As you are now the owner of this document which should have come to you for free, please consider making a donation of £1 or more for the upkeep of the (Radar) website which holds this document. I give my time for free, but it costs me money to bring this document to you. You can donate here https://blunham.com/Misc/Texas

Many thanks.

Please do not upload this copyright pdf document to any other website. Breach of copyright may result in a criminal conviction.

This Acrobat document was generated by me, Colin Hinson, from a document held by me. I requested permission to publish this from Texas Instruments (twice) but received no reply. It is presented here (for free) and this pdf version of the document is my copyright in much the same way as a photograph would be. If you believe the document to be under other copyright, please contact me.

The document should have been downloaded from my website https://blunham.com/, or any mirror site named on that site. If you downloaded it from elsewhere, please let me know (particularly if you were charged for it). You can contact me via my Genuki email page: https://www.genuki.org.uk/big/eng/YKS/various?recipient=colin

You may not copy the file for onward transmission of the data nor attempt to make monetary gain by the use of these files. If you want someone else to have a copy of the file, point them at the website. (https://blunham.com/Misc/Texas). **Please do not point them at the file itself as it may move or the site may be updated.**

It should be noted that most of the pages are identifiable as having been processed by me.

I put a lot of time into producing these files which is why you are met with this page when you open the file.

If you find missing pages, pages in the wrong order, anything else wrong with the file or simply want to make a comment, please drop me a line (see above).

It is my hope that you find the file of use to you.

Colin Hinson
In the village of Blunham, Bedfordshire.

TEXAS INSTRUMENTS

9900

TMS 9900 Family Software Development Handbook

MICROPROCESSOR SERIES™

First Edition

PREFACE

This handbook has three principal aims:

1) To introduce the concept of microcomputer software

2) To provide a survey of the systems software and development tools available for the TMS 9900 family of microcomputers

3) To act as a guide to the steps involved in developing software for a microcomputer-based product.

This book introduces a systematic approach to software design from the start. Previously, much microprocessor-based development was done in a haphazard, experimental manner, with users learning how to do it as they went along. This approach was inevitable as the tools provided by microprocessor manufacturers were not very sophisticated. In addition, few people (even within the manufacturers) had much experience in regard to microprocessors or microprocessor applications.

Recently, this situation has changed. With rapidly falling hardware costs and rising labour costs, software has become the major investment in developing a microprocessor-based product. It is no longer appropriate to adopt a "trial and error" approach -- this is both expensive and time-consuming.

It is therefore important that the software designer has a set of tools which make a job easier and more controllable. Texas Instruments is committed to providing these tools. A complete range of systems software and development tools is now available for the TMS 9900 range of microcomputer, comparable with those provided for minicomputers and mainframes.

Some of these tools (compilers, assemblers, link-editors, etc.) are very similar to those already existing for larger computers. Texas Instruments was in a good position to develop these due to its wide experience in minicomputers (from which the 9900 microprocessor was originally developed). The 9900 shares the same basic instruction set as the 990 range of minicomputers, for which extensive software exists, including compilers for most of the major high-level languages. Other tools, such as the AMPL emulator, which provides real-time in circuit testing, had to be designed from scratch.

This handbook is a guide to the tools currently available and to the methods of using them to best effect (something which is not always known to a user unless he has a background in software). Part of the book's purpose is to highlight the design choices; for example, providing information that would help the user in choosing the appropriate language or development system for a particular application. The design techniques introduced are intended to bring out design choices in the application itself.

Much can be learned from recent thinking in software design which, in historical terms, has just emerged from the Dark Ages (the science of software is little more than twenty years old) and is being established for the first time on a systematic basis. However, the techniques now increasingly being adopted for mainframe software design need some adapting to the special environment of the microcomputer. This book is directed to achieving a professional approach to software design for microcomputers, by combining state-of-the-art techniques with experience gained from designing and building microcomputer systems for real applicatons.

HOW TO USE THIS BOOK

The first chapter (Introduction) is directed at those who do not have experience with sofware or microprocessors. It describes the elements of a computer, what software is and how it fits into a microcomputer-based system, and what is special about microcomuters as opposed to minicomputers or mainframes.

The second chapter (Software Concepts) explores microcomputer software in more detail, presenting a range of ideas which are important in software design. It describes aspects particular to microcomputer software -- for example, ROM/RAM partitioning and the relationship between development and target systems. Chapter IV, "Data", and Chapter V, "Algorithms", introduce basic concepts of software design. These ideas, and the notations introduced to describe them, are used extensively in the rest of the book. These sections are worth reading even by those familiar with the rest of the material in this chapter. They form the basis of the systematic approach to software design adopted by this book.

Chapter III, Software Development, examines the various steps involved in developing a microcomputer-based system, from problem definition to implementation, with software particularly in mind. Once again, a systematic approach to software development is adopted as far as is possible. Along the way, the software and hardware tools used in software development and testing (editors, compilers, emulators, etc.) are described.

The remaining chapters describe each of the three programming languages in which it is possible to implement a software design for the TMS 9900: Pascal, POWER BASIC, and Assembly Language. These chapters are not intended to be complete tutorial manuals (these are available elsewhere) but they should give the reader a good feel for the capabilities of each language, and the applications for which it is best suited. Each chapter contains a Reference Section, which sets out in abbreviated form all the important constructs and features of the language. This will be a useful reference guide once the basic ideas of the language have been mastered. In addition, the Assembly Language chapter includes a large "Algorithms and Techniques" section, which describes ideas and techniques for implementing commonly used programming concepts in Assembly Language.

BIBLIOGRAPHY

Texas Instruments Publications:

97049-118-NI 9900 Family Systems Design and Data Book

943441-9701 Model 990 Computer/TMS 9900 Microprocessor Assembly Language Programmer's Guide

MP351 The Microprocessor Pascal System User's Manaul

946290-9701 TI Pascal User's Manual

MP305 TI Pascal Microprocessor Executive Library Reference Manual

MP318 Configurable POWER BASIC Reference Manual

MP308 TM990 POWER BASIC Reference Manual

1602030-001 TM990/302 Software Development Board User's Guide

Model 990 Computer DX10 Operating System Release 3 Reference Manuals Volumes:

946250-9700 II Production Operation

946250-9703 III Application Programming Guide

946250-9704 IV Development Operation

MP003 TMS 9901 Programmable Systems Interface

MP334 TM 990/201 and /206 Memory Expansion Board

Non-Texas Instruments Publications:

Osborne, A. An Introduction to Microcomputers, Vol. I and II, Osborne and Associates

Suppl, C and Kidd, D. Microcomputer Dictionary and Guide: Matrix Publishers

Wirth, N. Algorithms + Data Structures = Programs: Prentice Hall

Jackson, M. Principles of Program Design: Academic Press

Dijkstra, E., Dahl, O. and Hoare, C. Structured Programming: Academic Press

TABLE OF CONTENTS

CHAPTER I - INTRODUCTION

CHAPTER II - SOFTWARE CONCEPTS

CHAPTER III. SOFTWARE DEVELOPMENT

CHAPTER IV. PASCAL

CHAPTER V. POWER BASIC

CHAPTER VI. ASSEMBLY LANGUAGE

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER I

INTRODUCTION

1.1 WHAT IS A MICROCOMPUTER?

A microcomputer is a complete computer system implemented on a few square inches of printed circuit board. A microcomputer is built with a handful of standard integrated circuits. For small scale applications, it is possible to implement a complete computer system on a single chip of silicon (an integrated circuit). In larger applications, the heart of the system is a component called the microprocessor.

A microprocessor is a general purpose integrated circuit that can be programmed to perform a particular function. A microcomputer-based product can be constructed from a microprocessor, a selection of inputs and outputs and program memory. The inputs and outputs can be anything that is, or that can be converted to, digital electrical signals. The hardware design for a microcomputer product simply consists of interfacing inputs and outputs to the microprocessor - which is usually very straightforward. Its operation is specified by the program: a list of instructions to the microprocessor which is stored in the system's memory. In contrast with the hardware (the physical components of the system), the program which controls the system's operation is referred to as software.

There are several advantages to using a microcomputer instead of more conventional techniques. First, because a microcomputer uses standard, highly integrated components, it is inexpensive. The component count is much reduced compared with a conventional logic implementation (such as TTL); and the design cost is several orders of magnitude less than the development of a custom integrated circuit. In fact, the design is significantly quicker and less expensive than any form of hardware design because software is much easier to manipulate. In addition, software has the advantage of flexibility. If the software is well designed, changes can be incorporated, even late in the development process, with little disruption.

The microcomputer provides computer power small enough and inexpensive enough to be incorporated in almost any electrical device. This has two results:

1) Many existing devices can be built more inexpensively using a microcomputer

2) There are exciting new possibilities, due to the immense power of the microprocessor.

Unlike traditional computers, the inputs and outputs of a microcomputer are not restricted to standard peripherals such as card readers and line printers; devices which, fundamentally, can only handle paperwork.

The microcomputer allows computing power to be located where it is needed, rather than being locked away in a distant 'computer room'. It is small enough and inexpensive enough to permit its use in a dedicated application, where it does one thing all its life; and it is even economic to allow it to remain idle for a large proportion of its time, to ensure that it is there when required. The microcomputer makes it possible for users to determine what computers shall do. Previously, the economics of a computer dictated how it could be employed. For example, a large mainframe had to be kept running all the time simply to justify its expense. The microprocessor has gone a long way in taming computer power, and making it obedient to the needs of the user.

The major effort of microcomputer design goes into software. Software is in a number of ways easier to deal with than hardware. However, it must be treated with respect. Designing the software for a complex application is not trivial, especially as the availability of the microprocessor leads to more ambitious projects. This book shows what is involved in developing microcomputer software. As it is a new technique, new methods must be used: those developed for hardware design are not appropriate. Even techniques used in the design of software for 'mainframe' or 'mini' computers need adaptation, because of the special features and the different areas of application of microcomputers.

This introductory chapter explains what a microcomputer is, and how software and computer programs fit into the picture. The second chapter explores some of the concepts essential to the software designer. The third chapter examines the actual process of software development, and the steps that are involved in carrying through a design from problem definition to implementation.

1.2 BLACK BOXES AND DIGITAL ELECTRONICS

A mechanical or electrical device can be considered, very simply, as a black box with inputs and outputs:

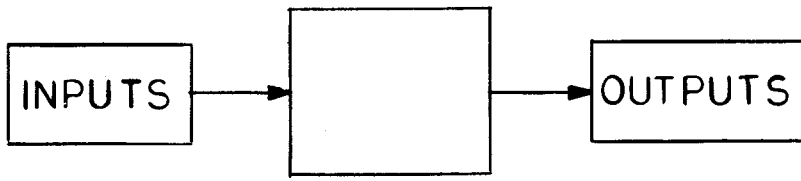


FIGURE 1-1. ELECTRICAL DEVICE

The black box processes these inputs and produces outputs in a well-defined fashion. For example, a typewriter takes key presses as input and produces printed characters corresponding to the key inputs as outputs. This is a particularly simple example: the processing may be time-dependent, and may also depend on previous inputs and outputs. All problems that are solvable by machinery can be analyzed in this manner. The black box, with its inputs and outputs, may be called a system.

How can such a black box be built? The traditional, non-computer method would be to design a dedicated piece of hardware: a mechanical device. Methods of implementation have varied. Early workers used wires, pulleys, cogs and a great deal of mechanical ingenuity.

More recently, electronics has made things much easier. Perhaps the most significant advance in black-box implementation was the invention of digital electronics, based on the binary digit, or bit.

A bit can be considered as a switch. It has two possible states: on or off, 1 or 0, high or low. Bits can easily be represented in electronic circuits, and they can be used to store information. Circuit elements can be designed that combine bits in various useful ways. One such element is the AND gate as depicted in the following figure.

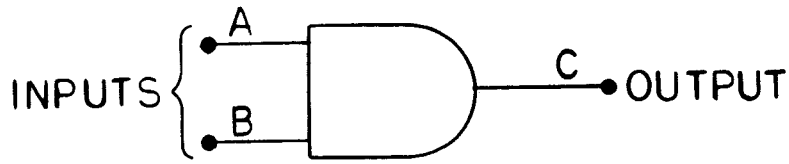


FIGURE 1-2. AND GATE

The possible states of A, B and C are conventionally represented as "0" and "1". For given conditions of the inputs A and B, the output C is completely determined. For an AND gate, C is 1 only when both A and B are 1.

By combining logic elements such as the AND gate, complex black boxes can be designed to perform a variety of functions. Solving a real world problem, of course, depends on translating real inputs (such as mechanical movements, temperature readings, etc.) into bits, and translating bits back into the real world.

This process of translation can be represented (adding to the black box diagram) as:

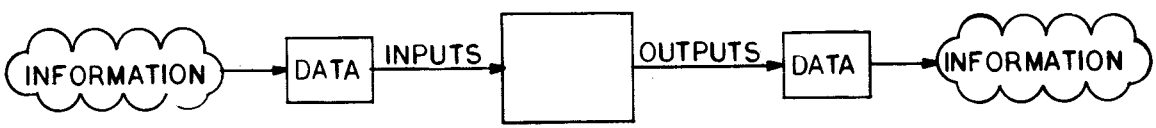


FIGURE 1-3. TRANSLATION PROCESS

'Information' is used here in a very wide sense. It may involve physical interaction - for example, turning on a motor.

'Data' is a term used for information divorced from its meaning - that is, information translated into a pattern of bits for processing by a digital circuit. The digital circuit does not know or care what the data represents; it simply processes bits according to the logic built into it.

Digital electronics is powerful because it is only concerned with bits. The bits can represent anything, and the same techniques can be used for a wide range of different applications. However, this can cause problems, because bits (data) are entirely abstract entities.

The designer must be very sure that he knows exactly what his data represents. Translating information into data (i.e., bits) in a well thought-out manner is probably the most important step in designing a digital system.

In the last 20 years, advances in technology have made it possible to place several thousand basic logic elements on a single chip of silicon. However, with the technological advance has come the problem of organization. Organizing all these logic elements to perform the desired action is a very difficult, time consuming, and expensive task, requiring a highly skilled designer (or team of designers). In addition, because an AND gate is a piece of hardware - a physical device - it is quite awkward to manipulate. Once a design has been put together, it is extremely difficult to change in any significant way without starting again from scratch.

This is where the computer comes in.

1.3 ELEMENTS OF A MICROCOMPUTER

Like other digital devices, computers work with bits. In fact, they usually work with groups of bits. The TMS 9900 family of microprocessors uses a basic unit of 16 bits, called a word. The possible operations that can be performed on words are strictly limited and well defined, which is what makes the computer possible.

Of the total range of operations, the most useful are selected to form the computer's instruction set. Each instruction performs one operation. For example, there is an operation to perform a logical AND on two words of data:

```
first word    0 1 0 1 1 0 1 1 1 0 0 1 0 1 1 0
second word   0 1 0 1 0 1 0 1 1 0 1 0 1 1 0 1
              -----------------------------------
result        0 1 0 1 0 0 0 1 1 0 0 0 0 1 0 0
```

Corresponding bits in each word are ANDed together to produce the corresponding bit in the resultant word. Here, a word is treated as containing 16 unconnected bits. The instructions which operate on words in this way are called logical instructions.

Using the binary number system, a 16-bit word can also represent a number. There is a group of arithmetic instructions which treat words as numbers, and perform the usual arithmetic operations on them. For example, add:

```
first word    0 1 0 1 1 0 1 1 1 0 0 1 0 1 1 0
second word   0 1 0 1 0 1 0 1 1 0 1 0 1 1 0 1
              -----------------------------------
result        1 0 1 1 0 0 0 1 0 1 0 0 0 0 1 1
```

(The binary number system is described in Section 6.11.2. The TMS 9900 instruction set also includes operations on bytes (1 byte = 8 bits) of data.

In addition there are instructions to read inputs and write outputs, and to move data around within the computer.

The operation of the computer can be specified by a list of these basic instructions. The list of instructions is called a program, and is stored in the computer's memory. A computer, then, looks like the following figure:

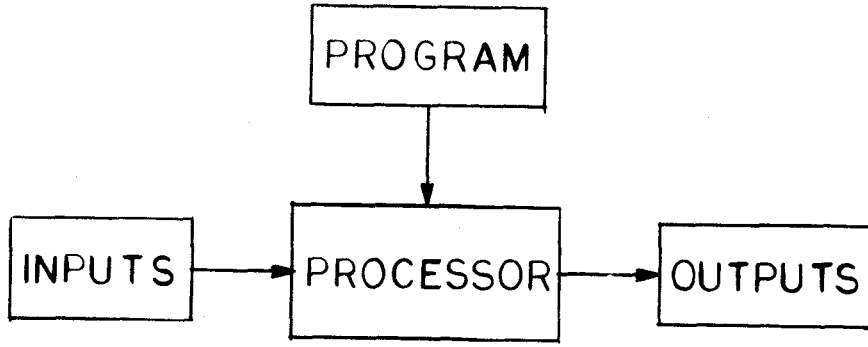


FIGURE 1-4. COMPUTER AND PROGRAM

The stored program controls the operation of the computer. The processor fetches the program instructions one at a time in sequence, and executes them. The sequential flow of the program can be altered by conditional instructions, which cause a jump to another part of the program, if the condition is met. One example of a condition is whether a particular data word is greater than zero. This makes very powerful programs possible.

The program completely determines the operation of the system. If the initial conditions and all of the inputs are known, the action of the computer will be entirely predictable.

Thus a computer is a black box - but a black box with a difference. Program memory is designed to be alterable. Simply by changing the program, the operation of the system can be altered. All the mechanical parts of the system (the hardware) are the same, but it is, in effect, a different black box. Computers are characterized by this new element (software) which determines their operation. Computer hardware can be regarded as a pool of resources, which is organized by the software. By placing the burden of organization on software, many of the problems of designing a digital system are solved.

Looking at a computer in more detail:

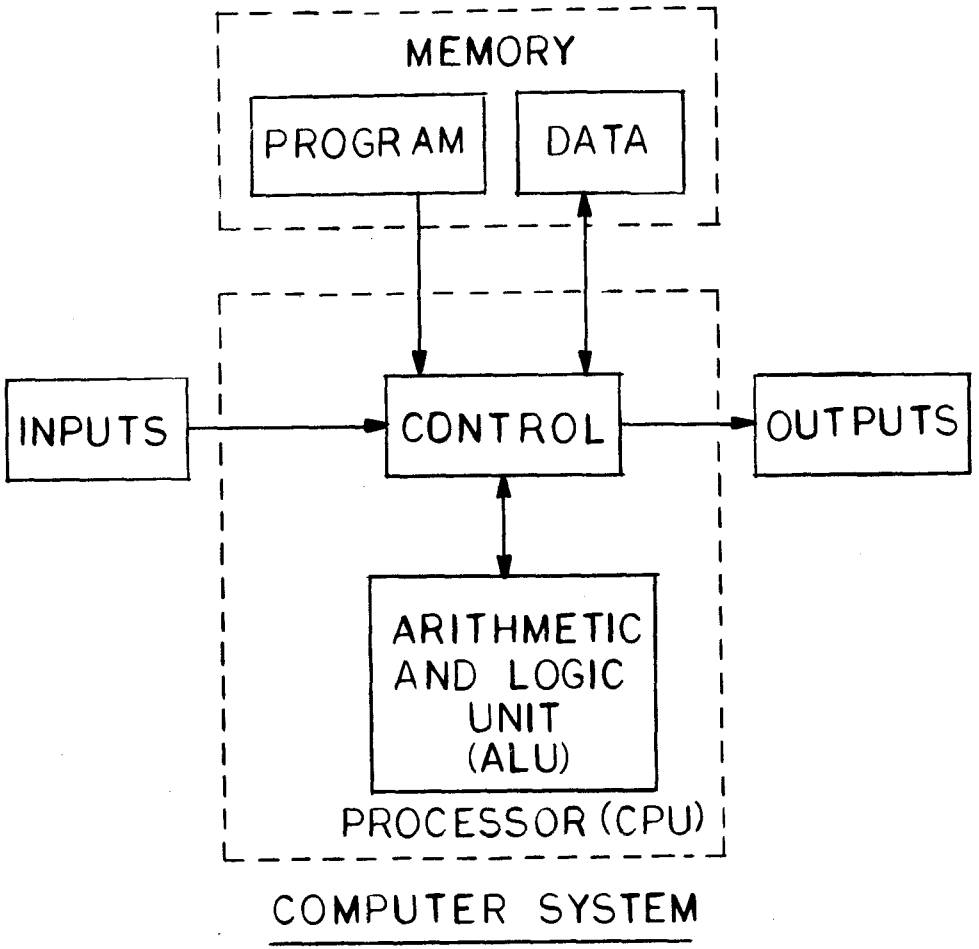


FIGURE 1-5. COMPUTER SYSTEM

The Arithmetic and Logic Unit (ALU) performs the operations requested by the program (addition, subtraction, logical ANDing, etc.). The Control Section supervises the reading and writing of program, data, and I/O, and ensures that everything happens in the proper sequence. These two elements are traditionally grouped together to form the Central Processing Unit (CPU), or Processor. When this is implemented on a single silicon chip it is called a Microprocessor, or MPU. The complete system is a Microcomputer. The Texas Instruments TMS 9940 is a complete microcomputer on a single chip.

Besides inputs and outputs, a computer will probably need a place in

which to store data (i.e., a scratchpad or filing system). Therefore a computer will generally have data memory as well as program memory. Unlike program memory, data memory must be capable of being changed by the program.

The inputs and outputs, more than anything else, determine what a computer system looks like to the user. When the usual peripherals (card reader, VDU console, line printer, magnetic tapes, etc.) are connected, it looks like everyone's idea of a computer. But interface it to motors, lights, switches, gauges and it could be anything from a washing machine to a car dashboard. A microcomputer is small and inexpensive enough to be hidden in almost any piece of electrical equipment, and the user need not even know that it is there.

1.4 MOST MICROCOMPUTERS ARE DEDICATED

Until a few years ago, the only computers in common use were general purpose machines. A general purpose digital computer consists of a central processing unit (CPU), main memory and a number of peripherals - devices which enable data to be input to and output from the computer. A typical configuration might look something like this:

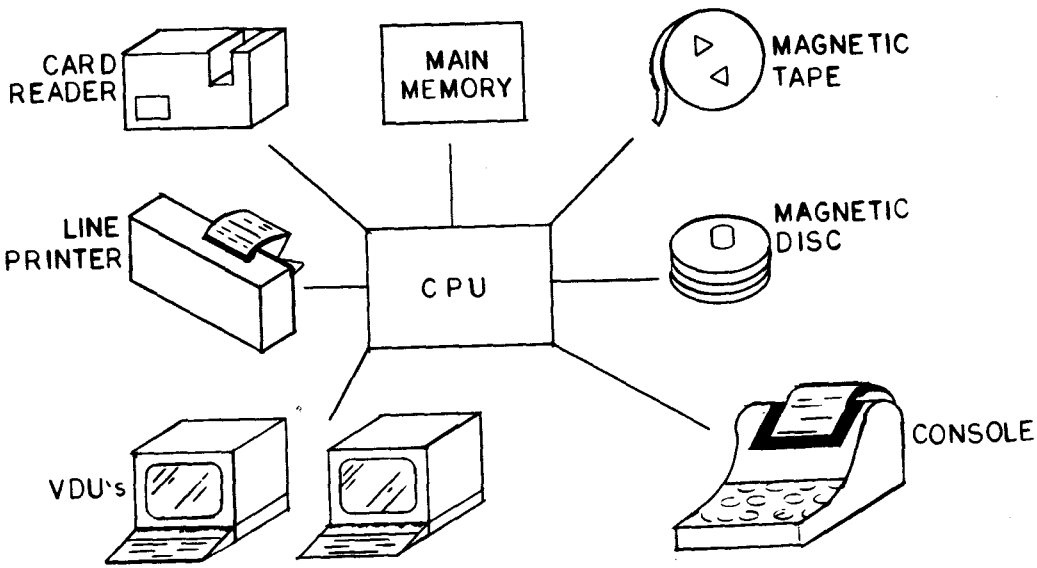


FIGURE 1-6. TYPICAL COMPUTER

One of the most important peripherals is the backing store. This is a memory device that is slower than the main memory, but has a large capacity. Its principal function is to load programs and data into the computer's main memory. A general purpose computer has a large repertoire of programs in its backing store, any one of which can be loaded and executed. Some of these programs are systems programs, which control the operation of the computer and provide commonly required tasks (these will normally be provided by the computer manufacturer), while others are applications programs developed by the user.

The most important systems program is that which runs the entire computer, and controls the loading and executing of other programs under commands from the user. This program is variously called the executive, monitor or operating system and is loaded into main memory

when the computer is switched on, remaining in control the whole time the system is running. Other systems programs provide software tools for developing applications programs. They can be called in as required by the operating system.

A general purpose computer is, therefore, a chameleon-like machine which can perform almost any function depending on the program which is loaded into it. However, the range of things it can do is limited by the peripherals which are connected to the computer. Standard peripheral devices include keyboard and visual display unit (VDU), teletype, line printer, punched card or paper tape readers and punches, and magnetic disc or magnetic tape devices. These last two are forms of backing store; the others are means of communicating with the user.

The input and output devices of a general purpose computer are designed to handle only alphanumeric characters (i.e., letters and numbers). This INPUT LIMITATION, as well as the physical size and expense of such computers, has limited their use to such things as ACCOUNTING, payroll MANAGEMENT, and scientific calculations. Although they have tremendous automatic processing power, a lot of human effort is required to translate input into machine readable form, and to interpret machine generated output. A general purpose computer requires spoon feeding: special purpose input must be converted into digestible punched cards. Computers have a bad name in many quarters because of the drastic changes which must be made to traditional patterns of work in order to to adapt them to the (until now) inflexible computer.

There is no reason why a microcomputer should not be constructed as a general purpose computer: the Texas Instruments TM 990/4 is exactly that. But the microcomputer has opened up a new possibility: the dedicated system. A dedicated microcomputer might look like this:

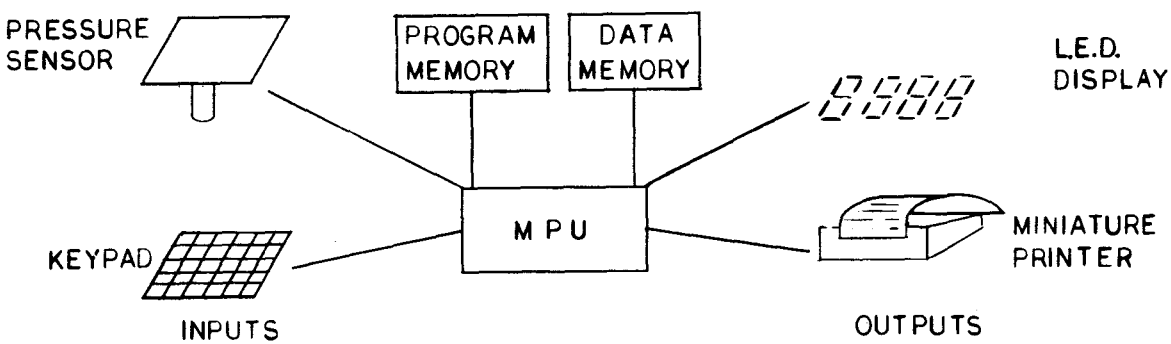


FIGURE 1-7. DEDICATED MICROCOMPUTER

This system could serve as a weighing scale. A program would be written to read the pressure sensor and the price (entered on the keypad), multiply the weight by the price, display the result, and print a ticket. With extra software, the system could become a complete cash register. The complete microcomputer and associated circuitry could be fitted into one corner of the case.

A dedicated microcomputer can only execute one program, which starts running when the system is switched on and only stops when it is switched off. Most real-time programs are endless loops. In the example pictured above, the program would spend most of its time in a loop repeatedly checking whether or not there was any input from the pressure sensor or the keypad. If there was, that portion of the program written to deal with that input would execute.

A dedicated microcomputer will probably have no backing store and no executive. The program will be stored permanently in memory.

1.5 SO WHAT?

The dedicated microcomputer has accomplished two things:

1) It has revolutionized the design of both small and large-scale electrical devices, from toys to cars

2) It has changed the role of the large general purpose computer, or 'mainframe'.

With the arrival of the minicomputer several years ago, the death of the mainframe has been predicted; that death sentence has been premature. It now seems there will always be a need in some applications for a large centralized database of information, and massive processing power. But microprocessors have changed the way information is input to, and output from, mainframes.

Microcomputers have been used to build 'intelligent' peripherals for mainframes (disc controllers, for example) which can handle some of the local 'housekeeping' functions required by the peripheral and take the load off the central processor.

One significant development in this regard has been the intelligent terminal, a visual display unit containing a microcomputer. The intelligent terminal provides local processing power for small tasks, can be linked to the mainframe for reference to central files and for handling a large amount of processing.

With the development of 'personal' computers, the microcomputer system is likely to be carried a stage further. A storekeeper, for example, might use a microcomputer to handle his daily transactions, and then transmit his accounts over a dial-up link to the central office computer. The British Post Office's Postal system is an example of how such a scheme could be implemented on a large scale. This is a public computer network which can be accessed by anyone with the right equipment (which can be as small as a TV set and a keyboard) via the telephone network. It provides information and services, and can even be used to transmit software to a user's computer.

The microcomputer allows the distribution of computing power to the place in which it is needed - the office, the factory floor, even the home. Local processors can be linked to larger computers, using the telephone network if permanent connection is not required. Special purpose microcomputers can be constructed to collect information where it is generated and in the form that it exists. These microcomputers can automatically translate information into data suitable for a mainframe automatically, without the tedious manual process of data preparation.

Microcomputer applications range from simple real-time control functions (such as a weighing scale) to sophisticated computer networks. (A 'real-time' application is one in which the computer is in direct control of a process, event, or phenomenon such as monitoring electronic ignition timing and fuel mixing, and modifying it during its actual occurence. The TMS 9900 family is particularly suited to cover a wide range of these applications. It has features that are useful for real-time control, such as the Communications Register Unit (CRU), which is a bit-oriented method of input and output. The CRU allows the 9900 to input and output data either in single bits or in groups of any size from 1-16 bits. It was developed from Texas Instruments' experience with the 960 and 980 series of minicomputers, in process control applications. In addition, the 9900 has a powerful minicomputer architecture, including multiply and divide instructions and multiple addressing modes. The 9900 family shares the same basic instruction set as the 990 range of minicomputers, including the powerful DS990/10, which operates in a multi-user environment with up to 1 Megabyte of memory. The 990/10 is currently being used in a wide range of commercial and industrial minicomputer applications. One advanced feature of 990 and 9900 architecture is the workspace register concept, lending itself naturally to the implementation of modern programming techniques such as high level languages. The architecture of the 9900 is described in detail in the Assembler chapter. Texas Instruments supplies extensive software support for the whole range of 990 and 9900 products.

The microcomputer has a dual personality: it is both electronic component and computer. This is why it provides such a rich field for applications. The technology and the opportunity exist for a wide range of products; the only real limit is the imagination of the designer.

CHAPTER II

SOFTWARE CONCEPTS

2.1 OVERVIEW

This chapter covers a wide range of topics with which the software designer should be familiar. Each section is relatively self-contained, although later sections tend to draw on ideas introduced in earlier ones. The chapter does not present any kind of sequential argument, but rather sketches out a broad spectrum of ideas and concepts. It is intended to provide the reader with the background, or context, from which further exploration can be undertaken.

2.2 ROM AND RAM - SEMICONDUCTOR MEMORY

Computer memory can be thought of as a collection of pigeon holes or locations in which values (i.e., numbers or patterns of bits) can be stored. These locations can be referred to by their consecutively numbered addresses.

Semiconductor memory systems are typically organized in bytes.

The TMS 9900 family can operate on both bytes and words (16 bits) of data. A word is stored in two consecutive memory locations, starting at an even address.

A general purpose computer requires a program memory that can be written to as well as read, since different programs must be loaded into it from the backing store. However, once the program is loaded, the portion of program memory in which the program is stored will not normally be changed until the operating system loads in the next program. (The program can change data memory, but not the program code.)

A dedicated system only executes one program. Normally it will not need an operating system, and the program need never be changed. Therefore a special type of program memory, called Read Only Memory (ROM) is used for dedicated microcomputer systems. A ROM memory chip is programmed (i.e., loaded with a program) once, outside the system in which it will be used, and retains its contents permanently (even when the power is switched off). This last feature is important because there will probably be no backing store from which to load the program when the device is switched on.

2.2.1 ROM Types

There are several different types of ROM, each with its own characteristics.

Mask ROM has the program inserted as part of the manufacturing process. A mask must be made to etch the pattern of binary digits which form the program on the surface of the silicon chip. Generating this mask is an expensive process, because it must be done with great precision. However, once the mask has been made, programmed ROMs can be manufactured very inexpensively. Where large quantities (thousands) of identical ROMs are required, this method is by far the least expensive.

Programmable ROM (PROM) is manufactured with fusible metal links in each memory cell. These links can be selectively fused by applying high voltage pulses to the PROM chip after manufacture using a device known as a PROM Programmer. Blank PROMs are supplied by Texas Instruments and can be programmed by the user (usually an equipment manufacturer) to put in his system. Once the pattern of 0's and 1's has been 'burned in' in this way the PROM cannot be erased. PROMs are more expensive per chip than mask ROMs, but work out cheaper overall for small to medium quantities (hundreds), because of the cost of manufacturing a mask.

Erasable Programmable ROM (EPROM) is supplied blank and programmed in the same way as PROM. But the high voltage pulses do not break fusible links: instead they selectively establish static charges in the memory cells, which turn on or off switching devices (transistors) that represent the 0's and 1's. An EPROM is a very useful device. It can be programmed permanently, like a fusible link PROM; by exposing it to ultraviolet light for a period of about 20 minutes, it becomes erased and can be programmed with something different. EPROMs are slightly more expensive than PROMs, but their special features make them valuable for many applications, particularly in development.

Most microcomputer systems require some memory that can be written to as well as read, for storage of intermediate results. This is achieved by using RAM (Random Access Memory) instead of ROM. RAM is actually a misleading term, since ROM can also be accessed randomly. (Read/Write Memory would be more descriptive, but 'RAM' is the traditional term.) In a general purpose computer, the main memory is implemented entirely with RAM. A microcomputer system is more likely to have a partitioned memory - some ROM and some RAM. One of the most important decisions to make when designing a microcomputer system is how much ROM and RAM to build in. The fact that memory is partitioned also has consequences for the design of software.

2.2.2 RAM Types

Semiconductor RAM is volatile; the contents disappear when the power is switched off. There are, in fact, two types of RAM:

- Static RAM retains its contents for as long as the power is switched on.
- Dynamic RAM must be refreshed, that is, read or written to every few milliseconds, or its contents decay.

Dynamic RAM requires some external circuitry to implement this refresh, and is therefore more difficult to design into a microcomputer. However, it is less expensive and smaller than static RAM. Static RAM is normally used for systems that require a relatively small amount of RAM; dynamic RAM for larger systems where the cost of refresh circuitry can be justified by the savings on memory chips.

2.2.3 ROM/RAM Summary

Both ROM and RAM are supplied by Texas Instruments as standard integrated circuits, and are therefore very suitable for microcomputer products.

The characteristics of semiconductor memory are summed up in Table 2-1 below.

TABLE 2-1. SEMICONDUCTOR MEMORY CHARACTERISTICS

	Mask ROM	PROM	EPROM	Static RAM	Dynamic RAM
Readable?	y	y	y	y	y
Writeable?	n	n	n	y	y
User programmable? (outside system)	n	y	y	-	-
Eraseable? (outside system)	n	n	y	-	-
Retain contents without power? (non-volatile)	y	y	y	n	n
Require refresh?	n	n	n	n	y

2.3 DEVELOPMENT SYSTEMS

In traditional forms of computing, software is usually developed on the machine on which it is to run. Such computers are general purpose machines capable of running many different programs, including the 'software tools' used in program development.

With microcomputers, this is not usually possible. Normally, a dedicated system cannot be used to develop the software that is to run on it. Most microcomputer systems are designed to run only one program, and probably will not have the peripheral devices (keyboard, printer, etc.), much less the software tools, required for program development.

