HANDBOOK OF

Software Engineering

Edited by

C. R. Vick, Ph.D.
C. V. Ramamoorthy, Ph.D.

Van Nostrand Reinhold Electrical/Computer Science and Engineering Series

HANDBOOK OF SOFTWARE ENGINEERING

Edited by

C. R. Vick, Ph.D.
Department of Electrical Engineering
Auburn University

C. V. Ramamoorthy, Ph.D.
Department of Electrical Engineering
and Computer Science
University of California at Berkeley

Van Nostrand Reinhold Electrical/Computer Science and Engineering Series

VAN NOSTRAND REINHOLD COMPANY
NEW YORK CINCINNATI TORONTO LONDON MELBOURNE

27

Software Development for Micro/Mini Machines

Robert Glaser
Telesaver

27.1. INTRODUCTION

Microprocessors and minicomputers continue to become more powerful every year. As a result, each has found growing application in areas where larger computers had previously been used and new uses that had earlier been impractical. With rapidly changing technology, the classifying distinctions between microcomputers, minicomputers, and conventional computers have become less clear. Because minicomputers fall between microprocessors and large computers, emphasis in this chapter is placed upon microprocessors. Generalizations about microcomputers apply to low-end minicomputers, whereas high-end minicomputers bear a closer relationship with large mainframe computers.

A multitude of microprocessor chips are available, each of which has its own peculiarities. This chapter, however, will focus on the features the various chips have in common.

A major difference between software for micros and larger computers is that the micro software is highly dependent upon the microcomputer hardware. Micro software is nearly always concerned with low-level input/output tasks. Communication with peripheral chips is of major concern. When dealing with microprocessors, it is not possible to effectively discuss software independent of hardware, and thus this chapter includes microprocessor-hardware-oriented material.

A microprocessor, as discussed in this chapter, is a central processing unit (CPU) contained within a single integrated circuit package. The CPU provides logical, arithmetic, and control functions. The instruction set is predefined. The control functions include instruction decoding, and memory address, data, read/write, and selection signals. This type of microprocessor is different from a microprogrammable processor, which is also sometimes called a microprocessor. This device requires a number of integrated circuits to implement the central processing section alone. User-written microcode defines the instruction set of this type of processor system. These microprogrammable processors find their chief use in specialized applications requiring high speed.

A microcomputer consists of a microprocessor CPU, program memory, data memory, input/output (I/O) devices, and supplementary control logic. This may require several chips on a single circuit board, or several boards. The program memory is often nonvolatile read-only memory (ROM). Data memory is read/write random-access memory (RAM), and I/O can be through a variety of peripheral chips.

Minicomputers generally have higher speeds and larger word sizes than microcomputers. They contain the same elements as microcomputers, though implemented from a larger number of integrated circuits. At one time minicomputers consisted largely of TTL logic gates, though now many are constructed with fast microprogrammable processor chip sets. The circuitry typically requires several circuit boards.

27.2. APPLICATIONS OF MICRO/MINICOMPUTERS

The applications of micro/minicomputers fall into two distinct categories: equipment con-

trollers and general purpose computers. The software for equipment controllers is somewhat different from that for large computers; hence, equipment controller applications are stressed here. The more general purpose uses of micro- and minicomputers, which can be different from those of larger computers, are included as well.

Equipment Controllers

There are several advantages of implementing controllers from microcomputers. Random-logic designs require different parts for each system. Low-quantity products do not justify the large capital outlay required to design, lay out, and manufacture specialized integrated circuits. Available parts must be used, and several circuit boards filled with logic gates are the usual result. The cost of design, layout, board manufacture, and testing must all be covered by a single controller product. Microcomputer implementation of the product permits much of the work to be useful for a variety of similar products. Off-the-shelf microprocessor parts that are mass produced provide low-cost alternatives to special-purpose integrated circuits. Several controller products can use a standardized microcomputer circuit board. The development cost of a controller is then reduced to that of the controller program. A smaller number of circuit boards is also possible because of the use of a cost-effective sophisticated microprocessor part. Product modifications are program changes requiring only a ROM change, whereas with a random-logic design, circuit board modifications and additions are necessary. The scope of possible product modification is greatly limited for random-logic designs as compared with microprocessor implementations. Through the use of microprocessor-based design, the special-purpose functions of an equipment controller are supplied by a special-purpose controlling program, and other aspects of the product can be used for a multitude of other products.

Computerizing an equipment controller imposes certain limitations on the microcomputer. The physical size, power drain, and cost of the computer must be no larger than that of the random-logic design being replaced. These restrictions become more important when new applications rather than replacement applications are considered. Intelligent products are possible that cannot be contructed without microprocessors because of limitations of size, power, and cost. The low size, power, and cost of microprocessor-based controller designs are precisely what create the new applications.

Microprocessor-based controller applications can be found in a variety of areas. Consumer-oriented devices show a large number of examples. Uses in radio receivers and home entertainment equipment proliferate and are but a sample of what can be done in consumer electronics. The Ahwatukee House is perhaps the ultimate in computerizing a home [1–3]; the possibilities for improving everyday living are immense [4]. The automotive controller [5] is a particularly high-volume application. Uses in traffic control [6] are growing. Very human-oriented applications include aid for the handicapped [7] and medical uses as common as aid in childbirth [8]. Commercial and industrial applications include controlling satellites [9], astronomical telescopes [10], and gas turbine generators [11]. Applications in the field of test equipment [12, 13] are enormous.

Many consumer, commercial, industrial, and research devices can be enhanced with intelligent controllers. The wide spectrum and sheer number of applications is limited only by the creativity of equipment designers. Other applications can be found in References 14 to 19.

General Purpose Computers

The microprocessor revolution has made it possible for micro- and minicomputers to be used in much the same ways as larger conventional computers. Standard business applications of payroll computation, inventory maintenance, general ledger, and address label sorting and printing have become feasible for small businesses utilizing micro- or minicomputers. Development systems for controller development and support are often implemented

with micro- or minicomputers. Section 27.8 deals with these systems. Small computer systems for educational uses have improved computer science educational programs. Small stand alone-systems can give students a better learning opportunity than punched card/batch submission methods. The introduction of microprocessor-based low-cost general-purpose computers spurred the entrance of the computer into the home. Home computing, which has grown to significant levels, includes games, computer-aided instruction, and text processing. As the hobby has grown, so has the number of magazines to serve it [20–24]. Personal computing equipment has proliferated to the point where some of it has been called into professional service [25].

27.3. MICROCOMPUTERS, MINICOMPUTERS, AND LARGER COMPUTERS

The classification of computer systems as micro, mini, or large can be based at best only on general characteristics. The dividing lines between each category are difficult to draw and keep changing.

Word Size

The more bits per word a processor has, the more powerful each instruction is. This is particularly noticeable in arithmetic computation. The number of operation codes is limited when dealing with small word sizes, so machines with small word sizes have a correspondingly greater number of multiple word instructions. Microcomputers generally have 4- or 8-bit words, although 16 bit microprocessors are available, and 32 bit chips are not far off. Minicomputers have between 12- and 24-bit words, with 16 bit being the most prevalent. Large computers have 32 bit words and up. Processor word size is not as reliable an indicator as it once was, although 4- or 8-bit processors would almost always be classified as microprocessors, geared to control applications and low-complexity processing problems.

Architecture

Processor architectures vary considerably within each category. Microprocessors tend to have more restrictive architectures than the others; data paths can be quite specialized, with one or more memory registers required to access general data memory. Larger word sizes facilitate easier memory access by supplying a greater number of bits for address specification. Separate program memory and data memory paths are more often encountered in microprocessors than larger machines. This is more natural for controllers because there are normally different types of memory for each function. These architectures usually require memory address pointers for data storage.

The simplest architectures provide a single accumulator for arithmetic, logical, and move operations. More sophisticated arrangements provide several accumulators, or general-purpose registers for these operations. Microprocessor chips tend to have single or double accumulators, since the small word size makes it difficult to have enough instruction code space to reserve register fields. This limitation creates data bottlenecks, which require data moves to achieve proper positioning for subsequent operations. Minicomputers tend to have a number of general-purpose registers, permitting operation code specification of register operands.

Instruction Set

The larger the word size, the greater number of instructions that can be specified in single word instructions. Consequently, in addition to the fact that larger machines have greater capabilities otherwise, microprocessors have much sparser instruction sets than minicomputers and larger computers. Internal stack space is sometimes found in microprocessors, a useful hardware feature but a software-limiting situation. Paged addressing, instead of relative or even absolute addressing, is often found in microprocessors. A bare minimum instruction set includes data moves, logical

shifts, AND, OR, exclusive OR, and addition. Some microprocessor sets include little more than this minimum, whereas others include a respectable number of additional operation codes (op-codes). Small machines use subroutines to replace missing op-codes. Codes such as multiply and divide are typically missing from microprocessors, and floating point instructions are often missing from minicomputers.

Speed

Naturally, the operation speed increases from micro- to mini- to large computer. This is partly inherent in the type of technology from which the devices are constructed, and partly because of the greater number of instructions that must be performed to provide the same throughput for smaller processors. Dealing with 32 bit integers on a 4 bit machine clearly requires over 8 times the execution time of a 32 bit machine. Accumulator and memory address register bottlenecks result in even more operations being necessary. Limited instruction sets call for greater use of subroutines. All of these restrictions produce the same result: a slower machine. Even if the instruction execution times of micro-, mini-, and large computers were equal, there would still be a speed contrast. Additionally, there usually is an inherent speed differential. Microcomputers can have execution times of several microseconds, with minicomputers slightly less, and large computers well into the submicrosecond range.

Hardware Interfaces

Microprocessors have simple interface requirements when compared with mini- and large computers. Input/output devices and general hardware can be added more easily to micros because the interfaces are less restrictive. Sophisticated handshaking and bus-driving requirements are necessary to obtain the higher operating speeds of mini- and large computers. Additional peripheral chips may be placed right on the bus on controller-size micros, where separate boards and bus extenders and terminators may be required for larger systems.

27.4. A MICROPROCESSOR SURVEY

There are scores of microprocessor chips available. A history of microprocessor development can be found in References 26 to 28. A brief description is given for some of the more popular ones.

Four Bit

These micros are intended chiefly for use in control applications, particularly when tasks are I/O intensive. Computations are often done in binary coded decimal form (BCD).

Intel's MCS-4 chip set [29], announced in 1971, is notable for opening the microprocessor era. Each member of the chip set is packaged in a 16 pin dual in-line package (DIP), made possible through the use of a multiplexed 4 line bus for address and data. This set consists of the 4004-CPU, 4001-ROM, 4002-RAM, and 4003-shift register. The system utilizes 12-bit addresses and 4-bit data words. The ROM, RAM, and shift register each provide I/O lines. The pMOS CPU supplies 46 instructions, contains a 3-level internal stack, and the minimum instruction time is 10.8 microseconds. An external clock and driver are required, and no interrupts are provided. Sixteen 4 bit general-purpose registers may also be used as eight 8 bit index registers. I/O addressing is set up by special control commands.

