Multiprogramming with the iAPX 88 and iAPX 86 Microsystems

George Alexy
Microcomputer Applications
Intel Corporation makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR 7-104.9 (a) (9). Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and may only be used to identify Intel products:

- BXP
- CREDIT
- i
- ICE
- ICS
- i
- Insite
- Intel
- Intelec
- iSBC
- iSBX
- Library Manager
- MCS
- Megachassis
- Micromap
- MULTIBUS
- MULTIMODULE
- PROMPT
- Promware
- RMX
- UPI
- μScope
- iSBX
- Promware
- μScope

and the combinations of ICE, iCS, iSBC, MCS or RMX and a numerical suffix.

MDS is an ordering code only and is not used as a product name or trademark. MDS® is a registered trademark of Mohawk Data Sciences Corporation.

*MULTIBUS is a patented Intel bus.

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051
Multiprogramming with the iAPX88 and iAPX86 Microsystems

Contents

Introduction ........................................ 1
What is Multiprogramming? ...................... 1
General System Requirements .................. 2
iAPX 86 and iAPX 88 Architectural Features .... 3
Segmentation ....................................... 3
Segment Registers and Usage ................... 4
Semaphores ....................................... 7
The 9 Chip Multiprogramming System .......... 8
System Overview ................................ 9
Hardware Description ............................ 9
Software Description ............................ 10
Adding More Users .............................. 20
Variant Considerations ......................... 20
Conclusions ..................................... 20

Appendix A
Considerations for Programs with Multiple Code and Data Segments ......... 21

Appendix B
Multiuser Tiny Basic Operating System
Program Listing ................................ 25
INTRODUCTION

An engineer, faced with the assignment of developing a multiprogrammed microprocessor based system, need no longer be concerned about the enormity of the task. The new technology and architecture available in today's microprocessors are directed at supporting the basic requirements of these systems. In particular, Intel's iAPX 86 and iAPX 88 microprocessors and support chips will handle this task very efficiently. The purpose of this Application Note is to provide a description of various system requirements for multi-programming and to show how to use the capabilities of the iAPX 86 and iAPX 88 to meet those requirements. As a demonstration of the applicability of the iAPX 86 and iAPX 88 architecture, a multiuser system with Tiny BASIC was created around the iAPX 88. More about this system will be discussed later. For additional information on the iAPX 86 and iAPX 88, the reader is referred to the 8086 Family Users Manual.

WHAT IS MULTIPROGRAMMING?

In any software group, around any desk, you can hear the "buzz" words: multiprogramming, multitasking, multiprocessing, multi-, multi-, multi. A single phrase which will cover all of those buzz words is concurrent processing, which is simply the ability of a system to process more than one function at a time. Multiprogramming, then, is a form of concurrent processing featuring the ability of the system to allow more than one user to access the system's resources at the same apparent time. This does not mean that more than one program is being executed simultaneously; that would require more than one processor and would be multiprocessing. Multiprogramming implies that a processor's time and resources are being divided in such a way that more than one program is executing in the system. If the system is executing much faster than the real time requirements of any program, it appears to an outside observer that all programs are executing simultaneously. What is actually occurring in the system is that the microprocessor, after an initial start-up sequence, will execute one of the programs and after some type of interrupt or polling routine, will process another. (Figure 1). Note this differs from a batch processing environment in which each program runs to completion before the next program starts execution. The interrupt or polling routine will save the state of the machine for the current program and then determine who is going to get service next. It will then restore the previously saved state of the next program to receive service and begin processing this program. After this user has received service for some period of time, the sequence is repeated with a different user. During each context switch, pointers and other required information relevant to the current user are saved and the system proceeds to identify the next user. If there is only one user on the system, that program may get virtually all of the processor's time and resources.

For our design example, a terminal based environment, the input and output are being performed by the operator at human speed, which is extremely slow relative to the speed of the microprocessor. Most of the processor's time in a single-user system is spent waiting for the operator to enter the required information, or for an output device to display the information being sent by the processor. The ratio of effective computer time usage to computer wait time can be very small. Multiprogramming takes advantage of this relatively large amount of wait time by using it to execute a request from one of the other concurrent users. Of course, as the number of users on the system increases, the response time (i.e., the amount of time it takes for the computer to respond to a specific request from an operator) will become longer and longer until it reaches some "unacceptable" limit. In order to maximize the number of users which may acceptably use the system concurrently, the operating system may be "tailored" to a particular type of application.

The operating system is the "master program" which keeps track of what the system is doing and what it needs to do next. It will handle all of the input and output functions such as disk read and write routines, terminal input and output, etc. The operating system also controls the use of system resources, (i.e. allocation of memory to a user) as well as housekeeping associated with switching from one user to another. For example, as a user requests access to a specific program, the requested program and/or data is loaded into the user's work area. The operating system not only loads the requested program, but also may monitor the program's use as dictated by the operator.

Imagine a small accounting system which provides limited service to two or three users. In addition, it has a programmer who maintains the existing programs as well as writing new ones. At any given time during the day, there could be up to four users on the system at the same time, each doing a different task. It is the responsibility of the operating system to ensure that each user's programs and data get loaded and that each user gets the needed service without interfering with the needs of the other users. The intent of this note is to discuss the capabilities of the iAPX 86 and iAPX 88 which support this type of concurrent processing.
GENERAL SYSTEM REQUIREMENTS

For a system to provide programming services to more than one user (program) at a time, it needs some method of partitioning the system resources among the users and some method of controlling access to those resources. This requires, from a simplistic view, an operating system which will monitor the user's requests for resources, control allocation of resources and switch the machine state from one user to another.

The primary resources to be shared in the system are the CPU and system memory. To maximize the efficiency and throughput of the system, it is desirable to allow multiple programs and associated data to reside in memory concurrently and switch from one user to another by minimal reloading of the system context. Optimally then, the system will support a simple mechanism for not only allocating areas of memory to each program but also guaranteeing the programs will not violate each others address spaces. This requires that the mechanism for specifying the currently active (accessible) areas of memory be decoupled (separated) from the general CPU based resources available to the programmer. Strongly associated with this is the requirement for the system to support position independent programs, i.e. programs which will operate correctly regardless of where they are loaded into memory. If the programs are not position independent, then specific programs must reside in specific areas of memory forcing the system to either save and reload memory during task switches or not allow a user to run until the appropriate areas of memory are available.

Another desirable attribute of the system is the ability to support reentrant programs. This capability allows a single copy of a program to be shared by more than one user. This reduces the amount of memory required to support each user by eliminating the need for separate copies for each user or preventing one user from waiting until the program is available for his use. This concept is particularly applicable to system library routines available to all users in the system.
The system also needs to support a mechanism for allocating system resources which cannot be shared (such as peripherals or non-reentrant programs) among the various users. For example, if the system has more users on the system than printers available to print reports, the system needs some method of allocating the printers as the users request their service assuming that each printer is allocated to no more than one user at a time.

In order to share the system, we need a method of determining when it is time to start processing another user's program. This is usually provided by an external interrupt which is input through the hardware interrupt structure of the microprocessor. The external event technique prevents one of the users from monopolizing all of the processor's time, keeping the other users from getting service. An alternate method would be to allow each user to execute until service from the operating system (usually for I/O) is required.

When the system is interrupted, it needs some method of determining who gets to use the system next thus allowing the amount of time the processor has available to execute programs to be allocated among the various users on the system.

APX 86 AND iAPX 88
ARCHITECTURAL FEATURES

Intel's iAPX 86 and iAPX 88 directly support many of the features required for multiprogrammed applications. The architectural capabilities of the family stem from the register structure, large memory address space, almost unlimited interrupts, a powerful segmentation scheme and addressing structures which support reentrant and relocatable programs. The segmentation scheme allows the processor to provide capabilities such as program sharing, dynamically relocatable code and functionally partitioning memory among multiple users with a degree of protection implemented directly by the CPU.

Segmentation

Segmentation is the partitioning of a program and its data into specific elements called segments. Basically, segments are assigned to logical (and often variable length) elements (i.e. code or data) and should not be confused with the term "page" which is typically associated with a fixed length area of memory. Using segmentation, a programmer may assign modules of his program to the segments, and his data structures to additional segments (Figure 2). All references to a specific logical element (either program or data) are made relative to the appropriate segment. Assuming that the user's program does not specifically modify the segment registers, the operating system may place each segment anywhere in available memory and the program will still function properly. This concept of position independence becomes very important in multiuser systems where the number of users and the number of different programs in the system varies over time. As a user begins to initiate tasks in the system, the operating system will allocate specific areas in memory for the required segments. As more and more users enter the system they also are given system resources with which to execute their programs. Using segmentation, it doesn't matter what areas in memory the segments are assigned because the programs reference the data and programs relative to the associated segments. If the programmer breaks the data and programs into different segments, they may be located separately anywhere in memory. This method provides efficient utilization of the system memory resources since memory is only allocated for the specific segment size. The concept of segmentation also provides a degree of isolation between users in the system if all program and data references are restricted to their own segments. The ability to separate code and data into separate segments supports the concept of shared programs by allowing each invocation of a single program to reference only data and temporary variables relative to the data and stack segments of the current active user. Segmentation is a powerful concept which, up to this time, has been available only in larger computers such as minis and mainframes. Note that in other architectures,

![Figure 2. Segmented Address Space Partitioning](image)
the notion of position independent data is supported through the use of based addressing where a base register resource is used to identify the location of data and all references are made relative to the base register. The primary drawback with this approach is its restriction on the flexibility with which based addressing can be used, particularly when dealing with data structures and arrays. The concept of segmentation allows defining the position (within memory) of the structure without consuming the base register resource (Figure 3). This concept is supported directly by the CPU architecture of the iAPX 86 and iAPX 88.