For this reason, a general purpose computer system called a development system is used to develop software for a microcomputer. The dedicated microcomputer in which the software will finally run is called the target system. The development system is often a minicomputer, such as the Texas Instruments 990/4 or 990/10. The 990/4 and 990/10 have the same instruction set as the TMS 9900 family of microprocessors, which makes software development a lot easier. (The 990/4 uses a TMS 9900 as its central processor). However, it is possible to develop software for a microcomputer on a large mainframe computer, such as an IBM 370. Texas Instruments provides software development tools for 990 minicomputers, and also cross-support software that can be used on other general purpose computers.

A microcomputer development system is likely to have one or two special purpose peripherals, such as a PROM Programmer. The AMPL package (Advanced Microprocessor Prototyping Laboratory) provided by Texas Instruments for the 990/4 and 990/10 minicomputers also allows target system emulation. The target hardware is connected by a cable to the development system. The emulator runs a program contained in the development system's memory, on the actual hardware of the target system. All the resources of the development system are available to monitor and to change the program if necessary. AMPL provides sophisticated testing aids for both hardware and software.

Using the peripheral devices and the software tools provided with the AMPL development system, it is possible to write a microcomputer program, translate it into machine understandable form (i.e., binary digits), test it under simulation on the development system, try it out in the target system hardware, and finally write it permanently into the memory of the target microcomputer system.

2.4 DESIGNING A MICROCOMPUTER SYSTEM

Developing a microcomputer application involves two steps:

1) Specifying the hardware of the system

2) Writing the software to drive it.

2.4.1 Hardware Design

The hardware of a system can be regarded as resources, to be manipulated by the software. The hardware needs to be fixed at a relatively early stage of a project, because it typically requires a much longer production lead time than software. If necessary, software can be changed in a matter of weeks or even (if the program is in PROM) days before production begins.

To fully exploit this software flexibility, some thought must be given to the initial hardware design. The detailed algorithms that will be used need not be considered at this stage, except for the resources they will require.

In particular, all the inputs and outputs (the I/O) need to be identified early and designed into the hardware. Discovery, at a late stage in development, that a vital input or output signal is missing can be both frustrating and expensive.

It is a good idea to design as much as possible of the system logic in software rather than hardware. Changes in the way the system operates (perhaps in response to altered market requirements) can then be made much more easily.

Trade-offs can often be made between hardware and software. Where an operation can be carried out either way, it is usually an advantage to do it in software (as the computer is already there) and save the cost of the extra hardware elements, provided the processor will not be overloaded. A rough estimation of the system load (e.g., seconds of processing time per second of real-time) needs to be made early in the effort.

With this approach, hardware design becomes simply a matter of interfacing signals to the computer.

Use of a ready-built microcomputer board (or boards) simplifies the process of hardware design. Texas Instruments supplies a range of microcomputer modules (the TM990 series) which are ready built microcomputers with a range of inputs and outputs, and memory configurations, to suit many requirements. Expansion boards are available to extend both memory and I/O, and to provide such additional functions as analog to digital and digital to analog conversion.

2.4.2 Software Design

Once the system hardware has been fixed, software design can continue in parallel with the building of hardware to the agreed system specification. Designing the software involves, first, considering the data structures that the program will use; second, developing the algorithms that will manipulate the data. The data represents all information about the application resident in the computer. If the data is inadequate or badly structured, it will be difficult to write a good program. An algorithm is a specification of what the system is to do in unambiguous terms -- that is, a detailed description of how the microcomputer will carry out the task. It is very desirable to separate the design of data structures and algorithms from the details of implementation, because these initial stages involve fundamentally understanding the problem. It is not wise to bring in details of implementation (programming languages, etc.) until the task to be performed is well understood. Novice software designers tend to rush into programming too early.

Studies have proved that time spent at the beginning of a project in thoroughly understanding the task to be performed, and developing the best possible design (not just one that works), is repaid many times over in time saved at the development, debugging and testing stages; even eliminating errors in the final product, which can be very expensive to correct.

Implementation of a design involves, finally, translating it into a form that the computer can understand -- a pattern of binary digits representing a program. However, the language of machines is not designed for humans to understand, and programming in binary is not really a practical proposition.

Programming languages were developed to make the implementation process easier. Texas Instruments has developed for the user superstructures of the Pascal and BASIC languages that are supported by the TMS 9900 family of microprocessors. The TMS 9900 can also run on assembly language to fit the user's needs. Of course, the appropriate implementation of language depends on the task to be performed. Chapters IV through VI provide descriptions of these language alternatives.

2.5 DATA

Data, which is just a collection of bits, can be used to represent any kind of information. Often it will be some form of numeric information, but this need not be the case. On input and output, each bit will probably signal the state of an I/O line; high or low, 0 or 1.

It is worth spending a good deal of time deciding how the information will be represented inside the computer, because this is the basis of every microcomputer application. The data must extract the essential elements of the task, to be manipulated by the program algorithm (see paragraph 2.6).

2.5.1 Data Types

The first step in determining how data is to be represented in the computer is to identify on the different kinds of information that need to be stored, and to define a data type for each. The different values that each data type can take should be enumerated. If a system needs to work with the days of the week, for example, a type called "day" can be defined as follows:

```
type day = (Monday, Tuesday, Wednesday, Thursday,
            Friday)
```

At this stage it is neither necessary or desirable to worry about how this data type will be implemented. Data items of type "day" must be capable of taking five different values representing the days of the week. These items could be stored as the values 0-4, 1-5 or as arbitrary patterns of bits. That decision can be made later. At this point it is important simply to understand the problem.

Declarations such as the one above simply specify a data type providing a rule for translating information into data. Such decalarations do not reserve storage within the computer. Storage is provided by actually declaring data items or variables as in the following example:

```
var startday, endday : day
```

This statement declares two variables, or storage locations, of type "day", named "startday" and "endday". Whatever implementation is later decided on for "day", that amount of storage and that interpretation will be assigned to "startday" and "endday".

The advantage of declaring data types at the start can easily be seen. The definition is localized in one place, and can be changed with minimal disruption (for example by adding Saturday and Sunday). If design choices are clarified and isolated as above, flexibility can be carried right through to the final program. The alternative method of implementation is to postpone thinking about data types until the actual variables are required, declaring, for example:

```
var startday : (Monday, Tuesday, Wednesday, Thursday,
                Friday)
```

each time a new variable is required. This is not recommended, because changing the definition could involve searching through the whole of the software for each variable declaration.

The notations used as examples in this section are not intended for direct implementation on a computer. Rather, they are part of design language allowing systematic thought about data structures, without worrying about the implementation details (syntax, punctuation, etc.) required by a particular programming language. The underlined words are keywords in the language that define its basic constructs.

This design language can be compared to the logic diagrams used by circuit designers. As yet there is no standard for software design languages. The notations used in this and the following sections incorporate most of the features generally agreed to be useful in software design. There is no reason why a designer should not adapt these notations to suit his own needs. The main requirement is that the language chosen must be readily understandable to other designers as well as to the author should he reexamine his design plans at a later date. The notations used here follow the good practices advocated by leading practitioners of the art of software (there is a list of references at the end of the chapter).

Where the values of a data type follow an obvious sequence, only the start and end need be enumerated:

```
type weeknumber = (1..52)
```

Designing good data types is not an easy matter, and there is no standard way to go about it. This is perhaps the biggest challenge of software design. It involves abstracting the elements of the real world that are relevant to the particular problem. One approach is to try a number of alternatives before deciding upon a solution.

In a microcomputer, data is closely related to input and output. Decisions made about I/O will strongly affect the choice of data and vice versa. It is also impossible to design good data structures without some conception of how algorithms and programs work (see the next section). However, data design logically precedes algorithm and program design. Data must exist before you can perform operations on it. For some programmers, data occurs only as a byproduct of programming: it is never considered separately. This leads to bad software.

Microcomputer design is usually an iterative process. Each aspect of the design (I/O, data, algorithms) must be considered separately, with the implications for the rest of the design. This is the only way to approach such a multidimensional problem. Several passes may be necessary before the pieces of the jigsaw fit together and the design

finally crystallizes. All this needs to be done before the design gets beyond the paper stage.

A systematic approach means considering each aspect in turn, so that nothing gets omitted. Data is one of the most important pieces of the design puzzle. In the early stages of a design, it may be useful to consider several alternative ways of organizing the data. This approach is more likely to yield an optimum design than picking one method and staying with it. It is a good design exercise to vary one element of the design and consider the implications. With a field as new as microprocessors, it is often possible to find fresh ways of doing things.

2.5.2 Data Structures

Data types of the kind described above can be structured in various useful ways. Various data structures include the record, the field, the array, and the list. These data structures are defined below.

One of these structures is the record, which enables like data to be grouped together. A record is simply a collection of (probably dissimilar) data types. Consider an application that controls a number of gas pumps at a self-service filling station. A record can contain various information about a pump as follows:

```
type pump_record = record
     status  : (off, filling, completed)
     grade   : (regular, premium, unleaded)
     gallons : (0..30)
     end

var pump1, pump2 : pump_record
```

The type declaration defines the structure of the record; the var statement declares two record variables, pump1 and pump2, of this type. End closes the record definition. "_" is used to make pump_record into one word.

The record in this example contains three fields, each of which has a unique name. The status field for the first pump can be referred to unambiguously as "pump1.status". All of the information about this pump can be referred to collectively as "pump1". This is a very useful shorthand when dealing with large and complex collections of data.

The fields in a record can be of any type, including structured types. This provides the possibility of building very powerful data structures. Types of fields in a record can be predefined, e.g.:

```
type status_values = (off, filling, completed)

type pump_record = record
     status  : status values
```

The algorithm for this application involves continually checking the status field of each pump record in turn. When a status of "completed" is read, the program calculates the cost of the gas delivered based on the grade and gallons fields of both the record and a table of prices, and displays the calculated cost at the cash desk. When the account is paid, the status of the pump is reset to "off" and the cycle can begin again.

Another structured data type is the array. An array is an ordered list of elements of the same type that can be referred to by the name of the array and an index. The index specifies the array member's position in the list of elements comprising the array.

```
type buffer = array [1..80] of character

var buf1 : buffer
```

or, equivalently

```
var buf1 : array [1..80] of character
```

"Character" is a previously defined type. The number of elements in the array (80 in this case) is specified by listing the possible values of the index, in square brackets.

The fourth element of the array (i.e., the fourth character in the buffer) can thus be referred to as "buf1[4]"; this element is of type "character".

Note that 1..80 in the array declaration has the same form as the right hand side of a type declaration. In fact, a type name can be used in place of an explicit list of values. The index values need not be numeric. Thus, an array containing the daily receipts of a store can be declared:

```
var daily_takings [day] of money
```

the receipts for Tuesday can then be referenced by

```
daily_takings [tuesday]
```

Arrays can be employed for any list of identical items. The elements can be any data type, including records and other arrays.

Arrays are useful principally because they can be referenced using a variable as the index. For example:

```
buf1 [pointer]
```

where "pointer" is declared,

```
Var pointer : 1..80
```

These declarations can be combined into a third:

```
type buf_size = 1..80

var bufl : array [buf_size] of character

var pointer : buf_size
```

The above declarations make changes to the buffer size much easier and also aids in documentation. With an appropriate choice of names, designs such as this can be self-documenting.

With an index variable, the same portion of the program can be used to operate on different array elements, according to the value of the index. This is relevant to the gas station example (above). As it stands, a separate piece of program needs to be written for each pump. Instead of declaring pump1, pump2 as separate variables, declare an array of pump records:

```
type no_of_pumps = 1..10

var pump : array [no_of_pumps] of pump_record

var pump_no : no_of_pumps
```

The same program can then be used for any pump, first setting pump_no to the required value, then referring in the program to:

```
pump[pump_no].grade
```

For the grade field of the pump specified by pump_no. Notice how the notation works:

```
pump is an array
pump[pump_no] is an element of the array, and is a record
pump[pump_no].grade is a field of the record, and is of
                    type: (regular, premium, unleaded)
```

Any array can be indexed by adding "[index]"; any record can have a field specified by adding ".field". By nesting definitions in this way, these data structures provide the necessary tools for managing the complex data found in the real world. Once learned, the notation is a very powerful tool for managing the complex data found in the real world.

It is not necessary to grasp the whole of a large data structure at once. Beyond a certain point, it is mentally impossible. Using a technique such as this, if each level of the structure is correct and well understood, the user can be confident that the whole is correct. This is the principle on which most modern software design techniques are based, and it applies to algorithms and programs as well as data.

Returning to the gas station example, one problem appears in the original design. In order to save the cost information, a customer cannot use a pump until its previous customer has paid his bill. Several solutions, however, are possible. An array of pump records for each pump, one record per customer. A decision will then have to be made as to how many customers will queue at each pump. In another solution, the cost information can be stored in a separate data structure (or printed out) as soon as it becomes available, and the pump cleared.

A third possibility is to structure the data not by pumps, but by customers -- one record per customer. A customer record might look something like this:

```
type customer_record = record
     pump_number : no_of_pumps
     status      : (off, filling, completed)
     grade       : (regular, premium, unleaded)
     gallons     : (0..30)
     end
```

Each time a customer arrives, a new record is created. An array of customer records can then be declared. These records can be assigned to customers as they arrive. However, customers leaving will create "holes" in the array which will have to be filled. This problem can be solved (e.g., by a "tidying up" algorithm). Such a solution, however, is rather messy. In the array structure in this application there seems to be no obvious meaning for the index. This is one indication that an array is not the right structure to use in this application.

A structure, called the list, is more appropriate to the situation spelled out above. Lists, and other useful data structures such as trees, are described in more detail in the references given at the end of this chapter. Records and arrays must have their size (the amount of storage allocated to them) defined when the program is written. These allocations cannot be changed while the program is running. Lists, on the other hand, allow data elements (usually records) to be dynamically assigned from a pool, or heap, of storage space while the program is executing Elements can be deleted from anywhere within the list when no longer required and the storage returned to the heap.

The different solutions illustrate a point made earlier: that data can be structured in many ways, and it is worth exploring the alternatives. Data design determines the 'objects' with which the system will work and affects both algorithms and I/O. The best way to arrive at an optimum solution is to be aware of the choices that can be made.

2.6 ALGORITHMS

An algorithm is a list of instructions: a statement of 'how to do' something. More precisely, it is the specification of a finite number of steps required to achieve a desired end. A function can be performed by a computer if and only if that function can be stated as an algorithm. A sample algorithm for making tea might include:

```
fill kettle;
put kettle on;
put tea in teapot;
while kettle is not boiling
    twiddle thumbs;
fill teapot;
for number of cups required
    pour cup
```

Some of the operations described can themselves be analyzed into algorithms. For example, 'pour cup':

```
if milk is required
then
    pour milk;
    pour tea
else
    pour tea
```

The words underlined are not part of the basic operations; these keywords form Control structures that determine which part of the algorithm is to be executed, depending on some condition. If the same sequence of operations had to be carried out every time, computers would not be very useful. But a computer is capable of making a simple decision -- provided all the possible outcomes are enumerated in the program, and the computer is told exactly where to look to test whether the condition is satisfied.

The operation of an algorithm can be varied, depending on the state of some input or the result of a previous operation. By combining control structures such as the ones shown here, extremely powerful algorithms can be developed to control, for example, the operation of a complex scientific instrument or an industrial process.

There are various control structures that can be devised. However, it can be proved that any sequential algorithm (and any computer program) can be written using only three basic constructs -- sequence, selection and iteration -- all of which are included in the above example.

The sequence is the basis for all algorithms and all computer programs. It is so fundamental that there is no keyword associated with it. A sequence is simply a list of operations carried out one

after the other:

```
fill kettle;
put kettle on;
put tea in teapot
```

A sequence can be represented diagrammatically as follows:

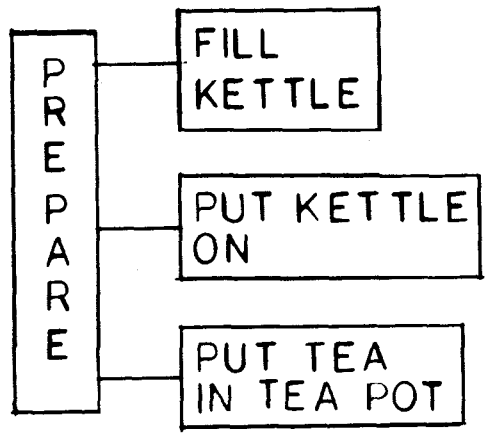


FIGURE 2-1. SEQUENCE

Which simply means: fill kettle, then put kettle on, then put tea in teapot. It is often useful to give a sequence a name, because it can then be treated as a single operation and included in a 'higher-level' algorithm. In the diagram, the long vertical box represents the sequence as a whole; the other boxes are the elements of which it is composed. The elements of the sequence are carried out in order, from top to bottom. The connecting lines show that these elements belong to that sequence (the lines do not indicate logic flow, as in a flowchart). The elements of a sequence might be simple operations, or they can themselves be any of the three basic constructs (sequence, selection or iteration).

The selection is a decision construct. Depending on a condition, one of two or more alternative operations is selected and performed. For example,

```
if weather is fine
then walk
else take car
```

diagrammatically, this is represented as:

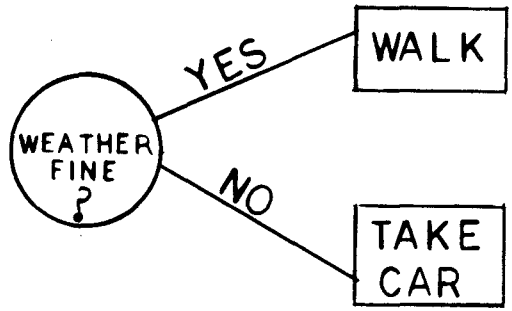


FIGURE 2-2. SELECTION

The circle represents the selection; that is, a single element which can be either of two things. The boxes are the components of the selection. For each execution of the selection, one and only one of the components is executed. Once again, the connecting lines express that the components are members of the selection (they are subordinate to it).

There is a selection in the example algorithm:

```
if milk is required
    then begin
         pour milk;
         pour tea
         end
    else
         pour tea
```

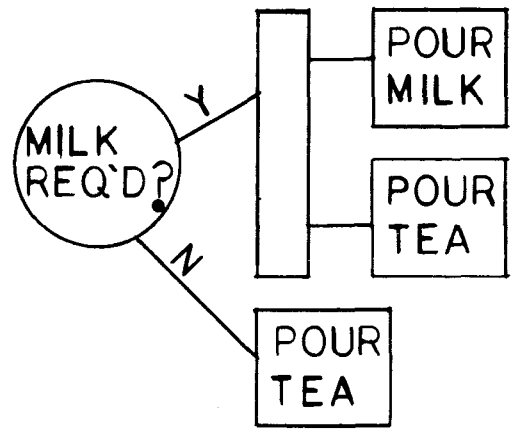


FIGURE 2-3. SELECTION WITH SEQUENCE

Here, the first alternative is a sequence of operations. The control words begin and end have been added to the verbal description to make quite clear that the sequence is regarded as one operation, as far as the selection construct is concerned. The words begin ..... end are used to bracket statements in the same way that parentheses are used to bracket numerical expressions:

$$5 \times (2 + 7) = 45$$

Note that only one of the alternatives is executed. When it is completed, the algorithm continues with the next operation after the *if* construct (if there is one).

A special case occurs when there is only one alternative, to be executed when the condition is satisfied. If it is not satisfied, nothing is done. This can be regarded as a selection in which one of the components is the null action, "do nothing". This component is usually left out of the diagram. In the example, 'pour cup' can be written:

if milk is required

then pour milk;

pour tea

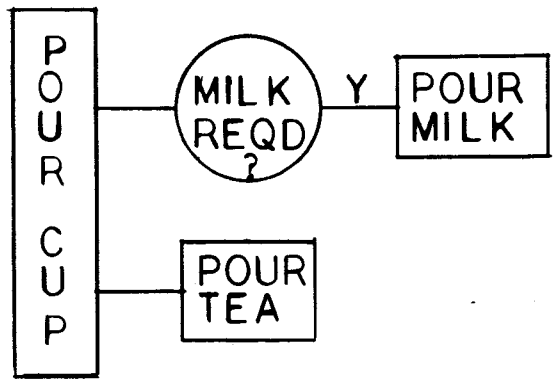


FIGURE 2-4. SELECTION: SPECIAL CASE

Here, 'pour cup' is a sequence consisting of an *if* construct (with only one alternative) and a simple operation. 'Pour tea' is always executed; 'pour milk' is executed only if milk is required.

This illustrates an important point about algorithms and programs. There are often several alternative ways of writing an algorithm to perform a particular function. It is worthwhile spending a little time determining which is the *best* solution. It is not sufficient that an algorithm works (in some circumstances): it must be clearly understood, and correct in all circumstances. The best algorithms are those that clearly reflect the structure of the problem.

Just as a good data structure extracts the essential elements of the information being represented, so a good algorithm extracts the essential elements of the process being performed and uses these elements as the basis of its structure.

It is possible to have a selection with more than two alternatives. This is represented in the design language by the *case* construct:

```
case weather of

     sunny:    go for walk;

     raining:  begin

                   put coat on;

                   go for walk

               end;

     snowing:  stay inside

end
```

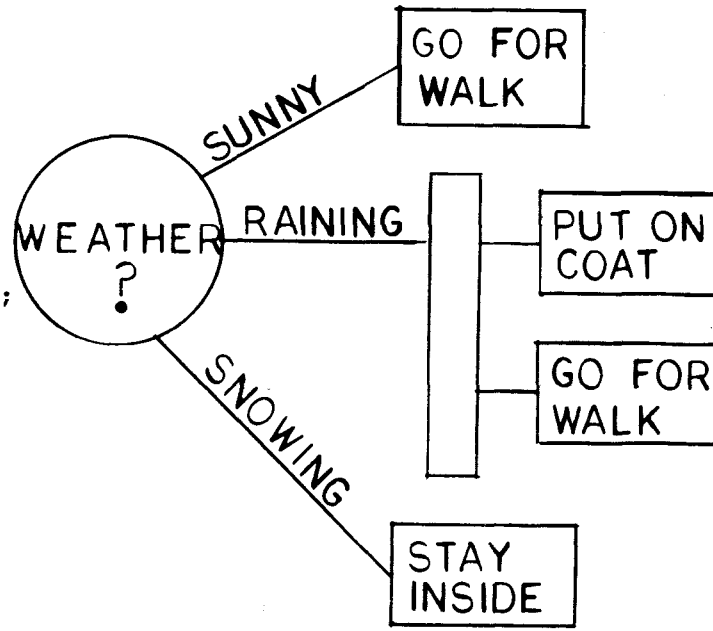


FIGURE 2-5. CASE CONSTRUCT

The case labels sunny, raining, snowing specify the possible values of the case expression weather (weather will have been declared as type (sunny, raining, snowing)), and the actions to be performed for each.

The case labels can specify a list or a range of values. There can be any number of case alternatives. Case constructs can have an otherwise clause that specifies an action to be carried out if the case expression has a value not expressed in any of the case labels:

```
case number of

    0..3,8 : add number to total;
    4,6,7  : subtract number from total;
    5,9    : divide total by 2;
    otherwise write ('number out of range')

end
```

Diagramatically, this is represented as:

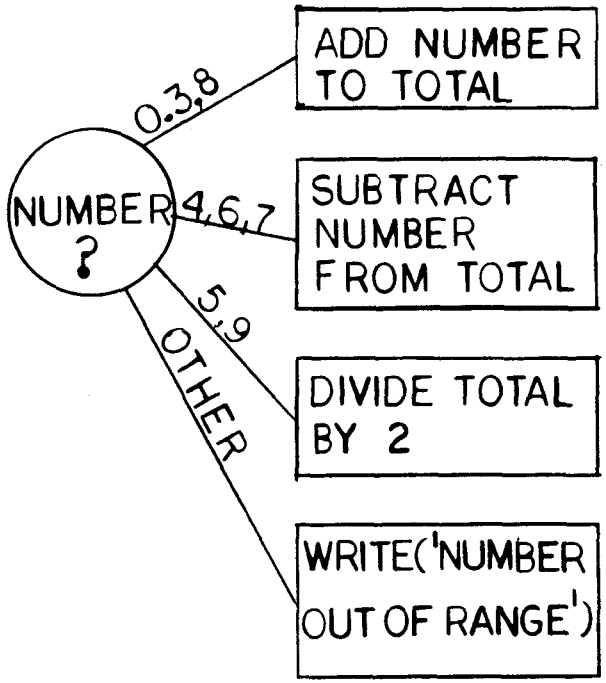


FIGURE 2-6. CASE WITH OTHERWISE

The iteration, or loop, is a powerful construct that allows an operation to be repeated either a specified number of times, or while some condition remains true. Usually a sequence of operations is repeated. Both forms of iteration are shown in the example:

```
for number of cups required
    pour cup
```

'Pour cup' is executed the required number of times.

```
While kettle is not boiling
    twiddle thumbs
```

This may not look like a repeated operation; but an algorithm is written on the assumption that only one thing can be done at a time. This is certainly true in a computer. While the executor of this algorithm is twiddling his thumbs, he cannot check whether the kettle is boiling. Therefore, after twiddling his thumbs for a while, he must return and test whether the kettle is boiling. If it hasn't boiled yet (i.e., condition is true), he can carry out some more thumb twiddling, otherwise he must continue with the next operation. The period set for thumb twiddling had better not be too long, otherwise the kettle will boil dry. This kind of consideration is often important in writing a real-time microcomputer program; and wait loops such as this are often required.

In the diagram, an iteration can be represented by a lozenge-shaped box:

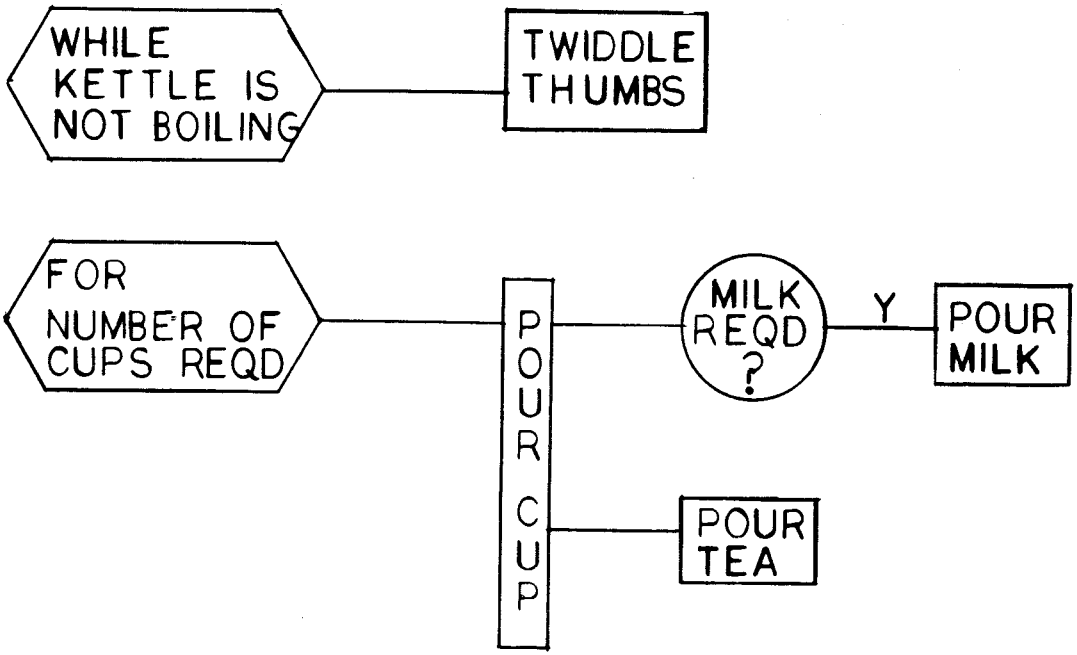


FIGURE 2-7. ITERATION

As most computer programs carry out some operation repeatedly (otherwise there would be little point getting a computer to do it), the iteration is a useful construct.

Although many programming languages provide additional control structures, it is wise to stick to the three described. Programs written using only these three constructs have been shown to be easily understood, easily amended, and above all likely to be correct. This discipline is known as structured programming, and is one of the most important techniques available to the software designer.

The three constructs presented here are basic mental structures, representing very closely the way the mind analyzes a problem. Consequently they are very easy and natural to "think in", once the notation has become familiar.

Other notations, such as flowcharts, have often been used for designing computer programs. Flowcharts may be useful at the lowest levels of implementation, when coding in Assembly Language for instance (see the next section). However, they are designed to

represent the way machines operate rather than the structure of an application. Trying to understand a problem using flowcharts involves bending the mind to work in the way machines do. This may be necessary at some point, but is not advisable in the earlier design stages.

Flowcharts concentrate on the details of process implementation, and have no way of representing hierarchical structure. Breaking a problem down into hierarchical levels (i.e., high-level algorithms that contain lower-level algorithms and so on) is the only way to understand it clearly, because the spread of information, from overall structure to fine detail, is so great. The notation used in this chapter may be unfamiliar to the user brought up on flowcharts, or used to programming in languages such as FORTRAN. It is worth making the effort to understand this notation, however, because of its value.

The diagrammatic notation used in this book was developed by Eric Richards, from an original notation designed by Michael Jackson (see References at the end of this section).

2.7 PROGRAMMING LANGUAGES

2.7.1 Assembly Language

The earliest computers were programmed directly in machine code; that is, binary digits. Each instruction in a computer is represented by a unique pattern of bits within a word of program code. For example, in the TMS 9900,

```
1010XXXXXXXXXXX   means   "add"
```

The X's carry other information and can be 0's or 1's. Some instructions require two or three words, because they contain data, addresses of memory locations, etc.

Programming in machine code is extremely tedious and very prone to errors. Therefore Assembly Language was invented. Using Assembly Language, a program can be written with meaningful mnemonics (e.g., MPY for multiply) instead of binary code for instructions, and symbols instead of numeric addresses for memory locations:

```
        C   @WORD1,@WORD2   COMPARE WORD1 WITH WORD2
        JEQ SAME            JUMP IF RESULT = 0 TO LABEL "SAME"
        .
        .
SAME    TB  7               TEST INPUT BIT 7
        .
        .
WORD1   BSS 2               RESERVE STORAGE (BLOCK STARTING
WORD2   BSS 2               WITH SYMBOL) FOR WORD1 AND WORD2
                            2 BYTES = 1 WORD EACH
```

Translation from Assembly Language to machine code, which must be done before the program can be executed, is a tedious but fairly straightforward process; the sort of thing computers do well. The translation is handled automatically by a computer program called an Assembler. This is one of the software tools that will be provided with a microprocessor development system.

One of the advantages of using an Assembler is that programs can easily be changed. For example, an extra instruction can be inserted in an Assembly Language program and the program simply reassembled. Inserting an extra instruction in a machine code program would involve going through the whole program changing jump addresses, because the position of all the code after the insertion would have changed.

2.7.2 High-Level Languages

Assembly Language, though a great improvement on machine code, still requires a problem to be translated to a large extent into terms before it can be programmed. Each Assembly Language instruction corresponds to one machine instruction, and the programmer must turn a selection construct, for example, into the low-level tests and conditional jumps that are the only things the computer understands. In addition, the programmer must manage all the resources of the computer, such as data storage.

High level languages (HLL) were introduced in an attempt to enable the computer to handle all these 'housekeeping' functions automatically, and to free the programmer to concentrate on the problem. One of the first high-level languages was FORTRAN, which stands for FORmula TRANslation. It allows programs to be written in a stylized language that combines elements of mathematics and English:

```
I = 5*J + 7
IF (I.EQ.27) THEN GOTO 100
```

I and J are variables which represent memory locations. But the programmer does not have to worry about where in memory they are; this is handled automatically by the compiler, which is a computer program that translates high-level language programs into machine code.

The input to a compiler or assembler is called source code; the output is object code. It is important to note that execution of a compiler or assembler is completely separate from execution of the resulting program. A compiler or assembler is a utility program used during software development, that translates a program written in a programming language into a machine executable form. In developing a microcomputer application, the compiler/assembler will run on the development system and the compiled or assembled program will be designed to execute on the target system.

In the FORTRAN example, the program will take the value stored in the memory location represented by J, multiply it by 5, add 7 and place the result in the memory location represented by I. If J contained 3, 22 would be assigned to I. The program then tests the value of I, and if it is 27 jumps to the place in the program that has the label 100.

It is much easier to write programs in FORTRAN than in assembly language. However, in some respects FORTRAN is still closer to the way a machine operates than to the way human beings think. The GOTO statement, for example, is obviously derived from the Assembly Language JMP; it is a machine construct and not a logical one.

Implementation of selections, for example, can be ambiguous, requiring GOTO statements and labels:

```
          IF (I.EQ.5) THEN GOTO 50
          .
          .
          .
          GOTO 100
    50    .
          .
          .
          .
   100    .
```

Not only is this confusing, but the order is inverted: the then action comes second. FORTRAN was designed before much research had been carried out into algorithms.

More recently, high-level languages have been designed with the intention of getting as close to the problem as possible. Many of these are based on ALGOL (ALGOrithmic Language), which was designed in the 1960s to be a natural language for writing algorithms.

One of the best modern high-level languages is acknowledged to be PASCAL. PASCAL has a coherence which some committee-designed languages lack. It implements most of the generally accepted good programming practices. Besides the basic algorithm constructs, PASCAL also has powerful data structures. Software designs can be turned into PASCAL programs with very little effort. A PASCAL program looks very similar to the design language introduced in Section 5:

```
TYPE NUMBER_RANGE = -128..127;

VAR MAX : NUMBER_RANGE;
    A   : ARRAY [1..10] OF NUMBER_RANGE;
    I   : INTEGER;
.
.

MAX := A[1];
FOR I := 2 TO 10 DO
    IF A[I] > MAX THEN MAX := A[I];
```

(:= is the assignment operator, read "becomes equal to")

2.7.3 Interpreters

Languages such as FORTRAN are compiled languages; that is, the source program is turned into machine code in a separate step (perhaps on a different machine) before it is executed.

With an interpreted language, such as BASIC, there is no separate compilation step. The program is not stored in machine code but in intermediate code which can be regarded as condensed source code with all unnecessary symbols removed. (During development the symbols are also stored so that the program can be printed out in source form and easily changed.) At execution time, the interpreter, which resides with the program in the target system, looks at each line of intermediate code, determines what it means and carries out the necessary action. The intermediate code is not executed directly; the interpreter contains machine code to carry out every operation that can be specified in the intermediate code, and it is this which is executed.

Intermediate code is much more compact than machine code; however, the overhead of the interpreter, must always be there also. Beyond a certain size, an interpreted program will take less memory than an equivalent compiled program. However, an interpreted program will run a lot slower, due to the extra work that must be done at execution time in actually interpreting the intermediate code.

BASIC is an extremely simple language in which it is easy to write programs. Development is also very easy and very quick because programs do not have to be compiled. They can be executed as soon as they are entered. The BASIC interpreter checks each line for syntax errors as it is entered, so mistakes are easy to correct.

Texas Instruments' POWER BASIC (see Chapter V) is designed to run on the TM990 range of microcomputer boards. A BASIC program can be developed and executed using, at minimum, one TM 990 board and a teletype terminal. No development system is required. BASIC provides a very powerful and inexpensive microcomputer system which is ideal for low-volume applications and experimental work.

TI's Microprocessor PASCAL (see Chapter IV) provides the user with the choice of executing either compiled or interpreted code on his target system. A Microprocessor PASCAL source program can be compiled to either interpretive code or 9900 machine code. The two versions will execute identically, apart from considerations of speed and code size. This feature allows the user to trade-off execution speed against memory, and to select which is more important for his particular application. Even if he selects machine code for the final version, interpretive code has a number of advantages when debugging the system.

2.7.4 High-Level vs Low-Level

Faced with the choice of which language is best, there is no easy answer. The solution depends on the application.

Low-level (Assembly) language allows the programmer direct access to all the features of the machine and thus the opportunity to write compact and efficient programs. To capitalize on this requires skill and time. The opportunity equally exists to make mistakes and to write inefficient programs.

