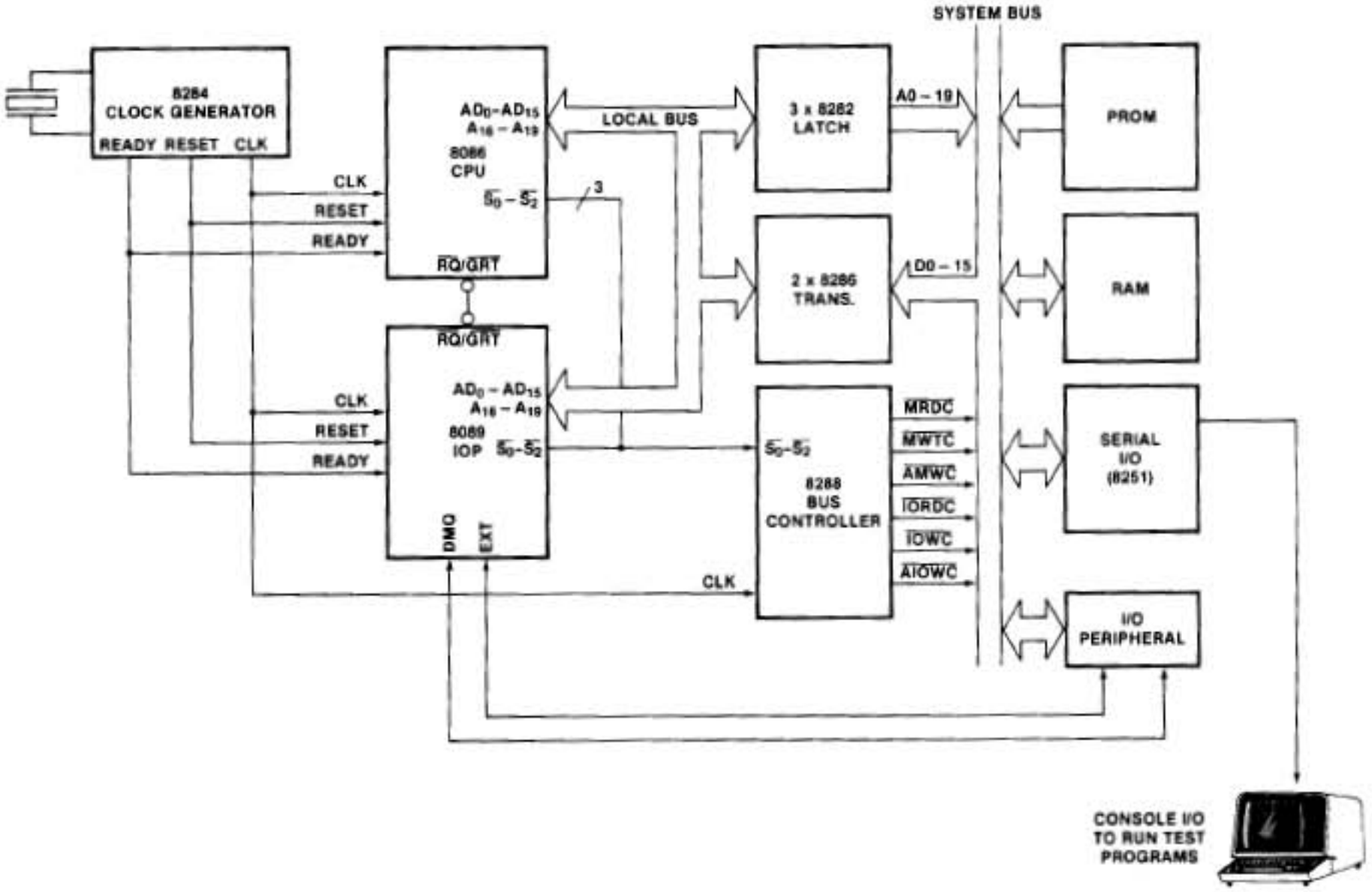


Figure 1. Generalized LOCAL Configuration—8086 in Max Mode

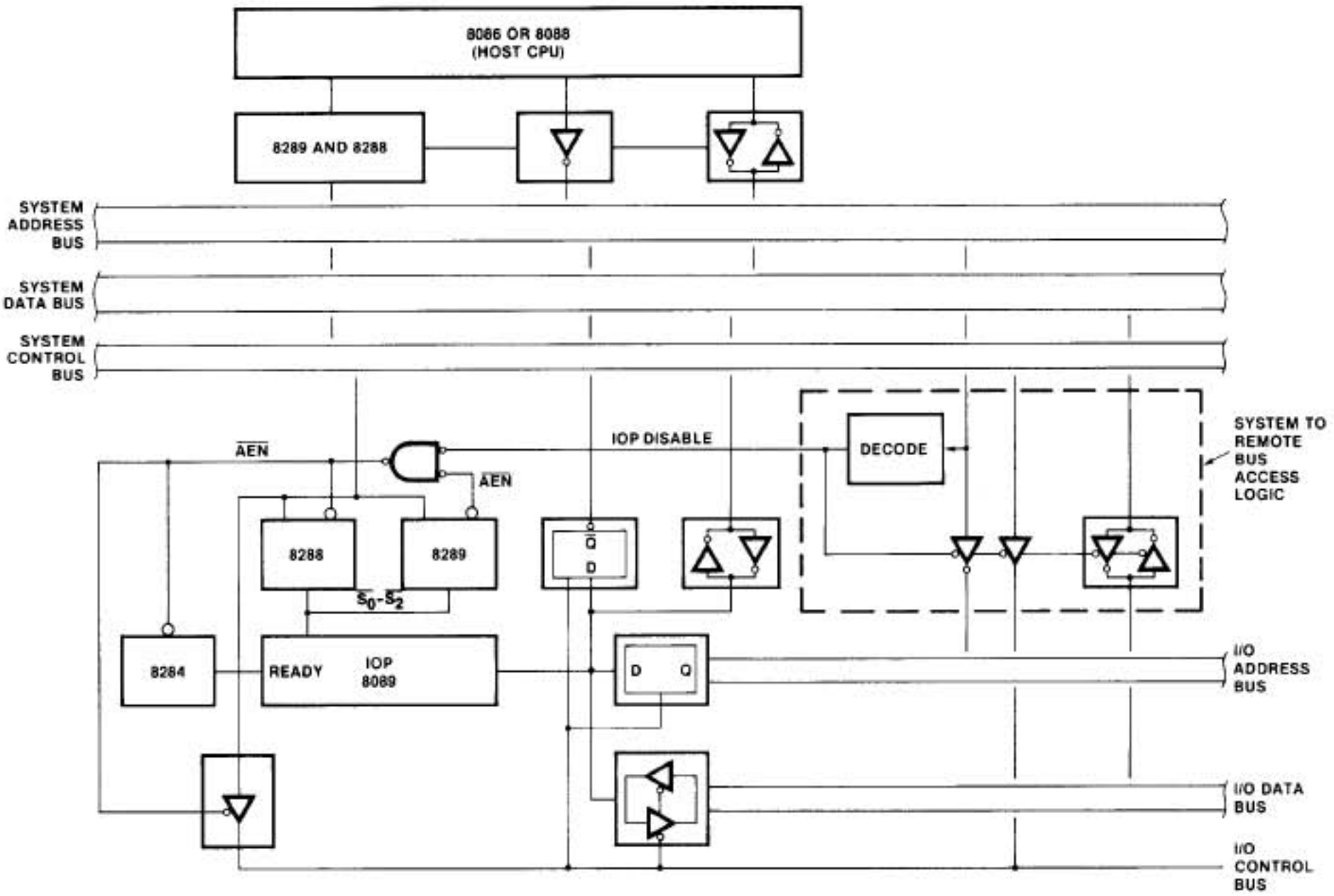


Figure 2. Remote Mode Bypass for Debugging

Running the IOP in the REMOTE mode, particularly if the MULTIBUS™ protocol is adhered to, has the advantage that the IOP can be exercised with any MULTIBUS-compatible processor. If the main processor is not amenable to being used as a debugging tool, another processor could be used to debug the hardware interface. If the microprocessor is of the same type as the intended host processor, software debugging can be done as well. A generalized REMOTE mode configuration using the MULTIBUS is shown in Figure 3.

External Processor Interface

A technique that can be used if the host processor cannot run any debugging or monitor routines is to have an external processor tie into the host processor's bus. This is useful if the main system CPU cannot run an interactive monitor or other debugging programs. If a MULTIBUS interface is being used, an 8289 bus arbiter and a set of address/data/control buffers can be used. A somewhat simpler system, similar to the remote bus access system mentioned above, could be used for static debugging of non-MULTIBUS systems. Again, if true bus arbitration is added (which brings us nearly to a MULTIBUS interface), it could also be used for run-time testing. Intel processors that have the MULTIBUS interface include the iSBC-80/20™, iSBC-86/12™, iSBC-80/10™, iSBC-80/05™, the Intellec® development systems, among others.

In the previously described systems, the external processor would disable the host CPU's access to the bus, either by some form of bus request or by a "brute force" disabling of the CPU's buffers. In the latter case, the external processor could only control the bus during a time that the CPU is halted, without destroying the program flow. Mapping the processor's memory space into the external processor memory space is the simplest method, but can impact programs being run on the external processor. If the processor under test utilizes the MULTIBUS interface (with bus arbitration), then a processor like the iSBC-80/30™ or iSBC-86/12™ could be used as the debug vehicle with no special hardware. A more flexible interface that would have less impact on the system memory space would have the addresses for the system under test generated from latches loaded by the I/O instructions from the external processor. This case must have software routines to interface to the I/O ports and handle the desired debugging routines (see Figure 4).

Software Testing

It is desirable to check as much of the IOP program as possible statically, since various tools and techniques are available which may not be usable during real-time