Eight Bit

There are more micros in the 8 bit category than any other. These are suited to character manipulation as well as I/O control and moderate arithmetic computation.

The 8008 CPU [30] was the first 8 bit CPU. Housed in an 18 pin DIP, an 8 line bus is multiplexed to supply 14-bit addresses and 8-bit data words. This pMOS-fabricated CPU supplies 48 instructions, contains a 7-level internal stack, and has a minimum instruction time of 20 microseconds. An external clock and driver

are needed, and a single interrupt input is provided. Seven 8-bit scratch-pad registers are available. The 8008's architecture, less limiting than the 4004's, opened up many applications for microprocessors.

A step up from the 8008, Intel's 8080 CPU [31] has a 2 microsecond minimum instruction time because of nMOS fabrication. The 8080 has separate address and data lines, requiring the larger 40 pin package. There are 16 address lines, permitting up to 64K memory bytes to be accessed. The register set is the same as that of the 8008, and the instruction set is an expanded set of the 8008, with 72 instructions. Several stack instructions and one register indirect branch instruction are added. The processor uses an external stack, has a single interrupt input, and requires an external clock. A hold mode is incorporated that permits direct memory access (DMA).

The Intel 8085 CPU [32] incorporates all of the features of the 8080. It has five interrupt inputs and an on-chip clock oscillator, and requires only a single power supply voltage. To retain the 40 pin package, a multiplexed address/data bus is utilized. The minimum instruction time is 1.3 microseconds. The chief advantage of the 8085 over the 8080 is a reduction in system support components.

Software-compatible with the 8080/8085, Zilog's Z-80 CPU [33] adds relative and indexed addressing modes to the base instruction set. Block transfer and search instructions are also provided, and the Z-80 executes 158 instructions. Seven alternate general-purpose registers and two index registers are included in addition to those of the 8080/8085 set. Hardware features include a single power supply requirement, dynamic memory refreshing, two interrupt inputs, and provision for DMA.

The 8080 processor series may be considered to be register oriented; instructions permit operations with the general-purpose registers. Motorola's 6800 CPU [34] has two accumulators and a 16 bit index register; the instruction set includes operations with memory locations. This CPU can be considered to be memory oriented. Accumulator, immediate, direct, extended, indexed, implied, and relative addressing modes are available. The minimum instruction time is 2 microseconds, and there are 72 instructions. Housed in a 40 pin package, the 6800's 16 bit address bus is separate from the data bus. An external clock generator is required, and only a single power supply voltage is needed. There are two interrupt inputs, and the chip is capable of DMA. No special I/O control lines or instructions are provided, mandating memory-mapped I/O.

The Motorola 6802 CPU [35] is virtually identical with the 6800 except that it has a built-in clock generator, and 128 bytes of internal RAM. The chief advantage of the 6802 is a reduction in system components.

MOS Technology's MCS6502 CPU [36] is also a memory-oriented device, with an accumulator and two 8 bit index registers. A number of addressing modes are available: accumulator, immediate, absolute, zero page, indexed zero page, indexed absolute, implied, relative, indexed indirect, indirect indexed, and absolute indirect. There are two interrupt inputs, and the only active devices required for clock generation are two inverters. A single supply voltage powers the chip, and the minimum instruction time is 2 microseconds. The 6502 has a 64K address space, but 28-pin package variants of this CPU have 4K or 8K addressing.

The RCA CDP1802 COSMAC CPU [37] operates over the wide temperature range of −55 to +125°C and can draw very little current at slow clock rates. The 1802 is powered from a single voltage, has one interrupt input, a built-in clock, and excellent DMA provision. The 40 pin COSMAC includes four input lines and one output line. There are I/O instructions and control lines. The instruction set is centered about its sixteen 16 bit registers. The COSMAC's architecture is uncommon; manipulation of the registers produces results that with most other processors are handled in a different fashion. Register, register-indirect, immediate, and stack addressing modes are available.

Intel's MCS-48 [38] series of CPUs finds utilization in physically small control applications. The 8048 contains 1K bytes of ROM program memory, 64 bytes of RAM data memory, an 8-level stack, 27 I/O lines, a pro-

grammable timer counter, an interrupt input, and a clock oscillator. There are 96 instructions and the minimum instruction time is 2.5 microseconds. The CPU operates from a single power voltage and the 40 pin package can be used with no support devices. Other members of the MCS-48 family contain twice as much RAM or ROM, erasable, programmable ROM (EPROM), no ROM, or are housed in a 28 pin DIP.

Sixteen Bit

Microprocessors with large word size can challenge the power of minicomputers. These processors are suited for high-throughput control systems, and relatively complex computations.

Texas Instruments' TMS9900 CPU [39] features a memory-to-memory architecture. A workspace pointer points to 16 workspace registers in memory. The register file can be changed by altering the workspace pointer. The 15-line address bus accommodates 32K words (64K bytes). A single interrupt input, operating in conjunction with 4 interrupt code inputs, provides 16 vectored interrupts with the addition of an external priority encoder. Housed in a 64 pin package, the 9900 requires 3 power supply voltages and an external 4-phase clock. The instruction set includes multiply and divide, and 8 addressing modes. The minimum instruction time is 2.7 microseconds.

Intel's 8086 CPU [40] can address 1M byte of memory with a 20 bit address bus, and has fourteen 16 bit registers. Housed in a 40 pin package, a single power supply voltage is required. There are two interrupt inputs and an external clock generator is needed. The instruction set includes multiply and divide, and has 24 addressing modes. The minimum instruction time is 400 nanoseconds.

Zilog's Z-8000 CPU [33] can address 8 megabytes of memory, and has sixteen 16-bit general-purpose registers. The Z-8000 operates from a single power supply voltage and is housed in a 48 pin package. The instruction set includes multiply and divide, and 8 addressing modes are available. The minimum instruction time is 750 nanoseconds.

Motorola's 68000 CPU [41] can address 16 megabytes and contains 17 32-bit registers. The instruction set includes multiply and divide, and has available 14 addressing modes. Data types associated with the 68000 are bits, BCD digits, bytes, words, and long words. Because of the 32 bit word option, this microprocessor can be considered a 32 bit machine with 16 bit memory access.

27.5. HARDWARE FEATURES

The microprocessor hardware knowledge that is most frequently required deals with interfacing: how to interface a CPU with memory and peripheral devices; how to interface support chips with the external environment; and how to interface several processors together to produce a larger system. The hardware interface method chosen determines the type of software driver that is necessary. Interchip interfacing problems are alleviated by manufacturer's chip sets. Understanding the processor buses is mandatory for peripheral connections, and various communication methods may be utilized between devices.

Chip Sets

Most manufacturers produce support chips for their microprocessors [42]. These include clock generators, ROM, RAM, bus drivers, and I/O. This permits an entire microcomputer system to be constructed from components that are guaranteed compatible. This can sometimes offer the advantage of multiple functions in support chips that are matched to the CPU's needs.

An Intel MCS-80 family [31] processor system could consist of the following: 8080, 8224, 8228, 8205, 8255, 8708, and two 8111s. The 8224 generates the clock and produces a reset signal. A data bus buffer, control signal demultiplexer, and single interrupt handler are combined in the 8228 system controller. The 8205 address decoder supplies chip selects. The 8708 gives 1K bytes of program memory, and the 8111s together provide 256 bytes of data memory. Three ports of I/O are handled by the 8255. This set of chips comprises a complete microcomputer package.

A similar Motorola 6800 system [34] would consist of the following: 6800 CPU, 6870 clock, 6830 ROM, 6810 RAM, and 6820 peripheral interface adapter. This combination gives 1K bytes of program memory, 128 bytes of data storage, and 20 I/O lines.

Combination support chips can reduce the component count greatly. An Intel MCS-85 system [32] could consist entirely of three chips: 8085, 8755, and 8155. The 8085 CPU contains the clock, the 8755 gives 2K bytes of EPROM program storage and 16 bits of I/O, and the 8155 supplies 256 bytes of RAM data memory, 22 bits of I/O, and a programmable timer/counter. The 8755 and 8155 support chips contain the demultiplexing circuitry required to interface with the 8085 bus. Because of the importance of matching chip sets, the choice of a microprocessor often is more dependent upon the availability of suitable support components for a particular application than characteristics of the CPU itself.

Processor Buses

Most microprocessors share bus characteristics. Figure 27.1 shows a typical CPU. The address bus is a set of output lines that is used to select a single word of memory or I/O. To permit DMA operations, this bus may have a high-impedance state to allow another device to generate a system address. The data bus is bidirectional, and the width sets the processor word size. This bus inputs data returning from memory or I/O during a read cycle, and outputs data going to memory or I/O during a write cycle. Data bus direction and access timing are set by the control bus. The control bus consists of the signals necessary to access memory and I/O, to distinguish memory from I/O, and other controls that may be processor dependent.

In an MCS-80 system, the 8228 provides four control signals: memory read, memory write, I/O read, and I/O write. With this set of controls, only one is active at any time. The appropriate line is gated with an address decoder output to activate the desired component. A ROM should only be activated during a memory read cycle, so that signal need only be used for ROM. RAM chip selects must be gated with the memory-read and memory-write lines to allow both operations. I/O device selects are gated with one or both of the I/O control signals. Devices gated with I/O control signals will only be activated during I/O commands; for the 8080 these are but two instructions: IN and OUT. Alternatively, I/O devices may be gated with the memory control signals. Naturally, addressing is required to distinguish these devices from memory locations. This is called memory-mapped I/O; I/O devices are treated the same as memory, and therefore can be addressed through any memory-accessing instruction. Processors that have separate I/O channels supply the option of using the separate I/O address space or using a portion of the memory address space via memory-mapped I/O. For those processors that have no special I/O address space, memory-mapped I/O must be used.

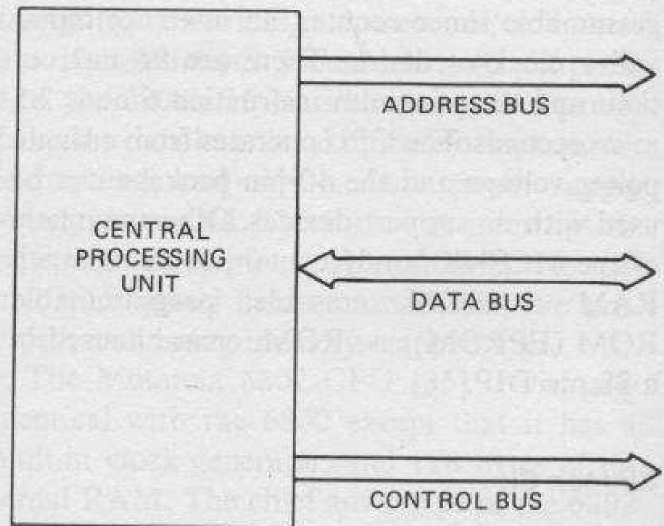

Figure 27.1. Processor buses.