High-level languages can shorten development time by a factor of 5 or more, and produce more reliable code. With a high-level language it is much more difficult to make expensive mistakes. High-level programs are more understandable (if properly written, they can be self-documenting), so that a project is less likely to be dependent on one programmer. Changes are easier to make in the late stages of a project. The cost is some code inefficiency because a compiler cannot optimize as much as a good assembly programmer. However, this becomes less true as the size of the program increases. Inefficiencies (and errors) may be introduced in a large assembly language program simply because of the intellectual difficulty of managing such a large amount of detail (especially when it is worked on by more than one programmer). Compilers do not suffer from this problem.

Restrictions on code size, particularly for high volume products, may dictate the use of assembly language in order to produce the most compact code possible. Unless this is the case, it makes sense to use a high-level language. Assembly language projects of more than a few K (= thousand) bytes should be considered very carefully because complexity increases very rapidly with size. (Some studies have estimated that complexity is proportional to the square of the size of the program).

For many projects, a compromise solution may be attractive. For example, the control aspects, where clarity of the design is important, can be programmed in high-level language, with assembly language routines for critical low- level areas such as input and output.

An alternative (or complementary) solution is to hand-optimize

compiler-produced code, once the program has been completely checked out; or even to rewrite it in assembly language after proving the design in (say) PASCAL. Both approaches have been used very successfully by Texas Instruments in internal projects.

2.8 MODULAR PROGRAMMING

With a project of any size, it is usually helpful to split the overall problem up into smaller tasks or modules which can be tackled separately.

When adopting this approach, two things must be considered:

1) The detailed nature of each module

2) How the modules will fit together to form a complete system.

To simplify the task of module interfacing, modules that are as self-contained as possible should be selected. In other words, the module boundaries should be drawn so that each module needs to communicate as little as possible with the other modules in the system. The ways in which each module interfaces with the rest of the system must be clearly defined.

The algorithmic notation described in Section V is a natural medium for designing such modular systems. A high-level algorithm can be written that describes the operation of the system. For example:

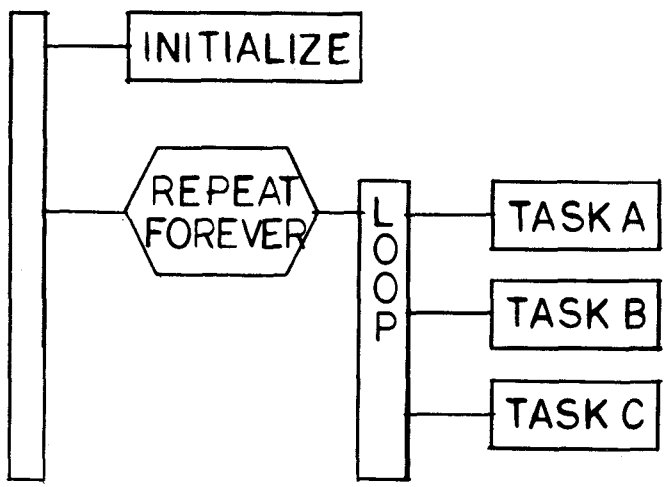


FIGURE 2-8. MODULAR SYSTEM

The terminal leaves of this diagram (INITIALIZE, TASK A, TASK B and TASK C), are natural candidates to be treated as modules and developed separately. The diagram defines how they fit together in terms of program structure. Their interaction with the data, however, still needs to be defined. Once the way they interact has been clearly specified, they can even be written by different programmers (INITIALIZE probably has to be written in consultation).

Each module can in turn be split into successively smaller modules, until the complete problem has been broken down into manageable segments. At every level in the structure, the modules can be regarded as 'black boxes' that perform specified functions and combine in clearly defined ways. The programmer can focus on a particular level in the structure, knowing that he can concentrate on the other levels at other times. If the ways in which modules can be combined to form larger modules are restricted to the three constructs described in Section V, the programmer can be sure that modules will not have 'side effects' or contain jumps to other modules that will upset the structure.

This hierarchical approach makes a complex problem intellectually manageable, and has been shown to lead to better, more correct, and more maintainable software.

The same approach can be applied to data using the structures described in Section IV. The data structures parallel program constructs; in fact, the diagrammatic notation, (described earlier) can also be used for data. The sequence construct can be used to represent records, and the iteration construct for arrays. Thus, the array 'pump' of 'pump_records' in subsection 2.5.2 can be drawn:

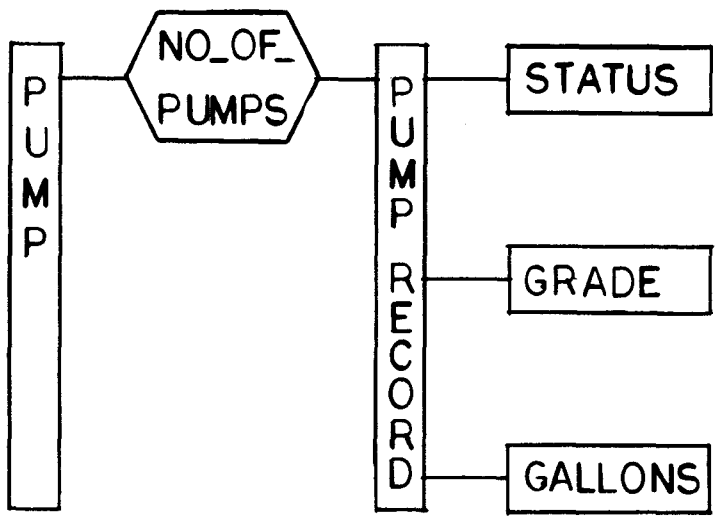


FIGURE 2-9. ARRAY CONSTRUCT

This means that data, too, can be treated in a modular fashion.

The selection construct can be regarded as representing the record variant, a record structure in which part of the record can have alternative forms. For example, a personnel record for a college might need to contain different information depending upon whether it represented a student, faculty member or a member of the administrative staff:

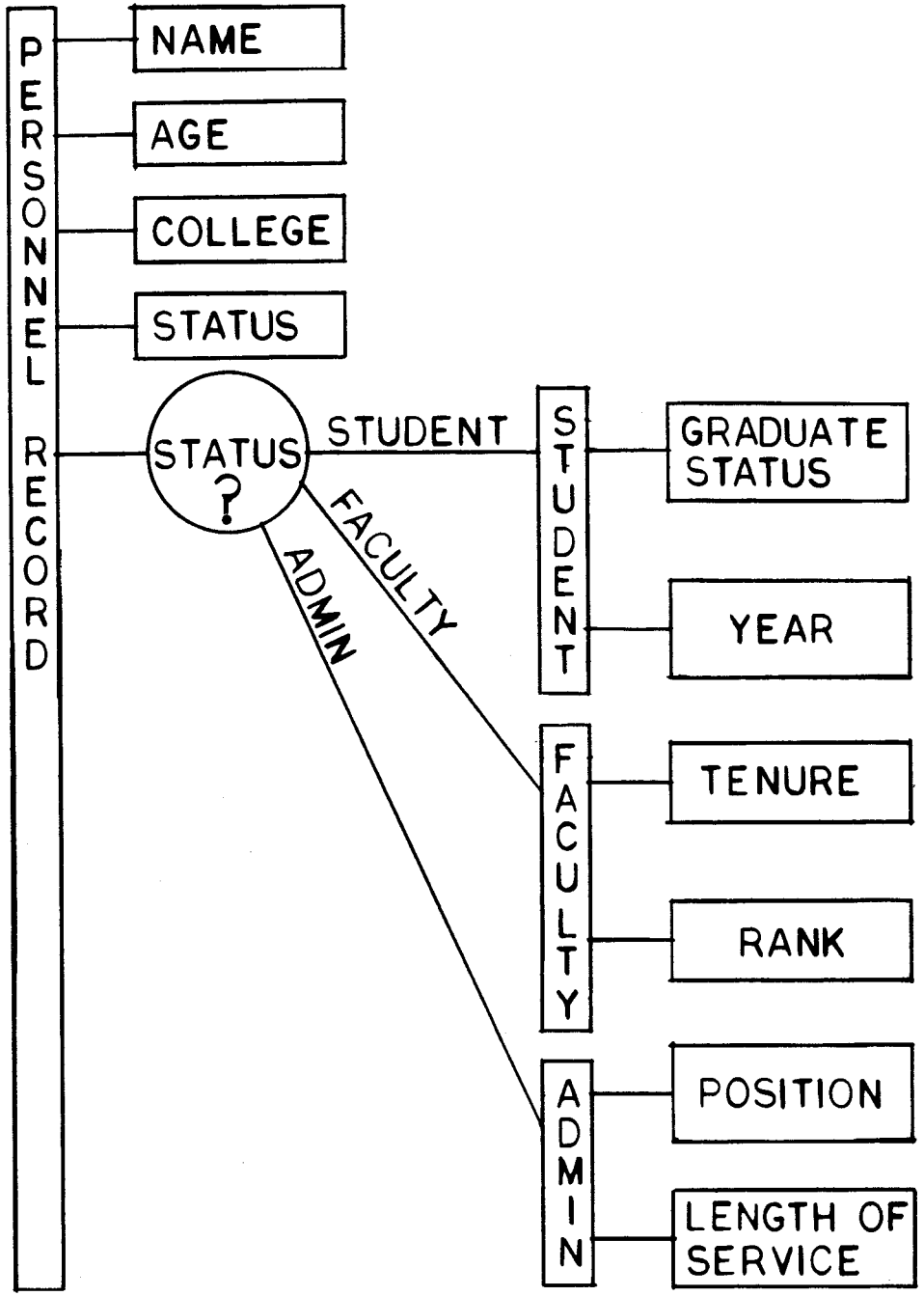


FIGURE 2-10. RECORD VARIANT

In the design language, this can be written:

```
type personnel_record = record
     name    : name_record;
     age     : 0..100;
     college : (cas, tech, music, jour);
     status  : (student, faculty, admin);
     case status of
          student : (graduate status : (graduate,
                         undergraduate);
                     year : 1..7);
          faculty : (tenure : boolean;
                     rank : (inst, asst, assoc, prof));
          admin   : (position : (asstdean, dean,
                         chairman, other);
                     length_of_service : 1..50)
          end
     end
```

According to the value of status (called the tag field), only one of the variants will be used to determine the structure of the record in any particular case.

2.9 PROCEDURES

A procedure (sometimes known as a subroutine) is a separate subprogram (a separate algorithm, or list of statements) that is declared within a program. A name is assigned to a procedure to enable the user to reference it.

Declaring a procedure is similar to defining a new statement or operation in the programming language. Once a procedure has been declared it can be referenced or called from the main program simply by writing its name. For example, if the programmer has written a procedure called CALCULATE_MEAN, to find the mean of a series of numbers, he can simply write

```
CALCULATE_MEAN
```

in the main program wherever this operation needs to be performed. (Some languages require a keyword, such as CALL, to precede the procedure name.)

In a case like this, the operation will probably have to be performed on several different sets of numbers which are stored as different variables. This can be accomplished by passing variable names as parameters to the procedure in order to specify the data objects on which it operates.

```
CALCULATE_MEAN (ARRAY_OF_NUMBERS)
```

Later the same procedure might be called by:

```
CALCULATE_MEAN (DIFFERENT_ARRAY_OF_NUMBERS)
```

When a procedure is declared, the number and type of parameters are specified in the procedure header. The variable names written here are used in the statements in the procedure body. They are the formal_parameters. When the procedure is executed (called), the formal parameters will be replaced by the actual parameters specified in the procedure call.

Procedure declaration:

```
PROCEDURE SEQ (A : INTEGER; B : REAL; C : ARRAY [1..80]
                          OF CHAR)
    BEGIN
    .
    .                     (* PROCEDURE BODY *)
    A := 5;
    B := 6.2;
    C[A] := 'P';
    .
    .
    END;
```

Procedure call:

```
SEQ (X, Y, Z);
```

The number and type of the actual parameters must exactly match the formal parameters. Thus, X must be declared as INTEGER, Y as REAL and Z as an ARRAY[1..80] OF CHAR.

A function is a procedure that returns a single value of a particular type. The type is specified in the function header:

```
FUNCTION NUMBER (A : BOOLEAN; B : CHAR) : INTEGER;
    BEGIN
    .
    .
    END;
```

and the function can be written as part of an expression:

```
P := 5 * NUMBER (TRUE, 'X')
```

Besides variables, values or expressions can usually be passed as parameters, provided they are the right type.

Procedures can declare local variables which are only used within the procedure. Usually, the procedure also has access to the variables of the program in which it is declared. (This depends on the programming language being used; in some languages, Pascal for example, procedures can be declared within procedures.)

Procedures form a natural method of writing modular programs, particularly if they can be nested (declared within other procedures) to any depth as in Pascal. In implementation, procedures save code. An instruction sequence that can be used in several places in the program only occurs once in the object code. When a procedure call is executed, the processor transfers execution to the procedure, saving the address of the the calling instruction in the main program. Once the called procedure has finished, the action returns to the instruction following the calling instruction, and action resumes.

Quite apart from code saving, procedures are a useful way of structuring a program, and may be used even when the procedure is called only once. In a block structured language such as PASCAL, variables declared within a procedure are completely local to that procedure, and cannot interfere with the operation of a procedure that is separately declared. (Procedures still have access to the variables of the program or procedure that contains them, so this has to be carefully controlled.) A procedure can even declare local variables with the same name as variables declared elsewhere in the program, and these variables will not interfere with each other. This means that, in a large application, program modules can be written by different programmers without risking incompatibility.

2.10 REAL-TIME SOFTWARE

Applications software for a general purpose computer usually has a beginning, a middle and an end. This software has a specific processing task to perform, and when this is complete, returns the resources of the computer to the operating system so that they can be allocated to another task.

Real-time software, on the other hand, usually has only a beginning and a middle. On power-up, the software will probably perform some kind of initialization procedure (such as clearing the data memory, setting all outputs to the required values, and perhaps sending a signal to the operator) and will then go into some form of endless loop, in which it monitors the inputs to the system and performs the required functions. Using the notation introduced earlier in this chapter, this can be represented as:

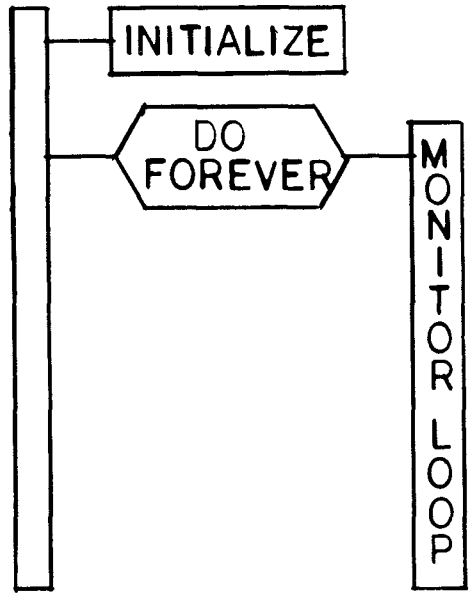


FIGURE 2-11. REAL-TIME SOFTWARE

There is no point in the software relinquishing control, because the system would then be 'dead' and unable to respond to any inputs. Real-time software only stops when the system is switched off. In fact, the software behaves rather like the operating system of a general purpose computer, sharing out resources to the different tasks to be handled, and remaining in control at all times. Computer operating systems can be regarded as a specialized application of real time software -- an application which has been extensively studied. Many of the techniques developed for writing operating systems can be successfully applied to other real-time situations.

The fundamental problem in designing software for a real-time system is that a sequential list of instructions (a program) must usually carry out a number of different parallel functions. A microprocessor can only do one thing at a time. How can it simultaneously handle all the inputs and outputs it has to deal with?

The answer, of course, is that the microprocessor performs its functions at a speed that makes it appear to be handling all functions in parallel. The system acts rather like a juggler keeping several balls in the air at once. It returns to give each one a push within a specified time, or a ball will fall to the ground. In a well designed system there should be no danger of this. The software is designed to service every requirement long before it becomes critical.

2.10.1 Software Organization

The challenge of writing real-time software is to organize all the loosely related tasks into a coherent whole that can be expressed as a sequential program. In most real-time systems, the processor must respond to a number of inputs which are asynchronous; that is, completely at random. The processor cannot tell, for example, exactly when an operator is going to press a button, or a sensor is going to register an input. In a typical real-time dedicated system, there will be a number of tasks, or processes, to be performed, each of which can be described by a sequential algorithm. The tasks may be completely unconnected, or they may be interrelated in a variety of ways. Some tasks will need to be performed at regular intervals (checking input lines, for example), others only infrequently, when a particular condition occurs.

For example, a system controlling an industrial process may need to monitor several different chemical reactions, and take corrective action if certain parameters are exceeded. The monitor function for each reaction can be described as a logically separate task.

There will also be a task associated with communicating with the operator and allowing him to display the status of each reaction, change parameters, etc. This task will need to pass information to and from the other tasks.

Two techniques have traditionally been used to convert a complex situation like this into a single sequential algorithm: polling and interrupts. A third alternative is to use an executive. An executive handles the details of software organization automatically, and allows the programmer to concentrate on writing the separate tasks and defining how they are related.

2.10.1.1 Polling. In a polled system, the tasks to be performed are simply written one after the other in the program. A polled program consists of an endless loop. As the program executes, it passes through check points corresponding to these tasks. At each check point, the program decides whether or not to perform the corresponding task. Once the list of tasks has been exhausted, the program begins execution again. The system deals with a asynchronous input simply by placing a check on that input somewhere within the loop, and checking (or polling) the input when it reaches that point. A polled system therefore does not respond to an asynchronous input immediately. The worst case response time corresponds to the maximum time taken to complete the polling loop.

Complex polling structures can be constructed in which some inputs are polled more frequently than others. However, this requires considerable thought, because the program structure required to do this is not naturally derived from the problem. In general, polling is a good technique for simple systems where immediate response to asynchronous inputs is not required. A complex polled system is

difficult to construct and not very flexible.

A simple polled system might look like this:

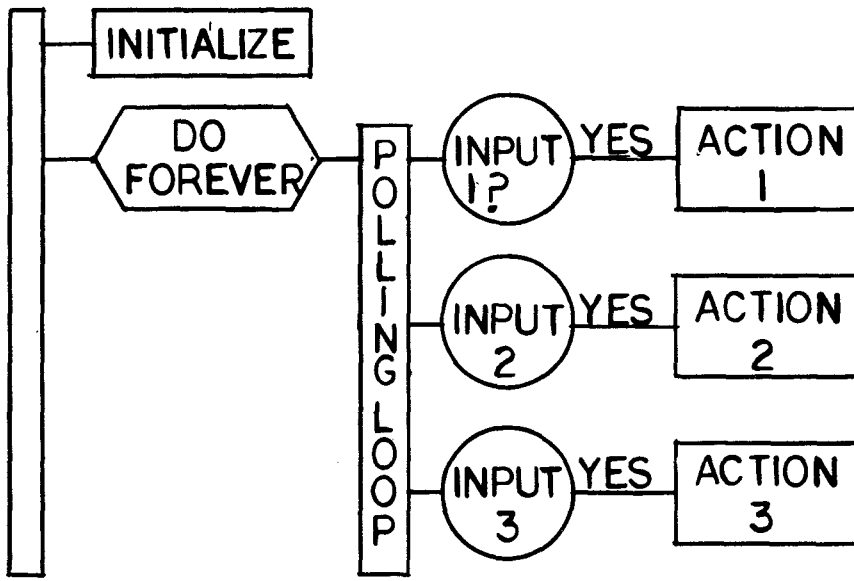


FIGURE 2-12. POLLING SYSTEM

The polling system displayed above carries out three separate tasks depending on the state of three inputs. Each task is completely transparent to the others (apart from the processing time it takes up).

Depending on how the system is constructed, the interval between polls of a particular input may be fixed or variable. In the previous example, the time taken for a single pass of the polling loop depends on what actions are performed. The worst case response time of the system is the maximum possible time between polls, plus the time needed to carry out the action. In evaluating a system design, it must be verified that the response time will be adequate for each action.

2.10.1.2 Interrupts. Some signals may require attention so urgently that they cannot wait to be polled. In this case a signal can be connected in hardware so that it interrupts the processor. The processor will suspend whatever it is doing and attend to the new signal. When the 9900 receives an interrupt it performs a context switch and executes an interrupt service routine written to deal with that particular interrupt. The context switch completely saves whatever the microprocessor was doing so that it can be resumed when the interrupt service routine is complete. (Reference the description of context switches in Section VI.)

The 9900 provides 16 levels of priority interrupts, so that very complex interrupt structures can be constructed. A higher priority interrupt can preempt a lower priority interrupt. While an interrupt service routine is executing, all interrupts of equal or lower priority are masked out (ignored). When calculating the response time for an interrupt, the possible masking effect of equal and higher priority interrupts must be considered. It is wise to keep interrupt service routines as short as possible (particularly high priority ones) because they prevent the system from doing anything else.

An interrupt routine is not part of the main program structure: it is a separate entity with its own structure which can temporarily 'borrow' system resources at any time during the execution of the main program. If there are portions of program which must not be preempted, interrupts can be temporarily disabled by setting the interrupt mask to zero. Interrupts are discussed in more detail in Section VI.

Many systems employ both polling and interrupts. Polling is an effective technique for small systems and for systems in which there are no tight timing constraints. Interrupts can introduce complexity into a system. With a complex interrupt structure it may be difficult to determine the exact behavior of a system; and hang-ups can be created. The classic 'deadlock' situation occurs when an interrupt routine waits for an event that can only be triggered by a lower priority routine. (The routine of lower priority cannot execute until routine has finished.) No way out of this situation exists;

obviously, software must be designed to avoid such deadlocks.

One common use for interrupts is to provide a time reference for polling. Consider a program in which a particular input or group of inputs require interrogation every 20 ms. Arranging the required checks may prove difficult in a complex system with a number of program paths. In order to satisfy the above polling requirements, an interrupt can be set to occur every 20 ms. For an input requiring polling every 100 ms, interrogation takes place every fifth interrupt. The 9940 has an internal timer which can be used to generate the interrupts; otherwise interrupts can be provided by the 9901 Programmable Systems Interface, or by using external hardware to divide the system clock.

When estimating system load, the average frequency of each interrupt must be calculated to determine the amount of time the processor will spend servicing interrupts. The minimum time between interrupts is also important, because if interrupts occur too close together, some may be lost.

2.10.1.3 Executives. To simplify the organization of real-time software, an executive can be used. Like high-level language and structured design, use of an executive is one more technique that brings software design closer to the problem. In particular, it allows a complex real-time system to be written as if it consisted of a number of separate, smaller processes executing simultaneously and in parallel. This simplifies software design, because it is closer to the reality of most real-time situations.

A system's processes can be regarded as competing for system resources; particularly for processor time. One of the most important parts of an executive is the scheduler, which allocates processor time among the various processes. There are various ways of doing this. One of the simplest is time slicing: each process in turn is allocated a fixed period, or slice, of processor time. The allocation is repeated cyclically.

Other scheduling techniques involve some notion of priority. A high priority process will be allowed to run in preference of a process with a lower priority.

A process will often need to communicate and synchronize its execution with other processes in the system. This can be done using the semaphore, which is basically a signalling mechanism between processes. A process may signal a semaphore to indicate that a particular event has occurred (for example, that a character has been received from an input device and placed in a buffer). Another process may be waiting for that signal (in our example, to take the character out of the buffer and print it). When the signal is received, the second process can be executed. Conceptually, both processes can be regarded as executing simultaneously.

A process waiting for a signal from a particular semaphore is said to

be suspended on that semaphore. More than one process can be suspended on a single semaphore. In addition, signals can be queued at a semaphore (for example, several characters can be placed in the buffer before any are processed). This gives the system a certain amount of elasticity.

Mechanisms such as semaphores support the scheduling of ready processes. At any one time, the highest priority process that is not suspended on a semaphore will be running. When this process cannot run any further (i.e., it becomes suspended on a semaphore and is waiting for an event), or terminates, the next highest priority process that is ready.

The executive maintains lists of processes ready but of lower priority than the currently executing process, and of processes waiting on semaphores. When a semaphore is signalled or the running process becomes suspended, the executive updates these lists and takes appropriate action.

Interrupts can be incorporated in such a system by treating them as signals to semaphores. The interrupt service routine is a process suspended on a semaphore, which is signalled directly by the interrupt. If the interrupt service routine is of a higher priority than the process currently executing (which will usually be the case), the interrupt routine will execute.

An executive makes it possible to design a system as if it were made up of a number of simultaneous, parallel processes, executing independently but communicating via semaphores:

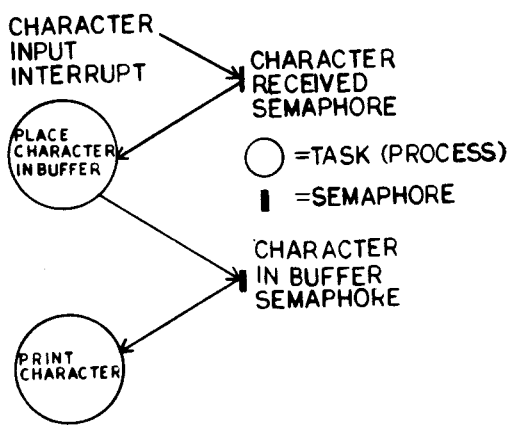


FIGURE 2-13. SEMAPHORE SIGNALING

Like all programs, the execution is in fact sequential. But the executive handles the work of adapting a sequential processor to a parallel world. It makes the design of real-time software much

simpler, and thereby makes possible more complex applications. For a complex real-time system, an executive is virtually a necessity.

Texas Instruments supplies a range of executives, tailored to different areas of application, for developing real-time software. These are designed to work with Pascal or assembly language, and can be adapted for use with other languages. The executive is normally supplied as a basic core or kernel, a library of routines implementing specific features. As a result, only those features of the executive which are actually used need be incorporated in the final system. In addition to semaphores, some executives provide more powerful synchronization mechanisms such as interprocess files.

CHAPTER III

SOFTWARE DEVELOPMENT

3.1 OVERVIEW

The end result of software development is a program — a pattern of bits residing in memory that instructs the processor what to do. To achieve this, several stages must be undertaken:

1) Definition of the problem

2) Design of the system - hardware and software, and how they will fit together

3) Design of the software (hardware development can be carried out in parallel

4) Programming the design (i.e., turning it into source program code)

5) Translating the source into binary machine code

6) Testing the software

7) Integrating the hardware and software

8) Evaluating the final system.

Each of these is an iterative process. Problems encountered at any stage may alter decisions taken at a previous stage, so that the true picture is probably more like this:

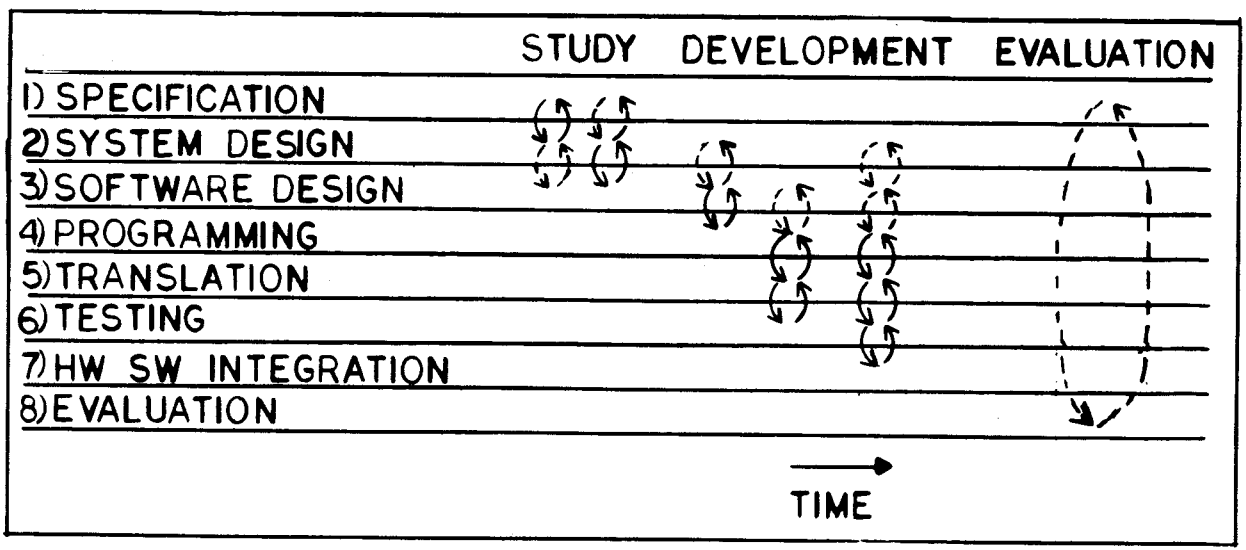


FIGURE 3-1. DEVELOPMENT STAGES

3.2 PROBLEM DEFINITION

The first step in development is to define the problem that is to be solved.

The way in which a problem is stated is often highly dependent on the implementation techniques believed to be available, that is, on the way similar problems have been solved in the past. Microprocessors have opened up a new range of possibilities. Therefore, when defining a problem, it is worth considering whether it can be restated to take advantage of the microprocessor's capabilities.

A microprocessor is both a programmable logic device and a computer. Where it is being used to replace conventional logic, its abilities as a computer may also be used to advantage, and vice versa. For example, a microprocessor might replace digital logic in controlling a scientific instrument. In this application, it can also be used to perform calculations on the results obtained by the instrument, something not easily achieved by digital logic. New forms of operator interface might also be considered; a keyboard and visual display screen, for example, rather than the traditional knobs and switches. The instrument might be given some degree of programmabilty to allow the user to set up a series of operations to be performed unattended. There are a whole range of new possibilities introduced simply by using a microprocessor.

A full problem definition for a microcomputer based product involves:

1) Defining the environment that is the devices and signals with which the product must operate, the operator controls and displays, and any special interfaces

2) Defining how the product reacts to this environment that is the actions it is required to take, the inputs it is required to respond to and the outputs it is required to produce.

This amounts to defining a black box (see Chapter 1, Section 1.2).

Once the black box has been defined, attention can be given to how to implement it. This is the field of system design. The system designer must decide how to integrate hardware and software, whether any special interfaces are required, if any additional hardware is needed (for analog to digital conversion, for instance), and so on.

Problem definition cannot be isolated from system design, particularly in such a new field. The way in which a problem is stated determines how the system will be designed, and vice versa. To extract the maximum potential from the technology, it is wise not to start with a rigid problem definition, and to be open to ideas that may come up in the system design stage.

3.3 SYSTEM DESIGN

Microprocessor system design differs from conventional digital design in that a microprocessor system is centralized. Hardware design consists simply of connecting all the inputs and outputs to the microprocessor, and ensuring that it has enough memory:

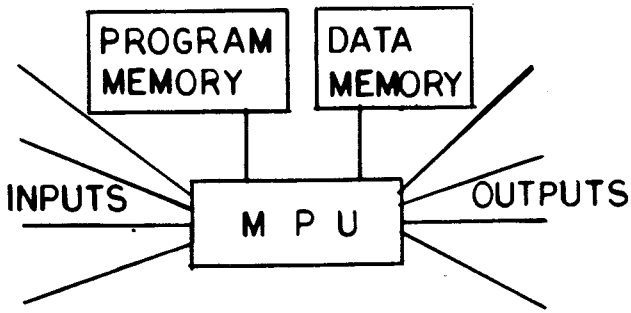


FIGURE 3-2. MICROPROCESSOR DESIGN

Determining the exact input and output requirements means a consideration of the software algorithms to be used (although they need not be programmed yet). Several iterations may be needed before a 'best fit' solution is achieved.

It is important to define the precise I/O configuration early in a project. From this base, both hardware and software designers can work. If the configuration is left at all vague, it is almost certain that the hardware and software will not work together or will work incorrectly.

Once the system configuration is established, hardware and software development can be carried out in parallel.

Some time can usefully be spent at the study phase of a project in sorting out the design issues. It is not necessary to make decisions at the outset, but rather to identify the choices to be made. The right solution can then be determined by investigation. Identifying (and documenting) various design choices at the beginning (as opposed to simply taking what seems to be the right choice at the time) facilitates backtracking when necessary. Therefore, it is worth keeping a record of the design process. Notes, and formal documents such as specifications, can be collected together to form a project notebook.

For example, an analog input (a voltage, for example) may be required. Decisions to be made include:

1) How much precision (i.e., how many bits) is required

2) How often a reading must be taken

3) What type of analog/digital converter can be used

4) Whether the input should be binary or coded decimal.

Decisions must be made on how much of the available information is required. For example, if a temperature is to be input, is the actual value required (to what precision?) or is a threshold indication enough?

Hardware/software trade-offs are important. When writing a number to a seven segment display, should the conversion from binary to decimal digits, and then from digits to the signals used to drive the display segments be handled by microprocessor software or by external hardware?

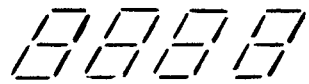


FIGURE 3-3. SEVEN SEGMENT DISPLAY
(4 DIGITS)

If processor resources are available, it makes sense to perform the conversion in software and save the cost of extra hardware. However, this depends on the processor having enough spare time to handle it.

3.4 ESTIMATING SYSTEM LOAD

One of the characteristics of a microcomputer system is that it can do only one thing at a time. If it is required to handle several things in parallel (as a real time system usually is) it must do so by handling each one in turn and at sufficient speed so that the effect on each is the same. An important part of specification is defining "sufficient speed". (For example: an analog input might need to be sampled every 5 ms, this being the minimum period in which it could change significantly in a particular application). An important part of system design is to determine that the processor can meet these specifications. Given a number of real-time tasks to be performed can the processor satisfy all of them simultaneously?

A useful measure of this is system load, which can be defined as:

$$\frac{\text{Processor Time}}{\text{Real-Time}}$$

For a given task, the load on the system is the processor time taken to perform the task, divided by how often the task must be performed (the "sufficient speed" specification). If the processor spends 2 ms carrying out a particular task, and the task must be performed every 10 ms, this represents a .2 or 20 per cent system load.

The total system load can be obtained by calculating the system load for each task that must be performed, and adding them together. System load is not a foolproof test of a design's practicality; but it does give the designer an indication of the magnitude of the task, and quickly shows up impossible specifications. Estimating the load for a given task involves a consideration of the software algorithm that will be used to perform it. This need not be very detailed at this stage. A rough calculation often shows that use of system resources is dominated by a very small number of tasks.

An estimation of 0.1 per cent could be out by a factor of 5 without making too much difference; a task calculated at 25 percent however needs careful evaluation. Usually, it is only necessary to look at a very small portion of program, which can be coded experimentally if necessary.

If the total system load comes out at more than 50 percent, the design should be reconsidered. There are two reasons for leaving a wide margin:

1) To allow for errors in the estimation, and for modifications to the software

2) Most systems have a degree of randomness: the average rate at which things happen may be predictable, but it may sometimes be exceeded by quite a large amount. It is wise to leave some power in reserve to deal with bursts of activity.

Besides the raw estimates of system load, timing constraints need to be considered. The straightforward estimate assumes that processing time is spread evenly over real-time. If the system needs to do a great deal within a period of 1 ms, and then nothing for 50 ms, this obviously must be taken into account. In this case, the load during the 1 ms period should be evaluated separately.

If the system load does come to more than 50 per cent, there are several alternatives:

1) Unload some of the work from software to external hardware

2) Reduce the specification of the svstem

3) Consider using a more powerful processor, or adding a second processor.

If the system load comes out very low (less than 1 per cent, for example) it is not necessarily a bad, provided design and cost criteria are met. However, if there are tasks being performed by external hardware that could equally be done in software, this is worth considering. Microprocessors have become inexpensive enough to make it economically feasible to have them lying idle most of the time. On the other hand, having to redesign because design parameters

have been pushed too far can be expensive.

Once the load has been calculated and the design fixed, the design engineer needs to beware of 'creeping enhancements'. Microprocessor systems follow a revised form of Parkinson's Law: designs expand to fill 150 percent of the resources available. To avoid this, the designer needs to evaluate carefully the effect of suggested enhancements, and consider them in relation to his loading estimates - which can be checked experimentally once the design is built.

3.5 SOFTWARE DESIGN

Software design consists of turning the specifications of what the processor is to do into precise software algorithms and data structures, using the system configuration established during system design.