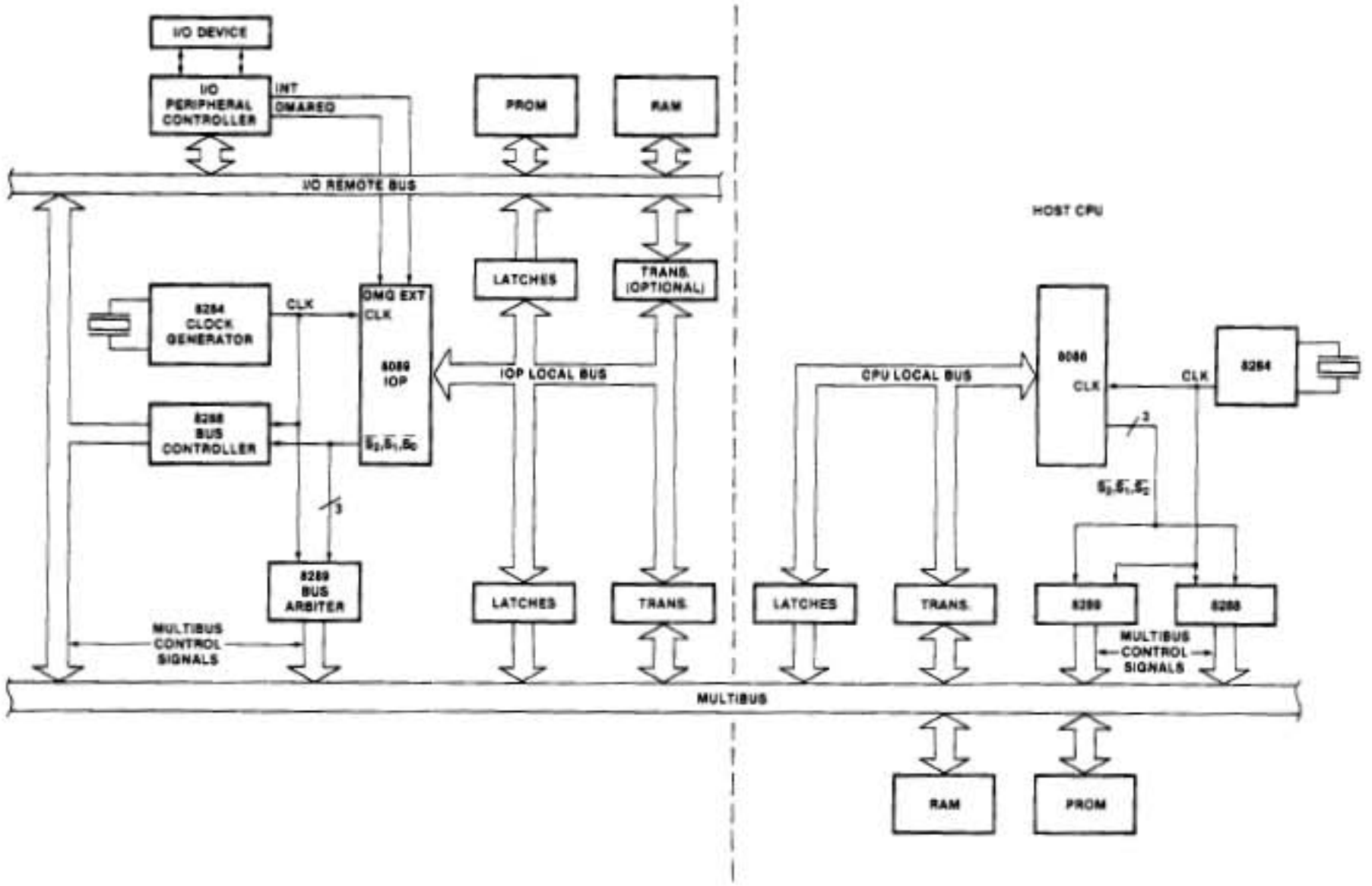


Figure 3. Generalized Remote Bus Using MULTIBUS Interface

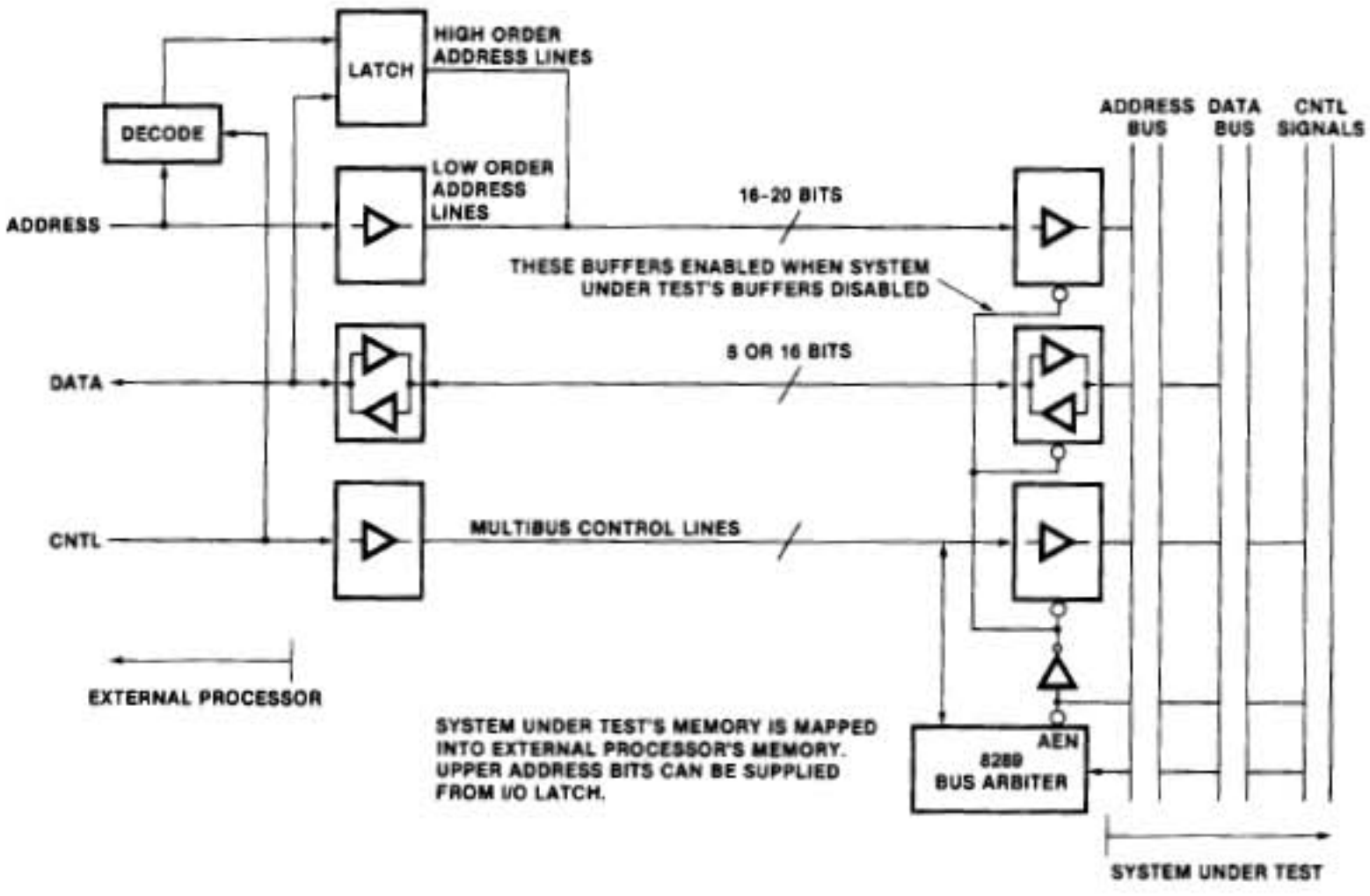


Figure 4. External Processor Interface

testing. This "static" software testing is not applicable to heavily I/O-dependent or DMA-dependent routines, but is best suited to longer computational or data handling routines. The idea is to test the correctness of algorithms, rather than seeing if the whole system runs.

There are two main approaches to functional software testing. One is to essentially run the program in real time and monitor program flow on a logic analyzer. The difference between this and real-time testing is that program subsections can be tested separately by using different TP (Task Pointer) starting addresses. If it is necessary to set up certain registers or parameters in memory, a small "setup" program can be run after initialization, which can load up registers or memory, then jump to the program section desired.

Another technique is to run the programs with breakpoint routines so that one can step through code segments and follow program execution. Software breakpoints are usually implemented by inserting a jump or restart to a monitor routine at the breakpoint location. This jump or restart is machine language dependent so, unfortunately, the existing breakpoint routines within monitors for the 8080 or 8086 are not applicable.

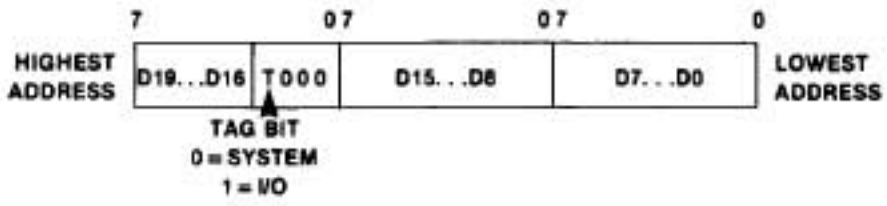
New routines tailored to the 8089 can be used, and, if done properly, can even be used to examine programs running on a REMOTE bus. Using breakpoints is somewhat complicated on the 8089 because the minimum instruction length is two bytes. There is no absolute CALL instruction, only a relative one (which would have to have its displacement recalculated each time it was used). But, with a several-byte absolute jump inserted at each place a breakpoint is desired, full breakpoint capabilities can be obtained.

There are many ways the breakpoints can be implemented. When a breakpoint is reached, the 8089 itself could output the machine state to a console through its own routines. Better suited to debugging, though, is a system that has the 8089 place its machine state in memory, alert the host processor, and then halt. The host then picks up the 8089's state and can treat it in the same way it runs its own breakpoint routines. Since the host processor is more likely to be running a monitor or some other kind of debugging routine (and most likely has at least temporary console I/O), it is the logical system to initiate and examine 8089 breakpoints. If the IOP is running in the REMOTE mode, and the host processor has access to the I/O bus via the scheme mentioned in the hardware debugging section, then IOP programs running on the REMOTE bus can be examined.

The breakpoint itself can consist of an escape sequence that is used to save the TP value and jump to the save routine, or just a jump to the save routine. This routine saves all register contents for the channel the breakpoint is in, signals the host processor, and stops the IOP. All user programmable registers (GA, GB, GC, IX, MC, BC, TP), as well as the pointer tags, are accessible. The PP (Parameter Pointer) and PSW are not normally accessible, but if the generation of the CA is such that the IOP can send itself a CA, then by sending a CA HALT, the PSW will appear at PP + 3. Remember that since the IOP doesn't have arithmetic or logical condition codes, the PSW is not as important as in other machines.

The most straightforward way to pass data from the IOP to the host processor is through the PB (Parameter Block) area since the PP will normally remain relatively fixed throughout the IOP program. In order not to infringe on the PB areas used by the programs, an area 18 bytes long should be allocated at the end of the PB block to hold the register contents. Using other areas to store the register data requires saving and reloading a pointer register as part of the breakpoint escape sequence.

The data returned from the breakpoint save routine will appear to the host processor as a sequential block of data in the PB area. Sixteen-bit data can easily be extracted, but 20-bit pointer data will have to be reconstructed from the move pointer (MOVP) format:



Several means are available to signal the host processor that a breakpoint has been reached. A bit could be set in memory or an interrupt sent to the CPU. The best way, though, is to use the BUSY flag (at CP + 1 or CP + 9). After starting the IOP, the BUSY flag is set to FF. When a breakpoint is reached, the IOP performs its save routine and does either a software or CA HALT. These result in clearing the BUSY flag, which then signals the CPU to obtain valid breakpoint data. The CPU can then restart the IOP by either a CA START or CA CONTINUE.

The breakpoint routine outlined above will work for a "one-shot" test. However, to be more useful as a general purpose debugging tool, some refinements must be added. To keep from destroying the program whenever a breakpoint is placed, the supervisory program running from the host processor must save the IOP code that is occupied by the escape sequence. When the breakpoint is completed and IOP execution is to resume, the host program restores the IOP code, sets the TP in the CB area back to where the breakpoint was placed, and sends a CA START. Since the length of each instruction can be easily found from bits 1-4 of the opcode, a single stepping function can also be done.* By the time this is implemented, the host program is becoming a full-fledged debugging routine. Appendix 3 describes a debugging program that makes use of the ideas presented here.

Breakpoint routines can be quite useful, but some restrictions and limitations should be mentioned. The processor examining the breakpoints must have access to the IOP program memory, either directly, or through IOP programs that simulate direct access. The program memory must be in RAM. The breakpoint must be

*The formula for length of instructions is: length (in bytes) = 2 + 1 (if bits 1,0 = 01) + 1 (if bits 3,2 = 01) + 2 (if bit 3 = 1) + 2 (if LPDI).

placed on an instruction boundary, and multiple breakpoints must not be placed so that they overlap. There may be some impact on the PB area. CA generation may have to be different than usual. But, despite these limitations, the breakpoints offer a useful and more conventional software debugging tool than analyzers.

REAL-TIME TESTING

Running an IOP program in its final environment with real I/O devices is the true test of dynamic operation. The program is no longer in a static, isolated environment. The demands of DMA and multiprocessing may reveal unplanned timing dependencies or critical section problems. There may also be sections of hardware or software, which couldn't be tested statically, that may have bugs. The whole purpose of static or functional testing is to dig these problems out while convenient debugging tools can be used. Since there are no simple techniques for real-time debugging, the use of a logic state analyzer and techniques to fully understand the IOP's real-time operation will be emphasized.

Multiprocessing operations and real-time asynchronous I/O requests can cause the timing complexity of the system as a whole to rise beyond the point of complete comprehension by an individual. It is then essential that techniques to ensure correctness are used. These include good design methods, especially a clean, well-structured design, as well as good testing. A thorough test requires the attitude that the system should be tested for failures, rather than tested for correctness. In other words, one should try to make the system fail, tests should be chosen that will put the worst stress on critical timing areas.

The best way to do this is to write a diagnostic program that puts the CPU, IOP, and I/O devices through the worst conceivable timing and program combinations. Ideally, the program should be self-checking so that it can be run without supervision, printing any data or program errors that occur, much like a memory test.