**Segment Registers and Usage**

The iAPX 86 and iAPX 88 concepts of segmentation define four types of segments: Code Segments, Data Segments, Stack Segments, and Extra Segments. Each is associated with a CPU based segment register which points to the currently active (being accessed) segment of each type.

Each segment register is 16 bits and defines the starting address of the segment within the iAPX 86's or iAPX 88's one megabyte address space. The segments can start on any 16 byte boundary and may vary from 16 to 64K bytes in length with length resolution on 16 byte boundaries. This implementation optimizes memory usage by allowing the segment size to be defined within 15 bytes of the size of the logical element the segment contains. The segment size assigned to each element is independent of other segment definitions and allows supporting anywhere from sixty-four thousand 16 byte segments to sixteen 64K byte segments within the one megabyte address space.

To reference memory, the value in the segment register selected for a specific memory reference is automatically multiplied by 16 by appending a suffix of four binary zeros, and is added to an offset address specified for each access (Figure 4). The result is a 20-bit address which may be used to access anywhere within the one megabyte of directly addressable memory. The data operand offset address calculation is determined by the selected or implied addressing mode given in the instruction. The available modes allow for greater flexibility in the manipulation of data structures than possible with other types of architectures (ref. 8086 Family Users Manual).

Since the offset address is 16 bits in length, programs that are: (a) less than 64K bytes of code (the maximum segment size); (b) do not change the segment register values and, (c) reference data contained within single data and extra segments (up to 128K bytes of static data), are considered directly relocatable and isolatable from other programs in the system. Considerations for programs which extend beyond a single code segment or two data segments are shown in Appendix 1. The system, therefore, has four segment registers: one to define the data, one to define the stack, one to define the code, and an extra segment which can be used to specify another data segment or global (shared) system data. Since the iAPX 86 and iAPX 88 support based stack segment relative addressing for access to operands on the stack, the stack is typically used for dynamic allocation of workspace and storage of temporary variables in addition to parameter passing during procedure invocation. The technique of dynamic allocation of memory for temporary space reduces the need for static data space resulting in more efficient use of memory. Using the stack for dynamic data provides support for reentrant procedures as demonstrated in Figure 5.
Using the segment registers, the system can provide each user not only a separate memory space for his data, but also an individual program area. By changing the segment registers, the processor may define which program is executing with which data. By specifying different segment values, users may execute different programs. In this way, one user can access a BASIC interpreter while another is using a FORTRAN compiler and a third is using something different. This becomes very useful in large systems where there will be more than one user executing concurrently. The basic system context which must be saved and restored to transition from one user to another consists only of the CPU registers and does not require reinitialization of memory or off chip address translation devices.

With regard to multiuser environments, segmentation not only provides the ability to partition the memory space, but also allows the system to change the areas in memory being accessed by a specific program by changing the segment registers before entering the program. Most iAPX 86 and iAPX 88 microprocessor instructions (all except those which specifically modify the segment registers or pass control of the processor to an area outside the current code segment) access memory relative to the current values of the segment registers. When the operating system wants to move the user to a different area, it only needs to move the program or data and change the appropriate segment register values (Figure 6). The program will continue execution unaffected by the relocation. Using this method, as new users enter the system, their programs are loaded into available memory areas by the operating system. When a program is invoked, the operating system will set the segment registers based on where the program and data are located. This also allows the operating system to reformat the allocation of memory and minimize memory fragmentation as users enter and leave the system. This provides the system with position-independent programs because the users are not dependent on executing at a specific memory location.
The ability to relocate programs anywhere in memory also facilitates the use of multiple 'master' programs by the system. The programs which are requested may be loaded into any user's workspace and not be dependent upon being in a specific location to be useable. In this manner, each user may have his own copy of a program while each program simultaneously resides in any number of locations.

Alternatively, by setting the code segments of two users to the same value, they will both access a single copy of the same program. Of course, in order for the program to execute correctly, it must be reentrant. This implies that as a user is accessing the program, the program cannot be self-modifying and the data the program accesses (including temporary variables defined within the program) must be uniquely defined and private for each user accessing the program. The reentrant program is independent of the users accessing it and is accessible to any number of users, independent of the user's status.

Using this concept, a program executed by more than one user such as a BASIC interpreter or a COBOL compiler may be written so that it always looks for specific pointers or values in locations relative to the active user's data, stack or extra segment registers. When a user wants to invoke this program, the operating system sets the segment registers to point to the user's workspace before initiating execution for this user. When the shared program accesses the memory, it would access the current user's memory.

The ability to have one program process several users' data by merely changing the appropriate segment register (usually the Data and extra segments) simplifies the implementation of multi-user programs. In order for several users to access the same program, it does not need to be duplicated in each user's workspace (Figure 7). As long as the program is reentrant, once the program has been loaded into system memory, any user may access the program by setting the Code Segment and instruction pointer to the entry point of that program.
Semaphores

The system, in addition to the features described earlier, needs some method of allocating resources (memory and external devices or peripherals) when the number of users on the system may exceed the number of system resources available. The system needs the ability to reserve a device or area in memory for a given user and to prevent other users from accessing the resource until the first user releases. It wouldn't work too well if after the first half of a balance sheet was printed, the system started to print a program listing. The purpose of semaphores is to control access to resources, providing a mechanism for a single user to gain access to the resource and allowing other users to determine easily whether the resource is available or not. There are several instructions available to the iAPX 86 and iAPX 88 user which will help him program semaphores into his system. They will be discussed in the following examples.

One method of implementing a semaphore is to reserve a byte for each resource for which access must be controlled. The byte will contain information as to whether a device is being used by someone or whether it is available for use. When a user wishes to reserve a specific resource he checks to see if it is available and if it is, a value specifying that the resource is busy is loaded into its semaphore. The user may then access or gain control of the resource. When someone else wishes to see if the resource is available, all he needs to do is test the semaphore value and see if the resource is available. If not available (busy), the user must wait until the resource becomes available. When the user controlling the resource is done, he must reset the busy indication in the semaphore to allow others to gain control of the resource.

Even in systems that use semaphores to allocate system resources, problems may still arise if the operating system is not programmed properly. For example, imagine a system which has one printer. In order to specify if the printer is being used or not, the system uses a byte in memory. When the system allocates the printer to a user, it places a 1 in the byte to tell other users that the printer is already allocated. Another user may see if the printer is busy by reading the byte and seeing if it is a 1. If not, the printer is available for use.

Now assume that User 2 and User 4 are both setting up reports to run on the printer. User 2 is currently being processed. He tests the byte which will tell him if the printer is busy or not and finds that the printer is not busy. He now knows that the printer is available and is about to reserve the printer for his report. But, before he can store the appropriate value in the byte telling the other users that the printer is busy, he gets interrupted and User 4 begins operations.

When User 4 checks the semaphore to see if there is a report already running on the printer by seeing if the semaphore has a value of 1, it finds that the printer is still available. User 2 did not change the value of the semaphore yet; he was interrupted too soon. So, User
4 sets the semaphore to a 1 to tell the other users that the printer is busy and begins to print his report on the printer. Eventually the processor returns to User 2. Now, User 2 has already checked the semaphore and "knows" that the printer is available so he now loads the semaphore with a 1. (This was also done by User 4 when he took the printer but was not re-checked by User 2 when he returned.) What follows is two different reports being merged as both users send information to the printer.

Within a multiprogrammed single processor environment, there are several methods of dealing with this problem. One is to disable the interrupts before User 2 tries to find an available printer. If this were done then he wouldn't be interrupted until he had had a chance to specify that the printer was in use by loading the semaphore with a 1. After this was done, the interrupts would then be enabled. However, in many systems it is undesirable to allow a user to modify the interrupt system and this method may not be acceptable. Another method, probably easier for the programmer to implement, is to use the iAPX 86/iAPX 88 XCHG (Exchange) instruction when programming this situation. The instruction exchanges the value in one of the operands with the other operand (Figure 8). For User 2 to check the status of the printer, he would first load a register with a 1, specifying that the printer is busy. This value is then exchanged with the value in the semaphore. If anyone interrupts User 2 after the exchange, they will find a 1 in the semaphore and they will know that the printer is busy. User 2, on the other hand, may examine the byte loaded into the register from the semaphore to see if the printer is available. If the printer was already assigned, User 2 would have exchanged a 1 with a 1, the semaphore would have been left the same and User 2, after checking the value in the register, would know that it was busy. If User 2 finds a 0 in the register, he knows that the printer is not busy and that it is now assigned to him. He has already indicated that the printer is being used so he does not need to reload the semaphore with a 1. Since the exchange operation is a single uninterruptable operation of reading and setting the semaphore, the problem in the previous example is avoided.

In a multiprocessor environment, an additional mechanism is required which will prohibit one processor from accessing the data bus while another processor is in the middle of the exchange operation. If both processors attempt to access the semaphore at the same time, one or the other may get erroneous results. While a multiuser system with only one processor will be interrupted only on an instruction boundary, in a multiprocessor environment, access to the bus is commonly shared on bus cycle boundaries. To allow the programmer or system designer to prevent other processors from gaining control of the bus during the exchange with semaphore operation, the iAPX 86 and iAPX 88 have a bus lock feature. LOCK is a special one-byte instruction prefix which will cause the processor to emit a bus-lock signal for the duration of the instruction that the LOCK precedes. The prefix may be placed before any processor instruction. Using the prefix bus arbitration circuitry will lock out all other processors for the duration of the instruction. The programmer may then protect any critical data areas from outside modification until the processor has had a chance to complete the operation being performed.