On the 6800, there are three control lines: R/W, VMA, and Ø2. The R/W line distinguishes between read and write operations. A valid memory address is on the address bus only when the VMA line is active. In addition, the clock phase two is reserved for memory accesses. Chip selects should be gated with all three signals and an address decoder. There are no I/O signals or instructions on the 6800, so memory mapped I/O is used exclusively.

The control bus emanating from the CPU may be converted into a different set of control signals used throughout the microcomputer;

the three 6800 signals can be converted into two control signals (memory read and memory write) if desired. Which set of control signals is preferable in any system is wholly dependent upon the accessing organization of the peripheral chips.

I/O Peripherals

Peripheral chips range from the very simple to the quite complex. The simplest output port is just a latch; the latch inputs go to the data bus, the gate is the chip select, and the outputs comprise the peripheral lines. The simplest input port is a tristate buffer; the processor reads the peripheral lines when the control signals activate the buffer. Several of these I/O ports can be combined into a single chip with additional steering logic, and are called peripheral interface adapters (PIA), or programmable peripheral interfaces (PPI). (For example, the 6820 and the 8255.) More sophisticated logic functions realized on a single chip can be used as peripheral devices, such as a universal asynchronous receiver/transmitter (UART). Most processor families contain at a minimum PIA and UART chips.

Complex peripheral functions are implemented with single chip microprocessors. Intel's UPI-41 [43, 44] is a universal peripheral interface chip that is a complete processor with controls configured as a slave device. The UPI-41 can be used to preprocess information to conform with any number of external devices, removing this burden from the main processor and permitting more time to be allocated to higher-level processing. There are other specialized peripheral chips that are actually single chip processors programmed by the manufacturer for a specific function; this is transparent to the user.

The most sophisticated peripherals can be coprocessors. These devices monitor CPU operation and perform functions as needed without specifically being requested by the main CPU; the effect is to extend the instruction set of the CPU. Intel's 8087 [45–47] floating-point processor and 8089 [40] I/O processor are in this category.

Communication Methods

The CPU must know when a peripheral is ready to receive data, and then have a method to send the data. Alternatively, when a device needs servicing a method must be provided for it to inform the CPU. Communication between peripheral chips and the CPU is usually through registers in the peripheral. A control register permits the CPU to configure the peripheral as required. A status register is read by the CPU to determine current device conditions. Data registers are used to transfer the actual information.

In the simplest control environment, the CPU constantly polls the peripherals to determine if any require servicing. Polling stops to permit peripheral handling when servicing is required, and continues when the request is met. A disadvantage of this polling method is that the CPU can spend a large portion of its processing time just checking service request bits of each device. During the servicing of a device, it may be desirable for higher-priority peripherals to have the ability to request servicing. Polling during device servicing can become complicated when a number of peripherals exist. There also can be a high or unpredictable latency period between when the request occurs and when the servicing is granted. These problems can often be eliminated by using interrupt-driven peripherals.

With an interrupt-driven system [48, 49], in addition to the device register communications, a peripheral output line is fed to a CPU interrupt input. When the device requires servicing, the processor is interrupted, and the service routine can communicate through the peripheral registers. Through proper handling of interrupt enables, masks, and priorities, all devices can be serviced optimally with interrupts.

Some peripheral devices require data rates that are too high to be handled through CPU/data register communications. An example is that of a floppy disk controller peripheral. These peripherals may achieve the necessary data rate through direct memory access (DMA). The peripheral activates an output line when data are either available or required.

This output feeds an input on the CPU which requests a DMA cycle. When the current instruction is completed, the CPU grants the DMA cycle and notifies the peripheral by activating a DMA-granted output line. This tells the peripheral that the CPU has freed the buses, and the peripheral can seize the buses and communicate with memory directly, instead of going through the processor. When the data transfer is complete, the peripheral lowers the DMA request line, and the CPU regains control of the buses. This method permits data rates as high as the memory bandwidth is capable. DMA may be handled completely by the peripheral device, or there may be a special DMA controller peripheral chip that provides the addresses and control signals for the data-requesting peripheral.

27.6. LANGUAGE FEATURES

Microprocessor instruction sets vary between processors, and machine language is greatly affected by architecture. The operations available are generally similar for many CPUs. Special architecture-related instructions are usually provided. A microprocessor language can be best understood through knowledge of the machine's architecture, the various addressing modes, how flags are affected and tests are made, and familiarity with the available instructions. The 8080 and 6800 are used in this section for illustration.

Addressing Modes

Machine operations must specify an operand or operands. The allowable addressing modes designate the operands that may be selected.

Register addressing is used when operands are general-purpose registers. The 8080 instruction "INR r" increments the contents of register r, where r is any of the seven 8080 registers. With other processors, register addressing is called accumulator addressing. The 6800 instructions "INC A" and "INC B" increment accumulator A and B, respectively. The register- (or accumulator-) addressing mode tends to be the fastest because the operands are internal to the CPU, and additional memory accesses are not required beyond the instruction fetch. Register addressing produces compact code because these are single byte instructions.

The immediate addressing mode is used when an operand is constant data. The 8080 instruction "ADI data" adds the data in the memory location following the op-code to the accumulator and stores the result in the accumulator. The 6800 instruction "ADD A,#data" does likewise with accumulator A. Immediate addressing can refer to single- or double-byte data, resulting in either 2- or 3-byte instructions.

Direct, absolute, or extended addressing specifies an exact memory location for use as an operand. The 8080 instruction "LDA addr" is a 3 byte instruction where the address of the memory location containing the data to be loaded into the accumulator is given. This is similar to the 6800 instruction "LDA A addr" when extended addressing is used—this 3 byte instruction performs the same action with accumulator A as the 8080 instruction does. A variant of extended addressing on the 6800 is direct addressing. This mode implies operands located at addresses within the first page of memory (addresses 0 through 255). Only a single byte is needed to specify the operand; hence these instructions are 2 bytes long.

Register indirect addressing specifies a register pair that contains the memory address where the data are located. The 8080 instruction "LDAX rp", where rp is a register pair, loads the accumulator from the memory location whose address is the contents of the specified 16 bit register. Indexed addressing takes this method a step further. An index register specifies a base address to which a single byte offset given in the second byte of the instruction is added, supplying the address of the data. The 6800 instruction "LDA A offset,X" takes the 8 bit quantity offset, adds it to the 16 bit index register, and uses the resultant 16 bit address to locate the data to be placed into accumulator A. Register indirect and indexed addressing are powerful addressing modes because the data address can be changed dynamically during program execution. These addressing modes are essential for efficient table lookup routines.

There are instructions for which an addressing mode need not be given, either because one is implied by the instruction, or because an addressing mode is not applicable. An example of the former is the 6800 instruction "ABA" which adds accumulators A and B and stores the result in A; an example of the latter is the 8080 instruction "STC" which sets the carry flag. These instructions are referred to as having either implied or inherent addressing.

Relative addressing references data or addresses to the program counter. The 6800 branch instructions use the second byte of the instruction as a signed 8 bit offset to the address, which the program counter would contain were a branch not encountered. There are two advantages to the relative addressing mode: 2-byte instructions result instead of 3-byte instructions, and the reference to the program counter produces machine code blocks that are not dependent upon memory location.

Flags and Tests

Flags are single bit flip-flops that are set and reset by processor operations and can be subsequently tested for the purpose of conditional branching. Flag bits are collected into a register called the processor status word (PSW) or the condition code register (CCR). Most of these flags are defined by Boolean arithmetic: overflow, carry/borrow, sign bit, auxillary or half carry, zero, or parity. Arithmetic and logical operations affect these flags in the usual sense.

The operations that do and do not set flags differ greatly from processor to processor. This can easily confuse the programmer familiar with one processor who is learning to use another. The instructions can perform the same operations on different processors yet affect flags quite dissimilarly. On the 8080, "MOV" operations affect no flags; on the 6800, the equivalent "LDA" instructions clear the overflow flag, and set or reset the sign and zero flags in accordance with the data. When flags have not been affected, and it is desired to make a test concerned with data from a previous operation, an additional flag-setting operation is required. The 8080 input instruction affects no flags. To use the zero or sign bit flags after inputting data with the "IN port" instruction, a flag-setting operation such as "ORA A" must follow the input instruction. Conversely, sometimes instructions will affect flags when it is not desired, and in these cases it is necessary to save them prior to the operation for later retrieval.

Flags can be used for simple parameter passing between routines. For example, a subroutine can be defined that exits with the carry set if an invalid condition is encountered, and exits with the carry clear otherwise. Software that called the routine can then easily branch to handle the different cases by performing a simple carry flag test upon return from the subroutine. Care must be taken to ensure that flags are not inadvertently modified in such instances.

Included in processor instruction sets are special flag-testing operations. The 6800 bit test instructions permit testing of particular bits without modifying data. Compare instructions are essentially subtract operations that modify only flags, not data. Conditional branch, jump, and return instructions test the flags in various combinations.

Other flags may be included in the CCR. The 6800 has an interrupt status flag, which signifies whether the interrupt mask is set or reset. The 8048 has two general purpose flags that can be used as the programmer desires. Both flags can be cleared and complemented, and tested by conditional jump instructions. The 8048 also has eight conditional jump instructions that test the individual bits of the accumulator; these are not flag bits but can be tested just as easily as flags.

Flags and tests permit conditional branching, which is the greatest source of program complexity. Great care must be taken so that flags are modified correctly to ensure proper branching.

Instructions

The specific instructions with which processors are equipped varies, but a certain number are shared by most, and fit into categories.

Data transfer instructions include move,

load, store, and exchange operations. A large part of controller software consists of data transfer, and this instruction block is heavily utilized.

Arithmetic instructions are add, subtract, increment, decrement, and decimal adjust. Logical instructions consist of AND, OR, exclusive OR, rotate and shift, clear, complement, negate, compare, and test instructions. Branch instructions are jump, conditional jump, subroutine call, conditional subroutine call, return from subroutine, return from interrupt, conditional return from subroutine, software interrupt, and wait for interrupt instructions.

Stack instructions are push, and pop or pull operations on registers. Machine control instructions include interrupt handling, halt, no operation, and I/O instructions.