The two main real-time problem areas are insufficient data rates or latency, and critical section problems. To test for data rate problems, run the system clock at its lowest expected frequency and use memory and I/O with maximum expected wait states. Identify the tightest program timings and try to have these sections coincide with worst case DMA or other heavy bus utilization (see dual channel operation later). Critical section problems can occur when two independent processors communicate with each other with improper "handshaking." This can result in one processor missing another's message, or even having both processors hang up, waiting for each other to go ahead. The 8089 provides aids to these problems, including the TSL instruction (to implement semaphores) and the BUSY flag. However, any interprocessor communication (including one channel of the IOP to the other) should be checked. Beware of cases when one processor is running considerably slower than the other (due to DMA overhead or chained instruction sequences).

The techniques for real-time debugging evolve from functional testing using a logic analyzer. For all but the simplest systems, an analyzer is essential, since it can graphically show program execution and timing relationships during real-time execution. Another aid is a delayed oscilloscope. Triggering the scope from the logic analyzer, the delay can be adjusted so that any signal in the system can be monitored.

To facilitate the use of the logic analyzer, especially if its memory is not very deep or when using it to trigger an oscilloscope, a repetitive system can be used to continually update the display. Using a repetitive reset helps to debug the software-hardware interface, since oscilloscope or logic analyzer probes can be readily moved around the circuit to observe new signals without manually retriggering the display. At its simplest, the reset to the host processor can be strobed, say every 10 ms. The processor will then provide the two channel attentions (CAs) that are needed to initialize the IOP. Where this isn't feasible, the CAs can be externally forced by either a string of one-shots or a simple processor with timing loops (such as a SDK-85 or SDK-86). See Figure 5 for initialization timing.

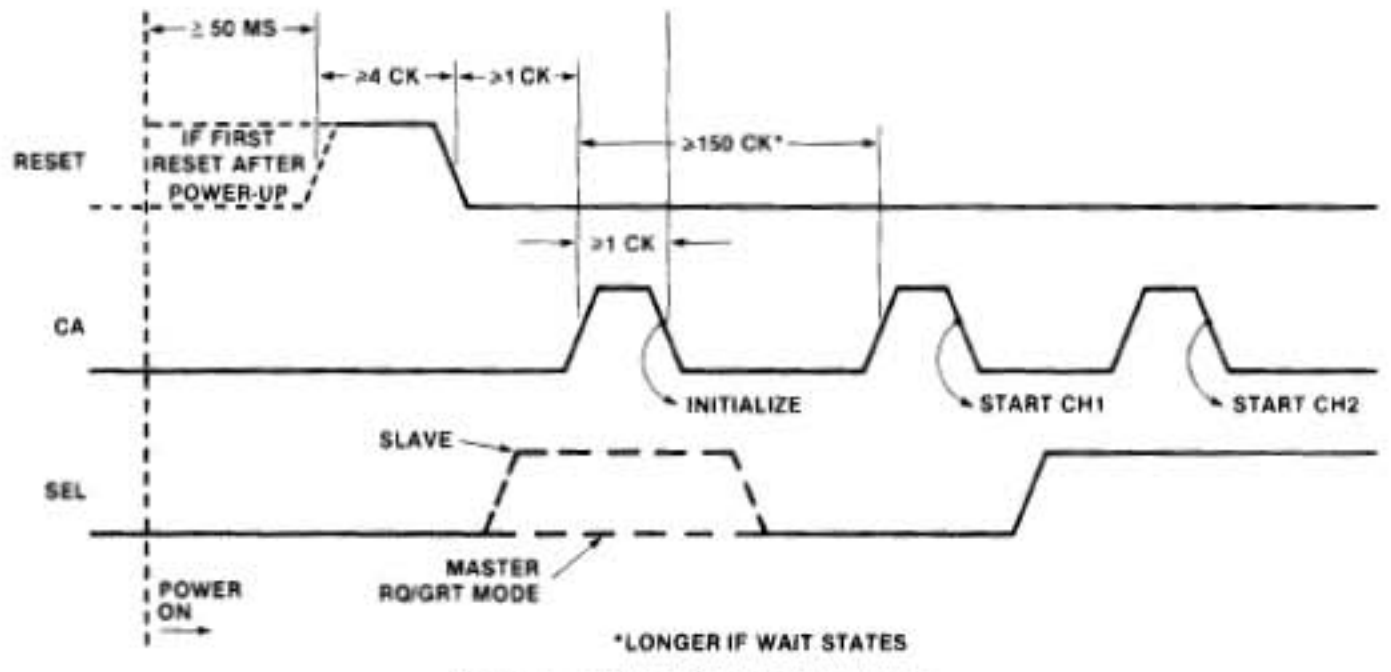


Figure 5. Initialization Input Sequence

Memory protection of the IOP and system programs is helpful when debugging DMA operation. It is quite easy for runaway DMA to wipe out memory. Another precaution to avoid this problem is to set an upper limit on the number of bytes transferred by always specifying a byte count termination.

Logic Analyzer Techniques

In the absence of other powerful debugging systems, the logic analyzer has shown to be an extremely useful tool. Because of its importance in debugging an IOP system, some basic techniques and observations that relate to monitoring IOP operation will be reviewed here. The particular brand or type of analyzer used is not too important, but would be desirable to have the following features:

- At least a 24-bit data width
- Flexible triggering and qualification control
- Display after triggering on a sequence of states
- Capability for hexadecimal data display

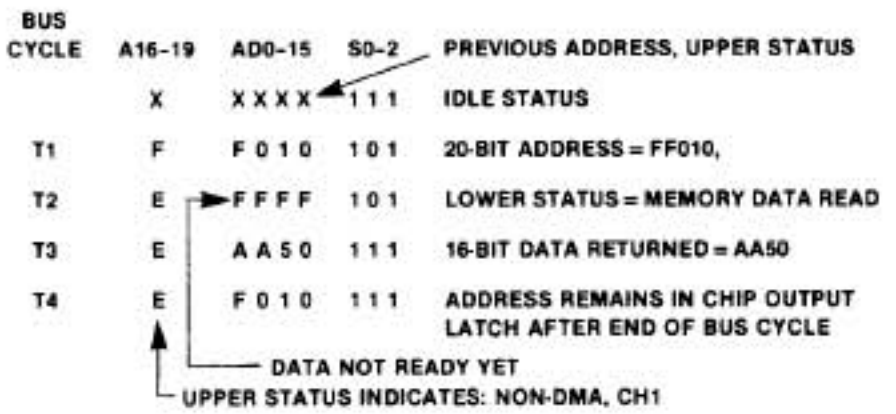
It is best to hook up to the address/data lines at the IOP, as opposed to looking at the separate address and data lines, since 39 lines would be required just to look at address, data and status lines. The three lower status lines should be monitored to show the type of bus cycle being run. Other lines can be connected where needed, at places like the DRQ lines, the EXT lines or other lines related to the system.

For general purpose debugging, triggering the analyzer on the rising edge of the IOP clock shows the most useful data concerning bus cycles. Of course, using the falling edge may be necessary to check certain signals, particularly ones that are active only while the clock is low. The following discussion is based on sampling data on the clock's rising edge.

One should be careful when setting up the triggering for the analyzer that the desired event is what is displayed and not a later event with the same trigger word. This can happen when the logic analyzer is in the repetitive trigger mode. It may retrigger before the system actually resets. A sequence restart feature is helpful.

The basis of following program execution and DMA on a logic analyzer is to follow an 8089 bus cycle, which is identical to a 8086 and 8088 bus cycle. The following diagram shows a typical 8089 bus cycle.

For general purpose debugging, displaying every clock is useful, but for quickly finding one's way around a program, the analyzer can be qualified so that only instruction fetches (status = 100 or 000), with ALE active, are trapped. A much more compact display of execution flow results.



As mentioned earlier, on a 16-bit bus, most instructions starting on odd addresses won't show the first fetch, since the internal queue is in use. It is a good idea in that case to use only even instruction boundaries as trigger words. When following dual channel operation, one should keep an eye on the upper status bits (S3-S6), since S3 indicates which channel is running (0 = CH1, 1 = CH2), and S4 indicates DMA/non-DMA transfer (0 = DMA, 1 = non-DMA).

A REVIEW OF IOP OPERATION (With things to look out for)

When trying to get an unfamiliar system going for the first time, it is too easy to stumble on apparent problems that are really just unexpected operation modes or peculiarities of the machine. For this reason the basic principles of IOP operation will be reviewed here with special emphasis on possible problem areas or pitfalls that a user might encounter when debugging a 8089 system. The topics are covered generally in the order encountered when bringing up a system. For complete details of operation and some design examples, see the 8086 Family User's Manual.

RESET

RESET must be active (HIGH) for at least four clocks in order to fully initialize all internal circuitry. On power up, RESET should be held high for at least 50 microseconds. The chip is only ready to accept a Channel Attention (CA) one clock after RESET goes inactive.

Note that the SEL pin is sampled on the falling edge of the first CA after RESET to tell the 8089 whether it is a master (0) or a slave (1) for its request/grant circuitry. If a master, it will assume it has the bus from the beginning. If a slave, it will strobe the $\overline{RQ}/\overline{GT}$ Line to request the bus back and will not start any bus transfers until it has been granted the bus. If the $\overline{RQ}/\overline{GT}$ line is not being used, make sure the IOP comes up in the master mode.

Initialization

Upon the first CA after reset, a sequence of instructions is executed from an internal ROM. These instructions pick up parameters and load data from the linked list sequence (Figure 6). The instruction sequence is essentially:

MOVB SYSBUS from FFFF6
LPD System Configuration Block (SCB) from FFFF8
MOVB SOC from (SCB)
LPD Control Pointer (CP) from (SCB) + 2
MOVBI "00" to CP + 1 (clears BUSY flag)

Remember that four bytes must be fetched during an LPD. If on a 16-bit bus, with even addressed boundaries, only two fetches are needed. Otherwise (8-bit bus or odd boundaries), four fetches are needed.

Even though no bus cycles are run to fetch these instructions, the CH1 Task Pointer (TP) appears on the address latches during the short internal fetch periods. On power up, this value is meaningless, but if a repetitive RESET is used, the TP remains unchanged from the end of the last program run. See Figure 6 for the start of a typical initialization sequence as viewed on a logic analyzer.

Bit 0 in the SYSBUS field sets the actual (or physical) system bus width that the IOP expects. In the 8-bit mode, only byte accesses are made, and all 8-bit data should appear on the lower eight data lines. In the 16-bit mode, word accesses can be made (if the address is even), all data on even addresses appears on the lower eight data lines, and all data at odd addresses appears on the upper eight.

Bit 0 in the SOC field sets the physical width for the I/O bus. The same rules for the system bus apply here. Note that these bits should reflect the actual hardware implementation and are not to be confused with the DMA logical widths set by the WID instruction.

The R bit (bit 1) in the SOC field is used to change the mode of the $\overline{RQ}/\overline{GT}$ circuitry. When the IOP is on the same bus as an 8086, it is required to have the R bit be 0, with the 8086 as the master and the 8089 as the slave.

CA	A_{19}-A_0	S_3-S_0	T	COMMENTS
1	FF F F F	111		Trigger CLK ↑
1	FF F F F	111		
	FF F F F	111		
	FF F F F	111		
	E0 0 0 0	111		Bus un-tristated
	E0 0 0 0	111		
	FF C6 D	111		
10 {				TP to latch
CK	FF C6 D	111		
	FF F F 6	101	T1	Address loaded to latch
	EF F F F	101	T2	Data not ready yet (nothing on bus)
	EF F 0 1	111	T3	SYSBUS loaded into chip (01)
	EF F F F	111	T4	Nothing on bus
	EF F F 6	111		After bus cycle, address remains in latch
	EF F F 6	111		
	FF C6 D	111		TP is loaded to latch, even though fetches are from internal ROM
14 {				
CK	FF C6 D	111		
	FF F F 8	101	T1	Address to latch
	EF F F F	101	T2	
	EF F F 0	111	T3	1st 2 bytes of LPD data fetched (FFF0)
	EF F F F	111	T4	
	EF F F 8	111		
	EF F F 8	111		
	EF F F 8	111		
	EF F F 8	111		
	FF F F A	101		
	EF F F F	101		
	EF F F A	111		2nd 2 bytes of LPD data fetched (FFFA)
6 {				
CK	EF F F A	111		
	FF C6 D	111		

Figure 6. Start of Initialization Sequence On a 16-Bit Bus

The master (8086 or 8088) can never take the bus away from the slave (8089); only the slave can give back the bus. In other words, during DMA transfers, the 8089 would not have the bus taken away. This is the only mode compatible with the 8086 or 8088.