THE 9-CHIP MULTIPROGRAMMING SYSTEM

To show the performance and architectural capabilities of the iAPX 86 and iAPX 88 in a multiprogramming environment, a small system was developed around the iAPX 88 microprocessor. The iAPX 88 system is based on an 8-bit bus CPU with the full programmer visible architecture of the 16 bit bus iAPX 86. This includes full object code compatibility between the iAPX 86 and iAPX 88 as well as 16 bit
data types, 1 megabyte address space and addressing modes. The advantage of the iAPX 88 in small systems such as the one we will discuss, is compatibility with the multiplexed bus memory and peripheral devices of the 8085 Family. As a result, this system uses nine chips to provide all of the system clock signals, I/O ports, interrupt signals, user workspace, and operating system. In this case, the multiprogramming system is a multiple user Tiny BASIC Interpreter. Written originally to demonstrate the 8085, the code was converted to execute on the iAPX 88 using CONV86. CONV86 is available as part of the general set of iAPX 86/iAPX 88 software development tools and converts 8080/8085 code to iAPX 86/iAPX 88. The iAPX 88 Tiny BASIC Interpreter is a reentrant program to allow multiple users.

**System Overview**

The software structure for the system is shown in Figure 9 and consists of a simple O.S., Tiny Basic and workspace for each user. The O.S. handles terminal I/O and time sharing of Tiny Basic between users. Each user is allocated a separate stack, temporary variable work space, I/O line buffer and BASIC program area. The physical address space for each user is defined by the contents of the segment registers.

The O.S. is effectively an interrupt handler for timer interrupts used for I/O. During each interrupt, the current users machine state is saved on his stack, the stack and data segments are switched to the next user and a return is executed. The return restores the machine state for the next user from his stack and returns to Tiny Basic execution for the next user. Between the saving of machine status for one user and restoring status for the next user, the O.S. performs pending terminal I/O.

The Tiny Basic program transfers control to the O.S. to perform I/O. This suspends Tiny Basic execution for the current user until the O.S. completes the requested I/O. Upon completion of the I/O, the O.S. will return control to Tiny Basic for this user. Tiny Basic makes no distinction between users and references all data relative to the current data and stack segment register values. Both users have the same values for the code segment register values. Both users have the same values for the code segment with the individual IP values depending on each users execution sequence.

**Hardware Description**

The total system consists of: an iAPX 88 microprocessor; 8284 Clock Generator to provide the clock signals to the processor and a programmable timer; 8155-2 to provide 256 bytes of RAM, 20 I/O lines and the programmable timer used to generate interrupts; two 8755A-2's which provide additional I/O lines as well as the EPROM where the BASIC Interpreter programs and the operating systems are stored; and two 8185's which provide 2K-bytes of RAM for the users' workspace. Communications between the user terminals and the processor board is accomplished with a 75189 and 75188 for level conversion from TTL to EIA standards.

The 8284, driven by a crystal, provides the clock signal for the iAPX 88 and the programmable timer on the 8155-2. The suggested crystal, 15 MHz, will provide a 5 MHz signal to the microprocessor and a 2.5 MHz signal to the timer/counter. The 8284 also provides the system reset to the processor and the other chips.

The 8155-2 provides the system with 256 words of RAM and is connected directly to the iAPX 88 multiplexed bus. The system uses the programmable timer on the 8155-2 to generate the baud rate interrupt signals to the processor. The timer output is connected to the non-maskable interrupt pin on the processor. When an interrupt is generated, the processor enters the interrupt routine which performs the...
proper input and output sequences for communication with the terminals and switches from one user to the other for execution of Tiny BASIC. The timer count is set during the initialization sequence to count to a specified value. When it reaches this value, an interrupt signal is sent and the timer begins counting again. Since this timer is set in software, the baud rate is software programmable. In addition to the timer, the 8155-2 provides two programmable 8-bit I/O ports and one 6-bit I/O port.

The 8755A-2 is a 2K-byte EPROM. Two of these chips were used to provide enough space for the operating system and the Tiny BASIC Interpreter. In the final system, approximately 60% of the total EPROM memory was used; the remaining space is available to expand the capabilities of Tiny BASIC or to increase the number of users the system can handle. Like the 8155-2, the 8755A-2 is directly compatible with the iAPX 88's multiplexed bus. In addition to the EPROM there are two 8-bit programmable I/O ports on the chip. Each line in the 8-bit port is individually programmable to be either an input or an output line. Two I/O lines on one of the 8755A-2's provide the I/O for one of the terminals. Two lines on the 8155-2 provide the communications to the other terminal.

The other chip used in the system is the Intel 8185. This chip is a 1K-byte RAM device. Two were designed into the system to allow each user 1K-bytes of memory for program and workspace while generating and executing BASIC programs. This chip, like the others, is compatible with the iAPX 88 multiplexed bus. Additional space to facilitate larger programs or increasing the number of users can be accommodated by increasing the number of 8185's in the system.

All of the chips used in this system are directly compatible with the 5 MHz iAPX 88 system bus. Therefore, no latches or data bus transceivers are needed in the system. Linear select techniques were used to select all devices and eliminate address decode logic. Figure 10 shows a functional diagram of the nine-chip system. To implement multiple users, the TIMER-IN line on the 8155-2 was wired to the PCLK line on the 8284 and, the TIMER-OUT line was wired to the non-maskable interrupt line on the iAPX 88. The timer was set to operate the terminals at 300 baud.

**Software Description**

In addition to the BASIC Interpreter, three programs were needed to provide the multiuser capabilities. The first was an initialization routine which is invoked after system reset. The next was the Character-In/Character-Out routine which is used to communicate with the user's terminal. The last was the Interrupt routine. This routine is called each time the 8155-2 sends an interrupt to the processor.

The two EPROM chips are selected between addresses FF000H and FFFFFH, each having 2K-bytes of memory. These were placed at upper memory since the system accesses addresses FFFFFH from system reset. When this address (FFFF0H) is placed on the address bus, the chip corresponding to FF800H to FFFFFH is activated. The two 8185 RAM chips are selected when addresses between 1000H and 17FFH are placed on the address bus. The first addresses (1000H to 13FFH) are used to hold the data and programs for User 1. The second addresses are used to hold the data and programs for the second user. The 256-bytes of RAM in the 8155-2 are used by the operating system for information it requires that is not directly associated with either of the users. The 8155-2 RAM is selected by addresses 00H to FFH and contains the interrupt vector table. The vector table contains the addresses of the routines the system will execute if the user attempts to divide by zero or when the 8155-2 sends an interrupt signal. These addresses are loaded into the interrupt vector table during the execution of the initialization routine following reset.

An address map of system memory is shown in Figure 11.

![Figure 11. Memory Map](AFN-10550A)
Figure 10. Schematic of 9 Chip 8088 Multiuser Tiny Basic System
First, the initialization routine (Figure 12) sets up the data segment registers so that they point to the data and program area reserved for User 1. The routine then resets all data and program pointers which are reserved for User 1. These pointers and data areas are used by the BASIC Interpreter in keeping track of the processing being done by User 1. This includes pointers which address the beginning and the end of the BASIC program, the data variables, and the information required to perform the I/O with the terminal. After the data and pointers have been initialized for User 1, the routine moves this information into the data and pointer area reserved for User 2, thus initializing the system for the second user. Usage of each user memory space is given in Figure 13.

After the routine has initialized the pointers for both users, it sets up the interrupt vector table. The first entry points to the error routine which will be called if the user attempts to divide by zero. The second entry is the location of the TIMER-OUT routine which will be invoked when the timer sends an interrupt to the NMI input. After the interrupt vectors are initialized, the iAPX 88 initializes the I/O ports which are used to communicate with the two terminals. Since each bit of each port may be programmed as either input or output, they must be defined by the system before they may be used. After the lines are programmed for their defined purpose (one input and one output line for each user), the system outputs a high signal to each of the output ports, sending a STOP bit. This STOP bit will remain valid until the BASIC Interpreter is ready to send a message or data to the user's terminal.

The system software consists of an operating system that handles terminal I/O and sharing CPU processing intervals between the users. There is a single shared user program (Tiny BASIC), and stack and workspaces for each user. To initialize both users with Tiny BASIC, the reset initialization procedure initializes User 2's stack to point to the starting address of Tiny BASIC, sets up the stack extra and data segment register to point to User 1's stack and workspace, enables the 8155 timer for baud rate generation and transfer control to Tiny BASIC for User 1. The CPU then continues to execute Tiny BASIC on behalf of User 1 until a timer interrupt occurs. The interrupt transfers control to the operating system which performs the proper I/O, switches to the next user's segments and returns to the next user. Note the state of the machine for User 1 is saved on his stack. Since User 2's stack was initialized to point to the start of Tiny BASIC, the system now begins executing Tiny BASIC on behalf of User 2. For each interrupt, the operating system may return to either user by loading the proper segment values and returning to where that user had previously been interrupted.

The programmable timer is initialized so that it will generate 900 interrupts per second. The system samples the incoming data from the terminal three times for each bit coming in, and the system communicates at 300 baud. The system samples each bit three times so that the accuracy of the input may be improved. When the system jumps to START (beginning of Tiny BASIC) for each user, the BASIC Interpreter will print "OK" on the terminal and wait for the user to begin entering data on the terminal. For the BASIC to print "OK" on the screen, and to monitor the input from the user's terminal, the program uses the Character-In/Character-Out routine in Tiny BASIC.
The Character-In/Character-Out routines are very simple. Figure 14a is the sequence for a user to enter and exit I/O while figure 14b is a more detailed flowchart for the CI and CO routines. The Character-In routine (CI) sets the count of bits received (INCYCLE) to zero and sets the status (STATUS) for this user to a 2 so that the interrupt routine will know that this user's program is waiting for something to be input on the user's terminal. It pushes onto the user's stack all of the registers and segment registers as if the user had been interrupted, then transfers control to a wait loop in the operating system. It will loop like this until the timer issues an interrupt. When in the I/O routines for a specific user, the system is waiting for a user to input data or waiting for the proper timing to output bits to the user's terminal, and will not process that user's program. This effectively suspends the user's execution until the user requested I/O is complete. When the system has completed the I/O (read a character from the terminal or output one to the terminal), the user's stack for the completed operation is modified so that the operating
system will return to the calling program. While one user is doing I/O, the other is able to process normally, and the processor will give virtually all of its time to the other user.