The instruction set differences between processors will cause different software techniques to be employed for different processors. Registers or memory for parameter storage, stack or reserved memory for temporary data save, and methods for parameter passing [50] between routines are choices that are processor dependent.

27.7. CONTROL PROGRAMS

Controllers require programs that are typically several thousand bytes long. Sufficient hardware is provided for the controller to satisfy the needs of the specific application. The resulting small size serves to keep the cost down. The size and execution speed of controller software have a direct effect on the amount of hardware required for a particular function. Control programs should not be viewed in the same fashion as large system software. This section discusses choice of language, software interfaces with peripherals, and some programming techniques and concludes with a controller example.

Language

Control programs are often written in assembly language, partly because of the fact that in the early microprocessor years flexible software packages were not available. This left the engineer with little alternative than to write an entire control program in assembly language. There are a number of advantages of assembly code for control applications.

Much of the software is at a very low level, consisting of software interfaces with peripherals. This is easily handled with assembly language, and little advantage can be found in other languages for this type of software—indeed, higher-level languages are often unwieldy when dealing with very low-level tasks. Assembly language lets the programmer utilize the full power of a processor's instruction set, with no other limitations imposed. Assembly code can be shorter and execute faster than higher-level languages, a particularly important feature for microprocessor controllers because shorter programs require less ROM, and the relatively slow microprocessor can be used in some applications only by optimizing software for speed. Control programs are short enough so that it is reasonable to code with assembly language, whereas for much larger programs it is generally recognized that a higher-level language would be preferable. For these reasons, assembly language has maintained a strong hold on controller software.

However, assembly language has its disadvantages. There are a large number of instructions required to perform a complicated task. If an equivalent program can be written with fewer instructions in a higher-level language, the software cost will be reduced. Following program flow in assembly coding can be difficult. Higher-level languages can be understood more easily. Although effort has been made to standardize assembly code [51, 52], more often than not a new assembly language must be learned for every CPU that is used. Structured programming can be enforced or encouraged with a high-level language.

High-level languages that permit low-level functions and assembly-level subroutine calls can remove the disadvantages of completely assembly-coded programs, while retaining much of the advantages of assembly coding. Some increase in program size must be granted, and time-critical routines can still be optimized in assembly code [53]. Software

portability can be obtained through languages that exist for several processors. Even interpretive languages may be suitable for some tasks [54]. Compiled languages can execute at respectable speeds and do not require a large run-time software package for execution. A number of high-level languages are available for microprocessors [55–64].

Some applications can be met through the development of a specific pseudocode. A set of subroutines may need to be called in varying order, and a list of addresses that point to subroutines can be formed into a pseudocode program. A small interpreter can be used to call the appropriate routines from the pseudocode program. Pseudointerpreters can range from the very simple to complete high-level, user-defined languages [65–68].

For complex control applications, control software can be simplified by utilizing a real-time operating system (RTOS) [69]. This permits multitasking software to be written for an application without the programmer becoming involved with the complexity of the multitasking software—only the application-dependent software. A prepackaged RTOS can be used to advantage in these instances.

Software Interface with Peripherals

A representative peripheral device communicates with the CPU through several peripheral registers. These registers are part of the peripheral chip and are accessed through proper manipulation of the address and control lines. Data are transferred through a data register. The CPU can determine the state of the peripheral by reading its status register. Status register bits are reserved to indicate that the device is ready to receive data, or that the device has received external data and that data are available to be read by the CPU. Through interrogation of the status register, the processor can send or receive information through the data register in a controlled fashion.

Many peripheral devices can be configured in different ways. The CPU can configure the peripheral by writing to its control register. This permits a single hardware device to serve different functions under software control. Complex peripherals may have additional parameter registers for storage of control parameters.

A processor can service several peripherals by polling their status registers and handling the devices as required. Interrupt-driven devices need the appropriate handling software. A peripheral requiring service raises its interrupt request line. If the processor has interrupts enabled and unmasked, the CPU executes an interrupt acknowledge cycle, and control is transferred to the interrupt service routine. If more than one device can cause the same interrupt, the relevant status registers are read to ascertain which is the interrupting device. The necessary servicing software is then executed, followed by a return from the interrupt routine.

When the interrupt is acknowledged, the processor interrupts are usually disabled, which prevents multiple interrupts from level sensitive interrupt inputs. If interrupts are to be permitted during the execution of the interrupt service routine, then the processor interrupts should be reenabled or unmasked near the beginning of the service routine. Otherwise, steps should be taken to ensure that interrupts are reactivated after the completion of the service routine. Some processors will automatically do this via a return from interrupt instruction. Others require interrupt-enabling instructions before the return instruction. If some means is not taken to restore the interrupt status after the completion of the interrupt-handling software, no further interrupts will be possible.

A controller may have several interrupt-driven peripherals utilizing different interrupt inputs. Sophisticated interrupt-handling software can be devised by selectively masking the interrupts. One device may have priority over the others; its interrupt service routine should not reactivate interrupts until after completion of its task. The other interrupt service routines, upon entry, should reactivate interrupts after masking all but the one priority device. Proper manipulation of interrupt enables and masks can produce any desired relationship between peripherals and the CPU. Minicomputers and some 16 bit micros support prioritized vectored

interrupts. This hardware function reduces software interrupt controls to modification of priority registers.

Programming Techniques

Control processors lack the high-powered instructions and high speed of large computers. This weakness can often be combatted with the extensive use of lookup tables. Values that can be determined by complex calculation are instead placed into a table in ROM. The advantage is that the complex calculation, which is difficult to implement in the restricted instruction set and requires a long execution time, is avoided. A properly indexed table lookup routine can execute rapidly. An example of a lookup table replacing a conventional gravity calculation is found in Reference 70. An additional advantage is that should the control equation require substitution with another in a program revision, instead of requiring a new complex calculation routine, table value changes are simply needed. Table data can be trimmed to match true specifications versus approximation by a mathematical formula. The disadvantage of this type of lookup table is that much ROM space can be occupied, particularly when small step sizes are utilized. Linear interpolation and clever methods for finding values through multiple tables can reduce table lengths to manageable sizes.

The proper use of interrupts can simplify software beyond the needs of peripheral interfacing. The segmentation of software between several interrupt routines and the main program serves to make the software structure more comprehensible. Placement of interrupt-driven peripheral data into a queue permits other routines to directly work with the data queue instead of the I/O details. A real-time clock interrupt may be desirable as a programming tool even if a clock function is not required in the controller. Multiple functions can be handled by a single controller almost as simply as dedicating a separate processor to each function if clock interrupts are used to allocate time to each task. The alternative is to write one integrated control program that performs each function; this is certainly more complex than writing separate, individual control programs and allocating CPU time to each through a clock interrupt routine.

Care must be taken to check the execution times of interrupt routines to ensure adequate time before another interrupt service is required. Perishable peripheral data must be handled before being overwritten and lost. It should be seen that the use of interrupts falls into two distinct categories: efficient peripheral communication and desirable software segmentation.

Special programming techniques are usually not required for DMA processing since DMA is inherently a hardware procedure. This can be limited to programming a special purpose DMA-handling peripheral chip. An example of this can be found in [71].

Control programs can have need for standard computations as do larger machines, but these must be written with the imposed restrictions of the less powerful CPU. Examples of pseudorandom sequence generation [72–74], multiply routines [75, 76], and fast Fourier transforms [77] can be found. Programming techniques can be found in References 78, 79.

A Controller Example

The software structure of a microprocessor based CRT terminal will be described. Before writing the control software, the programmer must first have knowledge of the hardware structure: which chips perform peripheral functions, and how communication is established between the CPU and peripheral devices. The terminal accepts serial data and displays the ASCII character on a CRT monitor connected to the video output of the terminal. Characters entered on a keyboard are translated into ASCII and transmitted over the serial line. Keyboard characters may optionally be echoed on the display. Activating a *mode key* temporarily replaces the displayed screen with a menu of available terminal options, such as baud rate, parity modes, tabs, scrolling, and cursor type. Keyboard entries change the options and modify the menu accordingly. A second depression of the mode key restores the original display screen and returns operation to that of a standard CRT terminal.

Figure 27.2 shows a block diagram of the

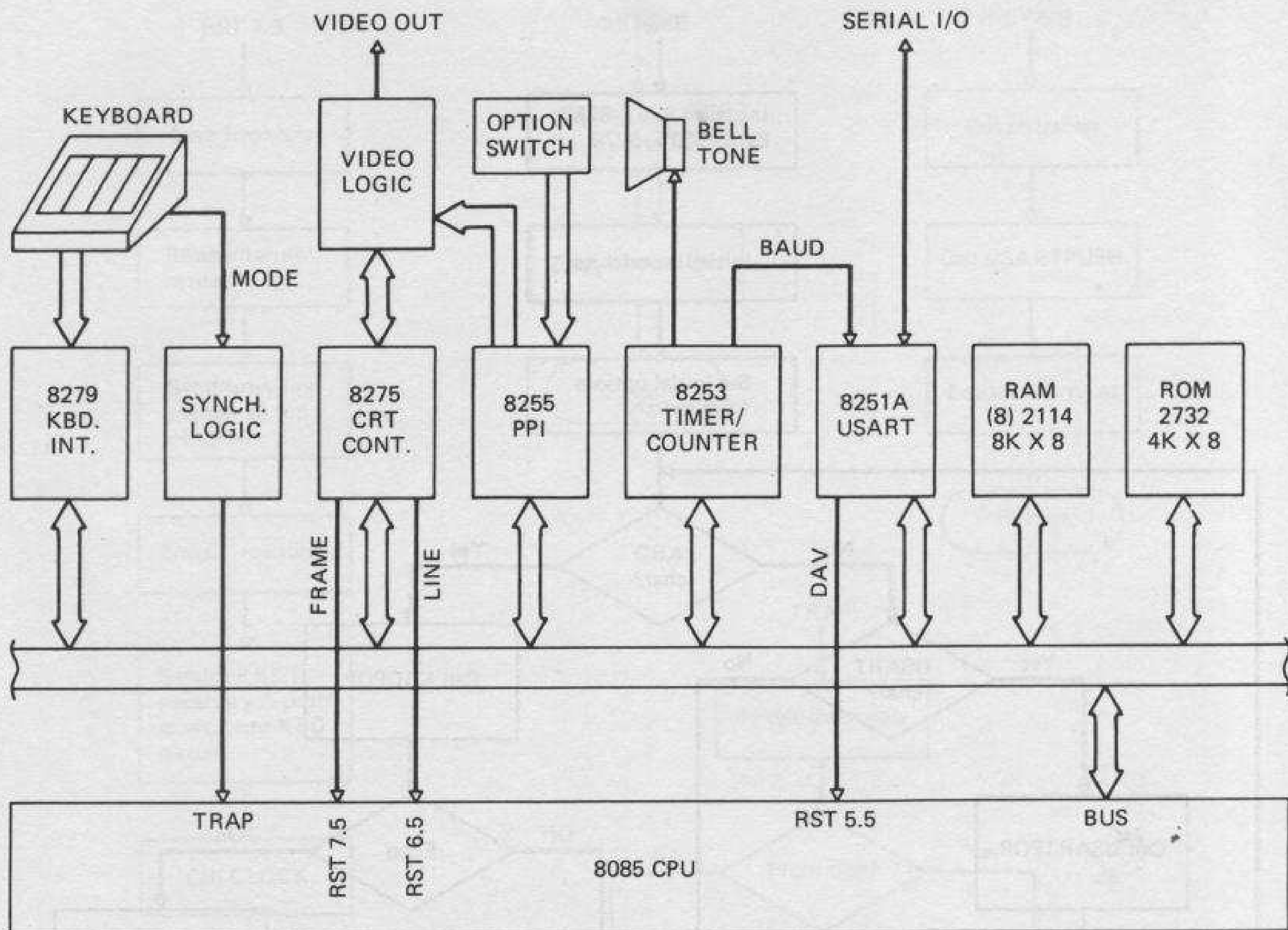

Figure 27.2. Sample CRT terminal controller.