When two IOPs are being used on the same bus, the $\overline{RQ}/\overline{GT}$ circuitry can be put into an equal priority mode by setting the R bit to one. A slave can only be granted the bus if the master is doing unchained instructions or running idle cycles. The master can request the bus back from the slave at any time. The slave grants it if doing unchained instructions or if it is idling. The master and slave are put on essentially the same priority.

At the end of initialization, the "BUSY" flag of CH1 is cleared. For systems where the 8086 is waiting for the initialization sequence to end before giving another CA, it can set the BUSY flag high prior to initialization. The BUSY flag going low is a sign that the IOP is ready for another CA. It is important to remember that the IOP will not respond to, nor latch, a CA during an initialization sequence.

Channel Attentions

The main system processor initiates communications with the IOP through the Channel Attention (CA) line. As mentioned earlier, the first CA after system RESET initializes the IOP. All subsequent CAs cause the IOP to do a two-step process. It first fetches the Channel Control Word (CCW) from the appropriate channel at (PP) for channel 1 or (PP + 8) for channel 2. (SEL at the time of CA falling determines the channel for all following actions.) The lower three bits of the CCW Command Field (CF) are examined and then cause the IOP to execute the desired function.

Command Field (CF)

Control of task block programs is accomplished through the command field. The various CF functions are:

CF

000 — Examine other field only and set BUSY flag

001 — Start task program in I/O space

011 — Start task program in system memory

The start command causes the following instructions to be executed out of the internal ROM:

LDP CP from (CP) + 2 (CH1) or + 10 (CH2)
LDP TP from (PP) (for TP in system) or
MOVB TBP from (PP) (for TBP in I/O)
MOVBI "FF" to (CP) + 1 or + 9 (set BUSY flag)

111 — HALT channel. BUSY flag cleared to "00"

110 — HALT channel. Save state of machine and clear BUSY flag by executing:

MOVP TP to (PP)
MOVB PSW to (PP) + 3
MOVBI "00" to (PP) + 1 or + 9

The channel will HALT and the machine will continue execution on the other channel or go to idle if the other channel is idle.

101 — Continue channel. The channel is revived after a HALT by executing:

MOVP TP from (PP)
MOVB PSW from (PP) + 3
MOVBI "FF" to (CP) + 1 or + 9
(set BUSY flag)

Do not do a CONTINUE after initialization without doing a CA START first since the (PP) register in CH1 is used as a temporary register (to hold SCB) and is only correctly loaded by a CA START.

The upper 5 bits in the CCW will have affect if CF = 000 or upon a CA START. Some things to note about these upper fields are:

- *Priority Bit* — If both channels are doing tasks of the same overall priority, the tasks with the higher priority bit will run. If the priority bits are the same, execution will alternate between the two channels.
- *BLL Bit (Bus Load Limit)* — Keeps nonchained instructions from occurring more often than once every 128 clocks. However, channel attention or termination cycles, even on the other channel, may disrupt the exact time interval to the next instruction.

It should be noted that the setting or clearing of the BUSY flag occurs after the loading or storing of registers, so that in a system where the main CPU uses the BUSY flag as a form of semaphore to tell when the IOP is truly finished, there is no danger that the SCB, CP, PP or TP could be changed before the IOP loads them.

Also since DMA termination cycles and chained instruction execution have a higher priority than CA, it is possible for CA to be "shut-out" by these higher priorities running on the other channel. However, since CA is always latched (except during initialization), it won't be forgotten.

How Can a Channel be Halted?

Sometimes a channel may stop its operation unexpectedly. To see what could cause this, and to show the impact of halting a channel, the various ways of stopping a channel are explained:

HALTED CHANNEL — If the channel has never started after initialization, if it has received a CA HALT command or a software HALT, channel operation is suspended. If the other channel can run, it will, otherwise idle cycles will run. Only a CA START or CONTINUE can resume operation.

WAITING FOR A DMA REQUEST — If the channel is in a source or destination synchronized DMA transfer mode, it will wait until DRQ is active before running its synchronized transfer. To minimize the impact on the overall throughput of the chip, the other channel can run during these DRQ wait periods.

WAITING TO GET THE BUS BY $\overline{RQ}/\overline{GT}$ — If the IOP has given the bus away via $\overline{RQ}/\overline{GT}$, it won't initiate any bus transfers until it has the bus back. The machine will run up to just before T1 of a bus clock cycle and will three-state its address/data and status pins until it has been granted the bus.

WAITING FOR READY — When running bus transfers, READY is sampled at T3 of a busy cycle. If inactive, the whole chip will wait until READY goes active.

The last two cases of waiting (or "wait" states) stop the whole chip and do not permit the other channel to run. However, with READY inactive or with the bus not acquired, there is not much that can be done on the other channel anyway. These two cases only stop the chip when running bus cycles. Any internal operations can proceed without having the bus or with the system not READY.

Note the difference between when the chip is HALTed when using $\overline{RQ}/\overline{GT}$ and an external arbiter (8289) for bus arbitration. Not having the bus due to $\overline{RQ}/\overline{GT}$ will inhibit the bus cycle from even starting. Since the 8289 stops the chip by forcing $\overline{AEN}$ inactive, which goes through the 8284 clock generator to force READY inactive to the IOP (or 8086/8088), a bus cycle has already been started, with ALE asserted, and the address on the address/data lines. When the bus is obtained, operation proceeds at T3 of the bus cycle.

As will be mentioned later, many invalid opcodes will cause the machine to hang up. In these cases the address/data lines will point to where the bad opcode was fetched.

Task Execution

Although optimized for fast and flexible DMA operation, the IOP is also a full-fledged microprocessor. The 8086 Family User's Manual deals with programming strategies and other details. Some of the things to be noted during debugging will be mentioned here.

Instruction Fetching

Unlike the 8085 (but like the 8086), the 8089 labels all fetches from the instruction stream, whether OPCODE, offset, displacement, or literal data, as an instruction fetch on the status lines. In some cases, such as MOV R,I and ADD R,I, the instruction fetch time greatly exceeds execution time because literals are treated as instruction fetches. When following programs on a logic analyzer, triggering on status = 100 or 000 (instruction fetch) and a known program address is the handiest way to trace the flow of the program.

When running programs on a 16-bit bus, a 1-byte queue register comes into play, saving the upper byte fetched from the last instruction fetch, if not used by the previous instruction. This reduces fetch time and bus utilization since the odd byte doesn't need to be fetched again. An internal four-clock cycle fetches data from the queue. Like the internal ROM fetches, the task pointer is put out on the address/data lines, but no bus cycle is run.

The queue can have some possible unexpected affects that have to be taken into account during debugging. These apply only to 16-bit systems and are:

1. Instructions that start on odd boundaries will not likely have bus cycles run to fetch the odd byte unless jumped to, unless preceded by LPDI (which clears the queue), or an instruction that modifies the task pointer is executed. The latter causes the queue to be cleared so that part of an old instruction won't become part of the new one.
2. There is a queue register for each channel so loading or clearing the queue on one channel has no affect on the other channel's queue.
3. The second word of immediate data fetched by a LPDI is done during a pseudo-instruction fetch cycle that cannot make use of the queue or already fetched data. Thus, if on an odd boundary, fetching an LPDI will be byte, word, byte, byte, byte, and the queue will not be loaded.

When Can the Other Channel Interrupt Instruction Execution?

This will be explained more in the "dual channel" operation section, but a few points will be mentioned here. All instructions are made up of internal cycles, with each cycle composed of two to eight clocks. Each bus cycle is one internal cycle, but there can be internal cycles with no comunications to outside the chip. Internal cycles will be extended by the number of wait states in each bus cycle. Between any of these cycles, DMA from the other channel can intervene if the priorities permit it. Instruction fetching and execution can only interrupt instructions on the other channel when the instruction has been completed, not between internal cycles.

Registers

All the registers have some special purpose use in the Instruction Execution or DMA, but all except the CC register can be used as general purpose registers during instruction sequences. A few are loaded specially:

- *CP* — Is only loaded during an initialization sequence. There is one CP register that handles both channels. (All others are duplicated, one set for each channel.)
- *PP* — Is only properly loaded during a CA START command. It holds the SCB value after the initialization sequence.
- *TP* — This is included as part of the registers in the RRR field, but cannot be operated on unless you plan on having your program execution jump around. Everytime this is operated on, the queue is cleared. The TP is loaded from two words (address and displacement) on a CA START, LPD, or LPDI, and loaded from 3-byte MOVP format (see illustration on page 5) on a CA CONTINUE, and can be operated on using any register oriented instructions.

The following registers are loaded during program execution, but can have special effects:

- *CC* — The only thing that affects instructions in the CC register is the chaining bit. If chaining doesn't matter (if only one channel is being used without channel attentions, for example), then the CC register can be general purpose. However, for portability of programs, it is strongly suggested not to use the CC register except for altering DMA parameters and chaining.
- *MC* — Is a general purpose 16-bit register, but is also used to do a masked comparison either for DMA search/match termination or for the JMCE and JMCNE instructions.
- *BC, IX* — Both general purpose 16-bit registers. In instructions that reference memory using the AA field, if AA = 11, the IX register is incremented by the number of bytes fetched or stored.
- *Pointer Registers (GA, GB, GC and TP)* — Are 20-bit registers, but can also be used as 16-bit registers. Adds will carry into the upper 4 bits, but other operations (COMP, OR, AND) are done only on the lower 16 bits. Note that when used as pointers to system memory, it is possible to add a large 16-bit number to the pointer and to put the pointer into another 64K block of memory.

Sign Extension

All program data brought into the chip, either literals or displacements in opcodes, or program data fetched from memory, is sign-extended. Offsets used for calculating addresses are not sign extended. Any 8-bit data brought in has bit 7 sign-extended up to bit 19. Sixteen-bit data is sign-extended from bit 15 to bit 19. It is important to note this, because it can affect logical operations. For example, if one wanted to OR 0084H with 1234H in register GC, you couldn't do ORBI GC, 84H, because bit 7 would sign-extend into the upper byte. Instead, you should code ORI, 0084H to do this properly (note that this has a word for the immediate data). The non-ADD operations will cause the upper four bits of the pointer registers to be invalid since the upper four bits of the ALU come only from the adder.

Tags

It should be noted that the way the IOP knows which bus to access (system or I/O) is via the Tag bit associated with the pointer register used. The TAG can only be set in these ways: loading as a 16-bit register (MOV R,M, MOV R,I) sets TAG to I/O space, loading as a pointer (LPD, LPDI) sets TAG to a system space), or bringing the TAG in from memory by a MOVP instruction.

Effects of Invalid Opcodes

The upper 6 bits of the 2-btye opcode actually determine which opcode will be executed. If these bits are a valid opcode, but lower bits are invalid, the chances are good that the bad bits will be ignored. But if the upper six bits are invalid, there is a very good chance that the chip will hang up and stop execution in that channel. The only way to get out of this mode is to reset the chip. If this hang-up occurs, it can usually be traced because the last address of the instruction fetch will still be on the

address/data lines, showing where the program went astray.

Going from Instruction Execution into DMA

The XFER instruction places the current channel into the DMA mode after the next instruction. This permits one last instruction to start up an I/O device (start CRT display on an 8275, for example). However, in order for the IOP to get setup for DMA, the GA, GB, and CC registers should not be altered during this last instruction. Failure to observe this will probably result in an improper first DMA fetch. The WID instruction can be placed after XFER.