After the current user is in the I/O mode, the processor attempts to identify who should receive service next. If either of the users is not in an I/O mode, he will receive the control of the processor. If both of the users are in an I/O mode, the system will loop until an interrupt occurs.

The OUTPUT routine works in the same manner as the INPUT routine. The character to be output to the user’s terminal is formatted with the appropriate START and STOP bits and is stored in the user’s memory (WORDOUT). The status byte (STATUS) is set to indicate to the interrupt routine that this user is going to be outputting a byte to the terminal. After the user’s memory is initialized to perform the output to the terminal, his stack is set, as in the CI case, to loop at IORTI until the whole byte has been sent to the terminal. The actual input and output of information to and from the terminal is accomplished by the Interrupt routine.

The interrupt routine is called each time the timer/counter on the 8155-2 reaches the count assigned in the Initialization routine. This count may be modified so that the system can run at any desired baud rate. To determine what the count should be, divide the number of clock cycles (2,500,000) by three times the desired baud rate.

Since the software must handle each bit in the serial I/O stream and samples at 3x the baud rate to eliminate synchronization problems, 300 baud was chosen...
Figure 14b. Character In/Character Out Routines

to provide reasonable I/O speeds and accuracy while avoiding impacting performance for individual users in the multi-user environment.

When the user is interrupted, the interrupt issued by the timer causes the processor to automatically save the flags and the return address onto the interrupted user's stack. This enables the system to tell where to return when it is that user's turn to be processed. The interrupt routine then saves all of the registers for the user. When the system returns to the user, it will have the same values in all of the registers so that the program can continue as if it had never been interrupted.

After all of the registers have been saved, the actual processing of the interrupt can begin. The interrupt routine has two functions. The first is to perform the proper inputs and outputs for the BASIC Interpreter. The second is to identify who is to get service the next time that the processor begins to execute a user's program. Note that if neither user is performing I/O, the interrupt service routine simply switches users (Figure 15).

In performing the inputs and outputs for the BASIC interpreter, it is imperative that the signals sent to the terminals and the checks for incoming data occur at the same point in time for each interrupt cycle. Doing this will ensure the accuracy of the data as it is read from the terminal and will ensure that the output is at the proper baud rate. To get the I/O to occur at the same time in the interrupt cycle, the processor must always perform the same steps. If the system is required to first check the status of the users and then perform the appropriate operation depending on the status found, the processor would be going through different steps, depending upon whether the user was in an INPUT mode, an OUTPUT mode, or a regular processing mode. This in turn would cause the I/O to come at different points in the interrupt routine. To allow the system to perform the I/O at the same time
in each interrupt cycle, the processor always performs input as soon as all of the registers of the current user have been saved. It doesn’t check to see if the user is in an INPUT mode; it doesn’t really matter. It then takes the data received from the input and saves it for future processing. Later on, after performing the time critical I/O functions, the processor will look at the status (STATUS) for each user to see if each user is in an INPUT mode. If the user is in an input mode, the processor will use the data received. If the user is not in an input mode, the data will be ignored.

After it has performed the inputs and saved the data, the processor performs the two outputs, one for each user. Again, it does not check to see if the terminal is in OUTPUT mode; it always outputs something to the terminal. If a user is not in the OUTPUT mode, the processor will send a STOP bit to the terminal. The system uses each user’s status byte (STATUS) to generate the STOP bit if the user is not in OUTPUT mode or to leave the data the same if the user is in OUTPUT mode. In either case, the steps followed to output a bit to the terminal are always the same to keep the timing signals constant.

It may be interesting to see how the processor outputs either a stop bit or the next bit of data. This processing is done in the OUTWORD subroutine (Figure 16). The first thing the routine does is check to see if it is time to send another bit. If not, it jumps around the code where the bit is generated and outputs to the terminal. If it is time, the processor loads the output character to the accumulator. The next bit of the character is transferred from the register to the terminal, low order bit first, one bit every CONNT interval.

The count in CONNT reaches 3 it is changed back to a 0. Each time the OUTWORD routine sees a 0 in CONNT it outputs the next bit to the terminal.

Figure 15. Sequence to Switch Between Users

Figure 16. Output Data Processing
After the appropriate value has been output to the port, the value of the byte being output is shifted to the right one bit, preparing the field for the next time a bit is to be sent to the terminal.

Once the proper outputs have been made to the terminals, the processor begins checking the status of each user and taking appropriate action. If the user is in OUTPUT mode, CONNT is checked to see if a new bit was just output to the terminal. If it was, the count of bits sent to the terminal (OUTCYC) is incremented. For each character the BASIC Interpreter wants to send to the terminal, 10 bits must be output. This includes the 8 bits of data in the byte plus a START bit and at least one STOP bit. When the routine has output the full 10 bits, the user status code is reset to 0 so that the processor will know that the user is no longer in the OUTPUT mode. Next the user's stack is modified so that it will return to the calling program rather than the wait loop.

If the user is not in an OUTPUT mode the system checks to see if he is in an INPUT mode and results in entering the INBYTE routine (Figure 17). If he is in an INPUT mode the system checks to see if the user has received a START bit from the terminal. If the user is waiting for a START bit, the input received from the terminal is tested for a START bit. If it is not, the processor ignores the input and continues to wait until a START bit has been received. When a START bit is finally received, the user status (STATUS) is set to indicate the START bit has been received. CONNT is also initialized so that the input will be received and interpreted correctly.

If the user is not waiting for a START bit and is in INPUT mode, the input received from the terminal is valid data. For each interrupt cycle the processor performs an input at the beginning and one more when it determines that it is in an INPUT mode. These two inputs are performed for each of the three cycles, giving six data inputs from which to determine the value of the bit being sent by the terminal.

The variable BITSS is initialized to OOFFH. (It is stored as a 2-byte word). As each of the inputs is received from the terminal, the value in BITSS is shifted to reflect the data received. If the data received is a 1, BITSS is shifted left once. If the data received is a 0, the value in BITSS is shifted right once. After all six inputs have been checked and BITSS has been shifted accordingly, the value which occurred most often will be indicated by the high order byte of BITSS. The newly received bit is OR'd into BYTE IN. After this sequence has occurred eight times, the bit which was entered first will be in bit 0 and the subsequent entries will follow. Once the whole byte has been received, the user's stack is modified so that the return address, is updated to return to the Character-In routine rather than the wait loop. Here, the value assembled in BYTE IN is placed in the accumulator and the system returns to the program where the Character-In routine was called.

After the interrupt routine has checked all of the I/O and has performed the appropriate action concerning the users' modes, the system determines which user is to receive service next. It first looks at User 1. It checks to see if he is in an INPUT or OUTPUT mode. If User 1 is in either, the system will not start processing for User 1 but will automatically begin processing for User 2. If User 2 is also in an I/O mode, the system will loop until the next interrupt from the timer.) If User 1 is not in an I/O mode, the system checks User 2's status. If User 2 is in an I/O mode, the system will automatically give User 1 service next. If neither of the users are in an I/O mode, the system will return to the user who has waited the longest for service. This is accomplished by examining who was executing when the system was last interrupted, and then setting the segment registers to the other user. In this way, the system is shared between both users. If one of the users must wait while in an I/O routine, then his time allocation is given to the other user until the user waiting in the I/O routine has completed the I/O.

The system changes users by performing an Exclusive OR of 40H with the segment register of the user who was first given service. After the system determines which will be serviced next, it restores the new user's segment registers and then restores the registers and flags which were pushed onto the stack. Thus, the user's status is restored before the interrupt return. The system performs an IRET (Interrupt RETurn) which restores the flags to their original value and returns control to the interrupted program. With all of the registers and flags restored, it appears to the BASIC Interpreter user as if there had never been an interrupt and processing will continue normally.

As long as the BASIC Interpreter references all data relative to the segment registers and does not change the segment registers (is reentrant), the system will handle the two users without difficulty. If the program attempts to change the segment registers, then User 1 may interfere with User 2's data or programs, or vice-versa.

Using this type of operating system, the Tiny BASIC used here could be replaced with any other program which is reentrant. Since all of the users use the same "master" program, there is no need to move one program out when the users change; the system only needs to save the registers of the current user on the
Figure 17. Inbyte Routine to Assemble Input from the Terminal
user’s stack before it attempts to go on to the next user. Code for these functions are included in Appendix 2. The complete multiuser Tiny BASIC program is available through INSITE, the Intel Users Software Library.

Adding More Users

With BASIC or any other “master” program set up this way, it is a simple job to change the operating system so that it will support more than two users. The system, as currently written, assumes that each of the users has a 300-baud terminal and a specific amount of memory available for his data and BASIC programs. These are the areas that someone who desires to add more users to the system will have to modify. The Initialization routine, which initializes the stack of User 2 and the I/O ports which communicate with the terminals, will also have to initialize the stack and the I/O ports of any additional terminals. This means that more memory will be required or that the existing memory will need to be subdivided among the number of allowable users. Also, of the remaining 48 I/O lines which are not currently being used, two lines for each additional terminal will be assigned for communications. If the memory boundaries are changed, one of the pointers in BASIC which defines the size of the buffers will have to be modified to reflect the new size of the buffers.