CRT terminal. A similar system can be found in References 71 and 80. An 8085 CPU with 8K of RAM and 4K of ROM forms the basis of the controller. Most of the RAM is used for screen memory and is large enough to permit off-page scrolling. The control program and its associated tables are contained in the ROM. Serial interface is handled by an 8251A USART (universal synchronous asynchronous receiver/transmitter). An 8253 timer/counter is used as a programmable divider operating from the CPU master clock to provide two signals: the baud rate clock for the USART, giving software-controllable baud rates, and an audio output tone to produce a bell signal. An 8275 CRT controller operates in conjunction with video circuitry to generate the video output. Power-on options are set by an internal switch and read by an input port of an 8255 PPI, which also has output ports to control the video circuitry. The keyboard is scanned by an 8279 keyboard interface device.

The controller operates with four interrupts. The mode switch signal is passed through synchronizing logic to the TRAP input. This is a nonmaskable interrupt input. When a character is received on the serial line, the USART raises the data available (DAV) output, requesting an RST5.5 interrupt.

The remaining two interrupts are used for communication with the CRT controller chip. The 8275 is designed to obtain display characters through DMA. One line of characters is loaded at a time, and video timing requirements are such that 80 characters must be loaded in less than 600 microseconds. This high-speed data transfer is best accomplished with DMA logic, a procedure that is shown in Reference 71 and that requires a DMA controller chip. A different method, using interrupts instead of DMA, is found in Reference 80. Hardware transforms the 8275 DMA control signals into signals that permit data transfer via a sequence of memory reads, initiated by an interrupt. The end of a display line interrupts the RST6.5 line. This interrupt service routine must send the appropriate characters to the 8275, which then converts them

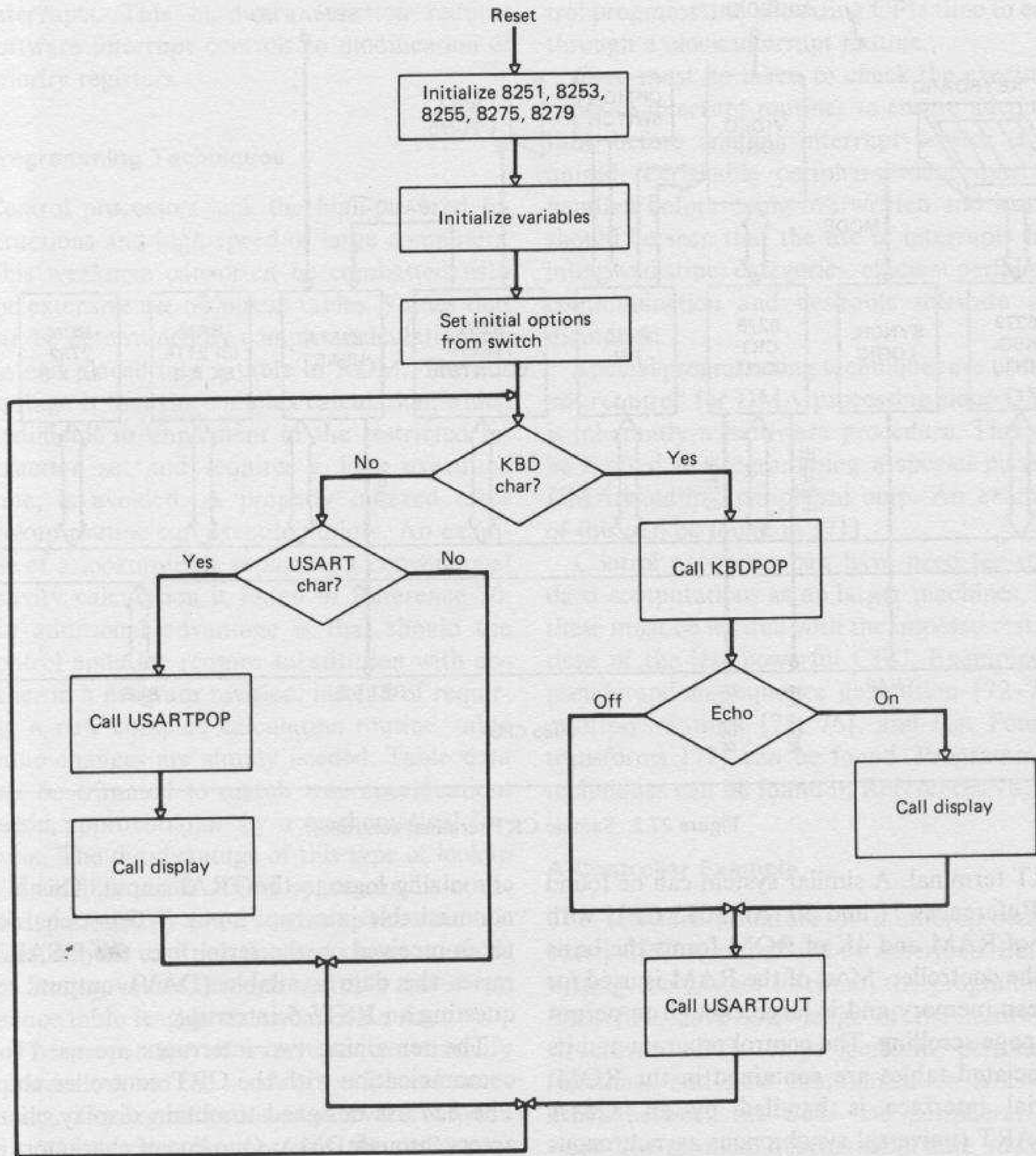

Figure 27.3. Main program.

into video signals with the external logic. When the display page is complete, the 8275 signals the CPU through the RST7.5 interrupt that the frame is complete. The CPU must then reset the display pointer to the top display line in preparation for the next RST6.5 interrupt. Video conventions cause the RST7.5 interrupt to be activated 60 times per second. This accurate interrupt is also used as a real-time clock interrupt.

Figure 27.3 shows the main program, and Figure 27.4 shows the interrupt routines. Upon power-up, variables are initialized. The peripheral devices are then initialized through their control registers. This device initialization must set the operating modes and the starting data. Figure 27.5a gives the software necessary to initialize the PPI.

The RST7.5 interrupt routine sets the display line pointer (used in the RST6.5 routine) to the top display line of the screen. The keyboard peripheral is checked for key depres-

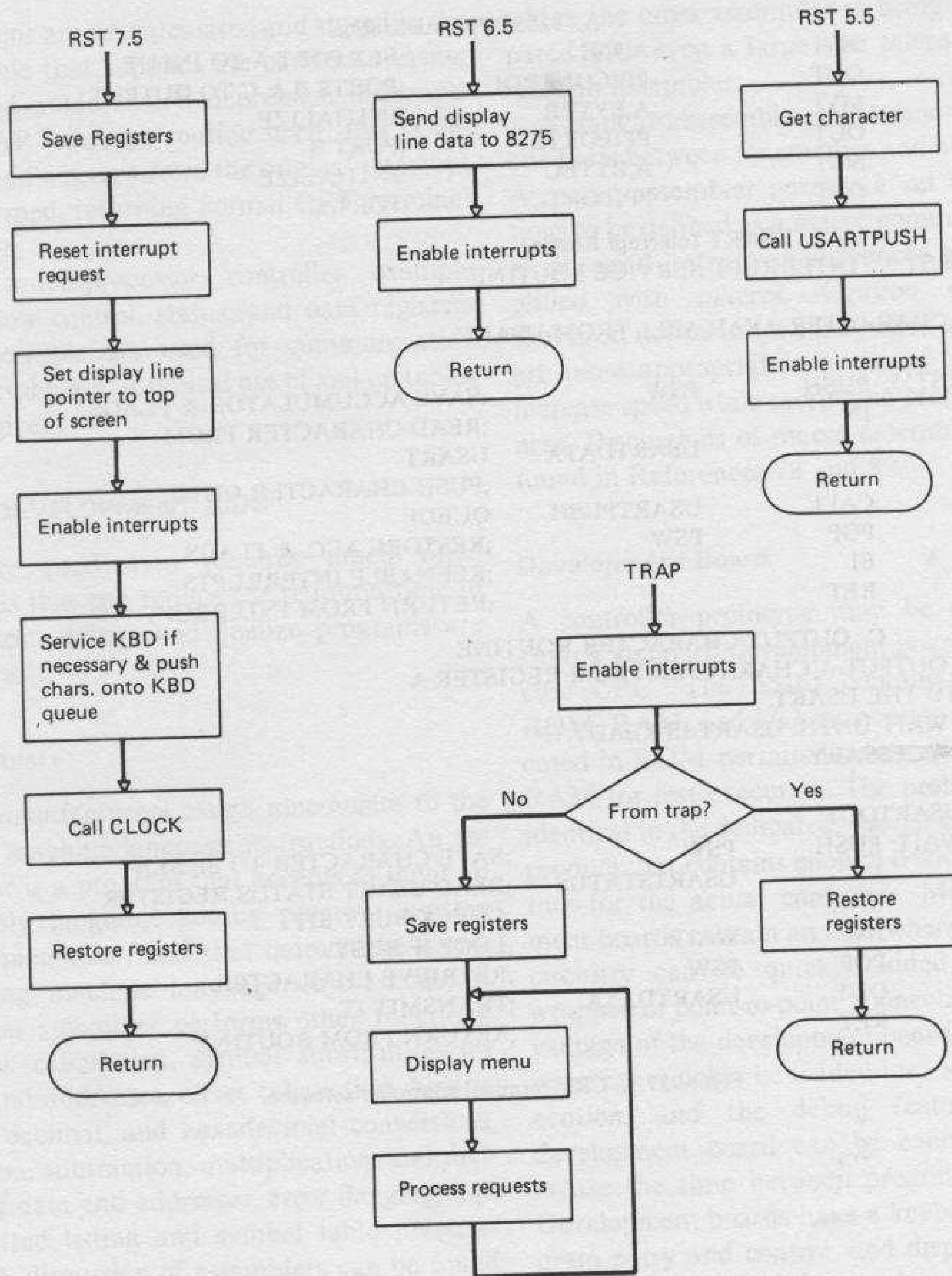

Figure 27.4. Interrupt routines.

sions; if any are found, the character is translated into ASCII and pushed into a keyboard queue. A real-time clock updating routine is called, and the interrupt routine exits. Interrupts are reenabled during the servicing of the RST7.5, permitting the much faster RST6.5 interrupts to occur during the clock and keyboard processing.