DMA Transfers

Incrementing/Non-Incrementing pointers

A memory or I/O pointer can be made to increment for each byte transferred during DMA or it can remain fixed. Incrementing is used primarily for memory block transfers, and non-incrementing is used to access I/O ports.

B/W Mode

Each DMA transfer is composed of separate fetch and store cycles so that 8/16-bit data can be assembled and disassembled, and translation and termination may also be easily handled. There are four possible transfers or B/W modes. They are:

B – B — 1 byte fetched, 1 byte stored
B/B – W — 2 bytes fetched, 1 word stored
W – B/B — 1 word fetched, 2 bytes stored
W – W — 1 word fetched, 1 word stored

The B/W mode used depends on the logical bus width (selected by the WID instruction), address boundary, and incrementing mode.

All systems with 8-bit physical buses will run in the B/B mode. On 16-bit physical buses the other modes are possible, depending on the logical widths selected. Note that the logical bus width can be different than the physical bus width since there are cases where an 8-bit peripheral may be used on a 16-bit bus. The selection of the logical width, and not the physical width, is what determines the B/W mode. Thus it is the responsibility of the programmer not to program an invalid combination (i.e., don't specify a 16-bit logical width on an 8-bit physical bus).

Any transfer on an odd boundary will be B/B but if the pointer is incrementing and on a 16-bit logical bus, after the first transfer, the pointer will be on an even boundary. The IOP will then try to maintain word transfers in order to transfer data as effeciently as possible. See the user's manual for details. The change in B/W mode occurs only after the first transfer or, as explained in the termination section, upon certain byte count terminations.

Synchronization

In the unsynchronzied mode, transfers occur as fast as priorities will allow. This is the IOP's "block-move" mode. Most I/O peripherals only want a DMA transfer on demand; the DRQ lines, along with synchronization specified, will handle this need. Source synchronization is used for I/O reads and destination synchronization is used for I/O writes.

If the IOP is waiting for a DMA request, it will run programs or DMA on the other channel, or execute idle cycles if nothing is pending. If running idle cycles when the DRQ comes, the transfer starts five clocks after DRQ is recognized. If running DMA or instructions on the other channel, the DRQ cannot be serviced until the current internal cycle is done, and may require a maximum of 12 clocks (without bus arbitration or wait states).

Consecutive DRQ-synchronized DMA transfers on the same channel are separated by four idle clocks (assuming no other delays) by an internal sampling mechanism. This happens between the 2-byte fetches on source-synchronized B/B-W cycles, and between the two stores on destination-synchronized W-B/B cycles. This delay between consecutive DMA cycles allows adequate time for proper acknowledgement of the current DMA request before the next request is processed. On destination-synchronized DMA, this isn't a problem, but on source-synchronized DMA, there will be four extra clocks per transfer. Unless one is running right at the speed limit, this won't be a problem. Near the maximum data rate, unsynchronized transfers can be used, with synchronization done by manipulating the READY line.

Translate Mode

When the translate bit is set, the data fetched during DMA will be added to the GC register. This new pointer will in turn be used to fetch, via a seven clock extra fetch cycle, new data, which will then be stored. Translate is only defined for byte transfers. The bytes are added to GC as a positive offset, so a lookup table for translating data can be a maximum of 256 bytes long. Even if the data to be translated falls within a smaller range (such as ASCII code), a full 256-byte lookup table is recommended so that erroneous data can be flagged and controlled.

Translate can be run on any of the B/B transfer modes, so it is useful for doing block translation within program execution as well as translation directly to or from an I/O port.

DMA Termination

One of the powerful features of the IOP is its varied DMA termination conditions and their close tie-in with resuming Instruction Block programs. However, because of the multitude of DMA modes, care must be taken in predicting the exact termination parameters. Various things to be careful about will be outlined here.

Byte Count (BC) Termination

The BC register is decremented for every byte transferred whether or not BC termination is set. If BC termination is set, the last transfer done is the one that results in BC being zero. To avoid the problem of missing BC = 0 on word transfers, if BC is odd between every transfer, the IOP detects when BC is 1, and forces the last transfer to be in the B/B mode. Since both the fetch and store cycles are complete, the source and destination pointers point exactly to the next byte or word that would have been fetched.

Masked Compare (MC) Termination

An MC termination occurs when a pattern matches (or doesn't match, depending on mode selected) the lower half of the MC register (the match pattern) with only the bits that are enabled by the upper half of MC (the mask pattern) contributing to a match. Thus the masked bits can be "don't cares" in both the data byte and the match byte.

The masked comparison is only done on store (deposit) cycles. Any bytes transferred (in B/B or W-B/B mode) will be compared. But, since the MC comparison is done on only one byte, any words stored (W-W or B-B/W) have only their lower byte compared. This may be fine, but if not, make the destination logical width 8 bits.

Just like BC termination, the pointers will point to the next data to be transferred. The BC will also be decremented correctly, except if the termination occurs on the first byte of a W-B/B transfer. In this case the BC will be decremented as if the entire transfer (both bytes) had taken place.

The store cycle that causes an MC termination will be lengthened by two extra clocks (or by one extra clock if there are wait states), to allow time to set up the termination cycle.

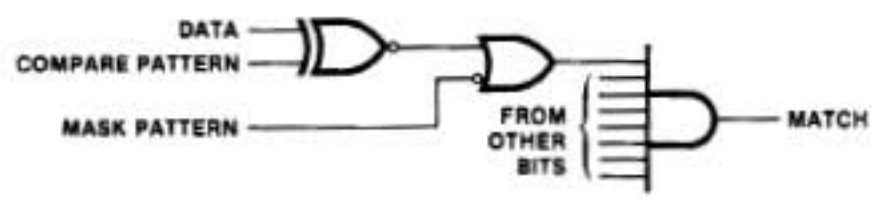


Figure 7. Masked Compare Logic for 1-Bit

External (EXT) Termination

External termination allows the I/O device or controller to use its own conditions to generate a termination. Basically, the IOP will halt DMA as soon as it recognizes an EXT terminate, even if a transfer is only partially complete. There might be concern that multibyte cycles (W-B/B or B/B-W) might have data lost if an EXT terminate stopped the store cycle. In unsynchronized DMA this would happen, but this mode is typically not used with I/O controllers that could generate external terminations. In synchronized DMA modes, it is assumed that the I/O controller will only do a DRQ for valid data transferred, and that it won't give an EXT terminate with its DRQ active. In destination synchronization, the possible problem occurs in the W-B/B mode, where EXT terminate comes after the first store but before the second. This is fine, since even though data was overfetched, the proper amount was actually transferred. In source synchronization, the B/B-W mode raises problems since if an EXT terminate came after the first byte fetched and before the second byte fetched, normally no store cycles would be done at all, thus losing the first byte fetched. In this case (i.e., source synced, DRQ inactive, and 1 byte already fetched), a single byte store cycle is run before the termination cycle, ensuring data integrity.

In order to prevent an invalid signal level from becoming trapped from the asynchronous EXT term lines, two clocks of delay and signal conditioning are done on these lines. In addition, a termination cycle can only be started at certain times during DMA (or TB on the other channel — see dual channel operation section). The EXT terminate lines should be valid eight clocks before the start of the DMA cycle to be stopped.

EXT is sampled even when the IOP is running something on the other channel. Remember though, that despite the high priority of termination, the current instruction on the other channel has to finish before the termination cycle is run. Simultaneous EXTs on both channels result in CH1 termination being done first.

In order to have enough time to process a byte count termination, the BC register is always decremented during DMA fetch cycles. Because of this, external or MC terminations that occur during W-B/B cycles will result in the byte count always being decremented by two, even if only one byte is stored. This also occurs in the block-to-block or block-to-port B/B-W modes. To find the exact number of bytes transferred, the source pointer address can be checked in the block-to-port and block-to-block modes during B/B-W cycles and in the block-to-port W-B/B mode. The destination pointer address can be used to find the number of bytes transferred in the port-to-block and block-to-block modes during W-B/B cycles.

Termination Cycles and Multiple Terminations

Upon termination, the user can run different task block programs, depending on which type of termination has occurred, by specifying an appropriate termination offset. That is, instruction fetching will begin after a termination cycle starting at either the TP value before the DMA started, TP + 4 or TP + 8. These offsets permit long or short jumps to termination routines.

The termination cycle is an add immediate instruction that runs from the internal ROM and adds the proper offset to the TP. It is 15 clocks long for TP + 4 and TP + 8 termination and 12 clocks long for TP + 0 termination.

As mentioned earlier, EXT terminate must come a certain time before the end of a transfer to ensure that the next transfer doesn't start. If it comes in time and MC termination also occurs on the current transfer, then the termination cycle with the largest offset is run. A simultaneous BC terminate cycle will have priority over MC and will result in the running the BC termination program.

Priorities/Dual Channel Operation

The IOP can share its internal and external hardware between two separate channels. The user sees two identical IOP channels with all registers, machine flags, etc., independent of the other channel. The only register in common is the CP register, loaded by the initialization sequence. The mechanism for achieving dual channel operation is time multiplexing between the two channels.

Since interleaving two channels affects their response time to external events and since interfacing to these events is the prime purpose of the IOP, several means of adjusting the priorities of the channels are provided.

Before going into the priority algorithms in detail the four types of cycles that are affected by the priorities will be outlined:

1. *DMA Cycles* — Any type of DMA transfer cycle, including single transfers and translate cycles. DMA can be interrupted after any bus transfer by the other channel.
2. *Instruction Cycles* — Any instructions that have been fetched out of I/O or system memory. Instruction cycles are made up of internal cycles, each two to eight clocks long (assuming no wait states). Some cycles may not run bus transfers. Instructions can be interrupted by DMA after any one of the internal cycles, but can only be interrupted by instructions on the other channel (normal ones or ones from internal ROM) after the current instruction is completed.
3. *Termination Cycle* — Performed when DMA transfers end and instructions resume (except on single transfers).
4. *Channel Attention Cycles* — Performed when channel attention is given, performs actions specified in the CCW field. Both termination and CA cycles can be interrupted by DMA after any internal cycle, but can only be interrupted by instruction cycles after the complete sequence of internal cycles is done.

Termination and channel attention cycles as well as the initialization cycle (which never runs concurrently with other operations) are sequences of instructions fetched from an internal ROM.

Recognizing the higher importance in doing DMA, termination and (to a lesser extent) CA cycles, the following priority scheme is built into the IOP. Any channel that has a higher-priority operation will run continuously until done. If both channels are running the same priority, execution will alternate between them.

Highest Priority

1. DMA transfers, termination, chained instructions
2. Channel attention cycles
3. Instruction cycles
4. Idle cycles

Lowest Priority

Two ways exist to alter the priority scheme. One way is to utilize the priority bits for each channel. If one is greater than the other, that channel will run at the expense of the other if both channels are otherwise running at the same priority. Thus the P bit only has effect on channels running at the same priority level.

If one wants to run instructions along with or in place of DMA on the other channel, the other technique is to set the chaining bit (in the CC register) which brings the instruction priority up to the level of DMA. Care should be taken with this since now CAs are at a lower priority than instructions and will not be serviced unless that channel goes idle. Chaining will also lock out normal instructions on the other channel. Chaining should thus be used with care.

In order to reduce the possibility of shutting out channel attentions, an exception is made to the above priority scheme. After every DMA transfer, whether synchronized or unsynchronized, the IOP will service any pending CA. However, chained task block execution will still shut out CAs on the other channel.

What is the importance of priorities? Well, as an example, let's say that we are running long periods of non-time-critical block moves (via DMA) on one channel and running short bursts of DMA that must be serviced promptly on the other channel. With the default priorities, the short DMA channel bursts would be interleaved with the longer DMA, reducing the maximum transfer rate for both channels. If, however, the priority bit was one on the burst mode DMA and zero on the other, the bursts would be serviced continuously at the fastest possible data rate.

An even more critical case would be the same low priority, long DMA transfers on one channel with DMA on the other channel that must terminate, run a short instruction sequence, and resume DMA again within a short, fixed time. (This might be the case in running a CRT display with linked list processing between lines.) Normally, the low priority, long DMA could indefinitely block the short TB sequence. By setting the high-priority channel's priority bit to one and putting it into the chained instruction mode, the low priority channel would stop its DMA entirely so that the termination/instruction sequence could run.