The Initialization routine will, as it does now, go to the “master” program’s START for the first user. When the first user is interrupted or begins I/O, the system would need some method of determining whose turn it is to receive service. This process could be handled much like the present routine. After the system finds out who is not in an I/O routine, it will process the user who has been waiting the longest for service by restoring that user’s registers, and returning to that user. If all users are in an I/O mode, it will loop until the next interrupt. As soon as the first user finishes with the I/O routine, the system will continue to process that user’s program until other users also continue execution. Then the system will switch users as before.

The Interrupt routine, which now performs the I/O for only two users, would need to perform the inputs and outputs for each additional user, and assemble input or disassemble output as they are received or sent. As long as there are remaining I/O lines and enough memory for each of the users, any number of new terminals could theoretically be added to the system if user response time is not a major consideration, and as long as the interrupt routine can complete execution before the next timeout interrupt occurs.

Variant Considerations

With the extra I/O lines, and additional ROM space not used by the BASIC Interpreter or the operating system, there are other features which could be added to the system if desired. A printer could be added to the system with a special output routine to tell the processor that the user’s program is trying to output to the printer instead of the terminal. This would allow the user to generate hard-copy reports or program listings. To do this, would require addition of commands to the BASIC Interpreter so that the user who desires to print on the printer can do so in his BASIC program. The BASIC Interpreter would also need to provide a method where the user could check to see if the printer is available. Some type of semaphore is required so that the other users can tell if the printer is in use, and reserve it for their use if it is available.

The Interrupt routine would then be changed so that it would give the proper output signals to the printer, both when in use and when idle. The type of output to be sent depends upon the type of printer and the signals the printer expects. Also, the initialization routine would need to be modified so that it would send the proper initialization signals to the printer when the system is reset.

Since there are multiple lines available on the I/O ports, other types of peripherals could be added to the system as long as there are methods by which the users could request and release the devices and there is enough room in the BASIC Interpreter to add the commands to allow the BASIC programs to access the device.

CONCLUSIONS

Admittedly, we have shown a relatively simple example of a multiprogrammed system with most of the system complexity evolving around totally software driven, simultaneous I/O for both users. It is just this point, however, which is noteworthy. The basic architectural characteristics of the iAPX 86 and iAPX 88 allowed us to simply (almost trivially) implement the reentrant Tiny BASIC interpreter and the operating system primitives for time multiplexing use of the interpreter between two separate users. In a similar vein, the architecture also supports the capabilities of dynamic relocation and controlled access to critical regions which would be required by more sophisticated systems. With the addition of operating system capabilities for memory allocation and management, loader and file I/O a more general purpose system could be developed.
APPENDIX A

CONSIDERATIONS FOR PROGRAMS WITH MULTIPLE CODE AND DATA SEGMENTS
To extend the concepts of relocatability to programs which consist of multiple code and data segments, the iAPX 86 and iAPX 88 support the ability to transfer control indirectly through memory and load data segment addresses from memory based tables. These capabilities may be implemented in various ways depending upon the characteristics of the code generators, load modules and loader. The basics of any implementation are:

1) Transfer of control to all external procedures or labels must be indirect through memory.

2) References to each data segment must be preceded by loading the data segment register from a table containing the location of each segment in memory.

The tables must be constructed by the loader at load time of the program and data, and maintained by the operating system if the segments are relocated. The location of the tables are implementation dependent. If programs are RAM based, the tables may be appended to the code segments and accessed relative to those segments, each segment maintaining its own table of data segments and external code references (Figure A.1).

---

**Figure A1.** Table of Pointers to other Code and Data Segments associated with each Code Segment
An alternate technique (Figure A.2) would allow a single table to be constructed for use by all code segments and contain the location of all code and data segments. If the programs reserve use of the extra segment for containing this table, the loader and O.S. need maintain only one table rather than one within each code segment. Another benefit of this approach allows the code segments to be in ROM without fixing their location in memory for all possible instances of their use. This is particularly applicable to library routines that will be used in a variety of end applications. A transfer of program control would then require loading the code segment and instruction pointer values from the ES based table. The displacement into the table is specified in the control transfer instruction and therefore, must be specified during linkage or preparation of load modules. Likewise, the location of each data segment must explicitly be loaded with the LDS instruction. The LDS instruction must reference the table in the ES segment and contain the displacement to the appropriate data segment address.

These techniques support programs and data which are relocatable at load time but not necessarily dynamically relocatable (i.e., operation is suspended, code and data are relocated and execution resumes). Since stack based return addresses and pointers are real addresses, dynamic relocation of code or data based on these techniques would require fixing up stack resident segment values, in addition to jump tables, before resuming task execution.

![Diagram](https://via.placeholder.com/150)

*Figure A2. Common Table of Pointers*
APPENDIX B
MONITOR LISTINGS
CODE SEGMENT WORD PUBLIC 'CODE'

; EQUATES FOR MULTIUSER OPERATING SYSTEM

INPORT EQU 0002H ; 8155 PORT B ADDRESS (INPUT).
PORTCNT EQU 0000H ; 8155 PORT CONTROL REGISTER ADDRESS.
OUTPT2 EQU OF00OH ; 8755A PORT B ADDRESS (INPUT).
INPRT2 EQU OF001H ; 8755A PORT A ADDRESS (OUTPUT).
OUTDR EQU OF002H ; 8755A PORT A DIRECTION CONTROL REG ADDRESS.
INDR EQU OF003H ; 8755A PORT B DIRECTION CONTROL REG ADDRESS.
OUTPORT EQU 0001H ; 8155 PORT A ADDRESS (OUTPUT).
TIMHI EQU 5 ; ADDRESS OF 8155 TIMER.

+1 $EJ
THIS ROUTINE Initializes VARIABLES FOR EACH USER INCluding STACK AREAS AND STATE CONDITIONS.

FB3F 8B---- R 1395 INIT: MOV AX, DGROUP ;INITIALIZE AX TO 0.
FB42 BEEB 1396 MOV DS, AX ;INITIALIZE DATA, STACK AND EXTRA SEGMENTS.
FB44 BEEC 1397 MOV ES, AX ;TO MEMORY LOCATION 0.
FB46 BEE0 1398 MOV SS, AX
FB48 BCFF1390 R 1399 MOV SP, OFFSET(STK)
FB4C BF0010 1400 MOV DI, 1000H ;SET DI TO USER1 VARIABLE AREA.
FB4F B91FOO 1401 MOV CX, 1FH
FB52 F3 1402 REP STOSB ;CLEAR CONTROL VARIABLES.
FB53 A4
FB54 B926A710 R 1403 MOV STACKP, SP ;INITIALIZE COPY OF STACK POINTER.
FB58 FF0E610 R 1404 DEC M1
FB5C BE0010 1405 MOV SI, 1000H
FB5F BF101A 1406 MOV DI, 1400H ;AX = OFFSET FOR USER 2.
FB62 B91FOO 1407 MOV CX, 1FH
FB65 F3 1408 REP MOVSB ;COPY USER1 VARIABLES INTO USER2.
FB66 A4 1409 MOV CRoUP: DZO, OFFSET GIOW ;DIVIDE BY ZERO INTERRUPT.
FB67 2EC706000088F5 R 1414 MOV CRoUP: DZS, CODE
FB6E 2EC7060200---- R 1415 MOV CRoUP: DIS, CODE
FB75 2EC7060000AFC R 1416 MOV CRoUP: TDD, OFFSET TIMOUT ;TIMER INTERRUPT.
FB7C 2EC7060A00----- R 1417 MOV CRoUP: TDS, CODE
1418 +1 #EJ
**LOC OBJ** | **LINE** | **SOURCE**
--- | --- | ---
1420 | . | INITIALIZE THE I/O PORTS FOR TERMINAL I/O
1421 | | 
1422 | MOV DX, PORTCNT | : LOAD ADDR OF 8155 COMMAND REG.
1423 | MOV AL, INOUT | : CONFIGURE PORTS: PA=OUTPUT.
1424 | | : PB=INPUT.
1425 | OUT DX, AL | : LOAD ADDR OF 8155 PORT A
1426 | MOV DX, OUTPORT | : LOAD PORT WITH FFH TO BEGIN
1427 | MOV AL, OFFH | : TRANSMITTING STOP BITS TO THIS
1428 | | : TERMINAL.
1429 | | 
1430 | OUT DX, AL | : SET UP USER2’S I/O PORTS ON THE 8755A.
1431 | MOV DX, INDR | : ZERO OUT AX TO ESTABLISH PORT B
1432 | XOR AX, AX | : AS AN INPUT PORT.
1433 | | : OUTPUT TO PORT B CONTROL REGISTER.
1434 | OUT DX, AL | : ADJUST DX TO INITIALIZE THE PORT A
1435 | DEC DX | : CONTROL REGISTER.
1436 | | : MAKE AL ALL ONES TO ESTABLISH PORT A
1437 | MOV AL, OFFH | : HAS AS AN OUTPUT PORT.
1438 | | : OUTPUT TO PORT A CONTROL REGISTER.
1439 | OUT DX, AL | : SET DX TO THE ADDRESS OF PORT A.
1440 | DEC DX | : OUTPUT STOP BITS TO THE TERMINAL.
1441 | | 
1442 | MOV AX, OFFH | : SET AX TO USER2 SEGMENT VALUE.
1443 | | : LOAD SEGMENT REGISTERS FOR USER2
1444 | MOV SS, AX | : INITIALIZE REGISTERS FOR USER1 AND THE STACK FOR USER2
1445 | MOV DS, AX | : FOR USER2.
1446 | MOV ES, AX | 
1447 | XOR AX, AX | 
1448 | MOV AX, 040H | 
1449 | MOV BP, AX | 
1450 | MOV DS, AX | 
1451 | MOV ES, AX | 
1452 | MOV BX, AX | 
1453 | MOV CX, AX | 
1454 | MOV DX, AX | 
1455 | MOV BP, AX | 
1456 | MOV SI, AX | 
1457 | MOV DI, AX | 
1458 | PUSHF | : INITIALIZE FLAG IMAGE FOR USER2
1459 | | : INITIALIZE CODE AND IP VALUES
1460 | PUSH CS | : TO POINT TO THE START OF TINY BASIC
1461 | MOV AX, OFFSET(START) | : GO PUSH ALL REGISTERS ONTO USER2’S
1462 | CALL SVREG | : STACK AND SWITCH TO USER1’S STACK.
1463 | | : THIS MECHANISM INITIALIZES THE REGISTERS
1464 | | : FOR USER1 AND INITIALIZES THE STACK FOR
1465 | | : USER2.
1466 | | 
1467 | | 
1468 | +1 $EJ | 