The basis of software is data, since this represents the information that will be manipulated by the algorithms. A system uses two types of data: input and output data, which is the system's means of communication with the outside world, and stored data, which is held in memory and represents those things of which the system must keep a record.

The first task of the software designer should be to determine:

- What data is required
- How it should be organized (structured).

The data should be structured to reflect as closely as possible the information it represents. This involves:

- Identifying those aspects of the information which are fundamental and not superficial
- using these as the basis for structuring
- wherever possible using structures instead of single unrelated data items. This makes the software more coherent and more manageable.

If this is done, then both data and program will be clearer, and easier to change if the requirements are modified.

'Data' (Chapter II, Section 2.5) considers data structuring in detail, and gives some examples.

Once the data structure has been established, the algorithms that will operate on the data can be constructed. Algorithms are described in detail in Chapter II, Section 2.6.

3.6 TOP-DOWN DESIGN

A completed software design consists of a complex multi-dimensional mass of information, ranging from overall structure to details of implementation. When constructing such an edifice from scratch, what is the best way to approach it?

At the start, two 'ends' of the problem are known:

1) What the system is supposed to do, and

2) The basic operations (i.e., instructions) the processor is capable of performing.

This leads to two approaches to software design:

1) Starting from the problem and working down towards the details of implementation. This involves splitting the problem into smaller segments, considering each in turn and further subdividing until the basic processor operations are reached

2) Starting from the basic operations, putting them together into larger units that will perform more complex operations, and so working up towards a solution of the complete problem.

The second method is the traditional way of designing software. It has been called the 'bottom-up' approach. For example, if it became clear that a system required a keyboard input routine and a display routine, these would be written, together with other routines, and used as building blocks to construct larger modules which would then be put together to make the complete system.

However, it has been found by experience that the first method, 'top-down' design, produces software that is better, clearer and easier to maintain. The problem with bottom-up design is that usually not enough thought is given to the way in which the blocks will fit together before constructing them. Therefore the designer ends up with blocks that are not exactly the right size or shape, and he either has to reconstruct the blocks, or (to pursue the analogy) use a lot of mortar and build a system that is not very robust, and is difficult to change without toppling the whole structure.

Actually, the situation is not quite as clear cut as this. Pure bottom-up design is not possible, because the designer must have given the problem some 'top-down' thought or he would have no idea what building blocks to construct. But traditionally this was not expressed (largely because there was no language or notation to express it in), and what is not expressed cannot be clearly thought about. The only language available to write down a design was program code. There was, therefore, a strong temptation to start programming very early, before the larger design issues were properly worked out.

Design languages and notations like those introduced in Chapter II, Sections 2.4 and 2.5 solved this problem. It is these notations which make top-down design possible, by allowing the designer to have a concrete grasp of his design at all levels, and to postpone consideration of details until important design issues have been worked out.

A design might be conceived initially like this:

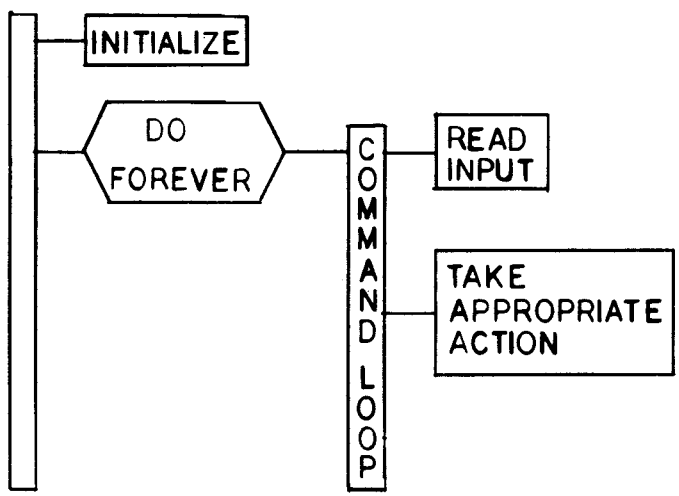


FIGURE 3-4. INITIAL DESIGN

This could be a device which, after initialization, would wait for an operator command, perform the appropriate action, and then return to wait for the next command. The device is specified in very general terms, but its basic operation is already clear.

The operator interface might be a teletype keyboard, on which the user would type a command telling the system what to do. Suppose a command consists of a line entered on a teletype keyboard, terminated by a carriage return (CR). The device prompts the operator for a command by outputting '?' to the teletype. READ INPUT could then be expanded like this:

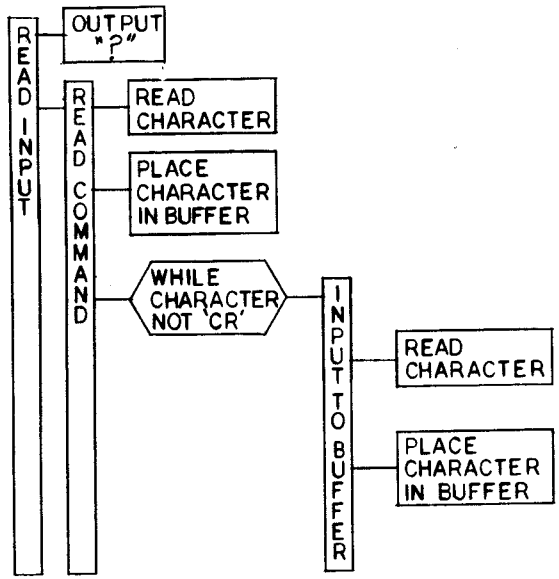


FIGURE 3-5. READ INPUT

The terminal boxes of this diagram can be further expanded when the design reaches that level of detail.

Because of the single entry and exit properties of the constructs used, the designer can be confident that however he expands the design of, for example, the box labelled 'TAKE APPROPRIATE ACTION', it will not affect any of the other boxes in the diagram, or the structure of the diagram. (This cannot be said of flowcharts, which is why it is difficult to use flowcharts without descending to the detailed level.)

It is this property of structured notation which makes it possible to hold off consideration of details and to design in a hierarchical fashion, from the top downwards.

In a practical system, top-down design must often be tempered with bottom-up considerations. It is impossible to start designing at the top without some idea of what is possible at the bottom. For example, it may be necessary to code and try out an I/O routine or a critical piece of code, in order to check the feasibility of the design. With a complex problem, it may be necessary to attack the intractable mass in the middle from both ends. However, the most important progression in design is from problem towards implementation. The reason for stressing it here (and elsewhere) is that, traditionally, software has not been designed this way - which is as logical as starting to build a house before the plans have been drawn.

3.7 PROGRAMMING

Programming involves turning a software design into source program code, following the syntax rules of a particular programming language. The amount of work involved depends on the programming language selected for implementation.

Pascal was designed as a problem-oriented language incorporating modern design techniques. Turning a software design into Pascal should involve little more than formalizing it and writing it to conform to the syntax rules. The constructs used in design can be implemented directly in Pascal. The routine work of translating the design into machine instructions is handled by the compiler.

BASIC, like Pascal, is a high-level language that handles much of the routine work (data allocation, for example) of translating the design into machine terms automatically. However, BASIC is designed as a simple language and is not quite as powerful as Pascal. It does not provide all the design constructs in a directly usable form.

BASIC does have other advantages. Being simple, it is easy to learn. As an interpreted language, it has special characteristics which are explained in Chapter V. Because it is designed to run on the TM990 range of microcomputer modules, a design can be developed very quickly and cheaply using standard hardware and a very low cost development system. BASIC is ideal for experimental and low volume designs.

Assembly Language is the most powerful, the most time consuming and the most difficult alternative. It gives the programmer complete control over all the resources of the microcomputer, but to exploit this control requires skill and discipline. Program development also takes much longer than in a high level language. Assembly language should be used where code size and efficiency is crucial (for example, for a large volume product). It can also be used to code critical areas of a program written in a high level language (I/O routines, for example). In general, assembly language can be used very effectively in small areas; large programs quickly become unwieldy.

Selecting which language to use depends very much on the application, the development facilities available, the development timescale, and the skills of the programmers. The remaining chapters of this book describe each language in more detail. They are not intended to be a complete description, but rather to give a feel for each language, so that the designer can select which one best meets his needs.

Programming, or coding, is a relatively mechanical process which involves expressing a software design in a precise, unambiguous form that conforms to strict syntax rules. The real creative work of development is done at the system design and software design stages. When choosing which implementation language and what type of development system to use, the designer is choosing how much of the programming process will be handled automatically by software development tools (compilers, linkers, etc.) and how much will be done by a human programmer.

3.8 TRANSLATION

Having written a program on paper, it must be physically entered into the development system in a machine readable form. This will be done by typing the source program in at a keyboard. The source program, which is in a programming language, must then be translated into machine executable form - that is, a pattern of binary 0's and 1's corresponding to the microprocessor's instruction set.

The steps involved in this process, and the concepts behind it, are covered in this section. Utility programs (software tools) are used at various stages in the process to perform particular steps: these are described, too.

3.8.1 Files

Much of the mechanics of program development consists of manipulating files on a development system. A file is a sequential list of information held on a backing storage device (disc, magnetic tape, etc). A file can be read as input data by a program running on the development system; the program can write back a file of output data.

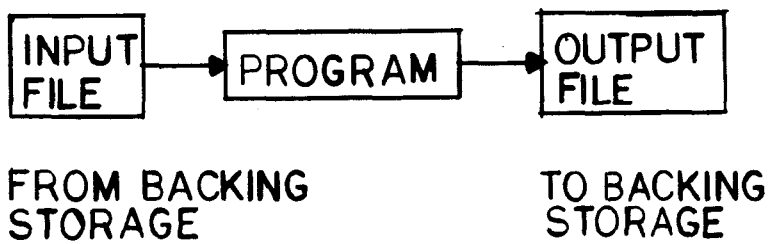


FIGURE 3-6. I/O

Utility programs are provided with a development system to perform many of the tasks associated with program development - for example, translating source code written in a high-level language into machine code that can be understood by the microprocessor. The source code is held on a file in backing storage; the machine code is written to another file.

A utility program may have several input and several output files, depending on the function it performs. An output file need not go to backing storage: if it contains textual information it might be sent directly to a printer. Similarly, an input file might be taken from a card reader or even be typed in at a keyboard - they both provide sequential information to

the program.

Utility programs are the principal tools of the software engineer. Once a design has passed the paper stage, it will consist of files held on the development system. Stored in this way, the design is manipulated using various software tools (utilities). To a hardware engineer, for instance, this medium may be unfamiliar; however it has a number of advantages over circuit diagrams, printed circuit boards and soldering irons. In particular, it can be manipulated by computer programs, thus partly automating the design process.

3.8.2 Text Files

In order to store textual information in a machine which recognizes only binary digits, some form of code must be used - that is, some rule for transforming textual information into binary data. The code adopted for the 990 and 9900 series is ASCII (American Standard Code for Information Interchange). The ASCII code specifies a unique bit pattern (number) for each member of the ASCII character set - letters, digits, punctuation marks and control characters. 7 bits are sufficient to uniquely identify an ASCII character. ASCII characters are usually stored one per byte (8 bits), with the most significant bit often being used for error detection (parity check).

Character	ASCII code	
	Binary	Hexadecimal*
A	01000001	41
T	01010100	54
1	00110001	31
5	00110101	35
?	00111111	3F
line feed	00001010	0A

This means that textual information can be held in memory, saved as a text file on backup storage and manipulated by utility programs.

It is the input and output devices (Visual Display Unit, printer, etc.) that recognize '01000001' as 'A', and so on. They translate key presses into ASCII coded data, and coded data back into displayed and printed characters.

Program manipulation of textual data is normally limited to moving it around in memory (to insert or delete text), searching for particular sequences of characters, and similar operations. Arithmetic operations on text do not make much sense.

---

* For the hexadecimal number system, see Section VI.

Numbers (decimal, hexadecimal or otherwise) can be represented in text as a string of digits. However, the bit pattern representing these digits in the computer is a code and bears no direct relation to the binary representation of that number - which the computer would use to perform any calculation. It is possible to carry out arithmetic with coded numeric data, but only by writing a series of subroutines to do it.

In many programs (particularly those which communicate with a user) there is an application for converting coded digits into binary numbers, and vice versa. In a large computer, this is usually handled automatically by standard input and output routines. The microprocessor user normally has to design his own input and output, and write the routines himself.

Program development, as far as the user is concerned, consists largely of manipulating text files on a development system - text files which represent program code.

3.9 SOFTWARE TOOLS

3.9.1 Text Editor

A text editor is a program which allows the user to enter text at a keyboard, and save it in a file on backup storage (cassette, floppy or hard disc). The text will usually consist of source code in assembly or high level language; however most editors will allow any kind of textual information to be entered. An editor also allows the user to modify text (hence its name) by entering editor commands at the keyboard.

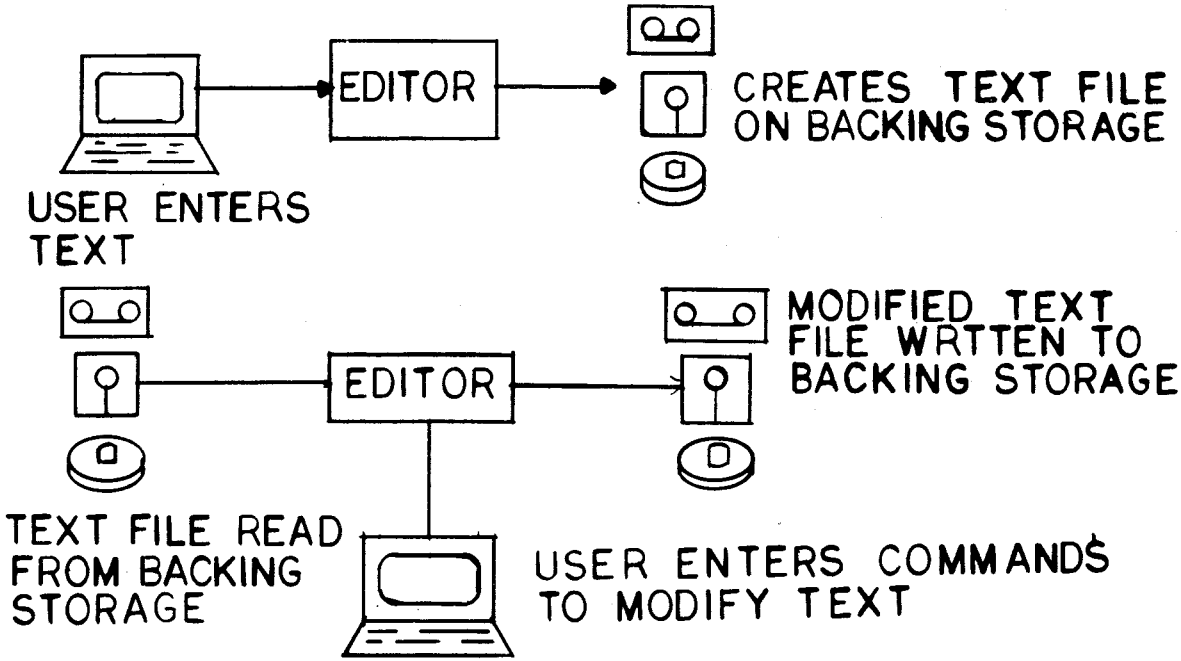


FIGURE 3-7. TEXT EDITOR

3.9.2 Assembler

An assembler converts assembly language source code into object code, for execution by the microprocessor. The input to the assembler will normally be a text file created by the editor. The output will be a file of object code. The assembler also generates a listing file, which is a text file containing details of the assembly, and any error messages.

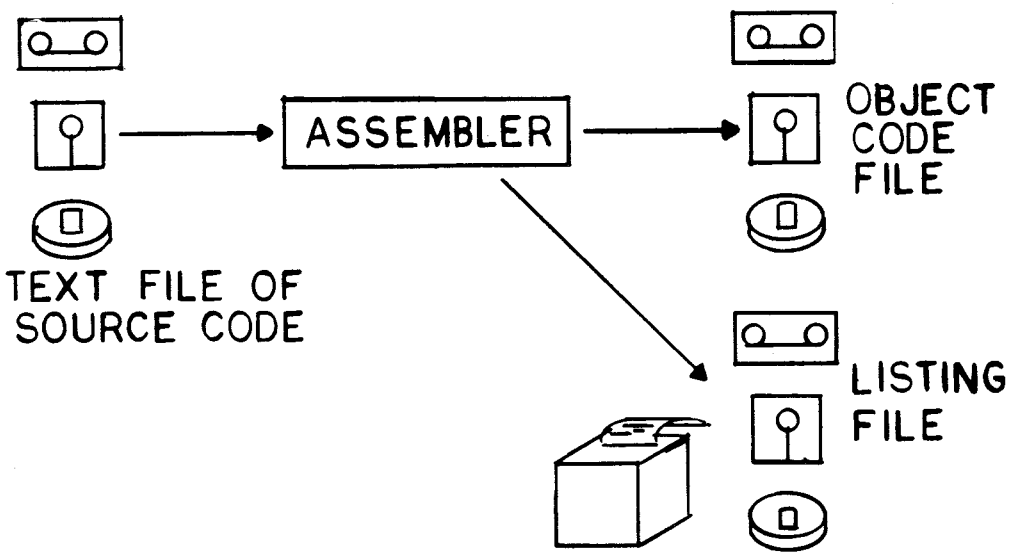


FIGURE 3-8. ASSEMBLER

3.9.3 Compiler

A compiler performs the same function as an assembler, but its input will be source code written in a particular high level language. Some compilers produce object code (machine code) directly; others generate assembly language source, which must be run through an assembler to generate object code. This is an extra step. but it does give the user the option of hand optimizing the compiler output before it is assembled.

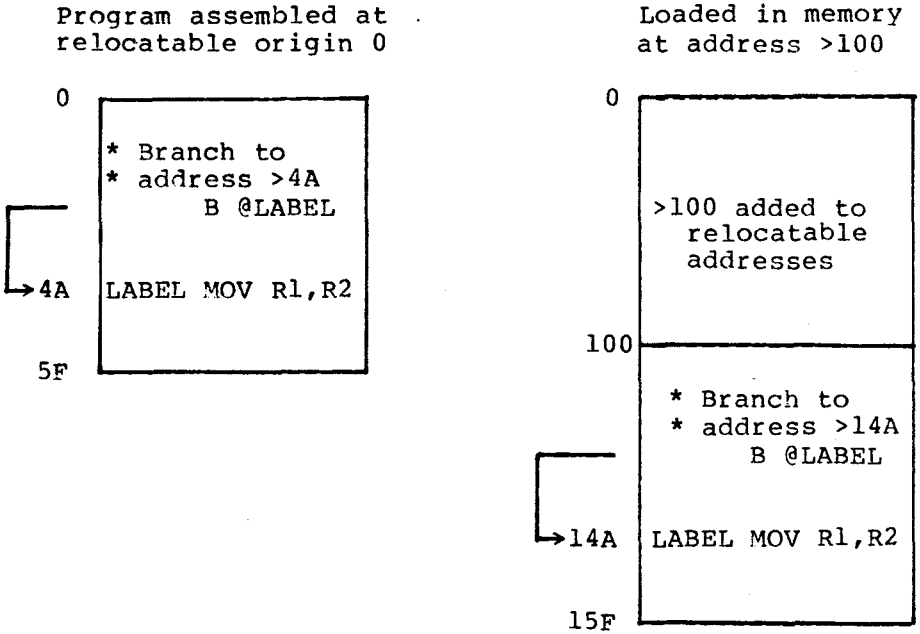
The TI Pascal compiler generates object code; but hand optimization is allowed for by providing a reverse assembler, which converts the compiler output back into assembly language source.

3.9.4 Absolute and Relocatable Code

Before a program can be executed, it must be located at a particular place in memory. Addresses in a program refer to particular memory locations, and the right data or program code must be present at those locations for the program to work.

Some assemblers for the 9900 (the Line-By-Line Assembler for example) produce only absolute code; that is, the position of the code is specified at the time of assembly, and cannot subsequently be changed.

However, other assemblers can produce relocatable code. Program and data addresses are calculated relative to the program base address - usually 0. Address fields are specified as "relocatable" in the object code output. When the program is loaded for execution, starting at, for example, address 100, the loader program can add this value to all the fields tagged "relocatable" so that the program will execute correctly.



Relocatable code allows the programmer to postpone deciding where the program will be located until the time comes to load it. This can be very useful when a system is being constructed from a number of different program modules. Each module can be assembled separately without needing to calculate exactly where it will fit in memory - which would involve knowing the lengths of all the other modules. More important still, one module can be changed (perhaps increasing its length) without the need to reassemble all the others in different positions to make room for it.

A system consisting of more than one module will probably need to be linked as well as loaded - see the following section for more information on the linker.

3.9.5 Linker

A linker, or link editor, is a program which will combine separately compiled or assembled object modules to form a complete system.

With a system of any size, it is much easier to break the program down into modules which can be written separately. Usually, these modules will be chosen so that each performs a fairly self-contained function and can be treated as a logical unit.

The interfaces between these modules - that is, the way that they will fit together to form a complete system - must be carefully considered when the system is being designed. Modules will often need to use programs or data contained in other modules. These can be defined as external references to symbolic names: they will be indicated (tagged) as unresolved addresses in the object code. Definitions to be used by other modules will also be included in the object code. The linker connects together, or resolves, these loose ends by linking references with their corresponding definitions.

Modules to be linked will usually be relocatable. The linker stacks them one after the other in memory, adjusting all the addresses accordingly. Output from a linker can either be a larger relocatable module, or absolute code, designed to be executed at a particular position in memory.

Linkers and relocatable code make a great difference to software development. It is possible to break a project down into manageable modules. One module can be changed without reassembling/recompiling the whole system. The linker automatically takes care of changes in module size and in the addresses of external variables. This can save a great deal of time (and money) in developing software.

A linker also allows the use of libraries of standard routines. It can provide, for example, mathematical capabilities or run-time support for a particular programming language. A library consists of a number of different modules, which can either be written by the user or supplied by a manufacturer. These modules are stored as relocatable object code. A user can reference any of these modules in his program; when the time comes to link, the linker will automatically select from the library the modules required by the program, and link them into the system.

With a linker, some modules can be written in high level language and others in assembler, according to their characteristics. This makes possible a very flexible approach to system design.

3.9.6 Loader

A loader is a software utility that loads an executable program from some form of backup storage into read/write (RAM) memory, for execution by the processor. It is therefore used on a general purpose computer rather than a dedicated microcomputer system, where the program is likely to be already in ROM memory and does not need loading. However, during debugging it may be necessary to load a program into RAM memory in a development system for test execution.

Some loaders are relocating loaders - that is, they can take a relocatable object program from backup storage and place it at any specified position in memory, adjusting the addresses tagged 'relocatable' so that the program will execute correctly.

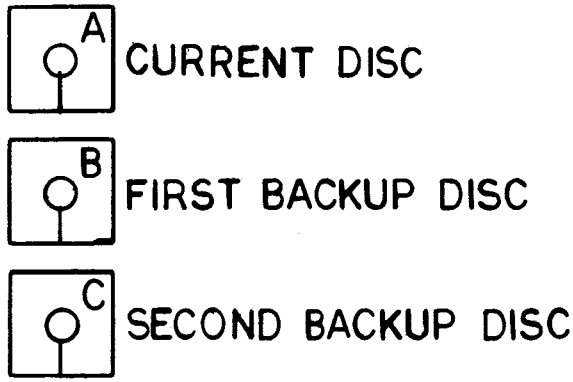
3.10 BASIC PROGRAM DEVELOPMENT

BASIC program development is a little different from any of the other languages because a separate compilation/assembly step is not required. A BASIC program is entered in source form using an editor which is part of the BASIC system. The program is stored in a condensed source form which is directly executable by the interpreter. This makes BASIC program development particularly simple. It is described in more detail in Chapter V.

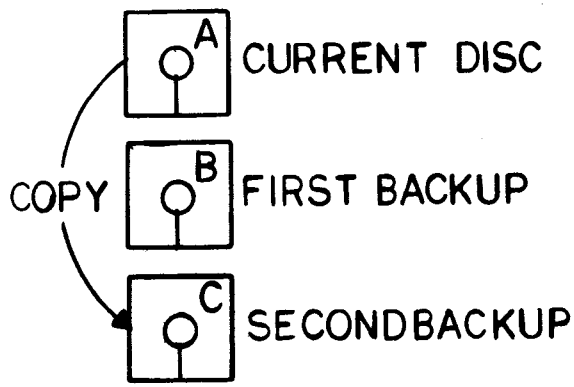
3.11 BACKUP

Once programming has begun, the work of the software designer will be held entirely on files in backing storage. While storage media are inherently very reliable, errors do occasionally occur (due, for example, to dust accidentally getting into a disc drive) which can wipe out days or even weeks of work. It is therefore necessary to have some form of backup for important files - an extra copy, stored away from the computer. There are many ways of doing this: for example, copying files at regular intervals to magnetic tape or paper tape.

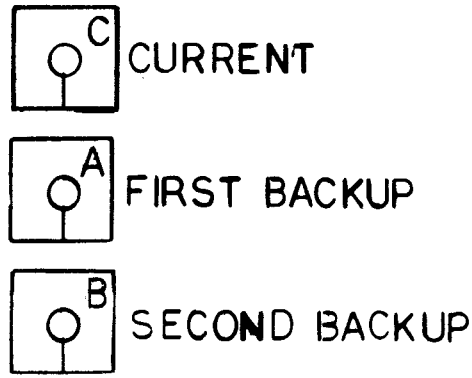
One method which works particularly well for floppy disc-based systems, and can also be used for hard discs, is to duplicate the complete disc (or discs) containing the files for a project. The suggested way of doing this is to have 2 backup discs for each disc in use. The 3 discs (labelled A, B, C for convenience) can be used in a backup cycle:



At regular intervals - at least once a week, but depending on how much updating has been done - the current disc is backed up. This is done by copying the complete disc to the second backup (C). The copy should be verified after it has been made.



Once this has been done, the second backup (C) becomes the current disc, the previous current disc (A) is relegated to backup, and the first backup to second backup:



There are two reasons for using C as the new current disc instead of continuing with A:

1) If the cycle is carried out regularly each disc will get the same amount of use

2) If for any reason the copy did not work, this will quickly become apparent when trying to use C.

If the current disc becomes corrupted at any time, the first backup can be used to restore the situation at the time of the last backup cycle.

The second backup provides an extra insurance policy against catastrophes - for example if a disc drive fault corrupts both the current disc and the first backup, or a power failure occurs during the backup process.

The extra expense of triplicating discs (not much for floppies) and the time spent backing up is more than paid for by the savings if a fault does occur. 3.12 TESTING

Once a program has been written, it must be tested. However, a microcomputer program is often designed to run on a system other than the one on which it is developed. The program is often ready for testing some time before the target system is built; and in any case the target system may not provide the facilities needed to test a program.

3.13 SIMULATOR

To overcome this problem, some means of simulating the target system environment on the development system is required. Texas Instruments provides a 9900 Simulator that executes on the 990/10 minicomputer, or, as part of the Transportable Cross Support package, on other machines. The simulation occurs entirely in software. In effect, the

simulator builds a software model of the target system on the development system. Inputs and outputs are simulated in software. The simulator records what would happen if the program was executed on the hardware it is designed for. It allows the user to trace exactly what goes on when the program is running - examining memory contents for example, and following the program's flow of execution.

The simulator can be operated interactively, with the user sitting at a terminal and directly controlling what happens, or by submittimg a list of commands and letting it run (batch mode).

3.14 INTEGRATION

While a simulator provides powerful debugging facilities, and can be used to check out completely the logic of a program, it does not prove that the software will work correctly with the target system hardware. The critical stage of hardware/software integration is best handled using an emulator.

3.15 EMULATOR

An emulator allows the software to be tried out in the target system hardware, while retaining the facilities of the development system to monitor program execution and change the program if necessary.

This is achieved by connecting the development system to the target by a special cable - in effect an umbilical cord. The microprocessor is removed from the target system and the cable plugged-in in its place.

The cable includes a buffer module containing a microprocessor and RAM memory. This emulator memory can be loaded from the development system with the program under test. The program executes in the buffer module exactly as it would in the target system (in real-time) and is connected to the target system hardware for input and output. But the development system can monitor program execution, trace the program flow and stop execution if specified conditions (breakpoints) occur.

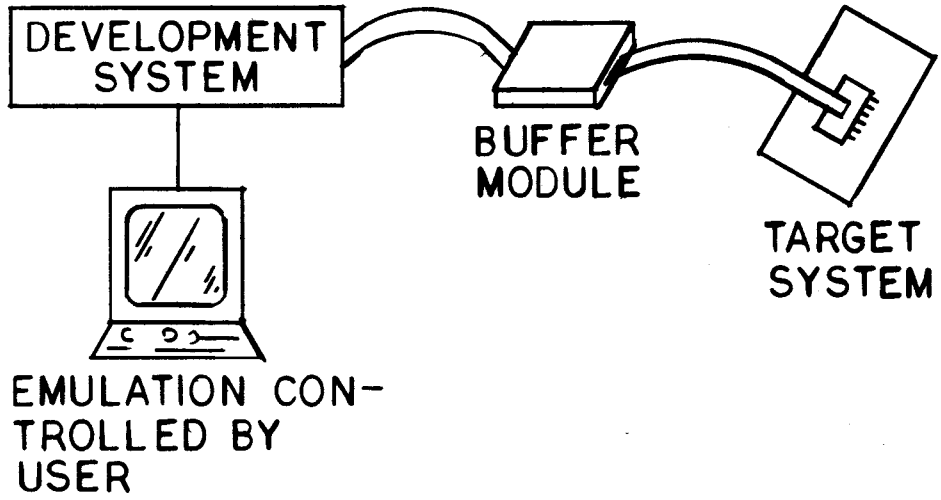


FIGURE 3-9. EMULATOR

Emulation is provided by the AMPL (Advanced Microprocessor Prototyping Laboratory) module, which can be used with a 990/4 or 990/10 minicomputer. The emulator is controlled by a powerful high-level language, in which sophisticated test procedures can be written.

Once the system is working in emulation, it can be programmed into PROMs and the umbilical cord to the development system can be removed. At this stage the device should undergo a thorough evaluation, preferably by someone not involved in its development.

3.16 PRODUCTION

Once a working system has been obtained that satisfies the design criteria, the hardware can be frozen and production of the device can begin. (If the device is 1-off, of course, it is the end of the road.) Hardware typically requires a much longer production lead time than software (for printed circuit board layout, tooling, etc.) and therefore needs to be frozen much earlier. Minor software changes and enhancements can still be made, provided they do not affect the hardware. This should not be carried too far: major changes to software can take a long time and may require hardware changes, too.

It is wise not to freeze the software until it has been tested with pre-production hardware. Minor problems introduced by the move from prototype to production may be able to be fixed in software. This will usually be much easier than modifying the hardware at this stage.

3.17 DEVELOPMENT SYSTEMS

Texas Instruments provides a range of development systems, from the simple to the sophisticated, for developing microcomputer software. The choice to be made depends on the size of the company, finance available, type of application, and the programming language selected.

In this area, investment usually pays off. For example, the cost of a TM 990/4 minicomputer with FS AMPL has to be weighed against the extra time spent by an expensive engineer if he does not have access to these facilities.

3.17.1 TM 990/4

The TM 990/4 is a minicomputer which uses the TMS 9900 microprocessor as its central processing unit (CPU). With dual floppy discs as backing storage, the 990/4 supports the Terminal Executive Development System (TXDS). This is a software package which provides a range of tools for the development of microcomputer software. These include:

TXEDIT - Text Editor

TXMIRA - Relocating Assembler

TXLINK - Linker

TXDBUG - Debug Monitor

TXPROM - PROM Programmer

TXPROM requires a Prom Programming Unit, a hardware module which plugs into the 990/4 chassis.

3.17.2 TM 990/10

the TM 990/10 minicomputer is designed for the medium to large user, who may be working on several projects at the same time. Under the hard disc-based DX10 operating system. the 990/10 provides an interactive multi-user environment with up to 1 Mbyte of main memory, and all the resources of a powerful general purpose computer. Software can be developed for a target system in assembly language or Pascal. A software simulator is available for testing target programs.

The 990/10 can be configured with a range of peripheral hardware, including several types of hard discs, floppy discs, magnetic tape and line printers. Each user interacts with the system through flexible menu-oriented command procedures, which prompt the user for required

parameters. The system is designed to be easy to use: most procedures supply default parameters, which are linked to other procedures. (For example, the "Print File" command defaults to the file most recently edited. The file name is displayed on the screen and can be accepted or changed by the user.) The command procedures themselves are written in a powerful interpretive language. The user can add new procedures or change existing ones as required. The system supports a powerful screen-based editor.

In addition to being a development system, the 990/10 is a general purpose computer that can be used for other applications. It supports several high-level languages including COBOL, RPGII, BASIC. A 3780 emulator package is available to allow connection to IBM mainframes.

3.17.3 AMPL

AMPL (Advanced Microprocessor Prototyping Laboratory) is a hardware and software package that can be added to a TM 990/4 or TM 990/10 minicomputer to provide complete emulation and trace facilities. AMPL is a very useful tool in the critical stage of hardware - software integration. It can also be used as a sophisticated software-driven logic analyzer - an essential tool for tracing faults in a microprocessor system.

3.17.4 TM 990 Boards

The TM990 range of microcomputer boards provides standard hardware that can be configured to suit many applications. Using these boards, hardware development is reduced to a minimum. A complete system can be built from a chassis, power supply and one or more TM 990 boards.

Assembly language and BASIC software for a system such as this can be developed using software utilities placed in ROM on the TM 990 boards, without the need for a separate development system. This approach is not recommended for projects of any size, because the facilities available on a full development system are much more sophisticated and can greatly improve programmer productivity. However, it is useful for low volume and experimental work, where the expense of a development system cannot be justified.

For assembly language programming, a system containing a TM 990/100 or /101 board and a /302 Software Development Board provides a text editor, symbolic assembler, loader and debugger. The /302 board includes a dual audio cassette interface, which provides a simple, low-cost form of backing storage. It also includes a PROM programmer.

A simple BASIC development system can be implemented using a single TM990/100 or 101 board (see Evaluation BASIC, Chapter V). More facilities can be made available, however, by adding a memory expansion board or /302 development board and using Development BASIC. A /302 board, with the Development BASIC Software Enhancement

Package, provides a PROM programmer and audio cassettes, as well as additional BASIC facilities.

CHAPTER IV

PASCAL

4.1 INTRODUCTION

Pascal was originated in the early 1970's by Professor Niklaus Wirth of ETH University, Zurich, Switzerland. Like the majority of modern programming languages, it is derived from ALGOL (ALGOrithmic Language). Previous 'high-level' languages, such as FORTRAN, were designed to take advantage of a particular computer's instruction set (FORTRAN was designed around the IBM 360) and can more properly be regarded as high-level assemblers. For example, standard FORTRAN makes certain restrictions on the form of array subscripts, DO loop expressions, and so on, because this makes the code particularly easy to implement on the 360. However, these restrictions also made the language difficult to remember (it has a lot of 'quirks'), and the restrictions quickly lost their significance when the language was implemented on later generations of computers with different instruction sets.

ALGOL was the first serious attempt to design a language that was independent of any particular machine's instruction set. The aim of the ALGOL designers was to construct a language that would make it easy to write clear, correct and maintainable programs. In this they largely succeeded. However, while ALGOL became popular with academic users, it was never very widely used in industry. This was partly because the ALGOL designers were uncompromising in refusing to consider implementation efficiency, and partly because ALGOL did not gain strong backing from computer manufacturers.

But ALGOL was the inspiration for a completely new generation of languages, of which Pascal is probably the most successful.

Pascal corrects most of the failings of ALGOL, while still retaining its ease of use. It leaves out some of the little-used but expensive (in code and time) features of ALGOL, and is designed with efficiency of implementation in mind. Therefore it is possible to implement Pascal efficiently on a small computer or a microcomputer. It is a very practical language. Pascal was developed principally by one man so it has a coherence that some committee-designed languages lack. Pascal is very regular (orthogonal): it has few 'quirks', and so is easy to learn. The features of Pascal make it equally suited for systems and applications work, so that there is no need to use two different languages.