The RST6.5 interrupt routine sends the display line of data specified by the display pointer to the 8275. Because this occurs at approximately 600 microsecond intervals, this consumes a large portion of the available CPU time, and this interrupt service routine must be trimmed for rapid execution.

The RST5.5 interrupt is straightforward—the status register need not be checked since a character must be available (it caused the interrupt). The character, therefore, is read from the USART data register and pushed into a received character queue. This software is shown in Figure 27.5b.

A. 8255 PPI Initialization

```
        MVI     A,90H           ;SET PORT A TO INPUT,
        OUT     PPICONTROL      ;PORTS B & C TO OUTPUT
        MVI     A,BYTEB         ;INITIALIZE
        OUT     PPIPORTB        ;PORT B
        MVI     A,BYTEC         ;INITIALIZE
        OUT     PPIPORTC        ;PORT C
```

B. USART Interrupt Routine

```
;RST 5.5 INTERRUPT SERVICE ROUTINE
;
; CHARACTER AVAILABLE FROM USART
;
RST55:  PUSH    PSW             ;SAVE ACCUMULATOR & FLAGS
        IN                      ;READ CHARACTER FROM
                USARTDATA       USART
                                ;PUSH CHARACTER ONTO
        CALL    USARTPUSH       QUEUE
        POP     PSW             ;RESTORE ACC. & FLAGS
        EI                      ;REENABLE INTERRUPTS
        RET                     ;RETURN FROM INTERRUPT
```

C. OUTPUT CHARACTER ROUTINE

```
; OUTPUT A CHARACTER FROM REGISTER A
TO THE USART.
; WAIT UNTIL USART IS READY IF
NECESSARY.
;
USARTOUT:
WAIT:   PUSH    PSW             ;SAVE CHARACTER TO BE SENT
        IN      USARTSTATUS     ;READ USART STATUS REGISTER
        RAR                     ;CHECK BUSY BIT
        JNC     WAIT            ;LOOP IF BUSY
        POP     PSW             ;RETRIEVE CHARACTER
        OUT     USARTDATA       ;TRANSMIT IT
        RET                     ;RETURN FROM ROUTINE
```

Figure 27.5. CRT terminal controller software.

After main program initialization, a simple loop program is executed to perform the CRT terminal function. The keyboard queue is checked; if it is not empty, a keyboard pop routine retrieves a character in a first in, first out (FIFO) order. If the echo is enabled, the display routine is called, placing the character on the screen. The character is then transmitted with the USART output routine (shown in Figure 27.5c). When all keyboard characters are sent, the received character queue is checked. If a character is found, a character is retrieved from this FIFO queue by the USART pop routine, and the character displayed.

Depression of the mode switch takes control from the main terminal loop and transfers it to the TRAP routine. Interrupts are reenabled, permitting video functions to continue. A program loop is entered that displays terminal options and permits changes via the keyboard. Any serial characters arriving during execution of this function continue to be placed into the received character queue, and will be displayed when the options-setting mode is departed. The various terminal options are stored in lookup tables, and one table-accessing routine serves for many options. Rather than calculating the required divider ratio to supply specified baud rates from a fixed master clock,

these ratios are precalculated and stored in the same table that supplies the baud rate menu. A second mode switch depression interrupts the TRAP interrupt routine itself, this is detected, and a return from the original interrupt is performed, returning normal CRT terminal operation.

This microprocessor controller example shows how control, status, and data registers and interrupts are used for communication with peripherals. A typical use of lookup tables is also given.

27.8. DEVELOPMENT AIDS

Software production requires much time. Methods that can reduce the required time to write, test, debug, and finalize programs are valuable.

Assemblers

CPU manufacturers assign mnemonics to the binary machine language instructions. An assembler is a program that accepts as input an assembly language source program written with mnemonics, and that outputs the corresponding machine language object code. A minimal assembler performs other functions: address calculation, symbol substitution for data and addresses, offset calculation, binary, octal, decimal, and hexadecimal conversions, addition, subtraction, multiplication, and division of data and addresses, error flagging, and formatted listing and symbol table preparation. A discussion of assemblers can be found in Reference 81.

An assembler program may execute on the same processor for which it is written, in which case it is called a resident assembler, or on a different host processor, where it is called a cross assembler. Controller software will always be assembled on a host machine, not on the controller itself; the host may be of the same type CPU as the controller or completely different. This is in contrast with software for large computers, which is usually assembled on the machine for which it is being written.

Cross assemblers are often more powerful than resident assemblers for micros because of the greater size and speed of the computer on which the cross assembler is being run, compared with even a large size micro hosting a resident assembler.

A standard assembler operates on a one-for-one basis between mnemonics and object code. A macro assembler permits a set of instructions to be defined as a macro command. Commonly used instruction sequences can be simplified with macros. Caution should be exercised not to use macros when subroutines are more appropriate; the use of macros can increase speed while sacrificing program shortness. Discussions of macro assemblers can be found in References 78 and 82.

Development Board

A controller prototype may be constructed from a standard development board for a specific CPU. The board contains the CPU, ROM, RAM, and I/O. A monitor program located in ROM permits loading programs into RAM for test execution. The prototype is not identical to the dedicated, special-purpose final product, but contains enough devices to substitute for the actual controller. Most development boards contain an area where specialized circuitry can be quickly added with wire-wrapped or point-to-point connections. The advantage of the development board is that software can quickly be loaded into RAM for execution, and the debug features of the development board can be exercised to decrease the time between program iterations. Development boards have a keyboard for program entry and control, and display readouts for memory interrogation and status checking. Down-line loading—the capability to directly load software from a host system to the board—greatly reduces turn-around time. Development boards are available for most microprocessors.

Debug Capabilities

The low-level ROM monitor resident on a development board permits the power of the CPU to be utilized for debugging purposes. At a minimum, after execution of test software, memory locations can be examined to determine the effect of the test program. Data val-

ues can be changed through the monitor and the test program again executed. Subroutines for I/O and other functions located in the monitor can be called by test programs to simplify testing. An example of a low-level monitor that can be used on a development board is found in Reference 83.

When all else fails, if the development board has the capability, single stepping through a program will usually locate a problem, although patience is advised with this method. Single stepping can be performed either with hardware or monitor software. The effect is to permit only a single instruction of the test program to be executed at a time. A more effective debug method consists of placing breakpoints at critical spots in the test program. When execution reaches a breakpoint, control is transferred from the test program to the ROM monitor, from which additional tests can be performed. Multiple breakpoint setting can permit zeroing in on a problem in short order. Sophisticated breakpoint capabilities allow a breakpoint to be passed a specific number of times before transfer of control takes place. This is useful in testing program loops.

Development Systems

Because of the inherent nature of a controller, software development must be done on a different system or on a host processor. Although general-purpose computers can be used to this end, development systems are produced specifically for this purpose [84–86]. The development system includes assemblers and high-level language compilers. Test program execution can be done to a limited extent directly on the system.

Simulation can be used to provide debug features that are not available on a development board or test controller. A simulator is a program that creates a virtual machine patterned after the CPU of interest, and the test software is supplied to the simulator as input. Simulation runs much slower than direct execution, and timing relationships may not hold. As a result, simulation may not be effective in testing software interfacing between devices that are time critical.

The most sophisticated of debug techniques is emulation. With this method the development system controls a special hardware package that behaves the same as an actual CPU. A plug from the emulator replaces the processor chip from a test controller. Timing remains the same as with the actual CPU, and the development system can track the behavior of the test controller to provide advanced debugging capabilities at actual operating speed in the actual controller. Descriptions of emulators can be found in References 87 to 91.

After debugging, the program must be placed, or burned, into EPROM. The development system must have the capability to burn the EPROM. A large number of EPROMs are available, and EPROM programmers generally have personality modules that are selected for individual EPROMs. This capability is most important when the debug features of a sophisticated development board or system are not available: in this case, the test programs must be placed into EPROM and tested on the actual controller for each program iteration.

It can be seen that the development aids for small microprocessor controllers can be very powerful computers when compared with the small controller for which they are created.

27.9. HARDWARE/SOFTWARE TRADEOFFS

There are many controller functions that can be realized through either hardware or software. A choice must be made between hardware peripheral devices and additional software for hardware replacement. A common example is that of a serial interface. Hardware implementation can be done with a UART, satisfying both sending and receiving functions. A program, called a software UART, may instead by written that, in conjunction with two bits of parallel I/O, performs the UART task.

The advantage of the software UART is clear—one integrated circuit package is eliminated (if two I/O bits are otherwise available). There are several disadvantages. Unless complicated software is used, it is not possible

to send and receive a character at the same time. The CPU is occupied with the actual timing of the serial data, and this time is generally wasted in software UARTs. An input character may be missed or received incorrectly if it arrives while the processor is not ready for it. With clever software these drawbacks can be removed. The complexity of the software must be contrasted with the almost trivial software interface that a hardware UART requires.

The decision to implement a function in software or hardware depends upon the demands on the CPU and the relative cost of either approach. If spare processor time is available, and a large number of controller units are to be produced, the additional software cost will be overshadowed by the reduction in hardware in production quantities. For low-quantity products, depending upon the additional software complexity, it may be preferable to use a hardware implementation simply to save on software cost. Software implementations may require an additional ROM for greater program storage, cancelling the advantage of eliminating peripheral chips.

Software implementation of several hardware functions may bog the processor down such that it barely has enough time to perform its task. Costs may indicate that the software approach is preferable, but if product updates are expected, it might not be possible to handle additional software because it could overextend the processor power. Eliminating hardware peripherals from a controller design may result in a compact, several-chip design. As a byproduct, address decoders for peripheral device selection may not be needed and may be left out. If the time comes when a peripheral needs to be added, the bus structure of the mostly software approach may preclude additional devices without extensive redesign. This tradeoff is clear: structuring a controller so that it has good capability for hardware expansion increases costs. This fact must be weighed against the lower flexibility of a wholly software implementation.