29

AFN-10550A
MACRO ASSEMBLER MTEASC

LOC OBJ LINE SOURCE
1469 1470 1471 . INITIALIZATION OF THE 8155 TIMER
1472 1473 1474

FBC0 BA0500 1475 MOV DX, TIMHI ; SET TIMER COUNT AND MODE IN THE 8155 TIMER.
1476 MOV AL, OCAH ; MODE=4 (11, AUTO RELOAD AND SINGLE PULSE
1477 OUT DX, AL ; OUTPUT ON TERMINAL COUNT) AND COUNT=OADAH.

FBC5 EE 1478 OUT DX, AL ; SET HIGH BYTE.
FBCA 4A 1479 DEC DX ; SET LOW BYTE. TIME DELAY = 1.111MS.

FBC7 30DA 1480 MOV AL, ODAH ; SET PORT CONTROL TO START COUNTING.
FBC9 EE 1481 OUT DX, AL
FBCA 3302 1482 XOR DX, DX
FBCB B0C1 1483 MOV AL, OCIH
FBCD EE 1484 OUT DX, AL
FBCF E9BEF9 1485 JMP START ; JUMP TO TINY BASIC TO START EXECUTION FOR
1487 +1 $EJ ; USER1.
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1488</td>
<td>CHARACTER INPUT AND CHARACTER OUTPUT ROUTINES</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1489</td>
<td>THESE ROUTINES TRANSFER THE USER INTO INPUT OR OUTPUT MODE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1490</td>
<td>AND SUSPENDS TINY BASIC EXECUTION FOR THIS USER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1491</td>
<td>UNTIL THE I/O IS COMPLETE.</td>
</tr>
<tr>
<td>1492</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1493</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1494</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1495</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FBD2</td>
<td>B002</td>
<td>1496</td>
<td>CI: MOV AL, 2; SET INPUT MODE STATUS.</td>
</tr>
<tr>
<td>FBD4</td>
<td>C606A</td>
<td>1497</td>
<td>CI: MOV INCYCL, 0; RESET INPUT CYCLES.</td>
</tr>
<tr>
<td>FBD4</td>
<td>A2AE</td>
<td>1498</td>
<td>COMP: MOV STATUS, AL; SAVE STATUS.</td>
</tr>
<tr>
<td>FBD4</td>
<td>9C</td>
<td>1499</td>
<td>PUSHF; SET UP STACK FOR IRET.</td>
</tr>
<tr>
<td>FBD6</td>
<td>9E</td>
<td>1500</td>
<td>PUSH CS</td>
</tr>
<tr>
<td>FBD7</td>
<td>B0E9FD</td>
<td>1501</td>
<td>MOV AX, OFFSET(CGROUP: IORTI); FORCE USER TO RETURN TO IORTI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1502</td>
<td>UNTIL INPUT IS COMPLETE.</td>
</tr>
<tr>
<td>FBD8</td>
<td>90</td>
<td>1503</td>
<td>PUSH AX</td>
</tr>
<tr>
<td>FBE3</td>
<td>EB9501</td>
<td>1504</td>
<td>CALL SVREG; SAVE REGISTERS FOR NORMAL RETURN.</td>
</tr>
<tr>
<td>FBE6</td>
<td>EB5E90</td>
<td>1505</td>
<td>JMP USER?; GO SWITCH TO OTHER USER.</td>
</tr>
<tr>
<td>FBE8</td>
<td>E8FE</td>
<td>1506</td>
<td>IORTI: JMP IORTI; LOOPS TO ITSELF UNTIL TIMEROUT.</td>
</tr>
<tr>
<td>FBE8</td>
<td>A0AD10</td>
<td>1507</td>
<td>CIRT: MOV AL, BYTEIN; RETURNS HERE WHEN CI HAS 7 BITS, LEAVE CHARACTER IN AL.</td>
</tr>
<tr>
<td>FBE8</td>
<td>C3</td>
<td>1508</td>
<td>RET; RETURN TO TINY BASIC FOR THIS USER WHEN THE CHARACTER IS RECEIVED.</td>
</tr>
<tr>
<td>FBEF</td>
<td>50</td>
<td>1509</td>
<td>PUSH AX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1510</td>
<td>OR AX, OFOOH; PUT STOP BITS IN WITH THE CHARACTER TO BE OUTPUT.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1511</td>
<td>SAL AX, 1; SHIFT LEFT TO SET UP START BIT (0) IN THE LOW ORDER BIT POSITION.</td>
</tr>
<tr>
<td>FBF5</td>
<td>A3AB10</td>
<td>1512</td>
<td>MOV WORDT, AX; TRANSFER CHARACTER TO TEMPORARY STORAGE FOR OUTPUT TO THE TERMINAL.</td>
</tr>
<tr>
<td>FBF8</td>
<td>C606A</td>
<td>1513</td>
<td>MOV OUTCYC, 0; RESET OUTCYCLES AND BIT COUNT.</td>
</tr>
<tr>
<td>FBF8</td>
<td>C606AF</td>
<td>100090</td>
<td>1514</td>
</tr>
<tr>
<td>FC04</td>
<td>B001</td>
<td>1515</td>
<td>MOV AL, 1; SET STATUS TO OUTPUT MODE.</td>
</tr>
<tr>
<td>FC06</td>
<td>EB02</td>
<td>1516</td>
<td>JMP COMP; GO SAVE STATUS, SETUP STACK FOR IRET TO IORTI AND SWITCH TO THE OTHER USER.</td>
</tr>
<tr>
<td>FC08</td>
<td>3B</td>
<td>1517</td>
<td>JUMP USER.</td>
</tr>
<tr>
<td>FC09</td>
<td>C3</td>
<td>1518</td>
<td>MOV CONNT, 0; START CIRT.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1519</td>
<td>Orc AX AX, 0; PUT STOP BITS IN WITH THE CHARACTER TO BE OUTPUT.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1520</td>
<td>SAL AX, 1; SHIFT LEFT TO SET UP START BIT (0) IN THE LOW ORDER BIT POSITION.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1521</td>
<td>MOV WORDT, AX; TRANSFER CHARACTER TO TEMPORARY STORAGE FOR OUTPUT TO THE TERMINAL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1522</td>
<td>MOV OUTCYC, 0; RESET OUTCYCLES AND BIT COUNT.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1523</td>
<td>MOV AL, 1; SET STATUS TO OUTPUT MODE.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1524</td>
<td>JMP COMP; GO SAVE STATUS, SETUP STACK FOR IRET TO IORTI AND SWITCH TO THE OTHER USER.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1525</td>
<td>JUMP USER.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1526</td>
<td>MOV CONNT, 0; START CIRT.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1527</td>
<td>OR AX AX, 0; PUT STOP BITS IN WITH THE CHARACTER TO BE OUTPUT.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1528</td>
<td>SAL AX, 1; SHIFT LEFT TO SET UP START BIT (0) IN THE LOW ORDER BIT POSITION.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1529</td>
<td>MOV WORDT, AX; TRANSFER CHARACTER TO TEMPORARY STORAGE FOR OUTPUT TO THE TERMINAL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1530</td>
<td>MOV OUTCYC, 0; RESET OUTCYCLES AND BIT COUNT.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1531</td>
<td>OR AX AX, 0; PUT STOP BITS IN WITH THE CHARACTER TO BE OUTPUT.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#EJ
THIS IS THE TIMER INTERRUPT SERVICE ROUTINE WHICH PERFORMS TERMINAL INPUT AND OUTPUT AND MULTIPLEXES THE CPU TIME BETWEEN THE TWO USERS.