When establishing the priorities to be run, care should be taken that both channels will run successfully under a worst case combination. This can be tricky when the channels are running asynchronously with fast data rates and/or short latencies, but must be taken into account. Of course, running only one channel on the IOP is an easy solution, but if more than one IOP is being used in the system, the priorities and delays of the bus arbitration used (either $\overline{RQ}/\overline{GT}$ or an 8289 bus arbiter) must be taken into account. It may be found that the on-chip arbitration between the two channels is faster and more powerful than external arbitration.

SUMMARY

It is hoped that the material presented here will aid those who are putting together and debugging an 8089 IOP system, and help them in understanding the operation of the IOP. Many of the debugging techniques should be familiar to those who have worked with micro- and minicomputer systems before. Other debugging techniques not mentioned here, which work well with microprocessor systems, could be just as applicable to the 8089. The unique nature of the IOP among LSI devices warrants special consideration for its I/O functions and multiprocessor capabilities.

Appendix I

CHECKLIST OF POSSIBLE PROBLEMS

HARDWARE PROBLEMS

- Is RESET at least four clocks long?
- Are both V_{SS} lines connected to ground?
- Does the first CA falling edge come at least two clocks after RESET goes away?
- Does the second CA come at least 150 clocks (16-bit system, no wait states) after the first CA?
- Is READY correctly synchronized and gated by local/system bus lines?
- Is SEL correct for first CA so that IOP comes up correctly as master or slave?
- If two IOPs are local to each other, is a 2.7K pull-up resistor used on RQ/GT?

SOFTWARE PROBLEMS

- Are the initialization parameters in the initialization linked-list correct?
- Is BUSY flag being properly tested by host CPU software before modifying PB or providing a new command?
- Has the chaining, translate, or lock bit in the CC register been erroneously set?
- Have DMA termination conditions been met? The IOP could be trying to do endless DMA.

Appendix II

BREAKPOINT ROUTINE AND CONTROL PROGRAM

The debugging program described here is an example of the kind of software development tool that can be developed for the 8089 IOP. It was written to try out various breakpoint schemes, and has been used to debug an engineering application test system. The program is not meant to be the ultimate debugging tool, but is an example of what can be put together to utilize the breakpoint routine described earlier in the application note.

The debugging program was tested on a 8086-based system that emulates the SDK-86 I/O structure, and uses the SDK-86 serial monitor. This enables it to use the SDK-86 Serial Downloader to interface to an Intellec® development system on which the software was created. The 8086 system is interfaced via a MULTIBUS™ interface to an IOP running in the REMOTE mode. The remote bus access technique, mentioned earlier in this note, is implemented on this system, but was not used in the software debugging program.

The breakpoint routine uses a simple jump to a save routine. The PL/M-86 supervisory or control program handles the placement of the jump within the users program. Since it can not normally access the remote bus, all IOP programs to be tested must run out of system memory.

When the control program starts, it assumes the IOP has just been reset. It then prompts the user for the CP and PP values. After this, it sends the first (initialization) channel attention. It then asks the user for the channel to be run, and the starting and stopping addresses. After the stopping address has been entered, a Channel Attention Start is given. If the breakpoint is reached, a HALT is executed, and the control program prints the register contents. If the breakpoint hasn't been reached, the user can type any character, and a Channel Attention Halt will be sent to the IOP. If the IOP responds within 50 ms, the TP where it was halted is printed. Otherwise, the control program issues an error message. If, at any time, the user wants to get out of the program, typing an ESC will pass control back to the SDK-86 monitor. Figure 9 shows the flow of the control program.

Note that, unlike a single CPU debugging routine, having the 8086 supervise the 8089 enables a clean exit from crashed IOP programs. The program code where jumps had been placed are always restored. The control program is a good example of how the power of dual processors can be put to good advantage.

Comments within the control program indicate parameters that need to be changed to run on different systems. It should be noted that channel attentions are invoked by the recommended method of using an I/O write to a port to generate CA and using A0 for SEL.

Source and object files of this program are available through Intel's INSITE™ User's Program Library as program 8089 Break. 89 (number AD6).