The major distinction between hardware and software function implementations is speed. Floating-point operations can certainly be handled in software—but the speed can be increased tremendously with the addition of a floating-point processor chip. The chip versus subroutine question will usually be answered in terms of the performance requirements.

27.10. FAULT TOLERANCE

The subject of fault-tolerant computing encompasses a large area. Considering microprocessor-based controllers instead of general-purpose computers causes a shift in fault-tolerant perspectives.

High Reliability Requirements

Processor-based devices often control critical mechanisms. The results of many microprocessor applications would be disastrous, were the processor system to fail in a particular fashion [92]. Automotive controllers, power plant controls, industrial process control, and air- and space-borne systems provide examples where reliability considerations must seriously be explored. Greater concern must be given to reliability in computer control applications than general-purpose computing applications.

There are two types of faults: permanent and intermittent or transient (I/T). Permanent faults cause physical damage and the malfunctioning device must be replaced. I/T faults can be due to many causes, some of which are the effects of a hostile environment. Electromagnetic interference from various sources can disturb processor systems, not physically damaging any circuit element, but nevertheless causing a system upset. ROM program storage in microprocessor controllers prevents I/T faults from modifying the program store, improving recovery chances. Depending upon the I/T fault severity and hardware and software configurations, the system may or may not be able to recover from an upset. A number of approaches can be used to increase the likelihood of system recovery.

Hardware Modifications

Additional hardware can be used to improve a controller's reliability. Checkbits increasing

the number of bits per word can augment processor memory with error-correcting codes. This approach safeguards a portion of a controller but does not guard the entire system.

Traps of several sorts can flag system errors. Invalid memory access is often a result of a program crash, and invalid address traps can be used to interrupt or reset the CPU. Invalid op-code fetch traps will catch system upsets that result in data being incorrectly interpreted as instructions. Special monitoring circuitry can detect undesirable states such as halts or interrupt disable modes. The primary concern of reliability constraints in controllers is to prevent a total program crash, in which event the control function is terminated. The secondary concern is to then preserve data integrity. If a system crash is aborted, the power of the processor can be used to determine the best place to resume control—once a fault causes a crash, unless the system is somehow restarted, all is lost.

Software Modifications

Internal state variables can be encoded in a fashion such that faults causing variable changes can be caught later as data inconsistencies. Proper segmentation of software modules permits subprograms to check one an-

A. Routine Correct for Defined Inputs, But an Undefined Input Causes Endless Loop

```
        ;BIT POSITION TO BINARY CONVERTER
        ROUTINE
        ;
        ;ENTER WITH A SINGLE BIT SET IN REGISTER
        A
        ;EXIT WITH REGISTER B CONTAINING THE
        BINARY
        ;POSITION OF THE SINGLE SET BIT.
        ;DESTROYS REGISTER A.
        ;

        MVI   B,-1    ;INITIALIZE REGISTER B
LOOP:   INR   B       ;BUMP COUNTER
        RRC           ;ROTATE REGISTER A RIGHT
        JNC   LOOP    ;CHECK FOR CARRY SET
        RET           ;RETURN FROM SUBROUTINE
```

B. Corrected Subroutine

```
        ;BIT POSITION TO BINARY CONVERTER
        ROUTINE
        ;
        ;ENTER WITH A SINGLE BIT SET IN REGISTER
        A
        ;EXIT WITH REGISTER B CONTAINING THE
        BINARY
        ;POSITION OF THE SINGLE SET BIT.
        ;DESTROYS REGISTER A.
        ;
        ;NOW WITH GUARANTEED EXIT.

        MVI   B,-1    ;INITIALIZE REGISTER B
LOOP:   INR   B       ;BUMP COUNTER
        ORA   A       ;SEE IF REGISTER A=0
        RZ            ;YES, ABORT AND RETURN
        RRC           ;ROTATE REGISTER A RIGHT
        JNC   LOOP    ;CHECK FOR CARRY SET
        RET           ;RETURN FROM SUBROUTINE
```

Figure 27.6. Effects of out-of-range input variables.

other. Obvious checks—such as verifying that the stack pointer and other data pointers are within the proper boundaries—can be made to permit a software reset.

A fairly simple, yet important constraint can be added to program modules to improve survivability, which may not be addressed other than in the context of upset phenomena. Program modules often execute on the assumption that some conditions hold on input variables. If no faults appear, these assumptions will be correct. A fault arrival can change a variable, either directly or indirectly, and the effect on the subsequent program module must be considered. Correct results must be guaranteed from a module if entry is with uncorrupted variables; module exit must be guaranteed under all input possibilities. An endless loop created by invalid input conditions causes a program crash. Figure 27.6a shows how this can occur. This module works on the assumption that upon entry, register A contains a single set bit. If this subroutine is called with register A erroneously containing all zeroes, an endless loop results. A simple test shown in Figure 27.6b can correct this problem.

Totally software checks can be made by performing calculations several times and checking to see if the results agree. This method can be effective by spreading the computations out over time so that environmentally induced faults will only affect some computations and not others. The byproduct of this fault tolerance method is lowered throughput, since one processor is repeating its calculations many times, where only once is required in the absence of faults.

Combined Hardware and Software Modifications

A commonly used procedure to guard against program crashes is that of a watchdog timer. A hardware timer is appended to the system, and the software is modified so that it periodically activates it. The timer is connected so that if the processor fails to activate it within a specified period of time, the system is interrupted or reset. This method works on the assumption that when the system crashes, it will fail to periodically activate the external timer.

Very elaborate fault-tolerant schemes are possible with microprocessor controllers. The use of voters and triple modular redundancy techniques [93], with appropriate system software exercising, can be used to achieve high reliability.

Much can be done with both hardware and software to improve controller reliability [94]. It is very difficult to be able to evaluate general systems for proper validation.

27.11. PERSPECTIVE FOR THE FUTURE

Microprocessor-based controllers will surely continue to replace a greater number of random-logic designs, and applications will place such controllers in almost all electronic equipment, providing intelligent interfaces with humans. This expanded controller usage will expedite the concern of fault tolerance in such devices. If chips become available that are designed for use with high-reliability measures, then built-in fault tolerance should be addressed.

The semiconductor industry is preparing itself for designs that can intelligently utilize a million transistors per integrated circuit. Design considerations for single-chip processors of the future can be found in Reference 95. Microprocessors that can challenge the power of current mainframe computers are appearing now. Intel's iAPX 432 micromainframe [96] is an indication of the future of sophisticated microprocessors.

The predicted software catastrophe resulting from transferring system design from hardware to software will be combatted with built-in operating systems [97]. This could alleviate software design problems in much the same way as standard digital logic packages aided random-logic synthesis.

Powerful computing systems constructed from many inexpensive microprocessor elements promise to be the way of the future. These distributed intelligence designs will depend upon much microprocessor software development.

REFERENCES

1. L. Waller, "Microcomputers to Run Test House," *Electronics*, vol. 52, no. 15, July 19, 1979, pp. 92–93.

2. P. J. O'Malley, "The Ahwatukee House," Motorola Semiconductor Group, Motorola Inc.
3. W. D. Pierce, "Microprocessor-Controlled Home Environment," Motorola Semiconductor Division, Motorola Inc.
4. "Microcomputer Net is Ultimate Thermostat," *Electronics*, vol. 53, no. 15, July 3, 1980, pp. 88–90.
5. J. G. Rivard, "Microcomputers Hit the Road," *IEEE Spectrum*, vol. 17, no. 11, Nov. 1980, pp. 44–47.
6. L. B. Sanderson and J. C. Lord, "Microcomputers Promise Less Stop, More Go," *IEEE Spectrum*, vol. 15, no. 11, Nov. 1976, pp. 30–32.
7. R. L. Ramey, J. H. Aylor, and R. D. Williams, "Microcomputer-Aided Eating for the Severely Handicapped," *Computer*, vol. 12, no. 1, Jan. 1979, pp. 54–61.
8. "Computerizing Human Birth," *IEEE Spectrum*, vol. 17, no. 5, May 1980, p. 26.
9. "C-MOS Processor to Control Attitude," *Electronics*, vol. 52, no. 18, August 30, 1979, pp. 42–44.
10. M. J. Ellis, G. R. Hovey, and T. E. Stapinski, "MTEC: A Microprocessor System for Astronomical Telescope and Instrument Control," *IEEE Transactions on Computers*, vol. C-29, no. 2, Feb. 1980, pp. 208–211.
11. C. Ringel and J. Tamburri, "Use of Microprocessors to Control and Monitor Operations of Gas Turbine Generators," *IEEE Transactions on Industrial Electronics and Control Instrumentation*, vol. IECI-23, no. 3, Aug. 1976, pp. 238–248.
12. V. Garuts and J. Tallman, "On-board Digital Processing Refines Scope Measurements," *Electronics*, vol. 53, no. 6, March 13, 1980, pp. 105–114.
13. L. Meyer, "Calculatorlike Controller Teaches Precision Multimeter New Steps," *Electronics*, vol. 53, no. 8, April 10, 1980, pp. 105–112.
14. A. J. Nichols, "An Overview of Microprocessor Applications," *Proceedings of the IEEE*, vol. 64, no. 6, June 1976, pp. 951–953.
15. "Smart Gamma Unit Forms Images Faster," *Electronics*, vol. 53, no. 17, July 31, 1980, pp. 44–46.
16. "6800 Varies Speed of Synchronous Motor," *Electronics*, vol. 53, no. 5, February 28, 1980, pp. 42–44.
17. L. Lowe, "Roundup: Smart Controllers Take Over," *Electronics*, vol. 53, no. 5, February 28, 1980, pp. 171–178.
18. "Control System Uses Multiple 8048s," *Electronics*, vol. 52, no. 25, December 6, 1979, pp. 43–44.
19. K. Karstad, "Microprocessor Adds Flexibility to Television Control System," *Electronics*, vol. 52, no. 24, November 22, 1979, pp. 132–138.
20. *Byte*, 70 Main St., Peterborough, N.H. 03458.
21. *Kilobaud Microcomputing*, 80 Pine St., Peterborough, N.H. 03458.
22. *Dr. Dobb's Journal of Computer Calisthenics and Orthodontia*, Box E, 1263 El Camino Real, Menlo Park, Calif. 94025.
23. *Creative Computing*, P.O. Box 789-M, Morristown, N.H. 07960.
24. *Interface Age*, 13913 Artesia Blvd., Cerritos, Calif. 90701.
25. "Apple Turns Pro to Aid Professionals," *Electronics*, vol. 53, no. 12, May 22, 1980, pp. 44–45.
26. "Fifty Years of Achievement: A History," *Electronics*, vol. 53, no. 9, April 17, 1980, pp. 375–381.
27. R. N. Noyce and M. E. Hoff, Jr., "A History of Microprocessor Development at Intel," *IEEE Micro*, vol. 1, no. 1, Feb. 1981, pp. 8–21.
28. S. P. Morse, B. W. Ravenel, S. Mazor, and W. B. Pohlman, "Intel Microprocessors—8008 to 8086," *Computer*, vol. 13, no. 10, Oct. 1980, pp. 42–60.
29. *MCS-4 Microcomputer Set—User's Manual*, Intel Corporation, Feb. 1973.
30. *MCS-8 Microcomputer Set—8008 User's Manual*, Intel Corporation, Nov. 1973.
31. *8080 Microcomputer Systems User's Manual*, Intel Corporation, 1976.
32. *MCS-85 User's Manual*, Intel Corporation, 1978.
33. *Microcomputer Components Data Book*, Zilog Inc.
34. "M6800 Microcomputer System Design Data," Motorola Inc., 1976.
35. "MC6802—Microprocessor with Clock and RAM," Motorola Inc., 1979.
36. "MCS6500 Microprocessors," MOS Technology Inc., May 1976.
37. *User Manual for the CDP1802 COSMAC Microprocessor*, RCA Solid State Division, 1976.
38. *MCS-48 Family of Single Chip Microcomputers User's Manual*, Intel Corporation, July 1978.
39. *TMS9900 Microprocessor Data Manual*, Texas Instruments Inc., Nov. 1975.
40. *The 8086 Family User's Manual*, Intel Corporation, Oct. 1979.
41. B. Hartman, "16-bit 68000 Microprocessor Camps on 32-bit Frontier," *Electronics*, vol. 52, no. 21, October 11, 1979, pp. 118–125.
42. J. G. Posa, "Peripheral Chips Shift Microprocessor Systems Into High Gear," *Electronics*, vol. 52, no. 17, August 19, 1979, pp. 93–106.
43. "Control System Fits on Chip," *Electronics*, vol. 53, no. 4, February 14, 1980, p. 198.
44. *UPI-41A User's Manual*, Intel Corporation, Apr. 1980.
45. *The 8086 Family User's Manual—Numerics Supplement*, Intel Corporation, July 1980.
46. J. Palmer, R. Nave, C. Wymore, R. Koehler, and C. McMinn, "Making Mainframe Mathematics Accessible to Microcomputers," *Electronics*, vol. 53, no. 11, May 8, 1980, pp. 114–121.
47. K. Rallapalli and J. Kroeger, "Chips Make Fast Math a Snap for Microprocessors," *Electronics*, vol. 53, no. 10, April 24, 1980, pp. 153–157.
48. R. T. Atkins, "What Is an Interrupt?", *Byte*, vol. 4, no. 3, March 1979, pp. 230–236.
49. T. A. Harr, Jr. and R. Phillips, "Interrupts Call the Shots in Scheme Using Two Microprocessors," *Electronics*, vol. 53, no. 4, February 14, 1980, pp. 166–170.
50. W. D. Maurer, "Subroutine Parameters," *Byte*, vol. 4, no. 7, July 1974, pp. 226–230.