AP-10550A
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>LINE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>FC6E 1F</td>
<td>1586</td>
<td>POP DS</td>
<td>;RESTORE USERS MACHINE STATE FOR TINY BASIC EXECUTION.</td>
</tr>
<tr>
<td>FC6F 07</td>
<td>1587</td>
<td>POP EB</td>
<td></td>
</tr>
<tr>
<td>FC70 5D</td>
<td>1588</td>
<td>POP BP</td>
<td></td>
</tr>
<tr>
<td>FC71 5F</td>
<td>1589</td>
<td>POP DI</td>
<td></td>
</tr>
<tr>
<td>FC72 5E</td>
<td>1590</td>
<td>POP SI</td>
<td></td>
</tr>
<tr>
<td>FC73 5A</td>
<td>1591</td>
<td>POP DX</td>
<td></td>
</tr>
<tr>
<td>FC74 59</td>
<td>1592</td>
<td>POP CX</td>
<td></td>
</tr>
<tr>
<td>FC75 5B</td>
<td>1593</td>
<td>POP BX</td>
<td></td>
</tr>
<tr>
<td>FC76 5B</td>
<td>1594</td>
<td>POP AX</td>
<td></td>
</tr>
<tr>
<td>FC77 CF</td>
<td>1595</td>
<td>IRET</td>
<td>;RETURN TO TINY BASIC FOR ACTIVE USER</td>
</tr>
<tr>
<td></td>
<td>1596</td>
<td></td>
<td>;OR IORTI IF BOTH USERS ARE IDLE (IN I/O).</td>
</tr>
<tr>
<td></td>
<td>1597</td>
<td></td>
<td>1598 +1 $EJ</td>
</tr>
</tbody>
</table>
LOC DSJ  
LINE SOURCE
1599 1600 1601 1602  
FC78 A9A10N R 1603 OUTWORD: MOV AL,CONNT ONLY OUTPUT EVERY 3RD CYCLE.
FC79 2A03  
FC7D 7514 1605 JNZ OUT1 NOT ON THIS CYCLE.
FC7F A1A10N R 1606 MOV AX,WORDT AX LOAD WORD OUT.
FC82 BAIAE10 N 1607 MOV BL,STATUS LOAD STATUS BYTE.
FC86 8C80FE  
FC8F FA93 1609 NOT BL FORCE THE
FC89 0A2C3 1610 OR AL,BL LOW ORDER BIT TO BE A STOP BIT
FC8D EE 1611 OUT DX,AL IF NOT IN CO ROUTINE.
FC8E D1FB 1612 SAR AX,1 SHIFT FOR NEXT BIT TO OUTPUT.
FC90 3A2A10 R 1613 MOV WORDT,AX SAVE FOR NEXT BIT TIME.
FC93 FE66A10 R 1614 OUT: INC CONNT INCREMENT COUNT.
FC97 803EAF1003 1615 CMP CONNT,3 CO USER IN OUTPUT MODE.
FC9C 750L 1616 JNE OTRT JUMP IF STILL TRANSMITTING THIS BIT.
FC9E C606AF100090 R 1617 MOV CONNT,0 RESET COUNT IF EQUAL TO 3, CONNT=0
FC9F C3 1618 INDICATES TIME TO BEGIN TRANSMITTING
FC44 C3 1619 OTRT: RET TO THE NEXT BIT.
FC4A 8A10E 1620  
FC4B FA310 F 1621 THIS ROUTINE CONSTRUCTS THE BIT RECEIVED AND ASSEMBLES THE BITS INTO CHARACTERS.
FC4C 743A  
1622 1623  
FC4A 8A10E 1624 INBYTE: MOV BL,STATUS LOAD STATUS.
FC4C 743A  
1625 1626  
FC4A 80E301 1627 AND BL,01H TEST FOR OUTPUT STATUS.
FC4E 743A 1627  
FC50 803EAF1000 R 1628 CMP CONNT,0 USER IN OUTPUT MODE.
FC55 7520 1629 JNZ BRET RETURN IF STILL TRANSMITTING THIS BIT.
FC57 FE66A10 R 1630 INC OUTCYC INCREMENT NUMBER OF BITS TRANSMITTED.
FC5B 803EAE100A R 1631 CMP OUTCYC,10 TEST IF ALL HAVE BEEN SENT INCLUDING
FC5C 7515 1632 JNE OTRT START AND STOP BITS.
FC5E 743A 1633 JNE BRET RETURN IF STILL TRANSMITTING THIS
FC62 C606AE100090 R 1635 MOV STATUS,00H IF ALL BITS HAVE BEEN TRANSMITTED,
FC6B B926A710 R 1636 MOV BX,OFFSET(CGROUP,3ORT) RESET STATUS AND MODIFY THE
FC6F B926A710 R 1637 RSST: MOV STACKP,SP USERS STACK TO RETURN TO CORT AND TINY
FC74 B926A710 R 1638 ADD SP,22 BASIC RATHER THAN THE WAIT LOOP AT OTRT
FC75 B926A710 R 1640 PUSH BX MODIFY IP ON THE STACK TO RETURN TO
FC77 59 1641  
FC78 59 1642 MOV SP,STACKP RESTORE STACK POINTER.
FC7D 59 1643 BRET: POP CX SAVE IF SO IT CAN BE RESTORED AFTER
FC7E 51 1644  
FC7F 51 1645 MOV STACKP,SP  
FC86 33C0 1646 XOR AX,AX SWITCH TO USER1 IN CASE THIS WAS USER2.
FC8B 8D8 1647 MOV DH,AX  
FC8E 8D8 1648 MOV DS,AX  
FC92 8D8 1649 MOV ES,AX  
FC97 8D8 1650 MOV SP,STACKP  
FC9C 51 1651 PUSH CX  
FC9D 51 1652 RET AND RETURN.
THESE ROUTINES ARE USED BY INBYTE TO CONSTRUCT THE INPUT CHARACTERS.

FCFI 80E704   R 1660 AND BH,0A0H ; SEE IF STILL WAITING FOR START BIT.
FCF4 7468 R 1661 JZ WAITST ; JMP IF STILL WAITING.
FCF6 B0E180   R 1662 AND CL,BH ; TEST THE BIT SAMPLED THE FIRST TIME.
FCF9 7407 R 1663 JZ CK1 ; IF BIT-IN IS A 0, SHIFT LEFT
FCFB D10EA510 R 1664 ROR BITSS,1 ; ELSE SHIFT BYTE RIGHT.
FCFF EBO;)'10 1665 JMP CK2

FCF1 80E70A R 1666 AND BH,04H ; SEE IF STILL WAITING FOR START BIT.
FCF4 7468 R 1667 JZ WAITST ; JMP IF STILL WAITING.
FCF6 B0E180   R 1668 AND CL,BH ; TEST THE BIT SAMPLED THE FIRST TIME.
FCF9 7407 R 1669 JZ CK1 ; IF BIT-IN IS A 0, SHIFT LEFT
FCFB D10EA510 R 1670 ROR BITSS,1 ; ELSE SHIFT BYTE RIGHT.
FCFF EBO;)'10 1671 JMP CK2
THIS ROUTINE SAVES THE STATE OF THE MACHINE FOR THE CURRENT USER AND RETURNS TO USER1.

```
; SAVE THE RETURN ADDRESS ON THE STACK
MOV BL1, DX
; SAVE THE CURRENT MACHINE STATE.
POP BX
; SAVE THE STACK POINTER.
MOV STACKP, SP
; SWITCH TO USER1.
MOV CX, SS
; LOAD SEGMENT REGISTERS.
XOR AX, AX
; SAVE THE STACK POINTER.
MOV SS, AX
; LOAD THIS USERS STACK POINTER.
MOV ES, AX
; SAVE PREVIOUS USERS STACK SEGMENT.
MOV DS, AX
; RESTORE THE RETURN ADDRESS.
MOV SP, STACKP
; AND RETURN.
PUSH DS
PUSH BP
PUSH SI
PUSH DI
PUSH BL
PUSH SI
PUSH CX
PUSH BL
PUSH AX
PUSH BX
RET
```

CODE ENDS
DATA SEGMENT WORD PUBLIC 'DATA'

ORG 1000H

iTINY BASIC VARIABLES.

BL1 DW ?
BL2 DW ?
ZERO DW 0
M1 DW -1
POLO DB 0
RICI DB 0
CURRNT DW 0
Z DB 0
STKGOS DW 0
STKINP DW 0
LOPVAR DW 0
LOPLMT DW 0
LOPLN DW 0
LOPPT DW 0
RANPNT DW

DATA ENDS

DATA2 SEGMENT WORD 'DATA' O.S. VARIABLES FOR EACH USERS I/O.

ORG 01020H

VARBGN DB 2*26 DUP (?)

TXUNF DW DGROUP:TXTBGN

DATA ENDS

COME HERE AFTER RESET AND JUMP TO INITIALIZATION.