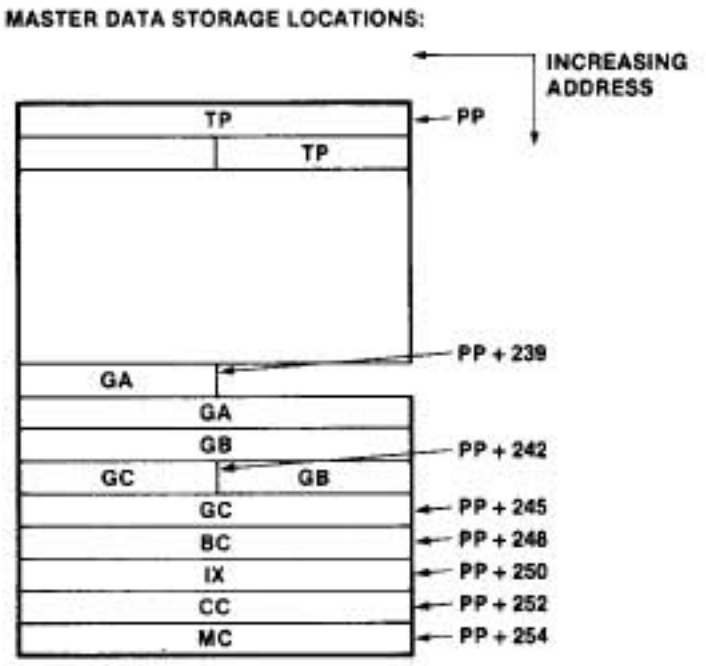


Figure 8. Breakpoint Routine to Run 8089 Program out of System Memory

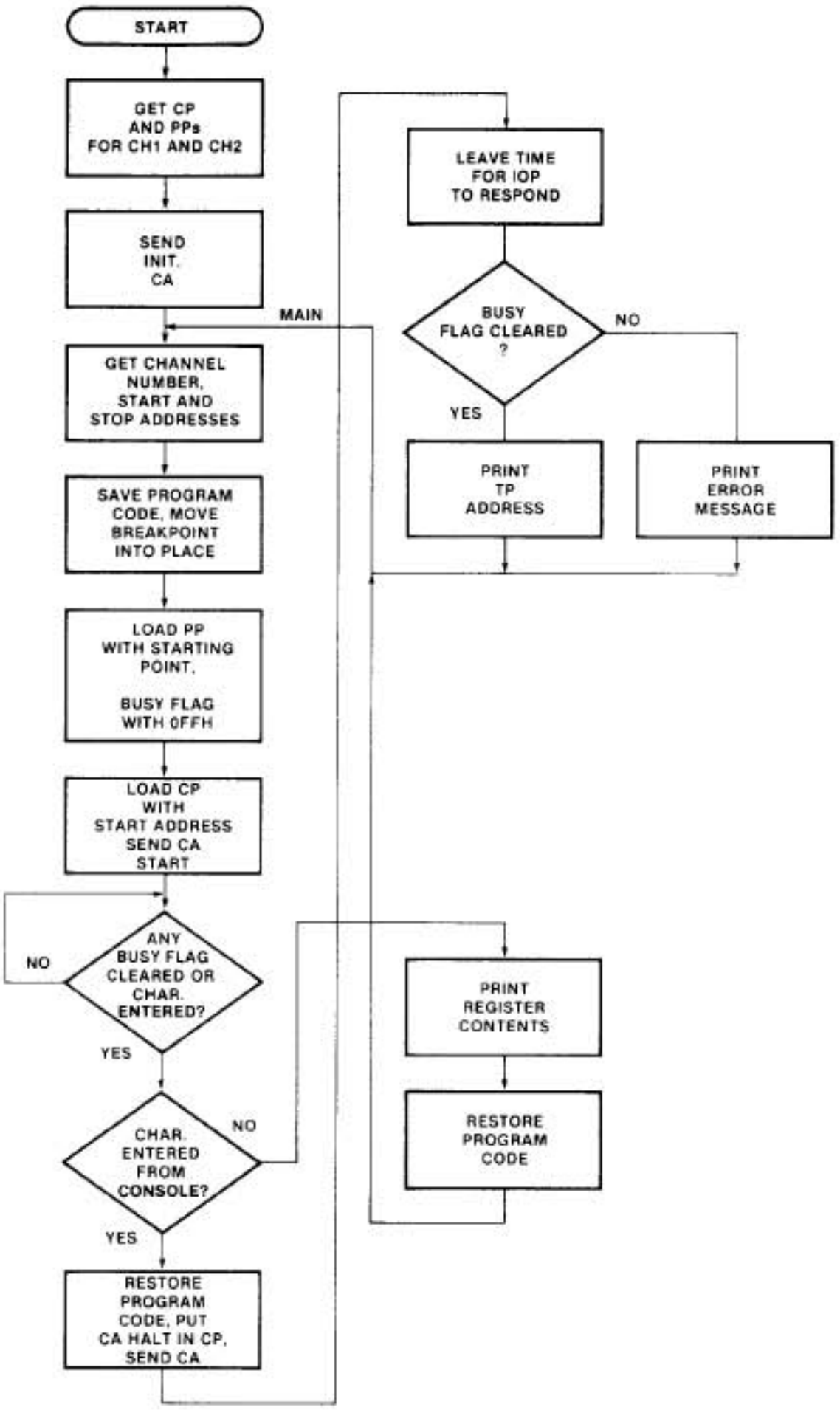


Figure 9. Breakpoint Routine to Run 8089 Program out of System Memory

```
PL/M-86 COMPILER     8089 BREAKPOINT ROUTINE                                    PAGE   1


ISIS-II PL/M-86 V3.0 COMPILATION OF MODULE BREAKPOINT
OBJECT MODULE PLACED IN BREAK.OBJ
COMPILER INVOKED BY:  F1:PLM86 BREAK.SRC PAGEWIDTH(100)


                $TITLE('8089 BREAKPOINT ROUTINE')
                /*.....................................................................

                    8089 BREAK POINT PROCEDURE
                    WRITTEN BY DAVE FERGUSON 2/2/79   REV 2  8/14/79
                    INTEL CORPORATION

                    .....................................................................*/

  1             BREAK$POINT:
                    DO;
  2   1             DECLARE I BYTE;
  3   1             DECLARE SAVECODE(4) WORD; /*BUFFER FOR STORAGE*/
  4   1             DECLARE ONEPP POINTER; /* CHAN ONE PP */
  5   1             DECLARE TWOPP POINTER; /* CHAN TWO PP */
  6   1             DECLARE STARTBYTES(4) BYTE; /* BUFFER FOR START ADDRESS */

  7   1             DECLARE STARTPOINTER POINTER; /* POINTER FOR START ADDR. */
  8   1             DECLARE ENDPOINTER POINTER; /* POINTER FOR END ADDR. */
  9   1             DECLARE PRESENT POINTER AT (@INPNTR); /* POINTER BUFFER */
 10   1             DECLARE TRUE LITERALLY 'OFFH', FALSE LITERALLY '000H';

                    /*  YOU MUST CONFIGURE YOUR I/O STRUCTURE AND
                       SYSTEM TO MATCH THE PROGRAM OR VISA VERSA */
 11   1             DECLARE CRTSTATUS LITERALLY 'OFFF2H', /* 8251 STATUS PORT */
                    CRTDATA LITERALLY 'OFFF0H', /* 8251 DATA PORTS */
                    CHANATTEN LITERALLY 'OFAH', /* CHANNEL ONE CHANNEL ATTENTION PORT */
                     /* CHANNEL TWO CHANNEL ATTENTION PORT = CHANATTEN + 1 */
                    CHANNELONE LITERALLY '00H',
                    CHANNELTWO LITERALLY '01H';

                    /* ASCII IS A STRING OF HEX CHARACHTERS IN ASCII FORM */
                    ASCII (*) BYTE DATA ('0123456789ABCDEF'),
                    TITLE$STRING (*) BYTE DATA (0AH,0DH,'8089 BREAKPOINT VER 1.0',
                                   0AH,0DH,'TYPE ESCAPE TO RETURN TO MONITOR.',
                                   0AH,0DH,0),
                    CHANGIVEN (*) BYTE DATA ('CHANNEL ATTENTION GIVEN TYPE ANY KEY TO ABORT.'
                                  ,0AH,0DH,0),
                    BKREACHED (*) BYTE DATA (0AH,0DH,'BREAKPOINT REACHED',0AH,0DH,0),
                    GETCP (*) BYTE DATA ('INPUT CP IN HEX',0AH,0DH,00),
                    GET$PP (*) BYTE DATA ('INPUT PP IN HEX FOR ',00H),
                    GETSTART (*) BYTE DATA (0AH,0DH,'INPUT STARTING ADDRESS IN HEX',0AH,0DH,00H),
                    STOPADDR (*) BYTE DATA ('INPUT END ADDRESS IN HEX',0AH,0DH,00H),
                    CHANNUMBER (*) BYTE DATA (0AH,0DH,'CHANNEL ONE OR TWO? ',00H),
                    ABORT (*) BYTE DATA (' FATAL ERROR - IOP DOES NOT RESPOND TO CHANNEL',
                    ' ATTENTION.  RE-INITIALIZE SYSTEM ',0),
                    ABORTAT (*) BYTE DATA (' TP WAS ',0),
                    ONE (*) BYTE DATA (' CHANNEL ONE',0AH,0DH,00H),
                    TWO (*) BYTE DATA (' CHANNEL TWO',0AH,0DH,00H),
                    GASTRING (*) BYTE DATA  ('GA = ',00H),
```

```
                    QBSTRING (*) BYTE DATA ('QB = ',00H),
                    QCSTRING (*) BYTE DATA ('QC = ',0H),
                    BCSTRING (*) BYTE DATA  (0AH,0DH,'BC = ',00H),
                    IXSTRING (*) BYTE DATA (0AH,0DH,'IX = ',00H),
                    CCSTRING (*) BYTE DATA (0AH,0DH,'CC = ',00H),
                    MCSTRING (*) BYTE DATA (0AH,0DH,'MC = ',00H)

 12   1             DECLARE CHAR BYTE;
 13   1             DECLARE ONETWO BYTE;

                    /* SDKMON IS A PLM TECHNIQUE USED TO FORCE THE CPU INTO AN
                       INTERUPT LEVEL 3.  IN ORDER TO USE THIS THE PROGRAM MUST
                       BE COMPILED (LARGE). */
 14   1             SDKMON:
                    PROCEDURE;
 15   2              DECLARE HERE (*) BYTE DATA (0CCH),
                    /* THIS IS AN INT.  3  */
                       WHERE WORD  DATA(.HERE);
 16   2                CALL WHERE;
 17   2             END;


                    /* CO SENDS A CHAR TO THE CONSOLE WHEN READY */
                    /* THIS ROUTINE IS WRITTEN TO RUN VIA THE SERIAL
                       PORT OF AN SDK86 */
 18   1             CO:
                    PROCEDURE (C);
 19   2              DECLARE C BYTE;
 20   2              DO WHILE (INPUT(CRTSTATUS) AND 01H) = 0; END;
 22   2              OUTPUT (CRTDATA) = C;
 23   2             END;

                    /* CI GETS A CHARACHTER FROM THE USER VIA THE SERIAL PORT */
                    /* CI AUTOMATICALLY ECHOS THE CHARACHTER TO THE USER CONSOLE */
 24   1             DECLARE ESCAPE LITERALLY '1BH';

 25   1             CI: PROCEDURE BYTE;
 26   2               DO WHILE (INPUT(CRTSTATUS) AND 02H) = 0; END;
 28   2               CHAR = INPUT (CRTDATA) AND 07FH;
 29   2               CALL CO(CHAR);
 30   2               IF CHAR = ESCAPE THEN CALL SDKMON; /* GO TO SDK MONITOR */
 32   2               RETURN CHAR;
 33   2             END;

                    /* VALIDHEX CHECKS THE VALIDITY OF A BYTE AS A HEX CHARACHTER*/
                    /* THE PROCEDURE RETURNS TRUE IF VALID FALSE IF NOT */

 34   1             VALIDHEX:
                    PROCEDURE (H) BYTE;
 35   2              DECLARE H BYTE;
 36   2              DO I=0 TO LAST(ASCII);
 37   3               IF H=ASCII(I) THEN RETURN TRUE;
 39   3              END;
 40   2              RETURN FALSE;
 41   2             END;
```

```
                    /* HEXCONV CONVERTS A HEX CHARACTER TO BINARY FOR MACHINE USE.
                       IF THE CHARACTER IS NOT A VALID HEX CHAR, THE PROCEDURE RETURNS
                       THE VALUE OFFH */
 42   1             HEXCONV:
                    PROCEDURE (DAT)  BYTE;
 43   2               DECLARE DAT BYTE;
 44   2               IF VALIDHEX(DAT) <> OFFH THEN RETURN TRUE;
 46   2               DO I=0 TO LAST(ASCII);
 47   3                 IF DAT = ASCII(I) THEN RETURN I;
 49   3               END;
 50   2               END;

                    /* HEXOUT WILL CONVERT A VALUE OF TYPE BYTE TO AN ASCII STRING
                       AND SEND IT TO THE CONSOLE */

 51   1             HEXOUT:
                    PROCEDURE(C);
 52   2               DECLARE C BYTE;
 53   2               CALL CO(ASCII(SHR(C,4) AND OFH));
 54   2               CALL CO(ASCII(C AND OFH));
 55   2             END;


                    /* WORDOUT CONVERTS A VALUE OF TYPE WORD TO AN ASCII STRING
                       AND SENDS IT TO THE CONSOLE */
 56   1             WORDOUT:
                    PROCEDURE (W);
 57   2               DECLARE W WORD;
 58   2               CALL HEXOUT(HIGH(W));
 59   2               CALL HEXOUT(LOW(W));
 60   2             END;

                    /* GETADDRESS IS A PROCEDURE TO GET AN ADDRESS FROM THE CONSOLE.
                       THIS PROCEDURE WILL ONLY CONSIDER THE LAST 5 CHARACHTERS ENTERED
                       */
 61   1             DECLARE INPNTR (4) BYTE;

 62   1             GET$ADDRESS:
                    PROCEDURE POINTER;
 63   2               DECLARE BUFF BYTE;
                    /*CLEAR ALL VALUES TO ZERO */
 64   2               INPNTR(0) = 0;
 65   2               INPNTR(1) = 0;
 66   2               INPNTR(2) = 0;
 67   2               INPNTR(3) = 0;

 68   2               BUFF = 0;
 69   2               DO WHILE BUFF <> TRUE;
                    /* THIS SEQUENCE OF SHIFTS ALLOW THE USER TO TYPE IN FIVE
                       OR MORE CHARACHTERS TO BECOME THE ACTUAL POINTER FOR 8089
                       OR 8086.  THIS PROCEDURE RETURNS THE LAST FIVE IN PROPER
                       SEQUENCE  STORED IN INPNTR(0-3).  THE STORAGE
                       IS AS FOLLOWS:
                            1. THE LAST CHARACTER INPUT GOES INTO
                               THE LOW FOUR BITS OF INPNTR(0).
                            2. THE NEXT TO LAST CHARACTER GOES INTO
                               THE LOW FOUR BITS OF INPNTR(2).
```

```
                        3. THE THIRD CHARACTER INPUT GOES INTO
                           THE HIGH FOUR BITS OF INPNTR(2)
                        4. THE SECOND CHARACHTER INPUT GOES INTO
                           THE LOW FOUR BITS OF INPNTR(3)
                        5. THE FIRST CHARACTER INPUT GOES INTO
                           THE UPPER FOUR BITS OF INPNTR(3).
                      THE 86 SHIFTS INPNTR (2.AND3) LEFT FOUR BITS AND ADDS THIS TO
                      INPNTR(0) RESULTING IN THE ADDRESS THE USER TYPED IN.  */

 70   3               INPNTR(3) = (SHL(INPNTR(3),4) OR (SHR( INPNTR(2),4) AND OFH));
 71   3               INPNTR(2) = (SHL(INPNTR(2),4) OR (INPNTR(0) AND OFH));
 72   3               INPNTR(0) = BUFF;
 73   3               BUFF = CI;
 74   3               BUFF = HEXCONV(BUFF);
 75   3             END;
 76   2             CALL CO(0AH); /*LINE FEED TO CRT*/
 77   2             CALL CO(0DH); /*CARRIAGE RET TO CRT*/
 78   2             RETURN PRESENT; /* PRESENT IS A POINTER TO THE ARRAY INPNTR.  */
 79   2           END;

                  /* STRINGOUT IS A PROCEDURE TO SEND THE CONSOLE AN ASCII STRING
                     ENDING IN THE VALUE 00.  STRINGOUT NEEDS A VALUE OF TYPE POINTER
                  */

 80   1           STRING$OUT:
                  PROCEDURE(PTR);
 81   2             DECLARE PTR POINTER,STR BASED PTR (1) BYTE;
 82   2             I = 0;
 83   2             DO WHILE STR(I) <> 0;
 84   3              CALL CO(STR(I));
 85   3              I = I + 1;
 86   3             END;
 87   2           END;


 88   1           DECLARE TAGIS (*) BYTE DATA (' OPERATING IN ',0),
                          TAGISONE (*) BYTE DATA ('IO SPACE',0AH,0DH,0),
                          TAGISZERO (*) BYTE DATA ('SYSTEM SPACE',0AH,0DH,0);
                  /* TAGTEST TESTS THE TAG BIT AND SENDS A MESSAGE TO THE CONSOLE
                     THE TAG IS LOCATED IN BIT THREE.  A TAG BIT OF ONE MEANS THE
                     POINTER IS TO I/O SPACE, AND A TAG BIT OF ZERO MEANS THE
                     POINTER IS TO SYSTEM SPACE */
                  /* THE CALLER MUST DECIDE WHICH BYTE HAS THE TAG AND PASS IT TO TAGTEST */

 89   1           TAGTEST:
                  PROCEDURE(TEST);
 90   2             DECLARE TEST BYTE;
 91   2             CALL STRINGOUT(@TAGIS);
 92   2             IF (TEST AND 01000B) <> 0
                    THEN
 93   2             DO;
 94   3               CALL STRINGOUT(@TAGISONE);
 95   3             END;
                    ELSE
 96   2             DO;
 97   3               CALL STRINGOUT(@TAGISZERO);
 98   3             END;
```

```
 99   2          END;
100   1          DECLARE SAVE$ADDR LITERALLY '2000H',
                     SAVE$SEG LITERALLY '00C0H';

101   1          DECLARE BREAK89 (4) WORD DATA (9B81H,0891H,SAVE$ADDR,SAVE$SEG);
                 /* BREAK89 IS AN 4 WORD ESCAPE SEQUENCE TO ADDRESS 2000H
                    CONSISTING OF AN  LPDI TP,SAVE$ADDR WITH SEGMENT
                        LOCATED AT 0C00H.    */

                 /* BRKRTN IS 33 BYTES OF CODE THAT STORES ALL REGISTERS
                    AS FOLLOWS:
                     GA STORED     AT PP + 239
                     GB STORED     AT PP + 242
                     GC STORED     AT PP + 245
                     BC STORED     AT PP + 248
                     IX STORED     AT PP + 250
                     CC STORED     AT PP + 252
                     MC STORED     AT PP + 254
                       */

102   1          DECLARE BRKRTN (33) BYTE AT (02C00H)
                 /* 02C00H IS ACTUALLY (SAVE$ADDR + (SHL(SAVE$SEG),4)), AND SHOULD
                    MATCH ADDRESS AND SEGMENT WHERE BREAK ROUTINE IS WANTED    */
                  INITIAL
               (03H,09BH,0EFH,023H,09BH,0F2H,043H,09BH,0F5H,063H,087H,0F8H,0A3H,087H,
               0FAH,0C3H,087H,0FCH,0E3H,087H,0FEH,020H,048H) ;
103   1          DECLARE PP POINTER;
104   1          DECLARE PPP BASED PP (1) BYTE;

105   1          START$PRGM
                   PROCEDURE(ONE$TWO,PPP),
106   2          DECLARE ONE$TWO BYTE,PPP POINTER,
                   WHERE BASED PPP (1) BYTE;

107   2              WHERE(0) = START$BYTES(0);
108   2          WHERE(1) = 0;
109   2          WHERE(2) = START$BYTES(2);
110   2          WHERE(3) = START$BYTES(3);
111   2          CPDAT((ONE$TWO) * 8) = 3,
                 /* IF ONETWO = 1 THEN OUTPUT TO PORT 0FBH, IF ONETWO
                    IS 0 THEN OUTPUT TO PORT 0FAH */
112   2          OUTPUT(CHANATTEN + (ONETWO )) = 0,
113   2          CALL STRINGOUT(@CHANGIVEN),
114   2        END;

                 /* THIS PART OF THE PROGRAM ALLOWS THE USER TO DEFINE THE
                   CP,PP OF EACH CHANNEL */
115   1        DECLARE BREAKOUT BASED ENDPOINTER (1) WORD;

116   1        DECLARE CP POINTER,
117   1        DECLARE CPDAT BASED CP (1) BYTE;

118   1        DECLARE ONEPPDAT BASED ONEPP (1) BYTE;
119   1        DECLARE TWOPPDAT BASED TWOPP (1) BYTE;


120   1        CALL STRINGOUT (@TITLESTRING);
```

```
121   1          CALL STRINGOUT(@GETCP);
122   1          CP = GETADDRESS;
123   1          CALL STRINGOUT(@GETPP);
124   1          CALL STRINGOUT(@ONE);
125   1          ONEPP = GETADDRESS;
126   1          CALL STRINGOUT(@GETPP);
127   1          CALL STRINGOUT(@TWO);
128   1          TWOPP = GETADDRESS;
129   1          OUTPUT (CHANATTEN) = 0; /* INITIALIZATION CA */

130   1      MAIN:
              CALL STRINGOUT(@CHANNUMBER);
131   1          CHAR = CI; /* GET CHANNEL NUMBER */
132   1          IF (CHAR AND 01H) <> 0 /* CHECK BIT ZERO TO DEFINE
                                CHANNEL NUMBER */
              THEN DO;
134   2            CALL STRINGOUT(@ONE);
135   2            ONETWO = CHANNEL$ONE;
136   2          END;
              ELSE
137   1           DO;
138   2             CALL STRINGOUT(@TWO);
139   2             ONETWO = CHANNEL$TWO;
140   2          END;

141   1          CALL STRINGOUT(@GET$START); /* GET STARTING ADDRESS
                              FROM USER */

142   1          STARTPOINTER = GETADDRESS;
143   1          DO I = 0 TO 3; /* MOVE STARTING ADDRESS INTO CP AREA */
144   2            STARTBYTES(I) = INPNTR(I);
145   2          END;
146   1          CALL STRINGOUT(@STOPADDR); /* GET STOP ADDRESS
                              FROM USER */

147   1          ENDPOINTER = GETADDRESS;
148   1          DO I = 0 TO 3; /* MOVE CODE TO SAFE AREA  */
149   2            SAVECODE(I) = BREAKOUT(I);
150   2          END;
151   1          DO I = 0 TO 3;
152   2            BREAKOUT(I) = BREAK89(I); /* MOVE ESCAPE SEQUENCE INTO PLACE */
153   2          END;
154   1          CPDAT(1) = OFFH; /* SET  CHANNEL ONE BUSY FLAG */
155   1          CPDAT(9) = OFFH; /* SET CHANNEL TWO BUSY FLAG */
156   1          DO CASE ONETWO;
157   2          PP = ONEPP;
158   2          PP = TWOPP;
159   2          END;
160   1          CALL START$PRGM(ONE$TWO,PP);
              /* WAIT FOR ONE OF THE FOLLOWING
                 1.CPDAT(1) = 0  CH1 NOT BUSY
                 2.CPDAT(9) = 0 CH2 NOT BUSY
                 3.THE 8251 REC.  BUFFER IS FULL BECAUSE USER HAS DEPRESSED A KEY
              */
161   1          DO WHILE (  (CPDAT(1) AND CPDAT(9)) AND (NOT (INPUT(CRT$STATUS) AND 02H))) = OFFH;
```

```
162   2           END;
163   1         IF (INPUT(CRT$STATUS) AND 02H) <> 0
                  THEN
164   1           DO;
165   2             CHAR = CI;
166   2             DO I = 0 TO 3;
167   3               BREAKOUT(I) = SAVECODE(I);
168   3             END;
                  /* IF ONETWO = 0 THEN PUT CHA HLT IN CPDAT(0)
                     IF ONETWO = 1 THEN PUT CHA HLT IN CPDAT(8)
                  */
169   2             CPDAT(ONE$TWO *8) = 06H;
                  /* IF ONETWO = 0 THEN OUTPUT TO PORT 0FAH, IF ONETWO
                     IS 1 THEN OUTPUT TO PORT 0FBH.
                  */
170   2             OUTPUT(CHANATTEN + ONETWO) = 0;
171   2             DO I = 0 TO 5;
172   3              CALL TIME(100);
173   3             END;

                  /* IF BUSY FLAG HAS BEEN CLEARED, THEN A CA HALT&SAVE
                     WAS EXECUTED.  IF SO, PRINT SAVED TP; IF NOT, ABORT  */

174   2             IF CPDAT(SHL(ONETWO,3) + 1) <> 0   /* CHECK BUSY FLAG */
                    THEN
175   2             DO;
176   3              CALL STRINGOUT(@ABORT);
177   3             END;
                    ELSE
178   2             DO;
179   3              CALL STRINGOUT(@ABORTAT);
180   3              CALL CO(ASCII(SHR(PPP(2),4)));  /* UPPER NIBBLE OF ADDR
                                STORED BY HALT */

181   3              CALL HEXOUT(PPP(1));  /* MIDDLE BYTE OF ADDR
                            STORED BY HALT */

182   3              CALL HEXOUT(PPP(0));  /* LEAST SIG BYTE OF ADDR
                            STORED BY HALT */

183   3             END;
184   2             CPDAT(ONETWO * 8) = 3H;  /* CA START IN CPDAT(0) OR CPDAT(8) */
185   2             GO TO MAIN;
186   2           END;
187   1         DO;

188   2           CALL STRINGOUT(@BKREACHED);

189   2           CALL STRINGOUT(@GASTRING);
190   2           CALL CO(ASCII(SHR(PPP(241),4)));
191   2           CALL HEXOUT(PPP(240));
192   2           CALL HEXOUT(PPP(239));
193   2           CALL TAGTEST(PPP(241));

194   2           CALL STRINGOUT(@GBSTRING);
195   2           CALL CO(ASCII(SHR(PPP(244),4)));
196   2           CALL HEXOUT(PPP(243));
```

```
PL/M-86 COMPILER     8089 BREAKPOINT ROUTINE                                PAGE   8


 197   2            CALL HEXOUT(PPP(242));
 198   2            CALL TAGTEST(PPP(244));

 199   2            CALL STRINGOUT(@GCSTRING);
 200   2            CALL CO(ASCII(SHR(PPP(247),4)));
 201   2            CALL HEXOUT(PPP(246));
 202   2            CALL HEXOUT(PPP(245));
 203   2            CALL TAGTEST(PPP(247));

 204   2            CALL STRINGOUT(@BCSTRING);
 205   2            CALL HEXOUT(PPP(249));
 206   2            CALL HEXOUT(PPP(248));

 207   2            CALL STRINGOUT(@IXSTRING);
 208   2            CALL HEXOUT(PPP(251));
 209   2            CALL HEXOUT(PPP(250));

 210   2            CALL STRINGOUT(@CCSTRING);
 211   2            CALL HEXOUT(PPP(253));
 212   2            CALL HEXOUT(PPP(252));

 213   2            CALL STRINGOUT(@MCSTRING);
 214   2            CALL HEXOUT(PPP(255));
 215   2             CALL HEXOUT(PPP(254));

 216   2            END;
                     /* RESTORE CODE TO ORIGINAL LOCATION  */
 217   1            DO I = 0 TO 3;
 218   2              BREAKOUT(I) = SAVECODE(I);
 219   2            END;

 220   1            GO TO MAIN;

 221   1       END.


MODULE INFORMATION

     CODE AREA SIZE.     = 0619H   1561D
     CONSTANT AREA SIZE = 01EFH    495D
     VARIABLE AREA SIZE = 0020H     32D
     MAXIMUM STACK SIZE = 0014H     20D
     427 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