Not only does Pascal have powerful program structures, directly implementing the constructs described in Section II, but it also has extremely powerful data structures which are very necessary for manipulating complex applications. In fact, the Pascal language is very close to the design language described in Chapter II because they

both come from the same root. Turning a software design into Pascal should involve little more than "tightening-up" the syntax and turning English-language descriptions into precise Pascal statements.

With rapidly decreasing hardware costs and increasing labor costs, software has become the major investment in developing a computer-based product. This cost trend has led to the move from low-level to high-level languages, necessitating standardization within high-level languages. At least as important as the investment made in existing software is the cost of retraining programmers to use a new language; and to use it efficiently.

With this in mind, Texas Instruments has made a commitment to use Pascal as a corporate standard for all software, whether for mainframes or microcomputers. Pascal has become the primary language for the 990 and 9900 range of mini and microcomputers. The majority of 9900 systems software is now being written in Pascal. Pascal also provides the base for a range of modular software to supply many commonly recurring needs.

Pascal provides a high-level standard that protects software (and the programming skills, to implement that software) from future obsolescence due to the introduction of new hardware. This form of standardization has now become more important than that on a particular low-level machine architecture.

Texas Instruments supports two implementations of Pascal: Texas Instruments Pascal (TIP) and the Microprocessor Pascal System. The languages are fundamentally the same, but provide slightly different features to support their different areas of application. Because microcomputer software is the main concern here, this chapter concentrates mainly on Microprocessor Pascal system.

4.2 TEXAS INSTRUMENTS PASCAL OVERVIEW

TI Pascal was developed prior to the Microprocessor Pascal system and was designed to compile and execute on larger machines (the Texas Instruments DS 990/10 and the IBM 70). TIP provides 'large machine' features such as dynamic arrays and extended precision reals. It also includes some extra compiler options allowing, for example, optimization probes to be inserted in the program to identify the most frequently executed paths.

TIP generates a conventional sequential program for execution under the control of an operating system. TIP was extended to allow execution in other environments (such as a target microcomputer system) by the introduction of TIPMX (TI Pascal Microprocessor Executive). TIPMX provides the run-time environment for a TI Pascal program, and also supports concurrency (see below). In a TIPMX system, concurrency is provided by procedure calls to the TIPMX executive.

4.3 MICROPROCESSOR PASCAL OVERVIEW

Microprocessor Pascal was designed from the start to produce code for a target microcomputer system, and to compile on the single-user floppy disc based FS990/4 computer and the multi-user DS 990/10.

The Microprocessor Pascal system provides a complete development environment for the design, coding, and debugging of Pascal systems for microcomputers.

Four major components assist in software development:

- an interactive, syntax-checking editor for source preparation and checking
- a compiler to compile source into interpretive code
- an interactive debugging interpreter
- a code generator to generate 9900 native object code

Two executives support the execution of the user's system on a target computer. One supports the interpretive code produced by the compiler; the other supports the object code produced by the code generator. These executives are functionally identical, so that the user has a choice of running either interpreted or compiled code on his target system.

Because microcomputer systems are real time, concurrency is an integral part of the Microprocessor Pascal System language (rather than being implemented as procedure calls as in TIPMX). A concurrent system consists of a number of independent processes executing in a single environment. Each process is a separate sequential program, and the processes are written as if they were executing simultaneously. In fact, the processor can only do one thing at a time; the executive divides processing time between the processes so that the effect is of simultaneous execution. Using this approach, a programmer can identify the various tasks that a real-time system has to perform, with their inputs and outputs, and write a separate process for each: the executive will handle the rest. This can greatly simplify a complex problem. Synchronization of processes is accomplished by signalling devices called semaphores. More complex communication between processes can be handled by interprocess files. Further information on concurrency is presented later in the chapter (subsection 4.5). Attention is now turned to Microprocessor Pascal system major components.

4.3.1 Microprocessor Pascal system Editor

Microprocessor Pascal features an interactive, screen-based editor that allows the user to create and modify Microprocessor Pascal system source files. When editing, a page of text is displayed on a video display unit (VDU screen). The text may be modified simply by positioning the cursor and typing new information. Characters can be inserted and deleted anywhere on the screen. The displayed page can be positioned anywhere within the text file; page boundaries are not fixed.

Alternatively, the user can press the command (CMD) key and enter a range of edit commands, including find string, replace string, etc.

When creating a source file, the editor assists line by line program layout by automatically positioning the cursor for a new line. The cursor can be moved forward or backward using the TAB keys. This helps in indenting text to reflect the program structure. The tab increment (number of columns for each indentation) can be set by the user.

When the program has been entered, the user can perform a Pascal syntax check without leaving the editor by entering the CHECK command. The editor is not equipped to detect semantic errors (such as undeclared identifiers), but will perform a complete syntax check that will find such errors as misspelled or missing keywords, incorrect punctuation, invalid constructs, etc.

When the editor finds an error, it outputs an appropriate error message to the screen, displays the relevant area of text and positions the cursor over the error so that the user can edit it immediately. When this is done, the CHECK command can be reentered and checking will resume from the earliest point at which the text was changed. (THE checker only 'backs up' as much as is necessary; it does not need to restart from the beginning of the file).

The syntax checker speeds up and simplifies the process of correcting syntax errors. It eliminates exiting the editor, executing the compiler printing the listing, and re-editing the source file for each mistake. The entire process becomes a single interactive step.

The CHECK facility is entirely optional. The Micprocessor Pascal system editor can be used for text files other than MPP source.

A full list of editor commands is presented later in this chapter (subsection 4.6).

4.3.2 Microprocessor Pascal Compiler and Code Generator

The Microprocessor Pascal compiler generates interpretive Pascal code from a Microprocessor Pascal source file. This code can be executed directly using the interpretive debugger or the Microprocessor

Interpretive Executive, or it can be passed through the Microprocessor Pascal system code generator to produce native 9900 object code that will run under the Microprocessor Pascal executive.

Thus, Microprocessor Pascal gives the user a choice of executing either interpretive or native code. Interpretive code and native code for the same Microprocessor Pascal system source file will be functionally identical, apart from considerations of speed and code size.

Interpretive code executes slower than native code; but (beyond a certain size, which accounts for the overhead of the interpreter) an interpreted system is much smaller. Interpretive code takes up about half the memory required by the equivalent native code. Therefore, for a large application, interpretive code can represent a great saving in memory.

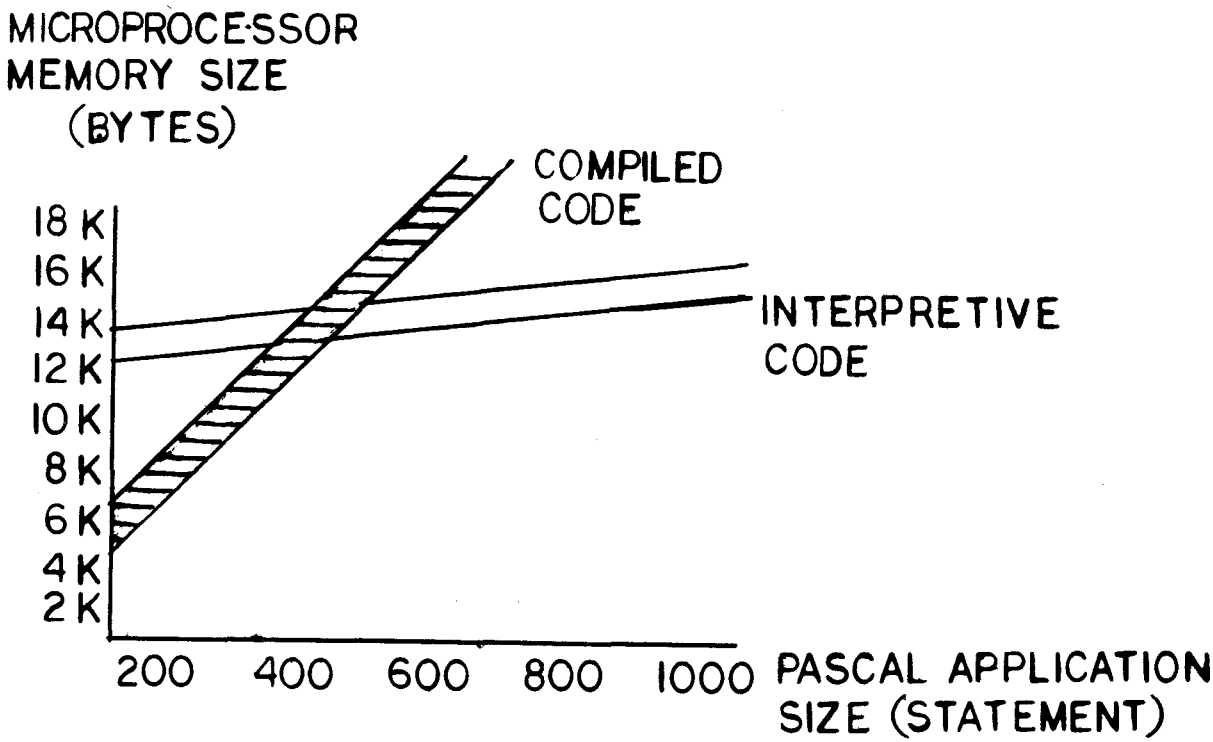


FIGURE 4-1. INTERPRETIVE vs COMPILED RUN-TIME CHARACTERISTICS

In selecting whether to use native or interpretive code, the user can trade off speed against memory size. One example of such a trade-off is the Microprocessor Pascal compiler itself. On the FS990/4 floppy disc based system, the compiler executes interpretively so that it will fit into the available memory space (it still runs at an acceptable speed). On the DS990/10, where there are no memory restrictions, it executes as native code to maximize the speed.

Various compiler options are available. These options include:

LIST	- generate source listing
MAP	- generate variable map
STATMAP	- generate map of displacements for each statement in the object module
DEBUG	- insert statement numbers in code for debugger
ASSERTS	- generate code for ASSERTS statement
CKINDEX	- insert run-time checks for array indices
CKPTR	- insert run-time checks for NIL pointers
CKSET	- insert run-time checks for set element expressions
CKSUB	- insert run-time checks for subrange assignments in bounds

4.3.3 Micprocessor Pascal Debugger

The Micprocessor Pascal debugger is an interactive interpreter that allows the user to control and monitor execution of a Micprocessor Pascal. This greatly simplifies the task of finding errors in a system (debugging).

The debugger is designed for use with a concurrent (multiple process) system. The user can monitor the execution of a single process, or examine and control process scheduling and communication. Debugging usually proceeds with one aspect of a system at a time.

The user can set breakpoints at any Pascal statement by specifying the routine and the statement number (printed on the source listing). The system can be executed in single-step mode (one Pascal statement at a time), or continuously until a breakpoint is reached. Three modes of tracing, trace process scheduling, trace routine entry/exit and trace statement flow, are possible.

The contents of a routine's stack frame (data area), heap, and common areas, can be displayed and modified. The scheduling algorithm can be overridden by holding (suspending) a particular process until an explicit release command is given.

The user can also reconnect interprocess files (discussed later in this section) using the Connect Input File and Connect Output File commands. The new file that results can be sent to an external file or to the terminal. The process concerned will then input or output

to the device specified. If it is a terminal, the system will prompt for input, and send a message identifying the source for output.

Interrupts can be simulated using the SIMulate Interrupts command.

The system has three ways of dealing with CRU I/O (for a description of the CRU, See Section VI). CRU statements can be directly executed, ignored, or simulated by the user. The "CRU" command is used to specify which option applies to a particular process. When simulated I/O is specified, the CRU address and value are displayed for output and the user is prompted for input. This feature can be useful when debugging software for a target system, which is likely to have a different CRU configuration from the development system.

The Micprocessor Pascal debugger is a very powerful high-level tool for verifying the detailed execution of a piece of software. It is designed to integrate closely with the other components of Microprocessor Pascal and to form a complete system in which designs can be smoothly carried through to implementation.

4.4 PASCAL STRUCTURE

4.4.1 Features

Pascal has structured statements which allow the user to produce a readable, maintainable, and easily checked program algorithm with mimimum effort. These structures, if used as intended, automatically generate hierarchical, nested code resulting in an easier understand, and (as has been proved) better, more correct software. Pascal's structured statements include IF CASE, FOR, WHILE and REPEAT; they are described in Paragraph 4.5.11.

Pascal provides extensive data structuring: RECORD and ARRAY data structures can be combined and nested to any level The POINTER data type allows powerful structures such as linked lists and trees. It also permits dynamic storage allocation. Pascal's data structures are described in Paragraph 4.5.3.

One of Pascal's most useful features is data typing. This allows data to be grouped according to use, and can clarify the design of a program so that, for example, it is easier to change at a late stage in development. Compiler checks on type compatibility can greatly reduce the risk of undetected errors in program code.

In addition to the standard data types, Pascal allows the user to define his own data types, which can have values represented by meaningful names as well as numbers. This can assist in program documentation. The type concept was discussed in Section II. Its Pascal implementation is described here in Paragraphs 4.5.4 to 4.5.7.

Pascal allows the user to define meaningful names for his identifiers (there are no arbitrary length restrictions). By using these

identifiers and standard keywords (IF...THEN...ELSE), the programmer creates a largely self-documenting program.

Pascal is a block structured language, which means that procedures (and processes) can be nested to any depth. It is therefore a natural language for writing modular software. Block structure and scope rules are described in Paragraph 4.4.6.

The concurrency features of Micprocessor Pascal allow a new approach to software design. particularly for microcomputers. A real-time problem can now be divided into separate parallel processes, each of which can be simply specified and coded. (A powerful extension of the concept of modular software). Concurrency was designed into Microprocessor Pascal from the start; all the development tools that make up the Microprocessor Pascal system were designed to support it. However, if the user wishes to develop a conventional sequential program in Microprocessor Pascal, he can do so without incurring any extra expense. The mechanisms involved in concurrency are described in more detail. Additional information can be obtained in the references stated at the end of this chapter.

4.4.2 Stack and Heap

Like the majority of modern high-level languages, Pascal has a stack architecture. The stack is an area of data storage from which sections (called stack frames) are allocated to a program or procedure at the time it is invoked. When the program or procedure has finished executing, its data storage area is returned to the stack for use by other routines. The workspace register concept of the 9900 (see Section VI) forms a natural basis for implementing stack frames.

Stack architecture means that data is completely separated from program code, so that Pascal adapts naturally to the ROM/RAM environment of a microcomputer. It also means that Pascal code is automatically re-entrant. If a routine is simultaneously invoked from different parts of a system (as can well happen in a concurrent system) both invocations can use the same program code; it is only necessary to create different stack frames.

In addition to the storage provided in the stack, Pascal is able to allocate storage dynamically, under program control, from an area called the heap. This is accomplished using the standard procedures NEW and DISPOSE, and the pointer variable described in Paragraph 4-5.6.5.

4.4.3 Systems and Programs

The largest unit in Micprocessor Pascal is a SYSTEM. A system may contain a number of processes, apparently executing in parallel. A Level 1 (highest level) process is declared, in Microprocessor Pascal, by the keyword PROGRAM. A conventional sequential program can be regarded as a special case of a system with only one PROGRAM.

4.4.4 Processes and Procedures

Each PROGRAM can contain within it subordinate processes that are declared by the keyword PROCESS. The keyword PROGRAM is used at the highest level because processes at this level have special properties. This also maintains compatibility with standard Pascal.

A system, program or process can contain within it procedures or functions.

Processes and procedures look similar but, in practice, are quite different A procedure is, logically, a part of the sequential program that calls it, whereas a process is a separate sequential task that executes in parallel with all the other processes in the system including the one that calls, or STARTs it.

4.4.5 Declarations and Statements

There are two principal parts to any Pascal system, program, process, or procedure: the Declarations, and the Statement Body.

Declarations define identifiers that can later be referred to by name (instead of by repeating the declaration). These identifiers specifv the data that the program is to work with; the statements specify exactly what is to be done with this data.

```
PROGRAM FACTORIAL;

VAR I,J.N : INTEGER;                    (*DECLARATIONS*)
                                  (*DECLARE VARIABLES NAMED*)
                                  (*I. J. N OF TYPE INTEGER*)

BEGIN (*COMPUTE FACTORIAL*)             (*BODY*)
RESET(INPUT);
READ(N);                          (* READ IN A VALUE FOR N *)
I := 1; J := 1;                   (*   SET I AND J TO 1    *)
WHILE I <> N DO
    BEGIN                         (*USE I AND J TO COMPUTE *)
    I := I + 1;                   (*     FACTORIAL N       *)
    J := I * J
    END;
WRITELN(J)                        (*    OUTPUT VALUE OF    *)
END.                              (*     FACTORIAL N       *)
```

The declarations also specify any subordinate processes, procedures, etc., and assign identifiers to them so that they can be referred to in the statement body.

Pascal programs are free format; the program can be laid out in any manner on the page. Statements for example, need not start in a particular column; nor are they restricted to one per line, though

this is usually good practice.

Pascal gives the programmer a free hand in formatting his program. However, for readability, it is a good idea to lay out the program to reflect its structure. This can be done by using indentation. In the example above, the BEGIN...END compound statement depending on the WHILE clause is indented to show that it is one level down in the program hierarchy. In fact, the indentation reflects the appearance of the structure diagram for the program (see Section II):

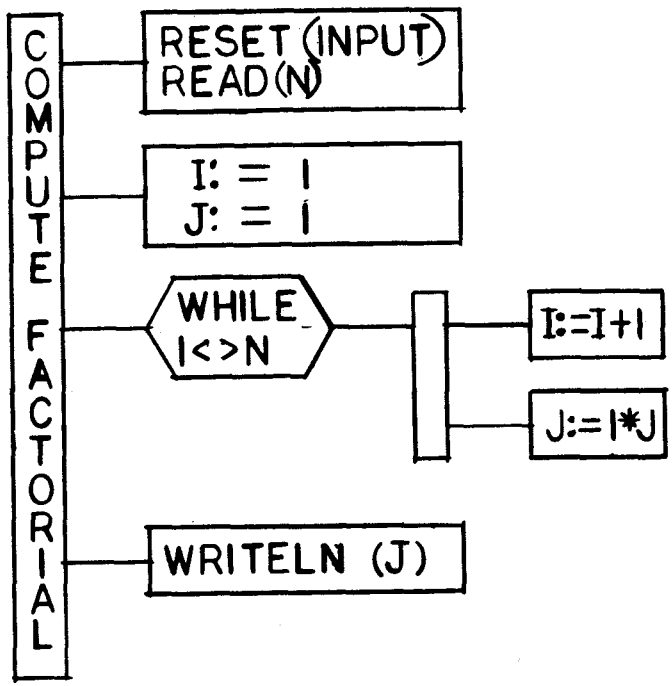


FIGURE 4-2. PROGRAM STRUCTURE DIAGRAM

Formatted in this way. the program is much more readable and the structure can be seen at a glance.

4.4.6 Block Structure

One of the most important features of Pascal is its block structure. Some of the basic ideas of block structuring are discussed in Section II.

A block is a self-contained area of program that contains both a statement body and the declarations (type, variable, procedure, etc.) Relating to it. A Pascal program consists of a hierarchy of blocks, nested one within another. An Micprocessor Pascal system block, which is a complete MPP system, contains a number of program blocks. which in turn can contain process blocks, procedure and function blocks, etc. The hierarchy is displayed below in the diagram in Paragraph 4.5.

The declarations made at the start of a block apply to that block and to any blocks nested within it. This is called the scope of the declaration. Scope can be formally defined as the range of system text over which the declaration is valid. Identifiers cannot be referenced outside their scope, i.e. outside the block in which they are declared. For example, consider the following:

```
SYSTEM X;
<declarations>                    (*system declarations*)

    PROGRAM A;
    <declarations>                (*program declarations*)
        PROCEDURE P;
        <declarations>            (*procedure declarations*)
        BEGIN
        .                         (*Procedure body*)
        .
        END;

        PROCEDURE Q;
        <declarations>            (*procedure declarations*)
        BEGIN
        .                         (*Procedure body*)
        .
        END;

    BEGIN
    .                             (*Program body*)
    .
    END;

    PROGRAM B;
    <declarations>                (*program declarations*)
        PROCEDURE R;
        <declarations>            (*procedure declarations*)
        BEGIN
        .                         (*Procedure body*)
        .
        END;

    BEGIN
    .                             (*Program body*)
    .
    END;

BEGIN
.                                 (*System bodv*)
.
END.
```

The declarations in PROGRAM A cannot be referenced in PROGRAM B or PROCEDURE R, but can be referenced in both PROCEDURE P and PROCEDURE Q. The declarations in PROCEDURE P cannot be referenced in PROCEDURE Q or in PROGRAM A.

If a reference is made to a declaration (variable, type, procedure, etc.) that is not in scope, the compiler will generate an error message. Block structure and scope rules are thus powerful tools for managing program structure. Procedure P, for example, can be written without worrying whether it will interfere with procedure Q. A

variable can even be declared in P with the same name as a variable declared in Q; they will be completely different variables because they are in different areas of scope. If a variable is declared in P with the same name as a variable declared in A, the compiler will create a new variable with this name, and references to it in P will always access this local definition. Where there is a possible ambiguity, the compiler always chooses the most local declaration.

This does not mean that it is good practice to declare different variables with the same name. But, should it happen (if, for example, modules are written by different programmers) there is no cause for worry. Note that in the example, both P and Q can access the declarations made at the start of program A; the interaction with data declared in higher-level modules needs to be clearly defined when writing a system. This should be part of the module specification.

As well as assisting program structure, block structuring (combined with Pascal's stack architecture) can save memory space. Data area is not allocated to a procedure from the stack frame until it is actually called. This means that if, say, procedure P is called followed by procedure Q, the space taken up by the variables of procedure P is returned to the stack when it has finished executing, and the same memory area can be used for the variables of procedure Q. The system only allocates data space to the routines currently executing.

A variable has an extent as well as a scope. Extent is the time during system execution for which storage space is allocated to the variable. Apart from dynamically allocated variables, this extent is the duration of execution of the block in which the variable is declared. In a concurrent system, a variable's extent continues as long as any of the processes declared in the same block are executing. The reason for this is that the variable is in scope in such a process and might be referenced.

4.5 PASCAL LANGUAGE

4.5.1 Basic Rules

A Pascal system, program, or process is made up of symbols from a finite vocabulary. The vocabulary consists of identifiers, numbers, strings, operators and keywords. These in turn are composed of sequences of characters from the underlying character set, which is

```
        the letters A-Z, a-z
        the digits  0-9
      and the special characters:
+ - * / " . , ; : = $ ' < > ( ) [ ] { } # ^ @ _
```

Special symbols are used for operators and delimiters. These include:

```
+ - * / := = <> < <= >= > :: @ ; ..
```

Keyword symbols are reserved words with a fixed meaning; they may not be used as identifiers. They are written as a sequence of letters and interpreted as a single symbol.

ACCESS	ELSE	MOD	REPEAT
AND	END	NIL	SEMAPHORE*
ANYFILE*	ESCAPE	NOT	SET
ARRAY	FALSE	OF	START*
ASSERT	FILE	OR	SYSTEM*
BEGIN	FOR	OTHERWISE	TEXT
BOOLEAN	FUNCTION	OUTPUT	THEN
CASE	GOTO	PACKED	TO
CHAR	IF	PROCEDURE	TRUE
COMMON	IN	PROCESS*	TYPE
CONST	INPUT	PROGRAM	UNTIL
DIV	INTEGER	RANDOM	VAR
DO	LABEL	REAL	WHILE
DOWNTO	LONGINT	RECORD	WITH

* not in TIP

Identifiers are names denoting user defined or predefined entities. An identifier consists of a letter or '$' Followed by any combination of letters, digits, '$' Or '_' (underscore). A lower-case letter is treated as if it were the corresponding upper-case letter. For example, the identifier Data_Size is the same as the identifier DATA_SIZE. A maximum length is imposed by the restriction that identifiers must not cross line boundaries so that they may not be more than 72 characters long. All characters in an identifier are significant. Process, routine and common names should be unique within the first 6 characters.

```
Legal Identifiers:
      X
      $VAR
      LONG_IDENTIFIER
      NUMBER_3
      READ

Illegal Identifiers:
      ARRAY        (Reserved word)
      _ROOT3       (Cannot start with _)
      3RDVAL       (Cannot start with number)
      MAX VALUE    (Cannot contain blank)
      TOTAL-SUM    (Cannot contain -)
```

Some identifiers are standard, that is they are predefined with a given meaning They can be redefined by the user, in which case the standard meaning no longer applies. For example, if the standard routine name READ is redefined, the standard routine READ cannot be

A comment is any sequence of characters beginning with { or (* and ending with } or *) (except within a string). A remark is any sequence of characters beginning with " and extending to the end of the line (except within a string). Comments and remarks are ignored by the compiler, and can be used to annotate program text.

4.5.2 Systems

In Microprocessor Pascal, a system is declared as follows (the full syntax definition is given in the Microprocessor Pascal System User's Manual.

```
SYSTEM <identifier>;

<system data declarations>;

<system routine declarations>;

<process body> .
```

The <system data declarations> declare global constants that are common to the whole system. The <system routine declarations> declare the programs and procedures that make up the system. The <process body> consists of the statements that make up the 'main program' of the system (the first process to be executed). In a concurrent system, this 'main program' will probably contain (besides initialization statements) a series of START statements to set up the various concurrent processes of which the system is composed. For example:

A program (Level 1 process) declaration is as follows:

```
PROGRAM <identifier> <program parameters>;

<program data declarations>;

<program routine declarations>;

<process body>.
```

The <program parameters> are optional. The <program data declarations> declare data objects local to the program. The <program routine declarations> can contain process, procedure or function declarations.

A process declaration is:

```
PROCESS <identifier> <process parameters>;

<process data declarations>

<process routine declarations>
```

```
<process body>;
```

The <process routine declarations> can declare subordinate processes, procedures, and functions.

Procedures are declared as follows:

```
PROCEDURE <identifier> <procedure parameters>;

<procedure data declarations>

<procedure and function declarations>

<compound statement>;
```

A procedure may declare subordinate procedures and functions, but not processes. The compound statement (described in Section 4.12.1) is simply a list of statements which describe the action of the procedure. Syntactically, a <process body> is also a compound statement.

Functions are the same as procedures, except that they return a single value of a specified type. The type is defined in the function header:

```
FUNCTION <identifier> <function parameters> : <type identifier>;
```

Types are discussed in the following sections.

Micprocessor Pascal regards a single sequential program as a system with only one program. The SYSTEM declaration can be left out, and the program declared as:

```
PROGRAM <identifier>;
<declarations>
<program body> .
```

(The syntax given here is not complete: a full syntax definition is given in the Micprocessor Pascal System User's Manual, and is included in the Reference Section of this chapter.)

The declarations hierarchy is represented in the following diagram:

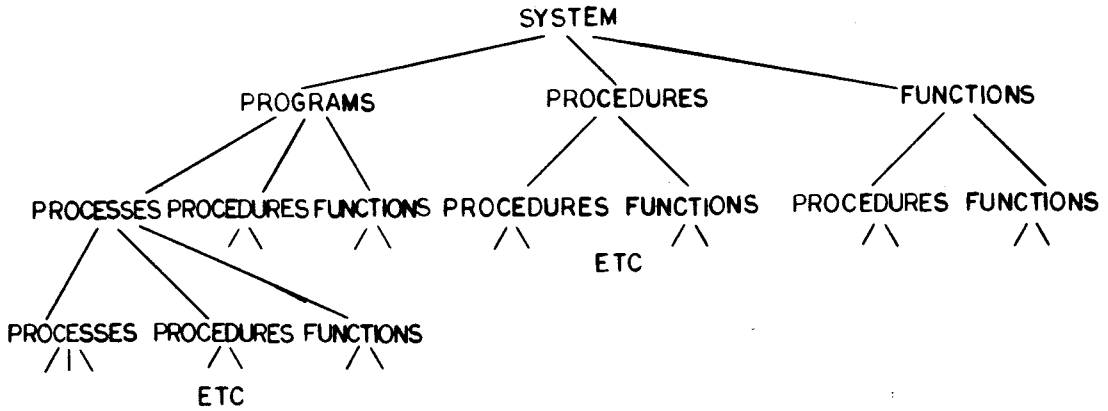


FIGURE 4-3. DECLARATIONS HEIRARCHY

The hierarchy for a sequential program which does not allow concurrent processes is represented as follows:

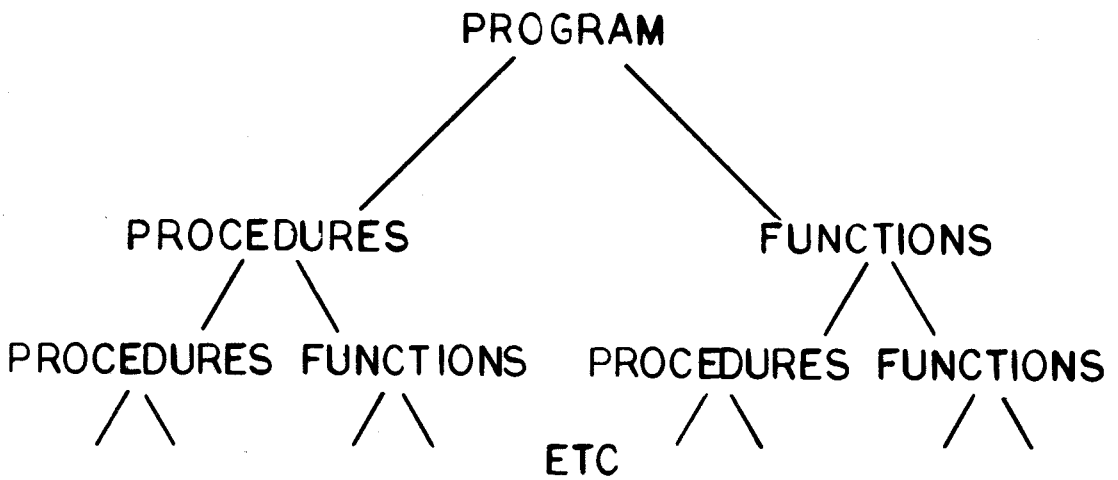


FIGURE 4-4. SEQUENTIAL PROGRAM NOT SUPPORTING CONCURRENCY

4.5.3 Data Declaration

The data declarations section of a program consists of four separate parts:

<constant declaration part>
<type declaration part>
<variable declaration part>
<common declaration part>

The <constant declaration part> allows an identifier to be used as a synonym for a constant. For example:

```
CONST MAX = 100;
      ASTERISK = '*';
      ONE_HALF = 0.5;
```

Constant declarations are described in detail in the Micprocessor Pascal User's Manual.

Type and variable declarations are described below. The COMMON declaration allows variables to be shared between modules; it is described in the Microprocessor Pascal User's Manual.

4.5.4 Type Declarations

The type concept allows the user to group data according to its use. Types are introduced in Section II.

A data type declaration defines the set of values a variable of the type specified may assume and the set of operations that may be performed on these values. Each variable is associated with one and only one type. The simple types consist of the standard types INTEGER, REAL, BOOLEAN and CHAR; plus the user-defined scalar or or subrange types. Structured types are made up of component types structured according to accepted methods. Structured types are declared by specifying the types of the components and the method of structuring. The structuring methods available consist of arrays, records, sets, pointers, semaphores (not TIP) and files.

A type declaration introduces an identifier as the name of a new data type. It can later be used to refer to that type; for example, to define variables, or to define structured types in which that type is included. The form of a type declaration is:

```
TYPE <type declaration list>
```

where <type declaration list> is one or more of the following:

```
<identifier> = <type definition>;
```

For example:

```
TYPE VECTOR  = ARRAY [1..10] OF REAL;
     DAYS    = (MON,TUE,WED,THU,FRI,SAT,SUN);
     DIGITS  = '0'..'9';
     COMPLEX = RECORD
               RE,IM : REAL;
               END;
```

The various forms of <type definition> are described in subsequent sections.

The TYPE declaration does not declare any actual variables (storage locations); this is accomplished in the variable (VAR) declaration (described in section 4.8

4.5.5 Simple Types

4.5.5.1 Integer and Longint. A value of type INTEGER is a whole number in the range -32768 to 32767 (signed 16-bit quantity). A value of type LONGINT ranges from -2147483648 to 2147483647 (signed 32-bit quantity).

The basic operators defined for INTEGER and LONGINT operands are:

+	unary plus or add
-	negate or subtract
*	multiply
DIV	divide and truncate result
MOD	modulus A MOD X = A - ((A DIV X) * X)

The operator / (divide) can be applied to integers, but always produces a REAL result. The relational operators =, <>, <, >, <=, >= can be applied to integers and produce a BOOLEAN result. Standard functions applying to INTEGER and LONGINT are described in the Reference Section.

4.5.5.2 Boolean. A value of type BOOLEAN is one of the logical values TRUE or FALSE. The following operators are defined for BOOLEAN operands and yield BOOLEAN results:

NOT	logical negation
AND	logical conjunction
OR	logical disjunction

TRUE and FALSE are predeclared keywords such that FALSE < TRUE. Thus the relational operators can be used with BOOLEAN operands to provide additional operations. For example:

=	equivalence
<>	exclusive OR

4.5.5.3 CHAR. Values of type CHAR are ordered according to their ASCII value. A character constant can be written either as a single character between single quotes, or by specifying its hex value, preceded by '[. For example,

```
'A'
'[0D'
```

4.5.5.4 Enumeration. INTEGER, LONGINT, BOOLEAN and CHAR are special cases of the enumeration type. An enumeration type is any simple type except REAL. The characteristics of an enumeration type are:

1) There is a distinct set of values which a variable of that type can take.
2) These values have a unique linear order in which each value (except the first and last) has a single predecessor and a single successor.

The integers

-32768,-32767,...-1,0,1,...32766 32767

Clearly follow these rules; so do the characters, which have a unique order (A,B,C, etc) defined by their ASCII representation. However, the user can also define his own enumeration types in a TYPE declaration simply by specifying a type name and an ordered set of values:

```
TYPE DAYS = (MON,TUE,WED,THU,FRI,SAT,SUN);
```

The values are represented by identifiers which must be unique and which can be regarded as primitive values (such as '7' or '125'). It is not necessary to translate them into numbers, or know how they are represented within the computer, any more than it is necessary to work out the internal-bit pattern used to represent '125'. 'MON', 'TUE', etc are values in their own right.

These user defined types are called scalar types. The relational operators (>, <, etc.) are defined for all enumeration types. The BOOLEAN expression MON < WED is TRUE because the values form an ordered set in which MON precedes WED. However, the arithmetic operators (+, -, etc.) are only defined for the standard types INTEGER and LONGINT (and REAL); it is meaningless to write MON + WED. The following standard functions apply to enumeration types:

SUCC(X)	the successor of X
PRED(X)	the predecessor of X
ORD(X)	the integer ordinal value of X within the set of values (not defined for INTEGER or LONGINT)

e.g., SUCC(WED) = THU, PRED(WED) = TUE, ORD(WED) = 3.

Scalar types are useful for counting purposes for example, to index into an arrav or control the number of iterations of a FOR loop:

```
FOR DAY := MON TO FRI DO
    TOTAL_TAKINGS := TOTAL_TAKINGS + TAKINGS[DAY];
```

The variable DAY is declared to be of type DAYS; the array TAKINGS is declared to be indexed by type DAYS.

4.5.5.5 Subrange. A type can be defined as a subrange of any previously defined enumeration type by specifying the smallest and largest values in the subrange:

```
TYPE WEEKDAYS = MON..FRI;
     ARRAY_INDEX = 1..25;
```

This is a useful feature, because a compiler option can be used to insert run-time checks to ensure variables do not exceed their specified subrange. This can be a great help in debugging. Types can also be used in declaring arrays, for example:

```
VAR TABLE : ARRAY [ARRAY_INDEX] OF INTEGER;
    SICKDAYS : ARRAY [DAYS] OF BOOLEAN;
```

This performs the double function of specifying the size of the array, and the type of the index variable. Constructs such as this make it easy to change the size of an array at a late stage in development, simply by altering one or two TYPE statements. (Arrays and variable declarations are discussed in this Chapter.) Section 4.8.)