DATA STRUCTURE DEFINITION

THIS SECTION DEFINES THE DATA STRUCTURE WHICH CONTAINS THE
VARIABLES AND CONTROL INFORMATION FOR EACH USER IN THE SYSTEM.
NOTE THE STRUCTURE IS DEFINED ONLY ONCE SINCE ALL OFFSETS ARE
FORMED RELATIVE TO THE SEGMENT REGISTERS WHICH ARE TRANSPARENT
TO THE USERS. THIS DEFINITION SERVES AS A TEMPLATE FOR CONTRACTING
THE OFFSETS USED, AND IS REPLICATED FOR EACH USER.
ASSEMBLY COMPLETE, NO ERRORS FOUND
<table>
<thead>
<tr>
<th>State</th>
<th>City</th>
<th>Address</th>
<th>Phone</th>
<th>TWX</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALABAMA</td>
<td></td>
<td>Intel Corp. 303 Williams Avenue, S.W. Suite 1422 Huntsville 35801 Tel: (205) 533-9553</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARIZONA</td>
<td></td>
<td>Intel Corp. 10210 N. 25th Avenue, Suite 11 Phoenix 85021 Tel: (602) 987-9695</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CALIFORNIA</td>
<td></td>
<td>Intel Corp. 7670 Opportunity Rd. Suite 135 San Diego 92111 Tel: (714) 268-3563</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Corp. * 2000 East 4th Street Suite 100 Santa Ana 92705 Tel: (714) 935-9642 TWX: 910-985-1114</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Corp. * 5530 Corbin Avenue Suite 120 Tarzana 91356 Tel: (213) 986-9510 TWX: 910-495-2045</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Corp. * 3375 Scott Blvd., Santa Clara 95051 Tel: (408) 987-9066 TWX: 910-338-9279</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Corp. * 3396 Royston Drive Suite 202 Scottsdale 85251 Tel: (602) 994-5400</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Corp. * 558 Valley Way Calaveras Business Park Milpitas 95035 Tel: (408) 946-8885</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Corp. * P.O. Box 1400 Cupertino 95014 Tel: (408) 257-9880</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Corp. * 558 Valley Way Calaveras Business Park Milpitas 95035 Tel: (408) 946-8885</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Corp. * P.O. Box 8763 Fountain Valley 92708 Tel: (714) 839-3341</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Corp. * 1321 Centinela Avenue Suite 1 Santa Monica 90404 Tel: (213) 829-4797</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Corp. * 20121 Ventura Blvd., Suite 240E Woodland Hills 91364 Tel: (213) 347-5900</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Colorado</td>
<td></td>
<td>Intel Corp. * 650 S. Cherry Street Suite 720 Denver 80222 Tel: (303) 321-8086 TWX: 910-931-2289</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Corp. * Peacock Alley 36 Paddockam Road Danbury 06810 Tel: (203) 792-8366 TWX: 710-456-1199</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Florida</td>
<td></td>
<td>Intel Corp. 1901 N W. 62nd Street, Suite 406 Ft. Lauderdale 33309 Tel: (305) 771-0900 TWX: 510-956-5407</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Corp. 5151 Addison Street, Suite 203 Orlando 32804 Tel: (305) 626-2393 TWX: 810-855-9219</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEORGIA</td>
<td></td>
<td>Intel Corp. 3300 Holcomb Bridge Rd. Norcross 30092</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ILLINOIS</td>
<td></td>
<td>Intel Corp. * 2550 Golf Road, Suite B15 Rolling Meadows 60008 Tel: (312) 981-7200 TWX: 910-651-5681 Technical Representatives 1520 North Lind Street Bloomington 61701 Tel: (309) 293-8080</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDIANA</td>
<td></td>
<td>Intel Corp. 9101 Weslayan Road Suite 204 Indianapolis 46286 Tel: (317) 296-0623</td>
<td></td>
<td></td>
</tr>
<tr>
<td>KANSAS</td>
<td></td>
<td>Intel Corp. 9393 W. 110th St., Ste. 295 Overland Park 66210 Tel: (913) 942-8880 Technical Representatives, Inc. 8245 Nieman Road, Suite 100 Lenexa 66214 Tel: (913) 884-9021; 3, 3 &amp; 4 TWX: 910-749-6412 Technical Representatives, Inc. 360 N. Rock Road Suite 4 Wichita 67206 Tel: (316) 681-0224</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MARYLAND</td>
<td></td>
<td>Intel Corp. * 7257 Parkway Drive Hanover 21076 Tel: (301) 796-7500 TWX: 710-862-1944</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Intel Corp. * 16021 Industrial Dr. Gaithersburg 20760 Tel: (301) 949-4350</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MASSACHUSETTS</td>
<td></td>
<td>Intel Corp. * 27 Industrial Ave. Chelmsford 01824 Tel: (508) 677-8126 TWX: 710-343-6333 EMC Corp. 381 Elliot Street Newton 02164 Tel: (617) 244-7700 TWX: 925231</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MICHIGAN</td>
<td></td>
<td>Intel Corp. 26500 Northwestern Hwy. Suite 401 Southfield 48075 Tel: (313) 353-0202 TWX: 810-244-4915</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINNESOTA</td>
<td></td>
<td>Intel Corp. 7401 Metro Blvd. Suite 355 Edina 55435 Tel: (612) 835-6122 TWX: 910-575-2967</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MISSOURI</td>
<td></td>
<td>Intel Corp. 502 Earth City Plaza Suite 121 Earth City 63045 Tel: (314) 291-1990 Technical Representatives, Inc. * 502 Earth City Plaza Suite 201 Earth City 63045 Tel: (314) 291-0001 Technical Representatives, Inc. * VSIBldg. Suite 501 406 W. 31st Street Kansas City 64111 Tel: (816) 756-3575 TWX: 910-771-0025</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW JERSEY</td>
<td></td>
<td>Intel Corp. * Raritan Plaza 2nd Floor Raritan Center Edison 08817 Tel: (201) 225-3000 TWX: 710-480-6238</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW MEXICO</td>
<td></td>
<td>BFA Corporation 1704 Moon N.E., Suite 7 Las Cruces 88012 Tel: (505) 523-0661 TWX: 910-983-0543 BFA Corporation 3705 Westerfield, N.E. Albuquerque 87111 Tel: (505) 292-1212 TWX: 910-989-1157</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW YORK</td>
<td></td>
<td>Intel Corp. * 300 Motor Fwy. Hauppauge 11787 Tel: (516) 231-3300 TWX: 510-257-6236 Intel Corp. 80 Washington St. Poughkeepsie 12601 Tel: (914) 473-2303 TWX: 510-248-0060</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEW JERSEY</td>
<td></td>
<td>Intel Corp. * 2255 Lyell Avenue Lower Floor East Suite Rochester 14605 Tel: (716) 254-6120 TWX: 510-253-7391 Measurement Technology, Inc. 159 Northern Boulevard Great Neck 11021 Tel: (516) 482-3500 T-Squared 4054 Newcourt Avenue Syracuse 13206 Tel: (315) 463-8592 TWX: 910-541-0554 T-Squared 2 E. Main Victor 14564 Tel: (315) 924-9101 TWX: 510-254-8042</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NORTH CAROLINA</td>
<td></td>
<td>Intel Corp. 154 Hoffman Mill Rd. Burlington 27215 Tel: (919) 584-3031</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| OHIO | | Intel Corp. * 6500 Polk Avenue Dayton 45415 Tel: (513) 980-5390 TWX: 810-450-2528 Intel Corp. * Chagrin Brancad Bldg., No. 300 2800 Chagrin Blvd. Cleveland 44122 Tel: (216) 454-2736 TWX: 810-427-5928 | | *

*Field Application Location
# U.S. AND CANADIAN DISTRIBUTORS

## September 1980

<table>
<thead>
<tr>
<th>State</th>
<th>City</th>
<th>Address</th>
<th>Phone</th>
<th>TWX</th>
<th>TWX Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW YORK</td>
<td></td>
<td>Tel: (212) 956-3324</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWX: 910-336-0026</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TELEX: 34-6372</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>NEW YORK (cont.)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel: (212) 956-3324</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWX: 910-336-0026</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TELEX: 34-6372</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>PA</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel: (717) 826-2000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWX: 810-427-6400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>OHIO</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel: (614) 228-5000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWX: 910-262-1182</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ONTARIO</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel: (416) 777-4400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWX: 610-262-1182</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>ORINDA (cont.)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel: (216) 965-2000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWX: 910-336-0026</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TELEX: 34-6372</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>OREGON</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel: (503) 641-9070</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWX: 910-845-2215</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>PENNSYLVANIA</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel: (215) 476-2000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWX: 910-262-1182</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>QUEBEC</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel: (514) 925-4384</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWX: 810-262-1182</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>WISCOSIN</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel: (602) 925-4384</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWX: 910-262-1182</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>WASHINGTON</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel: (206) 476-2000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWX: 910-845-2215</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>WISCONSIN</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tel: (206) 476-2000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWX: 910-845-2215</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Note: The above table lists distributors for U.S. and Canadian regions, providing contact information such as phone numbers and TWX codes.*
<table>
<thead>
<tr>
<th>Country</th>
<th>Address</th>
<th>Phone</th>
<th>Telex</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GERMANY</strong></td>
<td>Intel Semiconductor GmbH*</td>
<td>(089) 53 891</td>
<td>04186183</td>
</tr>
<tr>
<td></td>
<td>Senefelderstrasse 27</td>
<td>523 177</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Intel Semiconductor GmbH Mainzer Strasse 75</td>
<td>04186183</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Intel Semiconductor GmbH Mainzer Strasse 75</td>
<td>04186183</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Intel Semiconductor GmbH Mainzer Strasse 75</td>
<td>04186183</td>
<td></td>
</tr>
<tr>
<td><strong>ITALY</strong></td>
<td>Intel Corporation Italia, S.p.A.</td>
<td>2/34 93287</td>
<td>311271</td>
</tr>
<tr>
<td></td>
<td>Corso Sempione 39</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SWEDEN</strong></td>
<td>Intel Sweden A.B.*</td>
<td>(08) 98 53 90</td>
<td>12261</td>
</tr>
<tr>
<td></td>
<td>Box 20092</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Alpvagen 17</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NETHERLANDS</strong></td>
<td>Intel Semiconductor B.V.</td>
<td>781-28426</td>
<td></td>
</tr>
<tr>
<td><strong>NORWAY</strong></td>
<td>Intel Norway A/S</td>
<td>149122</td>
<td>22283</td>
</tr>
<tr>
<td><strong>UNITED KINGDOM</strong></td>
<td>Intel Corporation (U.K.) ltd.*</td>
<td>2/34 93287</td>
<td>311271</td>
</tr>
<tr>
<td></td>
<td>5 Hospital Street</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ISRAEL</strong></td>
<td>Intel Semiconductor Ltd.*</td>
<td>742 420</td>
<td>18018</td>
</tr>
<tr>
<td></td>
<td>P.O. Box 2404</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FRANCE</strong></td>
<td>Intel Corporation, S.A.R.L.*</td>
<td>62 65 60</td>
<td>36620</td>
</tr>
<tr>
<td><strong>FINLAND</strong></td>
<td>Intel Finland OY</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SWITZERLAND</strong></td>
<td>Intel Semiconductor A.G.</td>
<td>149122</td>
<td>22283</td>
</tr>
<tr>
<td><strong>SWEDEN</strong></td>
<td>Intel Sweden A.B.*</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BRAZIL</strong></td>
<td>Intel Semiconductor Brazil</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NORWAY</strong></td>
<td>Intel Norway A/S</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>UNITED KINGDOM</strong></td>
<td>Intel Corporation (U.K.) ltd.*</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SWITZERLAND</strong></td>
<td>Intel Semiconductor A.G.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Field Application Location