```
8089 ASSEMBLER


ISIS-II 8089 ASSEMBLER X004 ASSEMBLY OF MODULE AP50_BREAKPOINT_ROUTINE
OBJECT MODULE PLACED IN :F0:BRKASM.OBJ
ASSEMBLER INVOKED BY ASM89.4 BRKASM.SRC


                                        1 NAME    AP50_BREAKPOINT_ROUTINE
0000                                    2 BRKPNT  SEGMENT
                                        3 ;***************************************
                                        4 ;  BASIC 8089 BREAKPOINT ROUTINE
                                        5 ;    BY JOHN ATWOOD   REV 3  8/13/79
                                        6 ;    INTEL CORPORATION
                                        7 ;***************************************
                                        8
                                        9 ;  THE FOLLOWING CODE IS CONTAINED IN THE PL/M-86
                                       10 ;  CONTROL PROGRAM(BREAK.89) AND IS ASSEMBLED HERE
                                       11 ;  TO ILLUSTRATE HOW THE ESCAPE SEQUENCE AND SAVE
                                       12 ;  ROUTINE CODE WAS GENERATED.  TO USE THE 8089 BREAK-
                                       13 ;  POINT PROGRAM, THIS ASM89 PROGRAM WOULD NOT BE
                                       14 ;  NEEDED.  SAVE_ADDR IS THE SAME AS SAVE$ADDR IN THE
                                       15 ;  BREAK.89 PROGRAM.
                                       16
  2000                                 17 SAVE_ADDR       EQU     2000H   ;SAVE ROUTINE ADDRESS
                                       18
0000   9108 00200000                   19         LPDI TP,SAVE_ADDR       ;JUMP TO SAVE ROUTINE
                                       20
                                       21 ;***************************************
                                       22
                                       23 ;  REGISTER SAVE LOCATIONS WITHIN PB:
                                       24
                                       25 REGS    STRUC
0000                                   26 PBLOCK: DS      239     ;PARAMETER BLOCK
00EF                                   27 GASAV:  DS      3       ;GA AREA
00F2                                   28 GBSAV:  DS      3       ;GB AREA
00F5                                   29 GCSAV:  DS      3       ;GC AREA
00F8                                   30 BCSAV:  DS      2       ;BC AREA
00FA                                   31 IXSAV:  DS      2       ;IX AREA
00FC                                   32 CCSAV:  DS      2       ;CC AREA
00FE                                   33 MCSAV:  DS      2       ;MC AREA
0100                                   34 REGS    ENDS
                                       35
                                       36 ;  REGISTER SAVE ROUTINE:
                                       37
                                       38 ORG     SAVE_ADDR
                                       39
2000   039B EF                         40         MOVP [PP].GASAV,GA      ;SAVE GA
2003   239B F2                         41         MOVP [PP].GBSAV,GB      ;SAVE GB
2006   439B F5                         42         MOVP [PP].GCSAV,GC      ;SAVE GC
2009   6387 F8                         43         MOV [PP].BCSAV,BC       ;SAVE BC
200C   A387 FA                         44         MOV [PP].IXSAV,IX       ;SAVE IX
200F   C387 FC                         45         MOV [PP].CCSAV,CC       ;SAVE CC
2012   E387 FE                         46         MOV [PP].MCSAV,MC       ;SAVE MC
                                       47
2015   2048                            48         HLT                     ;STOP THIS CHANNEL,
                                       49                                 ;CLEAR BUSY FLAG.
                                       50 ;***************************************
2017                                   51 BRKPNT  ENDS
                                       52
                                       53 END
```