4.5.5.6 REAL. The type REAL can be used to represent real values with from 6 to 7 decimal digits of precision. The range of absolute values that can be represented is approximately 1.0E-78 through 1.0E75.

The following operators accept operands of type REAL and yield a REAL result:

+	unary plus or add
-	negate or subtract
*	multiply
/	divide

The relational operators are defined for REAL operands and yield a BOOLEAN result. The standard functions TRUNC, ROUND, LTRUNC, LROUND will truncate or round a REAL value to give an INTEGER or LONGINT result.

4.5.6 Structured Types.

Structured types can be constructed from other types which are called components. The components can be structured to form an array, record, set, file, pointer, or semaphore.

4.5.6.1 Array Type. An array type consists of a group of components which are all of the same type. The form of an array type definition is:

```
ARRAY [ <index type list> ] OF <component type>
```

The <component type> can be any type except FILE. This means that it is possible to have arrays of arrays, records or any other structured type. The <index type list> is a list of <index type>s separated by commas. These can be either explicit subrange definitions (such as 1..5) or the name of a suitable enumeration type (such as DAYS). The number of <index type>s in the declaration determines the number of dimensions of the array. There is no limit to the number of dimensions an array may have. Each <index type> definition determines both the size of that dimension of the array, and the type of the variable that will be used to index it. An <index type> can be any enumeration type; the types of different dimensions need not be the same. For example:

```
VAR HOLIDAYS : ARRAY [1..52, DAYS] OF BOOLEAN
```

An exactly equivalent definition is:

```
VAR HOLIDAYS : ARRAY [1..52] OF
                   ARRAY [DAYS] OF BOOLEAN
```

The assignment operator can be used between arrays of compatible types. For example:

```
VAR A,B : ARRAY [1..20, 25..50, 1..2];
.
.
A := B;
```

This causes every element in array A to be assigned the value of the corresponding element in array B.

4.5.6.2 Record Type. A record type consists of a number of components of possibly different types called fields. Each field in a record type is given a distinct name. A field of a record can be of any type (including array, record, etc.) Except file. The form of a record type definition is:

```
RECORD <field list> END
```

A <field list> is an arbitrary number of <record section>s separated

by semicolons. Each <record section> is of the form:

```
<field identifier list> : <type>
```

where <field identifier list> is a list of field identifiers separated by commas. For example:

```
TYPE COMPLEX = RECORD
                 RE, IM : REAL
                 END;

     DATE    = RECORD
                 MONTH  : (JAN,FEB,MAR,APR,MAY,JUN,JUL,
                           AUG,SEP,OCT,NOV,DEC);
                 DAY    : 1..31;
                 YEAR   : INTEGER
                 END;
```

The assignment operator (:=' can be applied to records of exactly the same type.

A field of a record is referenced by specifying the name of the record variable and the field name separated by a period. For example:

```
VAR START, FINISH : DATE;
    C1, C2, C3    : COMPLEX;
.
.
START.DAY := 20;
FINISH.YEAR := 1978;

C1.RE := 3.4;
C3.IM := 5.8;
```

and

```
START := FINISH;
```

which is equivalent to

```
START.MONTH := FINISH.MONTH;
START.DAY := FINISH.DAY;
START.YEAR := FINISH.YEAR;
```

Pascal also allows record variants which means that part of a record can be interpreted in more than one way. This allows, for example, a personnel record for a college to contain different information (different fields) according to whether It described a student or a member of staff. Record variants are described in detail in the Microprocessor Pascal System User's Guide.

4.5.6.3 Set Type. Pascal allows a set type where the possible values are subsets of the base type which can be any enumeration type. For example, with the base type 1..5, possible values of a set variable include:

```
[1,2,3]
[2,3,5]
[1,2,3,4,5]
[ ]              (the empty set)
```

A full range of operators is defined for sets (union, intersection, inclusion. etc.) The set type is described in detail in the Microprocessor Pascal System User's Guide.

4.5.6.4 File Type. A file type is a structure which consists of a sequence of components (of unspecified length) which are all of the same type. A file is usually associated with a mass storage medium, such as tape or disc. However, this is not necessarily the case in Microprocessor Pascal File variables can be used as a means of communicating between concurrent processes. One process can write information to a logical file and another can read it. The Microprocessor Pascal Executive or Microprocessor Interpretive Executive perform the file management without involving any external storage devices.

The form of a file type definition is:

```
FILE OF <component type>
        or
RANDOM FILE OF <component type>
        or
TEXT
```

The component type of a file can be any type except pointer or file. The number of components, that is the length of the file, is not specified and can grow to any size, depending on the storage medium with which the file is associated.

The prefix RANDOM denotes a random file in which components are accessible by their component number. This numbering is defined to be the natural ordering of the sequence of components with the first component being number zero.

A TEXT file is a sequential file of type CHAR which is divided into lines by end-of-line markers. INPUT and OUTPUT are standard predeclared TEXT files.

```
TYPE REC = RECORD
           NAME : PACKED ARRAY [1..15] OF
           ID_NUM : INTEGER
           END;

VAR F : FILE OF INTEGER;
    EMPLOYEE : RANDOM FILE OF REC;
    TEMP : TEXT;
```

Standard procedures and functions (READ, WRITE, etc.) are provided for file manipulation.

4.5.6.5 Pointer Type. Variables may be referenced indirectly by means of a pointer which can be thought of as the address of a variable. The form of a pointer type definition is:

```
@<type identifier>
```

A pointer variable can only point to the type for which it is declared. This goes a long way to 'taming' the potentially dangerous pointer type, which (in languages such as PL1) is allowed to roam freely throughout memory, and can cause chaos if the programmer makes a small error in manipulating it. (In Pascal it, is always possible to use the type transfer function; but the programmer is obliged to tell the compiler that he is doing something risky.)

The <type identifier> need not be defined before the pointer type is defined, provided it is declared later in the declaration section. This is a forward type declaration, which is only permitted with pointer types. All pointer types include the predefined value NIL, which points to no element at all.

```
TYPE PTR = @LIST;
     LIST = RECORD
            VALUE : REAL;
            LOC   : 0..FF
            END;
```

PTR is declared to "point to the type LIST".

The operators applying to pointer operands with compatible types are:

:=	assignment
=	equal (TRUE if the operands point to the same address)
<>	not equal

Pointers allow storage to be dvnamically allocated from a storage area called the heap through the use of stan dard proced ure NEW. Pointers can also be used to construct data structures such as linked lists and binary trees. A linked list is easily created by defining a record which contains one field that is a pointer to the next record. Similarly, a binary tree of records can be constructed by defining a 'right link' and 'left link' pointer within the record.

4.5.6.6 Packed Types. The symbol PACKED may precede a record or array type definition. If a structure is declared to be PACKED, several unstructured components of the structure are stored in one word if possible. Packing may economize the storage requirements of a data structure but at the expense of efficiency of access of the components.

One example of a packed array is a string defined to be:

PACKED ARRAY [<index type>] OF CHAR

Characters are stored one per byte.

Details of the packing algorithm are presented in the Microprocessor Pascal User's Manual.

4.5.7 Type Compatibility and Transfer

Pascal has strict rules for compatibility between types. In general, incompatible types cannot appear on opposite sides of an assignment statement, or as operands of the same operator.

Two types are distinct if they are explicitly or implicitly declared in different parts of the program. A type is explicitly declared using a TYPE declaration. A type may be implicitly declared in a VAR declaration or in other places where a name is not associated with the type (e.g., in specifying an array index type).

Two types are compatible if one of the following is true:

1) They are identical types

2) Both are subranges of the same enumeration type

3) Both are string types with the same length

4) Both are pointer types which point to identical types

5) Both are set types with compatible base types

6) Both are file types with compatible element types.

Arrays or records are compatible only if they are declared to be of the exact same type.

There is no implicit conversion of types except from INTEGER and LONGINT to REAL and between INTEGER and LONGINT.

The strict compatibility rules of Pascal give the programmer a means of checking that he is not using a variable in the wrong place (for example, using the wrong variable to index an array; or specifying the indices of a multi-dimensional array in the wrong way). It is possible to completely ignore this facility by (for instance) not declaring any new types and specifying all array indices as unnamed subranges of integer. However, intelligent use of the TYPE concept can greatly reduce the possibility of errors, and make a program more readable and easier to change.

It is possible to override the compatibility check by using the type transfer facility, which temporarily changes the type of a variable. The form of a type transfer variable is:

<variable>::<type identifier>

e.g., PTR::INTEGER

The variable is temporarily treated as if it were the type specified after the double colon. No conversion is performed; only the apparent type of the variable is altered. Use of this facility transfers responsibility from the compiler to the programmer, therefore he needs to be sure he knows what he is doing.

It is also possible to override the type structure by using variants in record structures without checking the tag fields (see the Micprocessor Pascal User's Manual).

4.5.8 Variables

Variables are used to reference areas of storage within a module. A variable declaration associates with an identifier a location which can hold a value of a specified type. The form of a variable declaration is:

VAR <variable declaration list>

where <variable declaration list> is one or more of the following:

<identifier list> : <type definition>;

Where <identifier list> is a list of identifiers separated by commas. A <type definition> can be a standard type (INTEGER, REAL, etc.), the Name of a type defined in a type declaration statement, or a new type definition taking any of the forms allowed in a type declaration. In the last case, the new type will not have any name associated with it. For example:

```
VAR NYEARS : INTEGER;
    AMOUNT,VALUE,RATE : REAL;
    TEN_YEARS : VECTOR;
    PROFIT : ARRAY [1..10] OF BOOLEAN;
```

'Type VECTOR was defined in the example earlier in this chapter).

A variable can either be a simple identifier that references the entire variable, or may be a qualified variable which is used to reference part of a structured variable.

4.5.8.1 Indexed Variable.

An indexed variable is used to reference an element of an array. Its form is:

<variable> [<expression>,<expression>,.....,<expression>]

e.g., VECTOR [5]

The expressions are used to subscript into each of the n declared dimensions. If an array variable is declared to have n dimensions, then the indexed variable may have from 1 to n subscript expressions. For example, if an array is declared

```
A : ARRAY [1..10  1..20] OF INTEGER
```

then

```
A [5]
```

is a legal indexed variable; it is an

```
ARRAY [1..20] OF INTEGER
```

This array can itself be indexed, e.g.,

```
A [5] [6]
```

which is exactly equivalent to

```
A [5, 6]
```

The types of the subscript expression must correspond exactly with the declared index types. There is a compiler option to check the value of a subscript to make sure it is within the declared bounds.

4.5.8.2 Record Variable

A record variable is used to reference a field within a record. Its form is:

```
<variable>.<field identifier>
```

where <field identifier> is one of the fields declared in the record type definition.

```
PUMP_ONE.GRADE
C1.RE
START.DAY
```

Any record can be qualified; any array can be subscripted. Since it is possible to construct arrays of records and records containing arrays, variables such as

```
ARR [5] . FIELD [4]
```

are possible. Thus:

```
ARR                 is an array
ARR [5]             is a record
ARR [5] . FIELD     is an array
```

ARR [5] . FIELD [4] is an element

Very powerful and complex data structures can be built in this way.

4.5.8.3 Pointer Variable

A pointer variable is used to reference the variable pointed to by a pointer type. Its form is:

<variable>@

Where <variable> is a pointer type. The value of a pointer variable is undefined until either a value is assigned to it or a NEW is performed on it to allocate an area of dynamic storage. The constant NIL can be assigned to any pointer variable in order to have it point to nothing at all. A compiler option (CKPTR) is available to check if a reference is made to a NIL pointer.

4.5.9 Expressions

Expressions combine the values of variables and constants using operators to generate new values. Expressions consist of operands, operators, and function calls.

4.5.9.1 Operands. Operands reference the values of constants or variables. An operand may be one of the following:

<integer constant>
<real constant>
<string constant>
<character constant>
<constant identifier>
NIL
<set>
<variable>
<function call>

4.5.9.2 Operators. An operator specifies an operation that is to be performed on one or two operands. An operator can only be applied to two operands if their types are compatible. Some operators accept mixed mode operands; if an INTEGER value is added to a REAL, the INTEGER is first converted to REAL and then added to give a REAL result.

Operators have a precedence specifying the order of their evaluation in a complex expression.

The operators are:

Group 1 : Multiplying operators:

*	multiplication; set intersection
/	real division
DIV	integer division (divide and truncate)
MOD	modulus A MOD X = A - ((A DIV X) * X)

Group 2 : Adding operators:

+	addition; unary plus; set union
-	subtraction; unary minus; set difference

Group 3 : Relational operators:

=	equal
<>	not equal
<	less than; proper set inclusion
>	greater than; proper set inclusion
<=	less than or equal; set inclusion
>=	greater than or equal; set inclusion
IN	set membership

Logical operators:

Group 4:	NOT	Negation
Group 5:	AND	Conjunction
Group 6:	OR	Disjunction

The list of operators is in order of precedence with groups of higher precedence listed first. In an expresssion, operators of highest precedence are evaluated first; operators of equal precedence are evaluated from left to right within the expression. Parentheses may be used to alter the order of evaluation.

Examples:

Expression	Value
2 + 3 * 5	17
15 DIV 4 * 4	12
NOT (5 + 5 >= 20)	TRUE
6 + 6 DIV 3	8
3 < 5 OR 2 >= 6 AND 1 > 2	TRUE

In a BOOLEAN expression of the form:

X AND Y

if X is false, Y is not evaluated and the value of the expression is FALSE. Similarly, in a BOOLEAN expression of the form:

X OR Y

if X is TRUE, Y is not evaluated and the value of the expression is TRUE. This is called short circuit evaluation.

4.5.9.3 Function Calls. A function is a subroutine that returns a single value of a specific type. It is invoked by a function call:

<function identifier> <parameters>

If the function has parameters, these are of the form

(<actual parameter>,.....,<actual parameter>)

where each actual parameter is a variable or expression. The actual parameters must match the types of the formal parameters declared with the function.

4.5.10 SIMPLE STATEMENTS

Simple statements used by Pascal are described below.

4.5.10.1 Assignment Statement. The assignment statement specifies an expression that is to be evaluated and assigned to a variable. Its general form is:

<variable> := <expression>

e.g., x := 5

The symbol ':=' can be read 'becomes equal to'. The type of <expression> must be compatible with the type of <variable>, except that an INTEGER expression is automatically converted to LONGINT or REAL, and a LONGINT expression is automatically converted to INTEGER or REAL. Direct assignments can be made to variables of any type (including records, arrays, etc.) except files (and semaphores in

Microprocessor Pascal).

4.5.10.2 Procedure Statement. A procedure statement invokes the specified procedure. Its general form is:

```
<procedure name> ( <parameter list> )
                or
        <procedure name>
```

e.g.,CALCULATE_MEAN (A, 5, 4*X)

(There is no keyword corresponding to CALL in other languages)

Parameters must match in number and type with those declared with the procedure.

4.5.10.3 START Statement (Microprocessor Pascal only). A START statement is similar to the procedure statement except that it invokes the activation of a specified program or process to execute in parallel with the system.

```
START <identifier> ( <parameter list> )
                or
        START <identifier>
```

Where <identifier> is a <program name> or a <process name>.

A procedure statement invokes the procedure as part of the current sequential program; a START statement creates the new process as a separate site of execution that will be provided its own share of processing time by the executive in parallel with the current program. Once a process has been STARTed, the initiating program effectively loses control over it; except via the synchronization primitives such as semaphores and interprocess files. The started process becomes a separate entity.

4.5.10.4 ESCAPE Statement. The ESCAPE statement is a 'structured jump'. It is used to terminate execution of a structured statement, procedure or program (or process in Microprocessor Pascal). It allows an orderly exit to be made through the normal exit point of the structure. Its form is:

```
ESCAPE <identifier>
```

Where <identifier> may be an escape label, procedure name, program name (or process name).

An escape label, followed by a colon, may prefix any structured statement. Each escape label is implicitly declared by its appearance in the program, and can only be referenced within the structured statement it precedes. Unlike GOTO labels, ESCAPE labels need not be

declared at the start of the program.

```
LOOP: WHILE I <= N DO
        BEGIN
        IF EOF THEN ESCAPE LOOP;
        READ (VAL);
        SUM := SUM + VAL;
        I := I + 1
        END;
```

4.5.10.5 GOTO Statement. The GOTO statement is an unstructured jump:

```
GOTO <label>
```

It transfers system execution directly to the statement having the specified label.

A statement label is an unsigned integer which must be declared in a LABEL declaration at the start of the block in which it is used.

```
PROGRAM SAMPLE;
LABEL 2;
.
.
BEGIN
.
2 : I := I + 1;
    IF VECTOR [I] < 100 THEN GOTO 2;

.
END.
```

GOTO statements should be used as little as possible (if at all) because they tend to lead to 'spaghetti code' that is difficult to follow and prone to error. In some languages (e.g. FORTRAN) GOTO'S are necessary because the constructs needed to implement control structures are not complete. This is not the case in Pascal, which has a complete set of sequence, selection and iteration constructs that are sufficient to implement any program algorithm. In almost every case in which a GOTO may be used, an ESCAPE statement can be used instead, or the program can be restructured to eliminate the need for any jump at all. This will result in clearer code.

4.5.10.6 ASSERT Statement. The ASSERT statement allows the programmer to check, using a BOOLEAN expression, a condition that should be true at a particular point in the program. Its form is:

```
ASSERT <expression>
```

Where <expression> must be of type BOOLEAN.

For example, if it is known that a particular variable should never exceed a value of 100, the programmer can write

```
ASSERT X <= 100
```

at a suitable point in the program. If the BOOLEAN expression is not TRUE when the ASSERT statement is executed a run-time error occurs.

The ASSERT statement is particularly valuable for system debugging. Code for ASSERT statements is controlled by a compiler option. When the program is debugged the ASSERT option can be turned off to prevent ASSERT code being generated.

4.5.11 STRUCTURED STATEMENTS

Pascal provides structured statements that directly implement the design techniques introduced earlier in this book (see Subsection 2.6, 'Algorithms'). These statements have a single entry and exit points, which means that they automatically produce hierarchical, nested code. There are no jumps to confuse the programmer and upset the structure.

Pascal does provide a GOTO instruction; but it is purposely made difficult to use. All labels must be declared at the beginning of the program, which means that any departures from structured code are clearly visible.

4.5.11.1 Compound Statement. A compound statement is a sequence of statements enclosed by the keywords BEGIN and END. A compound statement is treated as a single statement.

```
BEGIN <statement list> END
```

where <statement list> is a list of Pascal statements, simple or structured, separated by semicolons. The statements making up the list are executed one by one in the order that they appear, but the entire list is treated as a single statement.

```
BEGIN
EXCHANGE := X1;
X1 := X2;
X2 := EXCHANGE
END
```

The semicolon is used to separate Pascal statements and is not part of any individual statement. Therefore a semicolon is not needed following the last statement in the list. If one does occur, the compiler simply assumes that there is an empty statement between the semicolon and END.

The empty statement is quite legal in Pascal and can occur in many places without causing any harm. However, the presence of an extra semicolon can sometimes change the meaning of a statement:

```
IF A = B
THEN
    X := 1;
ELSE
    Y := 1
```

The IF statement is terminated prematurely by the semicolon; ELSE is treated as a new statement and will be flagged as an error (because there cannot be a statement beginning with the keyword ELSE.

This particular error is easy to find because it will be picked up by the compiler Other cases of extra or missing semicolons may be more subtle; code may be generated that is logically wrong but syntactically correct, so that the compiler will not find it. Therefore it is as well to know exactly where semicolons are needed and why.

The compound statement implements the sequence construct described in Chapter II.

4.5.11.2 IF Statement. The IF statement specifies execution of one of two alternative statements depending on a condition. The second alternative may be the empty statement. The form of the IF statement is:

```
        IF <expression> THEN <statement>
                        or
IF <expression> THEN <statement> ELSE <statement>
```

where <expression> must be of type BOOLEAN.

If the expression evaluates to TRUE the first <statement> alternative, the THEN clause, is executed; otherwise the second <statement> alternative, the ELSE clause, is executed if it is present. The <statement>s can be any Pascal statements, including compound statements and other IF statements.

Examples:

```
IF COUNT >= 0 AND COUNT <= LENGTH
    THEN READ (X[I]);
```

```
IF X < Y THEN MAX := Y
    ELSE MAX := X;
```

In nested IF statements, there is a possible ambiguity with regard to ELSE clauses. This is resolved by always associated an ELSE with the most recent unmatched THEN.

```
IF A > B THEN IF B > C THEN MIN := C
    ELSE MIN := B;
```

is equivalent to:

```
IF A > B THEN
    BEGIN
    IF B > C THEN MIN := C
        ELSE MIN := B
    END;
```

In cases such as this, it is wise to use explicit BEGIN...ENDs to make the logical structure perfectly clear.

4.5.11.3 CASE Statement. The CASE statement is an extension of the IF statement allowing more than two alternatives. A CASE statement allows a statement to be selected for execution depending on the evaluation of an expression at run time. The form of a CASE statement is:

```
CASE <expression> OF
    <case label list> : <statement>;
            . . .
    <case label list> : <statement>
    OTHERWISE <statement list>
END
```

where <expression> must be of an enumeration type. <case label list> is a list of one or more <case label>s separated by commas. The <case label list> : <statement> combination may be repeated any number of times within the CASE statement; each occurence must be separated from the previous one by a semicolon. The OTHERWISE clause is optional.

A <case label> is either a constant value or a subrange value of the same enumeration type as the <expression`. Each <case label list> specifies the list of values of <expression> for which the corresponding <statement> alternative will be executed.

The value of <expression> at run time is used as the selector into the CASE statement. If the <case label> indicated by the selector is present in the CASE statement. the corresponding <statement> is executed; otherwise the <statement list> following the OTHERWISE clause is executed. If the selected <case label> is not present and there is no OTHERWISE clause, a run-time error will occur.

Examples:

```
CASE NUM OF
    0..3,8 : TOTAL := TOTAL + NUM;
    4,6,7  : TOTAL := TOTAL - NUM;
    5,9    : TOTAL := TOTAL DIV 2
END;

CASE ALFA OF
    'A'..'M' : CH := SUCC(ALFA);
    'N'..'Z' : CH := PRED'ALFA)
    OTHERWISE
        WRITELN('NOT IN ALPHABET');
        INT := ORD(ALFA)
END;
```

The IF and CASE statements implement the selection construct described in Section 2 6.

4.5.11.4 FOR Statement. The FOR statement provides the repeated execution of a given statement for a progression of values that are assigned to the control variable of the FOR statement. This statement should be used if the number of repetitions required is known before the statement is executed. The form of the FOR statement is one of the following:

```
FOR <identifier> := <initial value> TO <final value>
     DO <statement>
                    or
FOR <identifier> := <initial value> DOWNTO <final value>
     DO <statement>
```

where <identifier> is the control variable, and <initial value> and <final value> are of the same enumeration type which must not be a set type.

The control variable is implicitly declared by its appearance in the FOR statement, and therefore may only be referenced within the FOR statement. If a variable of the same name has previously been declared, that variable will be temporarily inaccessible within the FOR statement. The value of the control variable may not be changed within the FOR statement.

The control variable is assigned the <initial value> prior to the first execution of the <statement>. If the <initial value> is greater (less) than the final value in the TO (DOWNTO) clause, the <statement> is never executed. Otherwise, after each execution of the <statement> the control variable is incremented (decremented) by one until the value of the control variable is greater (less) than the <final value>. Both <initial value> and <final value> are only evaluated

once on entering the FOR statement, so that the total number of repetitions is determined at that time.

Examples:

```
FOR I := N DOWNTO 1 DO
    SUM := SUM + A[I];

FOR DAY := MON TO FRI DO
    BEGIN
    READ(HRS, RATE);
    PAY[DAY] := RATE * HRS
    END;
```

4.5.11.5 WHILE Statement. The WHILE statement allows for the repeated execution of a given statement as long as a specified condition remains true. The form of the WHILE statement is:

```
WHILE <expression> DO <statement>
```

where <expression> is of type BOOLEAN.

<expression> is evaluated before each execution of <statement>. If the <expression> is false initially, <statement> is not executed at all; otherwise it is executed repeatedly as long as <expression> evaluates to true.

The WHILE statement is used where the number of repetitions cannot easily be predicted in advance. For example, <expression> might represent the state of an external input.

Examples:

```
I := 1;
WHILE I <= MAX DO
    BEGIN
    VALUE := AMT[I] + TAX[I+2];
    I := I + 1
    END;
```

There is an alternative form of WHILE statement called the REPEAT...UNTIL:

```
REPEAT
<statement list>
UNTIL <expression>
```

where <expression> is BOOLEAN.

The difference is that <expression> is evaluated after each execution of <statement list>; so that even if it is false <statement list> is always executed at least once.

It is a good idea to standardize either on WHILE or REPEAT to avoid confusion on what happens when <expression> is false. In general, the WHILE construct is more flexible because it includes the important special case of zero iterations. REPEAT....UNTIL can then be used as an optimization technique for the rare cases when an action must always be performed at least once.

The structure diagram iteration symbol is intended to be a WHILE, and is best kept as such. A REPEAT....UNTIL construct can then be written explicitly as:

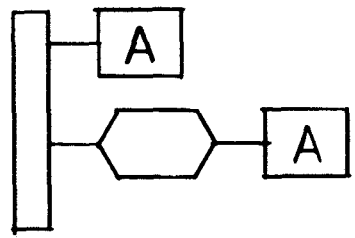
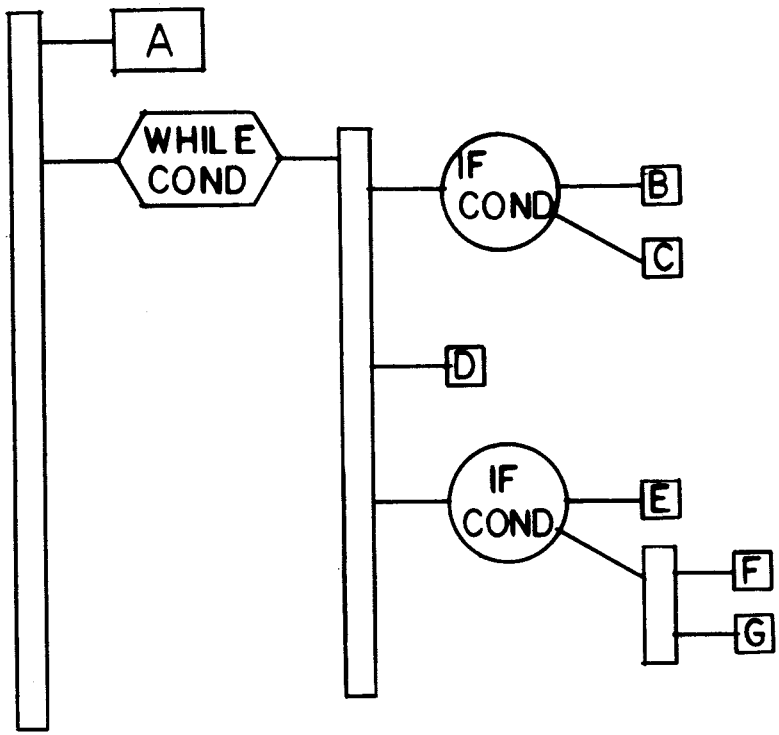


FIGURE 4-5. REPEAT UNTIL CONSTRUCT

This is often a truer reflection of the situation, because in a case like this there is usually something special associated with the first iteration.

With the sequence, selection and iteration constructs described, Pascal programs can be written directly from the software design:




```
BEGIN
A;
WHILE COND DO
  BEGIN
  IF COND THEN
    B
  ELSE
    C;
  D;
  IF COND THEN
    E
  ELSE
    BEGIN
    F;
    G
    END
  END
END
```

FIGURE 4-6. PASCAL PROGRAM

If the Pascal code is indented to reflect the structure, there is a strong visual resemblance between the program and the structure diagram, which can be used as a check.

4.5.12 File Manipulation

The standard procedures READ and WRITE are provided for input from and output to files. In addition, the procedures READLN and WRITELN (read and write line) apply to text files. These are described in full in the Microprocessor Pascal System User's Manual.

4.5.13 Standard Routines

Pascal provides a number of standard procedures and functions, of which a list is provided in the Reference Section.

4.5.14 Dynamic Storage Allocation

Dynamic memory areas referred to as heap packets may be allocated or deallocated during program execution through use of the standard procedures NEW and DISPOSE. These are described in the Micprocessor Pascal System User's Guide.

4.5.15 CRU I/O

Pascal supports direct 9900 CRU I/O (see Section VI) with the following standard procedures:

```
CRUBASE (BASE)
LDCR (WIDTH, VALUE)
SBO (DISP)
SBZ (DISP)
STCR (WIDTH, VALUE)
```

and the BOOLEAN function:

```
TB (DISP)
```

These are described in detail in the MPP User's Guide.

4.6 CONCURRENCY

The microprocessor executive and microprocessor interpretive executive (as well as the interactive Host Debugger provide for concurrent execution of a multiple-process system. This section describes some of the functions performed by the executive, and also the mechanisms provided for synchronization and communication between processes.

4.6.1 Processes

System, program and process declarations were described earlier in this chapter. When a system is executed, the system <process body> is automatically started. All other processes, including programs, must

be explicitly started using the START statement. The system <process body> therefore usually contains a series of START statements. When a process is started, stack space is allocated to it from the starter's own stack. The amount of stack space to be allocated to a process is set using the concurrent characteristic

```
(*# stacksize = required_stack_size *)
```

The concurrent characteristics are part of the process declaration.

A process's execution is terminated when it runs to completion or by using the run-time support routine P$ABORT to abort it.

A process can be in one of three states:

1) Ready - the process is able to run but there is a higher priority processes currently executing.

2) Active - the process is being executed. Under Microprocessor Pascal, the active process (there can only be one) is always the one with the highest priority.

3) Blocked - the process is suspended (waiting for a signal from another process, or perhaps from an interrupt) and unable to run until the event has occurred.

4.6.2 Process Record

Each process has a unique process record. This is used by the executive to access information concerning a particular process (where its stack is located, its identity, its priority, etc.). The process record is used for storing its volatile environment: display, program counter (PC), workspace pointer (WP), and status register (ST). (For an explanation of PC, WP and ST see Chapter VI.)

The display is a 16-word area containing addresses of the stack frames that can be accessed by the currently executing routine. The display is a 'short cut' means of access to remote stack frames that is quicker than tracing back through the stack frame linkage.

4.6.3 Semaphores

Processes are independent. However, it is often necessary to synchronize their actions. The simplest way of doing this is via the semaphore and its primitive operations wait and signal. Until they have completed, nothing must access the semaphore, the queues they operate on, or the operations themselves. This indivisibility is assured by setting the interrupt mask to zero on entry to the routines, and then resetting it back to its previous value on exiting them.) The basic idea of a semaphore is described below.

The semaphore is considered to be so fundamental to process synchronization that Micprocessor Pascal predefines it as a type structure composed of two elements: a non-negative counter of unserviced events and a queue (possibly empty) of suspended processes. The queue uses First In First Out (FIFO) ordering.

SIGNAL : Increments the counter if the semaphore queue is empty, otherwise it activates the first process in the queue. (The process is removed from the semaphore queue and reinserted into the scheduling queue.)

WAIT : Decrements the counter if it is non-zero, otherwise the issuing process (the currently active process) is suspended. (The process is removed from the scheduling queue and inserted into the semaphore queue.)

Before a semaphore can be used, it must first be initialized. This is performed using the INITSEMAPHORE routine.

A variable of type SEMAPHORE cannot be changed directly by a user's program. It can only be accessed via the EXTERNAL run-time support semaphore routines When declaring these external routines it is also necessary to declare the following:

```
TYPE semastate = (awaited,zero,signaled);
```

The Micprocessor Pascal Run-Time Support system gives greater flexibility in handling semaphores by providing additional routines to the wait and signal operations (a full list of these can be found in the Reference Section of Chapter VI).

4.6.4 Process Svnchronization

A process that is dependent on the occurence of an event can perform a WAIT to ensure that the event has occured. If it has, the waiting process executes. If not, the waiting process is suspended in that semaphore's queue until the event occurs. A SIGNAL operation performed on the associated semaphore allows a process to signal the occurence of an event. If some process is waiting for the event, it is made ready for execution (the process is removed from that semaphore's queue and inserted into the ready queue). In both of these cases, the process that called SIGNAL remains in the active state. The semaphores of the Executive RTS can thus be thought of as "counting" semaphores in that occurence of an event is never lost, even if no process is waiting when the event occurs. A count is kept in the semaphore of all events that occurred (by SIGNAL) but were not received (by WAIT).

When semaphores are used to ensure exclusive access to two or more resources, extreme caution must be exercised to prevent a condition

known as "deadlock". This takes place when a situation is created in which two or more processes are suspended, awaiting a condition that cannot happen because there is no active process to cause the needed event to occur.

For example, consider two simultaneously executing processes (A and B) both requiring exclusive access to resources (X and Y). The following sequence will result:

A gets X .. A requests Y B gets Y .. B requests X

In the above example, neither A nor B will ever resume execution, as A will be waiting for Y (which B has) and B will be waiting for X (which A has) To prevent a situation such as this, either and/or both processes must check the availability of succeeding resources and, if unavailable, release those already acquired.

4.6 5 Interprocess Communication

4.6.5.1 Shared Variables. The simplest form of interprocess communication is accomplished through the sharing of variables. A nested process can access all its parent's variables. (Heap variables can also be accessed since it is possible to pass pointers as parameters to a process.)

However, it is essential that only a single process is allowed to operate on any shared variable at a time. This can be achieved by representing the shared variable as a record structure containing a mutual exclusion semaphore, and enclosing any code sections referencing the variable with wait and signal operations on the semaphore. For example:

```
VAR b: RECORD
       mutex: SEMAPHORE;
       shared_variable: any_type;
       END;

WITH b DO
BEGIN
   wait(mutex);
   ($ access/modify shared_variable $)
   signal(mutex);
END;
```

4.6.5.2 Message Buffer. A message buffer is a shared data structure through which interprocess communication is possible. It allows a process to send messages to another process without the sender having to wait until the receiver is ready for the message (i.e., the messages are buffered). In this context a message is any structure that can be copied from one process to another.

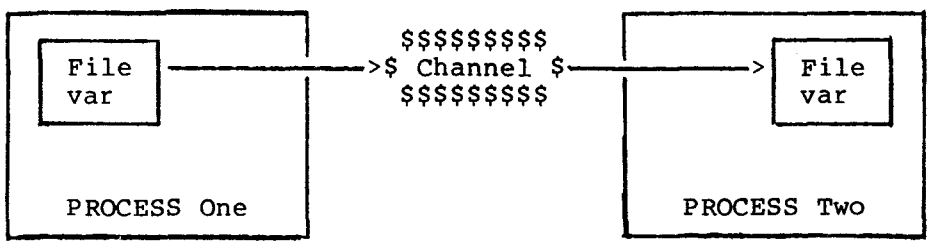
Note: Deadlock could result if the order of the wait operations is reversed in either routine.

Updating the buffer element pointers, NEXT_IN and NEXT_OUT, by MODing them with MAXUSERS and then adding one allows the message buffer to be used in a circular fashion (a buffer managed in this way is known as a circular or ring buffer).

Note: MESSAGE_IN and MESSAGE_OUT must be variables of type MESSAGE.

4.6.5.3 Interprocess Files. The third interprocess communication mechanism involves file variables that are linked together via channels. In this application, a file is not associated with any external storage medium; it is simply a logical device to enable one process to send information to another. The communication mechanism is handled by the executive. However, the same routines (READ, WRITE, etc) used are the same utilized with external files.

Channels are system maintained shared data structures that conduct information between file variables and synchronize the execution of participating processes.



A file variable has a character string name; initially this is the same as the variable's identifier, but it can be changed using the procedure SETNAME.

A channel also has a character string name. It is identical to the names of all file variables connected to it.

Files must be opened by calling the procedure REWRITE for write operations and RESET for read operations before any I/O can be performed. (If the file is already open, then it is automatically closed before it is reopened in the appropriate mode.) This also causes the file variable to be connected to a channel with the same name as the file variable. If no channel exists by that name, one is created and given the appropriate characteristics.