51. W. P. Fischer, "Microprocessor Assembly Language Draft Standard," *Computer*, vol. 12, no. 12, Dec. 1979, pp. 96–109.
52. M. Marshall, "Assembly Language Plan Raises Dust," *Electronics*, vol. 53, no. 2, January 17, 1980, pp. 98–100.
53. P. Caudill, "Using Assembly Coding to Optimize High-Level Language Programs," *Electronics*, vol. 52, no. 3, February 1, 1979, pp. 121–124.
54. M. Maples and E. R. Fisher, "Real-Time Microcomputer Applications Using LLL Basic," *Computer*, vol. 10, no. 9, Sept. 1977, pp. 14–21.
55. S. Crespi-Reghizzi, P. Corti, and A. Dapra, "A Survey of Microprocessor Languages," *Computer*, vol. 13, no. 1, Jan. 1980, pp. 48–66.
56. J. G. Posa, "Programming Microcomputer Systems with High-Level Languages," *Electronics*, vol. 52, no. 2, January 18, 1979, pp. 105–112.
57. M. Krieger, "Structured Assembly Language Suits Programmers and Microprocessors," *Electronics*, vol. 53, no. 2, January 17, 1980, pp. 118–122.
58. L. Waller, "High-Level Language Will Run on All Microprocessors," *Electronics*, vol. 52, no. 26, December 20, 1979, pp. 39–40.
59. "Pascal Resides in PROM," *Electronics*, vol. 52, no. 9, April 26, 1979, p. 212.
60. "C Language Befriends Microprocessors," *Electronics*, vol. 52, no. 4, February 15, 1979, pp. 41–42.
61. "PL/1 Shrinks to fit Microprocessors," *Electronics*, vol. 53, no. 10, April 24, 1980, pp. 41–42.
62. "CAP-CPP Writes Microcobol Software for Small-Business Applications," *Electronics*, vol. 52, no. 13, June 21, 1979, pp. 71–72.
63. R. R. Bate and D. S. Johnson, "Pascal Software Supports Real-Time Multiprogramming on Small Systems," *Electronics*, vol. 52, no. 12, June 7, 1979, pp. 117–121.
65. *8008 and 8080 PL/M Programming Manual*, Intel Corporation, 1975.
65. J. Brakefield, "A Coding Discipline for Microprocessors," *Computer*, vol. 13, no. 5, May 1980, pp. 120–121.
66. S. M. Hicks, "Forth's Forte is Tighter Programming," *Electronics*, vol. 52, no. 6, March 15, 1979, pp. 114–118.
67. T. Ritter and G. Walker, "Varieties of Threaded Code for Language Implementation," *Byte*, vol. 5, no. 9, Sept. 1980, pp. 206–227.
68. K. Meinzer, "IPS, An Unorthodox High Level Language," *Byte*, vol. 4, no. 1, Jan. 1979, pp. 146–159.
69. *iRMX 80 User's Guide*, Intel Corporation, 1980.
70. D. Kruglinski, "How to Implement Space War," *Byte*, vol. 2, no. 10, Oct. 1977, pp. 86–111.
71. J. Murray and G. Alexy, "CRT Terminal Design Using The Intel 8275 and 8279," *Peripheral Design Handbook*, Intel Corporation, Apr. 1978, pp. 2-119 to 2-176.
72. R. L. Harding, "Fractional-binary Program Creates Pseudorandom Integers," *Electronics*, vol. 52, no. 6, March 15, 1979, pp. 129–131.
73. C. B. Honess, "Three Types of Pseudorandom Sequences," *Byte*, vol. 4, no. 6, June 1979, pp. 234–246.
74. R. Grappel, "Randomize Your Programming," *Byte*, vol. 2, no. 1, Sept. 1976, pp. 36–38.
75. G. Sitton, "8085 Program Rapidly Computes 8- by 16-bit Product," *Electronics*, vol. 52, no. 6, March 15, 1979, p. 129.
76. J. Bryant and M. Swasdee, "How to Multiply in a Wet Climate," *Byte*, vol. 3, no. 4, Apr. 1978, pp. 28–35, 100–110.
77. R. H. Lord, "Fast Fourier for the 6800," *Byte*, vol. 4, no. 2, Feb. 1979, pp. 108–119.
78. "8080/8085 Assembly Language Programming," Intel Corporation, 1977.
79. "Chapter 2: Programming Techniques," *M6800 Microprocessor Applications Manual*, Motorola Inc., 1975.
80. "A Low Cost CRT Terminal Using the 8275," Intel Corporation, Nov. 1979.
81. I. Watson, "Comparison of Commercially Available Software Tools for Microprocessor Programming," *Proceedings of the IEEE*, vol. 64, no. 6, June 1976, pp. 910–920.
82. H. A. Cohen and R. S. Francis, "Macro-Assemblers and Macro-Based Languages in Microprocessor Software Development," *Computer*, vol. 12, no. 2, Feb. 1979, pp. 53–64.
83. R. C. Allen and J. Kasser, "Amsat 8080 Standard Debug Monitor: AMS80 Version 2," *Byte*, vol. 2, no. 1, Sept. 1976, pp. 36–38.
84. "Development Systems Thrive," *Electronics*, vol. 53, no. 8, April 10, 1980, pp. 164–165.
85. C. Bailey and T. Kahl, "Evaluation Delay Cut by Low-Cost Microprocessor Development Tool," *Electronics*, vol. 52, no. 18, August 30, 1979, pp. 121–125.
86. J. Kister and I. Robinson, "Development System Supports Today's Processors—and Tomorrow's," *Electronics*, vol. 53, no. 3, January 31, 1980, pp. 81–88.
87. B. Kline, M. Maerz, and P. Rosenfeld, "The In-Circuit Approach to the Development of Microcomputer-Based Products," *Proceedings of the IEEE*, vol. 64, no. 6, June 1976, pp. 937–942.
88. "Emulator Uses Multiprocessor, Multiple Bus Approach," *Electronics*, vol. 52, no. 18, August 30, 1979, pp. 192–193.
89. C. Zing, "Development System Puts Two Processors on Speaking Terms," *Electronics*, vol. 53, no. 17, July 31, 1980, pp. 93–97.
90. J. Moon, "Microcomputer for Emulation Bares Hidden Buses, Functions," *Electronics*, vol. 53, no. 16, July 17, 1980, pp. 126–129.
91. "Compact Emulator Simulates I/O," *Electronics*, vol. 53, no. 6, March 13, 1980, pp. 168–170.
92. D. R. Ballard, "Designing Fail-Safe Microprocessor Systems," *Electronics*, vol. 52, no. 1, January 4, 1979, pp. 139–143.
93. J. F. Wakerly, "Microcomputer Reliability Improvement Using Triple-Modular Redundancy," *Proceed-*

ings of the IEEE, vol. 64, no. 6, June 1976, pp. 889–895.

94. R. E. Glaser, *Upsets in Microprocessor Controllers,* Ph.D. Dissertation, Dept. of Electrical Engineering, Johns Hopkins University, Baltimore Md., 1981 (University Microfilms No. DA8205099).
95. D. A. Patterson and C. H. Sequin, "Design Considerations for Single-Chip Computers of the Future," *IEEE Transactions on Computers,* vol. C-29, no. 2, Feb. 1980.
96. "Microsystem 80 Advance Information," Intel Corporation, 1980.
97. J. Posa, "Intel Takes Aim at the '80s," *Electronics,* vol. 53, no. 5, February 28, 1980, pp. 89–95.