Closing an open file (using the procedure CLOSE, or by exiting a routine in which a file variable is declared) also disconnects the file variable from the channel. A channel is normally destroyed when all file variables have been disconnected from it

The following allows processes A and B to communicate with each other via the channel TRANSFER. Process A opens the file TRANSFER for

A message buffer is of the form:

```
CONST max_messages = ....   ($ some number $)
TYPE message_index = 1..max_messages;
     message = some user_defined_structure;
VAR message_buffer: RECORD
                    mutex.not_empty,not_full: SEMAPHORE;
                    next_in,next_out: message_index;
                    buffer : ARRAY[message_index] OF message;
                    END;
```

MUTEX - Ensures mutual exclusion (initialized to 1)
NOT_EMPTY - Indicates how many messages are in the buffer (initialized to 0)
NOT_FULL - Indicates how many vacant elements in the buffer (initialized to max_messages)

NEXT_IN - Where the next message is to be stored
NEXT_OUT - Where the next message is to be taken from

Initially, NEXT_IN=NEXT_OUT=0.

To deposit a message into the buffer:-

```
WITH message_buffer DO
BEGIN
   wait(not_full);
   wait(mutex);
   buffer[next_in]:=message_in;
   next_in:=next_in MOD maxusers +1;
   signal(mutex);
   signal(not_empty);
END;
```

To remove a message from the buffer:-

```
WITH message_buffer DO
BEGIN
   wait(not_empty);
   wait(mutex);
   message_out:=buffer[next_out];
   next_out;=next_out MOD maxusers +1;
   signal(mutex);
   signal(not_full);
END;
```

writing, while process B opens it for reading.

```
PROCESS a(.....);                PROCESS b(.....);
VAR transfer: TEXT;              VAR transfer: TEXT;
.                                .
rewrite(transfer);               reset(transfer):
writeln(transfer,...);           readln(transfer,...);
.                                .
```

A similar effect would be produced by:

```
PROCESS a(output:TEXT;...);      PROCESS b(input:TEXT;...);
.                                .
.                                reset(input);
writeln(.....);                  readln(.....);
.                                .

.
start a(filenamed('transfer'),...);
start b(filenamed('transfer'),...);
```

The function FILENAMED results in a file with the initial name equal to the specified string.

It is not necessary to perform a REWRITE operation in process A as this is automatically performed on the default output text file OUTPUT.

Note: Each peripheral device in the system is identified by an alphanumeric name (from 1 to 8 characters long) and has a dedicated channel of the same name.

4.7 INTERRUPT HANDLING

The 990 range of processors recognize 16 distinct interrupt levels, numbered 0 (highest priority interrupt) to 15 (lowest priority interrupt). A full description of the 990 interrupt structure is given in Chapter VI.

A device process is a process that has been written to service a particular interrupt level. These processes are identified by their priorities. All processes in this system are assigned a priority, in the range 0 to 32,677. The first 16 priorities, 0 to 15, are reserved for use by device processes.

A process with a priority of 5 may service level 5 through level 15 interrupts. A process's priority is set using the concurrent characteristic

```
($# priority = interrupt_level $)
```

If a number of devices all use the same interrupt level, then that level's device process must first determine which device actually

caused the interrupt before it can start servicing it.

All interrupts except the level 0 interrupt (RESET), are disabled by calling the procedure MASK. The procedure UNMASK enables interrupts which are more urgent than the priority of the calling process.

A correspondence is established between an interrupt level and a semaphore using the procedure EXTERNALEVENT. A device process executes a WAIT on the semaphore associated with its interrupt level. When an interrupt occurs, the executive performs a SIGNAL on the semaphore associated with the interrupt level, thus activating the suspended device process.

The procedure ALTEXTERNALEVENT allows the user to specify an alternative process that will be executed if the primary process is not suspended on the interrupt's semaphore. This procedure is intended to be used to service unexpected or spurious interrupts.

The correspondence between a semaphore and an interrupt level can be broken using the NOEXTERNALEVENT procedure, while the alternative process correspondence can be broken by the NOALTEXTERNALEVENT procedure.

```
PROGRAM level_7_handler(....);
VAR level_7_sem,spurious_level_7: SEMAPHORE;
.
    PROCESS interrupt_7(level: SEMAPHORE);
    .
    BEGIN
        ($# priority=7;..... $);
        WHILE true do
        BEGIN ($ do forever $)
            wait(level);
            .
        END   ($ forever loop $)
    END;
    PROCESS spurious_7(level: SEMAPHORE);
    .
    BEGIN
        ($# priority=7;..... $);
        wait(level);
        .
    END
BEGIN ($ level_7_handler $)
    .
    initsemaphore(level_7_sem,0);
    initsemaphore(spurious_level_7,0);
    externalevent(level_7_sem,7);
    altexternalevent(spurious_level_7,7);
    start interrupt_7(level_7_sem);
    start spurious_7(spurious_level_7);
END   ($ level_7_handler $)
```

Under TIPMX (the concurrent executive provided for use with TI Pascal)

interrupts are serviced using the WAITINTERUPT procedure. An active device process executes the WAITINTERRUPT procedure. This suspends the calling procedure until an interrupt occurs at the level equal to the process's priority. Using WAITINTERRUPT in a process whose priority is greater than 15 results in an error and the call fails. This procedure is included in the Micprocessor Pascal run-time support system for compatibility with earlier products. As WAITINTERRUPT is implemented using semaphores and several of the semaphore handling routines, the Micprocessor Pascal user is not encouraged to use it in new applications.

4.8 REFERENCE SECTION

4.8.1 System Commands

Create/edit a file	EDIT
Compile a Pascal program	COMPILE
Debug a compiled Pascal program	DEBUG
Execute a compiled Pascal program	EXECUTE
Save an edited file	SAVE
Display a stored file	SHOW
Terminate an MPP session	QUIT
Compile and save a Pascal program	$ BATCH
Print a stored file	$ PRINT
Execute sci command	$ SCI
Wait for background task to finish	$ WAIT
Delete files and synonyms used	$ PURGE
Copy text files	$$ COPY
File utility program	$$ UTILITY

$ only for DX990 users
$$ only for FS990 users

4.8.2 Debug Commands

Resume execution	G O
Help	HELP(<command>)
Select default process	SDP(<process>)
Terminate DEBUG session	QUIT
Display process	DP([<process>])
Display all processes	DAP
Assign breakpoints	AB(<routine>,<statement number>)
Delete breakpoints	DB(<routine>,<statement number>)
Delete all breakpoints	DAB(<process>)
List breakpoints	LB([<process>])
Execute single instruction	SS([<process>],<flag>)

Show stack frame	SF([<routine>],[<disp>],[<length>])
Show heap packet	SH([<routine>],[<disp>],[<length>])
Show common area	SC(common name,[<disp>],[<length>])
Show indirect variable value	SI(<routine>,<disp>,[<length>])
Show absolute memeory location	SM(<address>,[<length>])
Modify stack frame value	MF(<routine>,[<disp>],[<ver>],<value>)
Modify heap value	MH(<routine>,[<disp>],[<ver>] <value>)
Modify common value	MC(<routine>,[<disp>],[<ver>],<value)
Modify indirect variable	MI(<routine>,<disp>,[<ver>],<value>)
Modify memory	MM(<routine>,[<ver>],<value>)
Trace process execution	TP([<process>],<flag>)
Trace routine entry/exit	TR([<process>],<flag>)
Trace statement flow	TS([<process>],<flag>)
DEBUG the process	DEBUG(<process>)
Breakpoint process	BP(<process>)
Hold process	HP(<process>)
Release process	RP(<process>)
Connect input file	CIF(<file1>,[<file2>])
Connect output file	COF(<file1>,[<file2>])
Simulate interrupt	SIMI(<level>)
Select CRU mode	CRU([<process>],<cru mode>)

[<x>]
Indicates that the item <x> is optional.

<process>
If omitted it defaults to that set by SDP. It may be either a name (younges2 instance of the process) or an integer constant (older instance of a particular process), found using DAP.

<routine>
May be either a name (most recent activation of the routine) or an integer constant (earlier activation), found using DP. Optionally it specifies the process which activated it by preceding <routine> by <process> followed by '.' .

<flag>
Is an identifier that is either TRUE or FALSE : if TRUE command is enabled: if FALSE command is disabled.

<disp>
Is the byte displacement.

<ver>
Is the old value of the variable being modified, if it does not match the actual value an error occurs.

<file1>
8 character Microprocessor Interpretive Executive file name identifier.

<file2>
File pathname specified as a string (enclosed in double quotes). If omitted it defaults to user's terminal.

<cru mode> One of the following :

EXECUTE	Execute all CRU instructions
OFF	Ignore all CRU instructions
DEBUG	All CRU I/O is simulated by the user (default value)

NOTE : Parenthesis may be omitted if all the parameters are optional or defaulted. Trailing commas may be omitted.

4.8.3 Utility Commands (990/4 only)

Create a file	CF,<file name>
Compress a file	CM,<file name>
Change file name	CM,<old file name>,<new file name>
Change file protection	CP,<file name>,<U or W or D>
Delete file	DF,<file name>
Change listing file/device	DO,<file or device name>
Receive file across data link	DR,<file name>
Transmit file across data link	DT,<file name>
Map disc	MD,<disc name>
Display time and date	TI
Terminate utility program execution	TE

4.8.4 EDIT Commands

Help	CMD HELP
Edit/compose mode	F7 kev
Syntax check	CMD CHECK
Terminate and save edit	CMD QUIT
Terminate without saving	CMD ABORT
Change editing files	CMD INPUT
Scroll file down	F1 key
Scroll file up	F2 key
New line	RETURN key
Tab	SHIFT TAB SKIP key
Back tab	FIELD key
Set tab increment	CMD TAB(<character count>)
Move cursor up	up-arrow key
Move cursor down	down-arrow key

Move cursor right	right-arrow key
Move cursor left	left-arrow key
Move to home position	HOME key
Find [nth occurrence of] specified pattern	CMD FIND(<pattern>, [<occurence number>])
Relative positioning	CMD [+ or -]<line count>
Move to top of file	CMD TOP
Move to bottom of file	CMD BOTTOM
Insert line before	unlabelled grey key
Duplicate line	F4 key
Delete line	ERASE INPUT key
Skip to next tab setting	TAB SKIP key
Insert character	INS CHAR key
Delete character	DEL CHAR key
Clear line	ERASE FIELD key
Replace strings [n times]	CMD REPLACE(<original pattern>, <new pattern>,[<repeat count>])
Split line	F8 key

CMD HELP -
Strike the CMD key and then type in the word HELP.

[<exp>]
Indicates that item <exp> is optional.

<pattern>
Is either a string of characters enclosed within double quotes or an identifier.

NOTE:
Optional items may be omitted (they default to the value 1) along with any preceding commas.

4.8.5 STANDARD Routines

All functions marked '$' must be declared EXTERNAL.

DEFINITION	FUNCTION	ARGUMENT	RESULT
Absolute value	ABS	INTEGER,LONGINT, REAL	As argument
$ Arctangent	ARCTAN	REAL	REAL
Character corresponding	CHR	BOOLEAN,INTEGER, SCALAR type	CHAR
$ I/O column index	COLUMN	TEXT	INTEGER
$ Cosine	COS	REAL	REAL
End of file/medium	EOF	TEXT,FILE	BOOLEAN
End of line	EOLN	TEXT	BOOLEAN
$ Exponential	EXP	REAL	REAL
Create file connection	FILENAMED	string	ANYFILE
Real conversion	FLOAT	INTEGER,LONGINT	REAL
$ Natural logarithm	LN	REAL	REAL
Convert to longint	LINT	INTEGER,LONGINT, REAL	LONGINT
Address or entry point	LOCATION	variable, procedure,process	INTEGER
Round convert longint	LROUND	REAL	LONGINT
Truncate,convert longint	LTRUNC	REAL	LONGINT
Odd number?	ODD	INTEGER,LONGINT	BOOLEAN
Ordinal position	ORD	BOOLEAN,CHAR, SCALAR type	INTEGER
Predecessor	PRED	ENUMERATION type	As argument
Round	ROUND	REAL	INTEGER
$ Sine	SIN	REAL	REAL
Return size (bytes)	SIZE	type,variable; tagfields	INTEGER
$ Status of last I/O op	STATUS	ANYFILE	INTEGER
Square	SQR	INTEGER,LONGINT, REAL	As argument
$ Square root	SQRT	REAL	REAL
Successor	SUCC	ENUMERATION type	As argument
Truncate,convert integer	TRUNC	REAL	INTEGER

4.8.6 CRU Operations

```
PROCEDURE CRUBASE (base:INTEGER)
PROCEDURE LDCR (width,value:INTEGER)
PROCEDURE SBO (displacement:INTEGER)
PROCEDURE SBZ (displacement:INTEGER)
PROCEDURE STCR (width:INTEGER; VAR value:INTEGER)
FUNCTION TB (displacement:INTEGER): BOOLEAN
```

Where <width> is a number in the range 0 to 15.
<displacement> is a number in the range -128 to +127.

4.8.7 Standard Procedures

All procedures marked '$' must be declared **EXTERNAL**.

```
CLOSE (f);
```

Place file <f> in closed state.

```
DATE (v);
```

Get the current date in the form 'YY.DDD', where 'YY' is the year and 'DDD' is the Julian day.

```
DECODE (s,n,stat,q);
```

Convert string <s>, starting at the <n>th component of <s> into a form compatible with the read variable <q> (see NOTE 2) and store it in <q>. The status of the operation is returned in <stat>.

```
DISPOSE (p);
DISPOSE (p,t1,..,tn);   t1..tn -> tagfields
```

Deallocate the heap packet specified by <p> and set <p> to NIL.

```
ENCODE (s,n,stat,p);
```

Convert the write parameter <p> (see NOTE 1) into character format and store the result in <s>, starting at the <n>th component. The status of the operation is returned in <stat>.

```
HALT
```

Terminate program execution.

```
IOTERM (f,oldvalue,newvalue);
```

Change file <f>'s default error termination flag to <newvalue> and return the original in <oldvalue>.

```
MESSAGE (x);
```

Write the string <x to the system message file.

```
NEW (p);          NEW (p,tl,..,tn);   tl..tn -> tagfields
```

Allocate heap space to the variable referenced by <p> and return a pointer to it in <p>.

```
PACK (a,i,z);
```

Pack the components of array <a> into the packed array <z>, starting at the <i>th element of <a>.

```
PAGE (f);
```

Continue output of file <f> on a new page.

```
READ (f,vl,..,vn);
```

```
TEXT      READ (vl,..,vn); ---> READ(INPUT,vl,..,vn);
```

```
RANDOM    READ (f,recnum,vl,..,vn);
```

Read the components of a sequential, text or random file into the specified variables <vi> (see NOTE 2). If the first argument is not a file variable <f>, the file INPUT is used. For RANDOM files the second argument specifies the logical record number <recnum>, starting from zero. For sequential and RANDOM files, the remaining arguments must be compatible with the particular file components.

```
READLN (f,vl,..,vn);
READLN (vl,..,vn); ---> READLN(INPUT,vl,..vn);
READLN(INPUT);
```

Read the components of a text file into the specified variables and then carry on reading until the next end-of-line marker has been read.

```
RESET (f);
```

Opens a file <f> for input and positions it to its first component. If a sequential or text file is empty then EOF'F) becomes true, otherwise it is false.

```
REWRITE (f);
```

marks a file <f> as empty and then opens it for output. For a sequential or text file EOF(F) becomes true. This is automatically performed for OUTPUT.

```
SETNAME (f,name);
```

Associate logical channel <f> to the physical file <name>. <Name> may

not be the file OUTPUT.

```
TIME (v);
```

Get the current time of day in the form 'HH.MM.SS'. <V> is a packed array (1 to 8) of char.

```
UNPACK (z,a,i);
```

Unpack the components of the packed array <z> into the array <a> starting at the <i>th element of <a>.

```
WRITE (f,vl,..,vn);

TEXT      WRITE (vl,..,vn); ---> WRITE(INPUT,vl,..,vn);

RANDOM    WRITE (f,recnum,vl,..,vn);
```

Write the components to a sequential, text or random file from the specified variables <vl>..<vn> (see NOTE 2). If the first argument is not a file variable <f>, the file OUTPUT is used. For RANDOM files the second argument specifies the logical record number <recnum>. starting from zero. For sequential and RANDOM files, the remaining arguments must be compatible with the particular file components.

```
WRITELN (f,vl,..,vn);
WRITELN (vl,..,vn); --->WRITELN(OUTPUT,vl,..vn);

WRITELN(OUTPUT);
```

Write the components to a text file <f> from the specified variables <vl>..<vn> (see NOTE 1) and then write an end-of-line marker.

NOTE 1: WRITE VARIABLES for TEXT files may be of the form :

```
E | E:M | E:M:N
```

E is an expression of type CHAR, INTEGER, LONGINT, BOOLEAN, REAL or a string.

If M (an INTEGER expression) is present then it represents the minimum field width. If M is omitted, and E is REAL, then its value is written in floating point format.

If N (an INTEGER expression) is specified then the real number E will be output in fixed point format with N digits after the decimal point.

If E is INTEGER or LONGINT then the value may be written as a string of hex digits (not preceded by #) in the form :

```
E  HEX 2 E:M  HEX
```

If E is BOOLEAN then the identifier FALSE or TRUE is written preceded by M-5 blanks. If M<5 then the character T or F is written.

If E is a string (PACKED ARRAY of characters) then the whole string is output.

Default field widths for WRITE operations :

INTEGER	10
LONGINT	15
REAL	15
BOOLEAN	5
CHAR	1
Hex	10
String	length of string

NOTE 2: READ VARIABLES for TEXT files :

V is a variable to be assigned the value read and must be either CHAR, INTEGER, LONGINT, BOOLEAN, REAL or a string.

If V is a CHAR then V is assigned the next character read.

If V is a string of length L then the next L characters are read.

If V is BOOLEAN then either the character T or F is read or the identifier TRUE or FALSE.

If V is INTEGER. LONGINT or REAL then a sequence of characters that makes up the number is read. The sequence may be terminated by any character that is not part of the number. Preceding blanks and end-of-line markers are skipped. If the field is blank the value read is zero.

TEXT I/O RETURN CODES

0	Normal completion
1	Bad parameter passed to I/O routine
2	Field width too large for logical record
3	Incomplete data (READ only)
4	Invalid character in field (READ only)
5	Data value too large (READ only)
6	Attempt to read past end-of-file (READ only)
7	Field larger than logical record size

4.8.8 Microprocessor Pascal Executive

All Micprocessor Pascal executive procedures/functions must be declared EXTERNAL.

4.8.8.1 Processor Management (Scheduling) Routines.

```
TYPE non_device priority = 16..32766;

PROCEDURE setpriority(VAR oldvalue: non-oldvalue: non_device_priority:
                        newvalue: non_device_priority);
```

changes the priority of the first non-device process in the scheduling queue.

```
PROCEDURE SWAP:
```

removes the first non-device process from the scheduling queue and inserts it behind the last process with the same priority.

4.8.8.2 Semaphore Routines.

```
TYPE nonneg = 0..32766;
     semaphorestate = (awaited, zero, signaled);

PROCEDURE CSIGNAL(sema: SEMAPHORE; VAR waiter: BOOLEAN);
```

performs a conditional signal operation on <sema> - i.e., if a waiter exists on this semaphore a SIGNAL operation is performed on it and <waiter> is set to TRUE.

```
PROCEDURE CWAIT(sema: SEMAPHORE; VAR received: BOOLEAN');
```

performs a conditional wait operation on <sema> - ie if it has been SIGNALed a WAIT operation is performed on it and <received> is set to TRUE.

```
PROCEDURE INITSEMAPHORE(VAR sema: SEMAPHORE; count: nonneg);
```

initializes the semaphore <sema> to <count> and sets the queue management to FIFO.

```
PROCEDURE SIGNAL(sema:SEMAPHORE);
```

performs a SIGNAL operation on <sema>.

```
PROCEDURE WAIT(sema: SEMAPHORE);
```

performs a WAIT operation on <sema>.

```
PROCEDURE TERMSEMAPHORE(VAR sema: SEMAPHORE);
```

returns the space occupied by the semaphore <sema> to MPX. If there is anything waiting on it an error occurs.

```
PROCEDURE WAITSIGNAL(WAITFOR, signalthe: SEMAPHORE);
```

performs a WAIT operation on <waitfor> and a SIGNAL operation on <signalthe> in an indivisible manner.

```
FUNCTION SEMASTATE(SEMA: SEMAPHORE): semaphorestate;
```

returns the state of the semaphore <sema>.

```
FUNCTION SEMAVALUE(SEMA: SEMAPHORE): INTEGER;
```

returns the count of <sema>'s initial value plus the total number of SIGNALs performed on it minus the total number of WAITs performed on it.

4.8.8.3 Semaphore Attributes.

```
    TYPE interrupt_level = 0..15;

PROCEDURE ALTEXTERNALEVENT(SEMA: SEMAPHORE;
                  level:  interupt-level);
```

attaches the semaphore <sema> to the interrupt <level> as the alternative receiver of an interrupt.

```
PROCEDURE EXTERNALEVENT(SEMA: SEMAPHORE;
                  level: interrupt_level);
```

attaches the semaphore <sema> to the interrupt <level> as the primary receiver of an interrupt.

```
PROCEDURE NOALTEXTERNALEVENT(LEVEL: INTERRUPT_LEVEL);
```

detaches any semaphore which has been designated the alternative receiver of the interrupt <level>. Nothing happens if no semaphore has been designated thus.

4.8.8.4 Interrupt Routines.

```
    TYPE interrupt_result = -1..15;

FUNCTION INTLEVEL: interrupt_result;
```

returns the interrupt level of the interrupt currently being serviced (0 to 15) or -1 if no interrupt is being serviced.

```
PROCEDURE MASK;
```

disables all interrupts except for interrupt level 0.

```
PROCEDURE UNMASK;
```

enables all interrupts which are more urgent than the priority of the calling process.

PROCEDURE WAITINTERRUPT;

suspends the calling process until an interrupt of equal priority to the process occurs.

4.8.8.5 Process Management.

```
TYPE processid = @processid;
```

```
FUNCTION MY$PROCESS : processid;
```

Returns the process identification of the calling process.

4.8.8.6 Memory Management.

```
PROCEDURE FREE(VAR ptr: pointer);
```

Returns the area referenced by <ptr> to the heap, <ptr> is set to NIL.

4.8.8.7 Microprocessor Executive Error and Exception Codes.

In the process record, there is a field (word >36) which contains a CLASS CODE in the left-hand byte and a REASON CODE in the right-hand byte. The CLASS CODES provides information on the general area in which there is some problem; the REASON CODE, more specific information concerning the problem.

For example, an error code of >8508 would indicate:

1) The general area of the problem is in process management (CLASS CODE 85) and,

2) Specifically, there was not enough heap available to start a process (REASON CODE 08).

The CLASS CODES and REASON CODES are listed in the left-hand column below.

ERROR CODES

CLASS ERROR CODES

```
user_error           = 81;
scheduling_error     = 82;
semaphore_error      = 83;
interrupt_error      = 84;
process_mgmt_error   = 85;
exception_error      = 86;
memory_mgmt_error    = 87;
file_error           = 88;
host_file_error      = 89;
```

HEAP MANAGEMENT ERROR REASON CODES

```
heap invalid =          1;
heap overflow error =   2;
heap packet error =     3;
```

INTERRUPT ERROR REASON CODES

```
interuppt invalid =     1;
interrupt level
invalid =               2;
interrupt semaphore
invalid =               3;
interrupt not
handled =               4;
interrupt trap
vector error =          5;
interrupt handler
priority error =        6;
```

EXCEPTION HANDLER REASON ERROR CODES

```
exception handler not
established from
process =               1;
exception handler
cannot have
parameters =            2;
exception handler
cannot be in
assembly language =     3;
exception handler
local variables too
large for stack =       4;
```

```
PROCESS MANAGEMENT REASON ERROR CODES

not a process =          1;
aborted =                2;
not started invalid
priority =               3;
not started invalid
stacksize =              4;
not started negative
heapsize =               5;
not started process
is in assembly
language  =              6;
not started no memory
for semaphore =          7;
not started no memory
for process_heap =       8;
not started no memory
for process_stack =      9;
not started no memory
process_frame =          10;


SEMAPHORE ERROR REASON CODES

semaphore invalid =      1;
semaphore count
error =                  2;
semaphore operation

error =                  3;
semaphore count
overflow =               4;
semaphore interrupt
handler priority
error =                  5;


SCHEDULING ERROR REASON CODES

scheduler invalid =      1;
scheduler priority
error =                  2;


USER ERROR REASON CODES

stack overflow =         1;
stack overflow =         2;
```

4.8.9 ASCII Character Set

CHAR	HEX	CHAR	HEX	CHAR	HEX
NUL	00	+	2B	V	56
SOH	01	,	2C	W	57
STX	02	-	2D	X	58
ETX	03	.	2E	Y	59
EOT	04	/	2F	Z	5A
ENQ	05	0	30	[	5B
ACK	06	1	31	\	5C
BEL	07	2	32	]	5D
BS	08	3	33	^	5E
HT	09	4	34	_	5F
LF	0A	5	35	`	60
VT	0B	6	36	a	61
FF	0C	7	37	b	62
CR	0D	8	38	c	63
S0	0E	9	39	d	64
S1	0F	:	3A	e	65
DLE	10	;	3B	f	66
DC1	11	<	3C	g	67
DC2	12	=	3D	h	68
DC3	13	>	3E	i	69
DC4	14	?	3F	j	6A
NAK	15	@	40	k	6B
SYN	16	A	41	l	6C
ETB	17	B	42	m	6D
CAN	18	C	43	n	6E
EM	19	D	44	o	6F
SUB	1A	E	45	p	70
ESC	1B	F	46	q	71
FS	1C	G	47	r	72
GS	1D	H	48	s	73
RS	1E	I	49	t	74
US	1F	J	4A	u	75
SPACE	20	K	4B	v	76
!	21	L	4C	w	77
"	22	M	4D	x	78
#	23	N	4E	y	79
$	24	O	4F	z	7A
%	25	P	50	{	7B
&	26	Q	51	|	7C
'	27	R	52	}	7D
(	28	S	53	~	7E
)	29	T	54	DEL	7F
*	2A	U	55		

4.8.10 HEX-DECIMAL Table

EVEN BYTE				ODD BYTE			
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4,096	1	256	1	16	1	1
2	8,192	2	512	2	32	2	2
3	12,288	3	768	3	48	3	3
4	16,384	4	1,024	4	64	4	4
5	20,480	5	1,280	5	80	5	5
6	24,576	6	1,536	6	96	6	6
7	28,672	7	1,792	7	112	7	7
8	32,768	8	2,048	8	128	8	8
9	36,864	9	2,304	9	144	9	9
A	40,960	A	2,560	A	160	A	10
B	45,056	B	2,816	B	176	B	11
C	49,152	C	3,072	C	192	C	12
D	53,248	D	3,328	D	208	D	13
E	57,344	E	3,584	E	224	E	14
F	61,440	F	3,840	F	240	F	15

4.8.11 BACKUS-NAUR Form (BNF) SYNTAX Definitions

```
::=     "is defined to be"
< >     for enclosing non-terminal symbols
        (ie entities defined by a production)
[ ]     for enclosing optional entities
{ }     for enclosing entities that may be repeated
        zero or more times
|       for representing alternatives
"["     indicates symbol [ is to appear in the text
```

4.8.12 COMPILER Options

```
<option control comment>::=
        "{" $ <option list> "}"

<option list>::=
          <option> { , <option> }

<option>::=
          [ NO ] <option identifier> |
          [ RESUME ] <option identifier>
```

Below is the full list of available compiler options along with the default value for each.

72COL DEFAULT=TRUE
Only scans the first 72 columns, when turned off the whole source line is scanned.

ASSERTS DEFAULT=TRUE
Generates object code for ASSERT statements.

CKINDEX DEFAULT=FALSE
Disables run-time checks for array indices.

CKPTR DEFAULT=FALSE
Disables run-time checks for pointers equal to NIL.

CKSET DEFAULT=FALSE
Disables run-time checks for set element expressions.

CKSUB DEFAULT=FALSE
Disables run-time checks for subrange assignments in bounds, also checks result bounds of PRED and SUCC.

DEBUG DEFAULT=TRUE
Statement numbers are incorporated into the code for use by the source level interactive debugger.

LIST DEFAULT=TRUE
Enables printing of source listing, error lines are always listed.

MAP DEFAULT=FALSE
When TRUE prints a map of the routine's variables and commons after listing the routine.

NULLBODY DEFAULT=FALSE
Should not be used in user programs. Used by the configuration processor to insert 'dummy' routine bodies in a program to be compiled.

OBJECT DEFAULT=FALSE
When TRUE generates object code.

PAGE DEFAULT=FALSE
When TRUE printing continues at the top of a new page, set to false at the begining of each source line.

STATMAP DEFAULT=FALSE
When TRUE a map of displacements for each statement in the object module is to be generated by CODEGEN.

4.8.13 CONCURRENT CHARACTERISTICS

<concurrent characteristics>::=

"{" # <concurrent characteristic list> "}"

<concurrent characteristic list>::=

<concurrent character> { ; <concurrent character> }

<concurrent character>::=

<concurrent character keyword> = <parameter identifier> |
<concurrent character keyword> = <integer constant>

Where <concurrent character keyword> is one of the following:

HEAPSIZE	Number of words allocated to the system, program or process heap
PRIORITY	Priority of the system,program or process
STACKSIZE	Number of words allocated to the system, program or process heap

These may only appear immediately following the initial BEGIN of a system, program or process declaration.

4.8.14 SYSTEM DECLARATION

A system may consist of a single program so long as no processes are declared within it. The syntax for such a system is:

PROGRAM <identifier ; <program block> .

<system>::=
SYSTEM <identifier> ; <system block> .

<system block>::=
<label declaration part><constant declaration part>

```
        <type declaration part><common declaration part>
        <access declaration part><system routines><body>

<label declaration part>::=
        LABEL <statement label> { , <statement label> } ; |
        <empty>

<empty>::=

<statement label>::=
        <digit> { <digit> }

<constant declaration part>::=
        CONST <constant declaration> { ; <constant declaration> } ;
        <empty>

<constant declaration>::=
        <identifier> = <constant> |
        <identifier> = <integer constant expression>

<type declaration part>::=
        TYPE <type declaration> { ; <type declaration> } ; |
        <emptv>

<type declaration>::=
        <identifier> = <type>

<variable declaration part>::=
        VAR <variable declaration> { ; <variable declaration> } ; |
        <empty>

<variable declaration>::=
        <identifier list> : <type>

<identifier list>::=
        <identifier> { , <identifier> }

<common declaration part>::=
        COMMON <variable declaration> { ; <variable declaration> } ;
        <empty>

<access declaration part>::=
        ACCESS <identifier list> ; | <empty>

<system routines>::=
        { <system routine> }

<system routine>::=
        <program declaration> | <procedure declaration> |
        <function declaration>

<program declaration>::=
        <program header><program block> ; |
        <program header> FORWARD ; |
```

```
        <program header> EXTERNAL [ Pascal ] ;

<program header>::=
        PROGRAM <identifier>[ <program parameter list> ] ;

<program parameter list>::=
        ( <program parameter> { , <program parameter> } )

<program parameter>::=
        <identifier list> : <type identifier>

<program block>::=
        <label declaration part><constant declaration part>
        <type declaration part><variable declaration part>
        <common declaration part><access declaration part>
        <program routines><body>

<program routines>::=
        { <program routine> }

<program routine>::=
        <process declaration> | <procedure declaration> |
        <function declaration>

<procedure declaration>::=
        <procedure header><block> ; |
        <procedure header> FORWARD ; |
        <procedure header> EXTERNAL [ Pascal ] ;

<procedure header>::=
        PROCEDURE <identifier> [ <parameter list> ] ;

<parameter list>::=
        ( <any parameter> { ; <any parameter> }  )

<any parameter>::=
        [ VAR ] <identifier list> : <type identifier>

<block>::=
        <label declaration part><constant declaration part>
        <type declaration part><variable declaration part>
        <common declaration part><access declaration part>
        <routines><body>

<routines>::=
        { <routine> }

<routine>::=
        <procedure declaration> | <function declaration>

<function declaration>::=
        <function header><block> ; |
        <function header> FORWARD ; |
        <function header> EXTERNAL [ Pascal ] ;
```

```
<function header>::=
        FUNCTION <identifier>[ <parameter list> ] : <result type> ;

<process declaration>::=
        <process header><program block> ; |
        <process header> FORWARD ; |
        <process header> EXTERNAL [ Pascal ] ;

<process header>::=
        PROCESS <identifier> [ <program parameter list> ] ;

<body>::=
        <compound statement>
```

TYPE SYNTAX

```
<type>::=
        <simple type> | <structured type>

<simple type>::=
        <scalar type> | <subrange type> | <type identifier>

<type identifier>::=
        <identifier> | ANYFILE | SEMAPHORE | TEXT | REAL
        INTEGER | LONGINT | BOOLEAN | CHAR

<scalar type>::=
        ( <scalar identifier> { , <scalar identifier> } )

<subrange type>::=
        <enumeration constant> .. <enumeration constant>

<enumeration constant>::=
        <character constant> | <boolean constant> |
        <integer constant> | <scalar identifier>

<scalar identifier>::=
        <identifier>

<structured type>::=
        [ PACKED ] <unpacked structure> | <pointer type> |
        <file type> | <set type>

<unpacked structure>::=
        <array type> | <record type>

<array type>::=
        ARRAY "[" <index type> { , <index type> } "]"
             OF <type>

<index type>::=
```

```
        BOOLEAN | CHAR | <scalar type> | <subrange type> |
        <identifier>

<record type>::=
        RECORD <field list> END

<field list>::=
        <fixed part> | <fixed part> ; <variant part> | <variant part>

<fixed part>::=
        <record section> { ; <record section> }

<record section>::=
        <field identifier> { , <field identifier> } : <type> |
        <empty>

<field identifier>::=
        <identifier>

<variant part>::=
        CASE [ <tagfield> ] <tagfield type> OF <variant>
             { ; <variant> }

<tagfield type>::=
        BOOLEAN | CHAR | INTEGER | LONGINT | <identifier>

<tagfield>::=
        <identifier> :

<variant>::=
        <variant label list> : ( <field list> ) | <empty>

<variant label list>::=
        <variant label> { , <variant label> }

<variant label>::=
        <enumeration constant> |
        <enumeration constant> .. <enumeration constant>

<set type>::=
        SET OF <simple type>

<pointer type>::=
        @ <type identifier>

<file type>::=
        [ RANDON ] FILE OF <type>

<result type>::=
        BOOLEAN | CHAR | INTEGER | LONGINT | REAL |
        SEMAPHORE | <identifier>
```

STATEMENT

```
<compound statement>::=
        BEGIN <statement> { ; <statement> } END

<statement>::=
        [ <statement label> : ]<simple statement> |
        [ <statement label> : ][ <escape label> : ]
            <structured statement>

<simple statement>::=
        <empty statement> | <assignment statement> |
        <procedure statement> | <escape statement> |
        <assert statement> | <goto statement> | <start statement>

<empty statement>::=
        <empty>

<assignment statement>::=
        <variable> := <expression>

<procedure statement>::=
        <procedure identifier> [ <actual parameter list> ]

<actual parameter list>::=
        ( <actual parameter> { , <actual parameter> } )
<actual parameter>::=
        <expression> | <variable>

<procedure identifier>::=
        <identifier>

<start statement>::=
        START <process identifier> [ <actual parameter list> ]

<escape statement>::=
        ESCAPE <escape label> | ESCAPE <routine identifier>

<escape label>::=
        <identifier>

<routine identifier>::=
        <program identifier> | <process identifier> |
        <procedure identifier> | <function identifier>

<goto statement>::=
        GOTO <statement label>

<assert statement>::=
        ASSERT <expression>

<structured statement>::=
        <compound statement> | <conditional statement> |
        <repetitive statement> | <with statement>
```

```
<conditional statement>::=
        <if statement> | <case statement>

<if statement>::=
        IF <expression> THEN <statement>
        [ ELSE <statement> ]

<case statement>::=
        CASE <expression> OF <case element> { ; <case element> }
        [ OTHERWISE <statement> { ; <statement> } ]
        END

<case element>::=
        <case label list> : <statement> | <empty>

<case label list>::=
        <case label> { , <case label> }

<case label>::=
        <enumeration constant> |
        <enumeration constant> .. <enumeration constant>

<repetitive statement>::=
        <for statement> | <while statement> | <repeat statement>

<for statement>::=
        FOR <control variable> <generator> DO <statement>

<control variable>::=
        <identifier>

<generator>::=
        := <initial value> TO <final value> |
        := <initial value> DOWNTO <final value>

<initial value>::=
        <expression>

<final value>::=
        <expression>

<while statement>::=
        WHILE <expression> DO <statement>

<repeat statement>::=
        REPEAT <statement> { ; <statement> }
        UNTIL <expression>

<with statement>::=
        WITH <with variable list> DO <statement>

<with variable list>::=
        <with variable> { , <with variable> }
```

```
<with variable>::=
        <record variable> | <identifier> = <record variable>
```

EXPRESSION

```
<expression>::=
        <boolean term> | <expression> OR <boolean term>

<boolean term>::=
        <boolean factor> | <boolean term> AND <boolean factor>

<boolean factor>::=
        <boolean primary> | NOT <boolean primary>

<boolean primary>::=
        <simple expression> |
        <boolean primary><relational operator><simple expression>

<relational operator>::=
        = | <> | < | <= | > | >= | IN

<simple expression>::=
        <term> | <adding operator><term> |
        <simple expression><adding operator><term>

<adding operator>::=
        + | -

<term>::=
        <factor> | <term><multiplying operator><factor>

<multiplying operator>::=
        * | / | DIV | MOD

<factor>::=
        ( <expression> ) | <set> | <unsigned constant> | <variable>
        <function identifier> [ <actual parameter list> ]

<function identifier>::=
        <identifier>

<set>::=
        "[" <element list> "]"

<element list>::=
        <element> { , <element> }

<element>::=
        <expression> | <expression> .. <expression>

<unsigned constant>::=
```

```
        <constant identifier> | <boolean constant> |
        <scalar identifier> | <character constant> |
        <string constant> | <integer constant> | NIL |
        <real constant>

<constant identifier>::=
        <identifier>
```

VARIABLE

```
<variable>::=
        <variable identifier> | <component variable> |
        <type-transferred variable>

<variable identifier>::=
        <identifier>

<component variable>::=
        <indexed variable> | <field designator> |
        <referenced variable>

<indexed variable>::=
        <array variable> "[" <expression> { , <expression> } "]"

<array variable>::=
        <variable>

<field designator>::=
        <record variable> . <field identifier>

<record variable>::=
        <variable>

<field identifier>::=
        <identifier>

<referenced variable>::=
        <pointer variable> @

<pointer variable>::=
        <variable>

<type-transferred variable>::=
        <variable> :: <type identifier>
```

INTEGER CONSTANT EXPRESSION

```
<integer constant expression>::=
        <integer constant term> |
```

```
        <adding operator><integer constant term> |
        <integer constant expression><adding operator>
             <integer constant term>

<integer constant term>::=
        <integer constant factor> |
        <integer constant term><intmult operator>
             <integer constant factor>

<intmult operator>::=
        * | DIV | MOD

<integer constant factor>::=
        ( <integer constant expression> ) |
        <integer constant identifier> | <integer constant>

<integer constant identifier>::=
        <identifier>


LANGUAGE ELEMENT


<symbol>::=
        <special symbol> | <keyword symbol> | <identifier> | <constant>

<constant>::=
        <enumeration constant> | <real constant> | <string constant> |
        <constant identifier>

<separator>::=
        <space> | <end of logical source record> | <comment> | <remark>

<comment>::=
        <open comment><any sequence of graphic characters
        not containing <close comment> ><close comment>

<open comment>::=
        "{" | (*

<close comment>::=
        "}" | *)

<remark>::=
        " <any sequence of graphic characters extending
            to the end of the logical source record>

<special symbol>::=
        +  |  -  |  *  |  /  |  =  |  <  |  >  |  (  |  )  |  .  |  ,  |  ;  |  :  |
        @  |  "[" |  "]" |  "{" |  "}" |  <=  |  >=  |  <>  |  ..  |  :=  |  ::  |
    Note : The following substitutions may be used.
        (* --> "{", *) --> "}", (. --> "[", .) --> "]", @ --> ^

<keyword symbol>::=
```

ACCESS | AND | ANYFILE | ARRAY | ASSERT | BEGIN | BOOLEAN |
CASE | CHAR | COMMON | CONST | DIV | DO | DOWNTO |
ELSE | END | ESCAPE | FALSE | FILE | FOR | FUNCTION |
GOTO | IF | IN | INPUT | INTEGER | LABEL | LONGINT |
MOD | NIL | NOT | OF | OR | OTHERWISE | OUTPUT | PACKED |
PROCEDURE | PROCESS | PROGRAM | RANDOM | REAL | REPEAT |
SEMAPHORE | SET | START | SYSTEM | TEXT | THEN | TO |
TRUE | TYPE | UNTIL | VAR | WHILE | WITH

<identifier>::=
<letter> { <letter> | _ | <digit> }

<letter>::=
A | B | C | D | E | F | H | I | J | K | L | M | N | O |
P | Q | R | S | T | U | V | W | X | Y | Z | $

<digit>::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<boolean constant>::=
FALSE | TRUE

<character constant>::=
' <character> '

<string constant>::=
' <character><character> { <character> } '

<character>::=
<graphic character> | # <hexdigit><hexdigit>

<graphic character>::=
<special character> | <letter> | <digit> |
<space> | <nonstandard character>

<special character>::=
+ | - | * | / | = | < | > | (|) | . | , | ; | : |
@ | '' | " | ## | _ | "[" | "]" | "{" | "}"

<space>::=
" "

<nonstandard character>::=
<any other character available on a particular
system or device>

<hexdigit>::=
<digit> | A | B | C | D | E | F

<integer constant>::=
<digits> [L] |
<hexdigit> { <hexdigit> } [L]

<real constant>::=

```
        <digits> . <digits> |
        <digits> . <digits> E <scale factor>
        <digits> E <scale factor>

<digits>::=
        <digit> { <digit> }

<scale factor>::=
        [ <sign> ] <digits>

<sign>::=
        + | -
```

CHAPTER V

POWER BASIC

5.1 INTRODUCTION

BASIC (Beginner's All Purpose Symbolic Instruction Code) is a high-level interpreted language. Although it does not support the full block structured approach of the Algol based languages (Algol 68, PASCAL, etc.), the BASIC language is easy to learn and supports a variety of useful features. These features are discussed below.

In an interpreted language, no machine code is produced. Instead, as each source line is entered, it is checked for syntax errors (does the source line conform to the language specifications?) and, if valid, is stored in a condensed and encoded form called interpretive code. Because interpreted languages are normally used in an interactive mode, syntax errors are immediately reported to the user. Before the next source lines can be entered, the line containing the error(s) must be corrected. The stored code can be 'executed' at any time (it is not necessary to wait until the whole program has been entered) by issuing the RUN command. The interpretive code is not directly executed. Instead, the interpreter examines each statement in the interpretive code and calls in a machine language subroutine (which is part of the interpreter) to carry out the desired operation.

Semantic errors (non-existent variables and arrays, incorrectly referenced arrays, etc.) And run-time errors (incorrect program logic) require that the line(s) containing the errors be revised before the program can be rerun. With a compiled language, the whole program must be recompiled after modifications are made. It may also be necessary to link edit the compiled program should it contain any external references.

The advantages of using an interpretive language follow:

- Because the interpreter calls in complete assembly language subroutines to perform each function, each statement in the interpretive code can specify a complex operation. This results in compact, memory efficient code.

- There is no need to go through separate compilation, link edit steps, etc. to produce executable code. As part of the edit step, each source statement is translated into 'executable' interpretive code as it is entered.

- Each source line is checked for errors as it is entered; it is impossible to enter a syntactically incorrect statement.

- Interpretive programs are usually developed interactively. As a result, it is only necessary to retype the relevant line(s) and rerun the routine in order to change the program. The user is able to see the result of his change immediately. Also, the interpreter provides excellent error diagnostics and good recovery techniques.

- Because the interpreter is in control the total time, it is more difficult for the programmer to find himself in irrecoverable error situations.

- To transport a program to another machine it is only necessary to provide a version of the interpreter written in the new machine's instruction code. Any program written in interpretive code can then be run on the new machine.

Because of the extra work done by the interpreter in reading interpretive code, calling subroutines, etc, interpretive code executes several times slower than compiled code. This is the principal disadvantage to using interpretive code. In addition, BASIC was designed as a simple language, and does not provide the powerful program and data structuring techniques of, say, PASCAL. As such, it is probably not a suitable language for developing large or complex applications. However, for small to medium sized applications, and for experimental work demanding speed in program development, BASIC is very acceptable.

5.2. POWER BASIC

POWER BASIC is a family of software products designed for the industrial user. It provides all of the facilities of BASIC plus specially designed features to support real-time industrial control applications. The POWER BASIC family consists of three members : Evaluation POWER BASIC, Development POWER BASIC, and Configurable POWER BASIC. In addition, there is an Enhancement Software Package available for use with Development POWER BASIC.

POWER BASIC is designed to run on the TM 990 range of microcomputer modules (it can also be adapted to run on other systems). It is possible to set up a POWER BASIC development system with a minimum of capital outlay. A chasis containing two or three microcomputer modules from the TM990 board range, a 733 ASR terminal, a single audio cassette recorder and a PROM programmer, provide all the facilities necessary to develop a POWER BASIC application program and store it in Programmable Read Only Memory (PROM). The floppy disc based FS990/4 system provides more sophisticated features, which allow a POWER BASIC program to be tailored for any application to achieve minimum code size.

5.2.1 Evaluation POWER BASIC

Evaluation POWER BASIC is a four-EPROM package that resides on either a TM 990/100M or a /101M CPU module. Additional RAM in the form of TM 990/201 or /206 memory expansion boards may be configured into the system as necessary.

Apart from the standard features of BASIC, Evaluation, POWER BASIC allows the user to access control equipment in real-time (timing is provided by the TIC function) by either memory-mapped I/O (MEM function) or via TI's standard bitwise Communications Register Unit (BASE, CRB and CRF functions). It also allows the user to load a program from (LOAD command) and save a program (SAVE command) on digital cassettes.

Evaluation POWER BASIC is intended for users to try out the features of POWER BASIC. It is not meant for serious development work, apart from experimental applications.

Used with the /101M CPU board. Evaluation POWER BASIC supports the following execution environments:

- Single-user single-partition
- Single-user two-partition
- Two-user two-partition

Selection of the appropriate environment is implemented via the 5-pole DIP on the /101M CPU board. Section 2.9 of the TM990 POWER BASIC Reference Manual describes this feature in greater detail

Communication between partitions is made possible by the system defined common array:COM(0) to COM(9). Thus Evaluation POWER BASIC can be used to control two separate tasks, the execution of each being synchronized using the COM array. For example, one partition can be used to control an industrial process while the other collects control data (from a terminal for example).

PARTITION #1

```
 10 REM GATHER DATA
 20 COM(0)=0
 30 INPUT V1,...,V9
 40 IF COM(0)<>0 THEN GOTO 40
 50 COM(1)=V1::COM(2)=V2
    ...
    ...
 90 COM(0)=1
100 GOTO 30
```

PARTITION #2

```
 10 REM CONTROL PROCESS
 20  'initialize' V1,...,V9
 30 IF COM(0)=0 THEN GOTO 120
 40 V1=COM(1)::V2=COM(2)
    ...
    ...
110 COM(0)=0
120  'use' V1,...,V9
130 GOTO 30
```

Partition #1 gathers input from the terminal and passes it across to partition #2 via the COM array. COM(0) is used to synchronize the

data transfer; mutual exclusion is guaranteed by allowing #1 to access the array only when COM(0)=0; when COM(0)=1 only #2 can access it. After loading the array, #2 is informed that fresh data is ready by setting COM(0) to 1. This also prevents #1 from modifying the array contents until #2 has copied them. Once the contents have been copied, #1 is given exclusive control of the array by setting COM(0) to 0.

In a single-user two-partition environment, CTRL T (pressing the T key while holding down the CTRL key) will transfer control from one partition to the other.

5.2.2 Development POWER BASIC

Development POWER BASIC is a six-EPROM package that resides on either a TM 990/100M or a /101M CPU board plus either a TM 990/302 Software Development Board or a TM 990/201 memory expansion board. Additional memory expansion boards can be included if required.

In Development Power BASIC, the two-partition feature is removed to allow the inclusion of additional features. With the CALL statement, Development POWER BASIC allows the user to access assembly language routines that have been burnt into PROM. Development POWER BASIC also allows the user to write interrupt service routines in POWER BASIC and to associate these with particular interrupt levels (using the TRAP, IMASK, and IRTN statements). Development POWER BASIC also provides full character handling facilities (character search, match and conversion functions), better control structures (including the ELSE, ON and ERROR statements) and more varied print formatting (hexadecimal formatting and direct output of hex ASCII codes).

In addition, the Enhancement Software Package is accessible from Development POWER BASIC. The Enhancement Software Package is a two-EPROM package used in conjunction with Development POWER BASIC. This provides the additional capability of loading a program from and saving a program on low cost audio cassettes. If the TM 990/302 Software Development Board is configured into the system, it is possible to program POWER BASIC applications into TMS2716 EPROMs (using the PROgram command). The software package also provides decimal print formatting and complete error message reporting.

5.2.3 Configurable POWER BASIC

Configurable POWER BASIC is a floppy disc based development package that is designed to run on a 990/4 minicomputer under the TX990 operating system (version 2.3 or later). The package has two parts, a configurator and an interpreter. Configurable POWER BASIC allows the user to generate an applications system of minimum size by deleting the POWER BASIC editor along with any parts of the interpreter that are used.

The configurator determines what POWER BASIC features are required by

the user's application program and builds a link control file. With this file along with the POWER BASIC object library, the TX990 Link Editor produces a POWER BASIC run-time module. This module contains both the user's application program and an interpreter customized to the user's requirements (only the POWER BASIC features used by the application program are built into it). Using the TXPROM utility, the module can be burnt into 2716 EPROMs.

If the application program's EPROM(s) are inserted at address >3000, toggling the reset switch causes the program to execute automatically. However, if the EPROM(S) are inserted elsewhere, the following command must be used to execute the program:

```
LOAD   <address>
```

where <address> is the address of the first pair of EPROMs containing the POWER BASIC application program.

The interpreter provides all the features of Development POWER BASIC, its Enhancement Software Package, plus a number of other features.

Configurable POWER BASIC supports a comprehensive file management package that allows the user to create, access and delete files (either sequential or random access) on the 990/4's floppy disc units. In accordance with 990 philosophy, all file and device I/O operations are performed via conceptual links called logical unit numbers or lunos. The physical connection between a luno and a specific file or device is made (opened) by the BOPEN statement and is broken (closed) by the BCLOSE statement. The RESET statement closes all lunos that are open at the time the statement is executed. Files can be created by either the BDEFS (define sequential file) or the BDEFR (define random file) statements, and deleted by the BDEL statement. Reading from and writing to files or devices can be performed by the COPY statement or by:

```
BINARY   <exp>
```

where <exp> specifies the required I/O operation.

The '@' operator has been added to the PRINT statement to give the user complete cursor control. With this the user can specify an exact starting position for output on the screen (911 or 913 VDT) either by supplying the 'x' and 'y' co-ordinates or by using these positioning commands:

```
B     Move cursor to begining of line
C     Clear screen and move cursor to HOME position
D     Move cursor down
H     Move cursor to HOME position
L     Move cursor to left
R     Move cursor to right
```

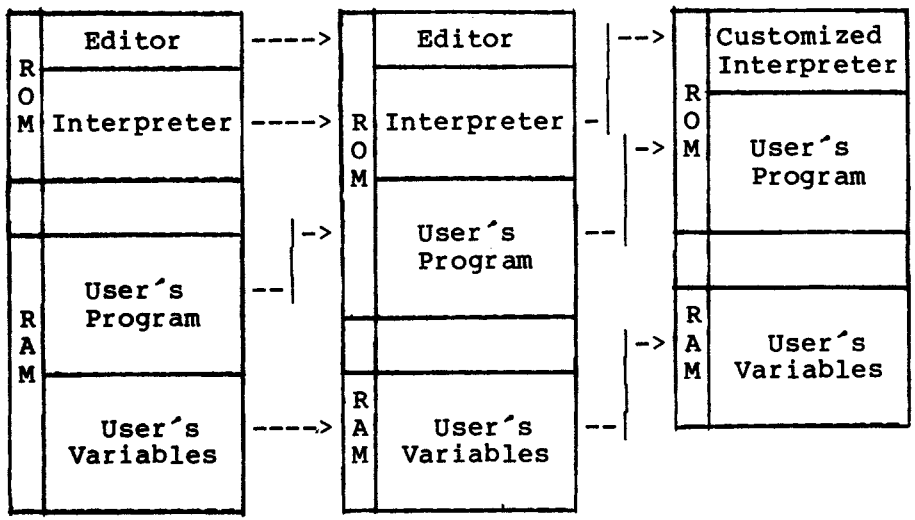
For example; To clear the screen and print the message 'INPUT NAME' to the VDT screen, starting on the fifth line at the twelfth character

position, either of the following commands is required.

```
        PRINT @"C5D12R";"INPUT NAME"
  or    PRINT @"C";@(5,12);"INPUT NAME"
```

Other features of Configurable POWER BASIC include: DIGITS (specify the number of digits to be printed in free format), STACK (interrogate the GOSUB stack) statements, NUMBER (set initial and increment values for the automatic line numbering facility), PURGE (delete the specified lines), SOURCE (show how much room the program will occupy when saved), and BYE (terminate a Configurable POWER BASIC session) commands.

The diagram that follows illustrates Configurable POWER BASIC as it relates to memory.



After program is developed

After PROM Programmer

After Configurator and PROM Programmer

5.3. BASIC LANGUAGE OVERVIEW

BASIC is an uncomplicated,easy to learn language, based upon a few simple concepts. A BASIC program consists of a series of numbered statement lines. The statements are executed in ascending numerical order. A line normally contains one BASIC statement; although the concatenation operator (::) can be used to write more than one statement on a line. One of the simplest statements, the assignment statement, is used to assign the value of an expression to a variable:

```
50 A2 = 5 + 7
```

When the above line is executed, A2 will be assigned the value of the arithmetic expression '5+7' (the integer 12).

There is no variable declaration; a variable is implicitly declared by its first appearance on the left-hand side of an assignment statement. Variable names are restricted to from one to three letters or a combination of a letter and a number in the range 0-127. There is no typing of data. Variables can have integer, real or character string values, depending on the context. The only data structure provided is the array, which can have one or more dimensions.

The principal device for structuring a program is the GOTO statement, which transfers execution directly to a statement number. The IF..THEN statement implements selection; it must be combined with the GOTO statement if the alternatives are longer than one or two statements. The FOR..NEXT statement implements iteration (see Subsection 2.5). In general, programming constructs have to be built by the programmer using IFs, FORs and GOTOs.

Subroutines, 'procedures' (see Subsection 2.8) can be called using the GOSUB statement, which simply places the address of the statement following the GOSUB on a last-in-first-out stack, from where it is retrieved when a RETURN is executed. Subroutines are not declared separately from the main program. The GOSUB simply specifies a statement number; the statements between that number and the next RETURN are treated as a subroutine. Scope rules are simple. Once a variable has been introduced, it can be referenced anywhere in the program. Subroutines can be nested (up to 10 deep), but the programmer needs to check that the GOSUBs and RETURNs match (the interpreter does not perform this check). Subroutine parameters are not allowed.

The main attraction of BASIC is its simplicity. Programs can be entered and executed easily even by users who are not skilled programmers. BASIC is a high level language, and as such automatically handles such details as storage allocation (to which the assembly language programmer devotes a lot of attention). The development environment provided by BASIC is particularly simple and easy to use; even novices can learn to develop a BASIC program in a matter of hours. BASIC is ideal for the rapid development of relatively simple applications.

However, it does have limitations. Because of its simplicity, BASIC performs very few checks on the integrity of program and data (such as are performed automatically by the PASCAL compiler, for instance). It is quite legal, for example, to assign an integer value to a character string variable. This may be valuable in some circumstances. However BASIC supplies no warning if it is done by mistake. In addition, the structuring and self-documenting features of PASCAL are missing. For a complex application, PASCAL is probably a better alternative.

5.4. POWER BASIC OPERATION

5.4.1 Operating Modes

POWER BASIC has two modes of operation:

1) Keyboard mode is automatically entered when POWER BASIC is initialized. Entering a numbered line causes that line to be stored in the appropriate place in the program space. Entering an unnumbered line causes the statement(s) to be immediately executed and keyboard mode to be re-entered as soon as the necessary processing has been performed.

2) Execution mode is entered by issuing either a RUN, a CONT or a GOTO statement causing the POWER BASIC interpreter to execute the previously stored program; RUN starts at the lowest line number in the program; CONT continues from the last line that was previously interpreted; GOTO proceeds from the line specified. This mode is terminated by any one of the following conditions:

   - Error condition arising.
   - STOP or END statement executed.
   - Pressing the ESCape key on the terminal.

Note : There are a number of statements which can only be issued in keyboard mode. The POWER BASIC Reference Manual refers to these as commands. These commands are also listed in the Reference Section at the end of this chapter.

5.4.2 Editing Source Statements

POWER BASIC supports a simple editor that allows the user to easily modify (or edit) already entered source statements. The available edit commands are:

CR or LF	Enter the edited line
ctrl H	Backspace the cursor one character
ctrl F	Forwardspace the cursor one character (An attempt to forwardspace past the last character will have no effect apart from sounding the bell on the terminal)
RUBOUT	Backspace and remove character
<ln> ctrl E	Display the line <ln> for editing

Development POWER BASIC supports two additional commands that are not available in Evaluation POWER BASIC:

ctrl I<n>	Insert <n> blanks
ctrl D<n>	Delete <n> characters

'Ctrl E' strike the E key while holding down the CTRL key.
'Ctrl I<n>' hold down the CTRL key while striking the I key, then strike the numeric key corresponding to the value <n>.

When the Carriage Return or LineFeed key is pressed, all characters displayed are entered, regardless of the position of the cursor.

Entering only a line number (and nothing else) causes the specified line to be deleted from the stored program. Entering a statement with a line number that already exists causes the original statement to be replaced by the new one.

The editor is automatically invoked when the interpreter encounters a syntax error in a line being entered via the ASR. However, if the program is being loaded from cassette (using the LOAD command) and a syntax error is encountered, the interpreter will display the number of the line containing the error. The whole line is ignored as it can not be stored correctly and the load operation will continue.

5.4.3 Automatic Line Numbering

The automatic line numbering facility is invoked by terminating an input line with a linefeed instead of a carriage return. This causes the interpreter to output the incremented line number and keyboard mode to be re-entered. The incremented line number is 10 greater than the last line number entered. Entering a line containing just a linefeed initializes the line number to 10. Terminating a line with a carriage return disables this facility.

5.4.4 System Initialization

Toggling the reset switch causes POWER BASIC to clear and scan the system RAM area to determine how much memory is present. This operation begins at location >FFDC and continues on down through contiguous memory to location >4000 or until a read/write mismatch is encountered. (A fully populated /100M microcomputer board only holds 1K of RAM . This is addressed from >FC00 to >FFFF. Any additional

memory in the form of memory expansion boards, must be configured so that it terminates at >EFFF. The interpreter ignores the resulting memory 'hole', from >F000 to >FBFF.

The POWER BASIC interpreter next performs the auto-baud sequence. This initializes the serial I/O interface for terminal communication. After the user has struck the A (or carriage return) key on the terminal, the interpreter measures the time of the start bit and determines the baud rate of the terminal. The onboard TMS9902 Asynchronous Communications Controller is then set to this baud rate (all terminal I/O is performed through the 9902.)

In Development POWER BASIC the UNIT flag is restored to a value of one; all output is directed to Port A on the microcomputer board.

Once all POWER BASIC pointers have been initialized, the following message is output:

```
TM990 BASIC REV X.n.m
*READY
```

where : X = language level

n = release number

m = revision number

At this stage, POWER BASIC is in keyboard mode waiting for user input.

Refer to the POWER BASIC Reference Manual for instructions on setting up the hardware configuration.

5.5. VARIABLES

A POWER BASIC variable can be used to store either an integer number, a real number, or a character string depending on the context in which the variable is used. Thus, although a variable may contain a number (integer or real) it can be used as though it contained a character string, and vice versa. All variables, whatever their type, occupy the same amount of storage (6 bytes in Development Power BASIC, 4 bytes in Evaluation POWER BASIC).

5.5.1 Variable Names

A variable name is either an alphabetic character followed by a number in the range 0 to 127 (e.g. Z100) or an alphabetic string up to three characters long (e.g. A, ST, and LST). The variable name cannot be identical to a POWER BASIC keyword; nor can it form the beginning of a keyword. The following variable names are not valid:

LIS Beginning of LIST which is

	a POWER BASIC statement
MEM	A POWER BASIC function
TOT	First 2 letters form the POWER BASIC keyword TO

for a full list of reserved words refer to the Reference Section.

5.5.2 Variable Declarations

Variables are not explicitly declared in BASIC. Instead a variable is implicitly declared by assigning a value to a valid variable name. For example, to declare the variable TST and assign it the value 100 the statement:

```
TST=100
```

is used.

A value can be assigned to a variable either by a READ (read a value from a DATA statement), an INPUT (accept input from the terminal) or a LET statement. The statement 'TST=100' is an implied LET, as are statements of the form:

<variable>=<expression>

where <expression> may contain function calls:

```
FRD=SIN(PI*NUM)
```

with both PI and NUM having been previously declared.

An attempt to use a variable that has not been declared (assigned a value) will result in error 40 (UNDEFINED VARIABLE).

5.5.3 Numeric Representation

If a number can be represented in a 16-bit two's complement form, it is stored in integer format, otherwise it will be stored in floating point format.

5.5.3.1 Integer Variables. An integer variable can store a value in the range -32768 to +32767.

5.5.3.2 Floating Point Variables. Floating point format allows a real number in the range 10E-75. ('E' represents the multiplier 10, the integer number following is the power to which 10 is raised.) This representation provides approximately 7 digits of accuracy for Evaluation POWER BASIC and approximately 11 digits of accuracy for

Development POWER BASIC.

5.5.4 Character String Variables

A character string is a string of characters enclosed within single or double quotes. Paired double quotes can be used to enclose single quotes and vice versa. In Development POWER BASIC, non-printable characters may be included in a character string by writing their hexadecimal ASCII representation enclosed in angle brackets. The angle brackets (< >) are stored along with the character string and are interpreted when the string is read from a DATA statement or when the string is being printed.

A variable is specified as containing a character string by preceeding the variable name by a dollar sign ('$'). In this form, a variable should be used to store a string of 5 characters for Development POWER BASIC, or 3 characters for Evaluation POWER BASIC. The last byte is used to terminate the string and contains the null character (zero).

5.5.5 Array Variables

An array can be thought of as a list of variables stored consecutively with each variable being represented as an array element. The area of memory reserved for the array is referenced by the array name. This is followed by a number enclosed in parentheses or square brackets (internally the parentheses are converted into and stored as square brackets). The number is known as the array subscript and indicates which element in the array is to be accessed.

Note : A and A(0) refer to two completely different variables.

To allocate the array STR with 10 elements the following statement is required:

```
DIM  STR(9)
```

The elements are referenced by

```
STR(0), STR(1), ...., STR(9).
```

The size parameter supplied to the DIMension statement is one less than might be expected as BASIC automatically allocates space starting from element zero.

Although an array may be used to hold character strings, it is declared (in the DIMension statement) without the dollar sign.

POWER BASIC allows an array to be declared with any number of dimensions; however, for most practical applications, a two dimensional array is usually sufficient.

5.6. POWER BASIC PROGRAM

A POWER BASIC program consists of a number of statements, each with a line number. Statements may either perform some action, such as adding two variables together and assigning the sum to a third variable ('A=B+C'), or may be control statements, that change the execution flow of the system. A full list of POWER BASIC statements is provided in the Reference Section at the end of this chapter.

POWER BASIC allows the user to write a number of statements on one line, with each statement being executed in turn. The general syntax for an input line is:

```
-line number- <statement> [ :: <statement> ]  -! comment-
```

where - - Indicate optional items
[ ] Indicate item is repeated as many times as required - 0,1,....

Exceptions :

- A NEXT statement should be the first statement on an input line; otherwise it will not be located to terminate the FOR loop.
- A DATA statement should be the only statement on an input line.
- A REM statement takes the remainder of a line as comment; statements following will be treated as comments.

5.6.1 Control Statements

POWER BASIC statements are normally executed in ascending line number order. However, it is not usually possible to write an effective applications program in a straightforward sequential manner. For this reason, POWER BASIC supports a number of control statements that allow the user to dictate the order in which program statements are executed.

5.6.1.1 GOTO Statement. The first of these control statements is the 'GOTO'. This provides a simple, yet very powerful, mechanism for changing program flow. The syntax for this statement is:

```
GOTO <ln>
```

This causes contol to be transferred to line <ln>.

Restraint must be exercised with this statement; too liberal a usage will lead to an unintelligible and unnecessarily complex program.

Possibly the best use of this statement is in building constructs that are not included in BASIC (the WHILE, DO FOREVER and REPEAT UNTIL

loops; more about these later).

5.6.1.2 IF THEN Statement. Occasionally it is necessary to perform some specific action only if a certain condition is met. For example, the only time the telephone should be answered is if it is ringing. To provide for this situation, Power BASIC provides the IF THEN statement. The above operation can now be expressed as 'IF the phone is ringing THEN answer it'. The syntax for this is :

```
IF <condition> THEN <sequence>
```

Statements in <sequence> must be separated from each other by the statement separator ('::'). <Condition> may be any valid expression that yields a value of true or false.

The POWER BASIC statements <sequence> is only executed if <condition> proves to be true.

Note: The statement separator does not delimit the IF THEN statement, it only separates the statements in <sequence> from each other.

```
100 IF cond1 THEN statement1::IF cond2 THEN statement2
```

Is not the same as:

```
100 IF cond1 THEN statement1
101 IF cond2 THEN statement2
```

In the first case, <statement2> is only executed if BOTH <cond1> AND <cond2> are true. In the second case, <statement2> is executed if <cond2> is true, irrespective of <cond1>.

The number of statements that can be associated with the THEN keyword is limited by the length of the input line. This can be overcome using the following:

```
        IF NOT(cond1) THEN GOTO 150
        .
        . Sequence of statements to be performed
        . when <cond1>=true
        .
150     REM end the IF THEN clause
```

The REM statement is a remark (comment), and is ignored by the interpreter.

If <cond1> is false, then NOT(cond1) is true and program control is passed to the REM statement following the sequence. But if <cond1> is true, then NOT(cond1) is false and program execution continues from the line following the IF THEN statement.

A WHILE loop can be built up as follows:

```
  10 IF NOT(cond1) THEN  GOTO 200
     .
     . Sequence to be performed
     . WHILE <cond1>=true
     .
     GOTO 10
 200 REM <cond1>=false
```

A DO FOREVER loop can be expressed as:

```
  50 REM start forever loop
     .
     . Sequence to be performed continuously
     .
     GOTO 50
```

A REPEAT UNTIL loop is:

```
 145 REM start repeat loop
     .
     . Sequence to be performed
     . UNTIL <cond1>=true
     .
     IF NOT(cond1) THEN  GOTO 145
     REM drop through to here when <cond1>=true
```

An IF THEN ELSE construct can be implemented as:

```
     IF  NOT(cond1) THEN  GOTO  100
     .
     . Sequence to be performed
     . when <cond1>=true
     .
     GOTO  200
 100 REM start ELSE part
     .
     . Sequence to be peformed
     . when <cond1>=false
     .
 200 REM end IF THEN ELSE
```

This can be easily expanded to allow an ELSEIF:

```
     IF  NOT(cond1) THEN  GOTO  192
     .
     . Sequence to be performed
     . when <cond1>=true
     .
     GOTO  475
 192 IF  NOT(cond2) THEN  GOTO  320
     .
     . Sequence to be performed
     . when <cond2>=true and <cond1>=false
     .
     GOTO  475
 320 REM start ELSE part
     .
     . Sequence to be performed
     . when <cond1>=<cond2>=false
     .
 475 REM end IF THEN ELSEIF ELSE
```

NOT is a recognized Development POWER BASIC boolean primitive that returns a value of TRUE if its argument evaluates to FALSE; otherwise it returns a value of FALSE. However, this is not supported by Evaluation POWER BASIC. It is simple to effect the NOT function by taking the complement of the relational operator in the condition.

A condition can be written in the form:

```
      <exp1><relop><exp2>
```

The negation of a condition can then be written :

```
      <exp1><relop*><exp2>
```

where <relop*> is the complement of <relop> and is derived from the following table.

Relationship	<relop>	<relop*>
Equal to	=	<>
Greater than	>	<=
Less than	<	>=
Greater than or equal to	>=	<
Less than or equal to	<=	>
Not equal to	<>	=

For example:

NOT(a > b) becomes (a <= b)
NOT(p = q) becomes (p <> q)

An expression is considered to have a truth value of TRUE if it evaluates to a non-zero value, otherwise it is considered FALSE. Thus the statement :

```
IF <expression> THEN statement(s)
```

is shorthand for

```
IF <expression> <>0 THEN statement(s)
```

5.6.1.3 ELSE STATEMENT. Development POWER BASIC supports the ELSE statement. This is normally used in conjunction with the IF THEN statement. The syntax for this is:

```
ELSE <sequence>
```

where the statements in <sequence> are separated from each other by ':'.

The ELSE statement uses the ELSE flag (set or reset by the last IF THEN statement depending on whether the condition is true or false) to determine whether the statement(s) following the ELSE keyword are to be executed. Several ELSE statements may appear between IF THEN statements. Each will be executed if the condition proved to be false, otherwise they will be skipped.

Typically, this statement will be used as:

```
100 IF condl THEN seql
110 ELSE seq2
120 REM end IF THEN ELSE
```

In the above, <seql> is only executed if <condl> is true; if <condl> is false then <seq2> is executed. After executing the appropriate sequence, control is passed to the REM statement (line 120).

<Seq2> may itself consist of an IF THEN ELSE:

```
100  IF  cond1 THEN  seq1
110  ELSE  IF  cond2  THEN  seq2
120  ELSE  seq3
130  REM  end IF THEN ELSEIF
```

Here <seq3> is executed only if both <cond1> and <cond2> are false; <seq2> if <cond1> is false and <cond2> is true; and <seq1> if <cond1> is true.

5.6.1.4 FOR NEXT Statement. A simple loop construct (perform a sequence of statements a known number of times) can be implemented by as follows.

```
     Num=int
 100 IF num>lst THEN  GOTO 350  ! IF NOT(num<=lst)
     num=num+1                  ! increment loop count
     .
     . Sequence to be performed
     . while num<=lst
     .
     GOTO  100
 350 REM end iterative loop
```

where 'int' = initial value
'lst' = final value
'num' = loop counter

The above loop is performed until the final value is exceeded.

To implement a count-down loop, the test and increment statements would have to be changed to:

```
 100 IF num<lst THEN  GOTO 350  ! IF NOT(num>=lst)
     num=num-1                  ! decrement loop counter
```

These simple loop constructs can be made more powerful by modifying the increment (decrement for the count-down loop) statement to:

```
     num=num+stp
```
where 'stp' = required increment/decrement

As this type of loop is used often, BASIC provides its own loop construct in the form of the FOR NEXT statement. The syntax of this is:

```
     FOR <var>=<start> TO <final> STEP <step>
     .
     . Sequence to be performed
     .
     NEXT <var>
```

The <start>, <final> and <step> values can be any valid numeric

expression. If the <step> value is one, the STEP keyword may be omitted. The variable <var> specified by NEXT must coincide with that used by the FOR.

The FOR statement opens the loop and the NEXT statement closes it. If the condition:

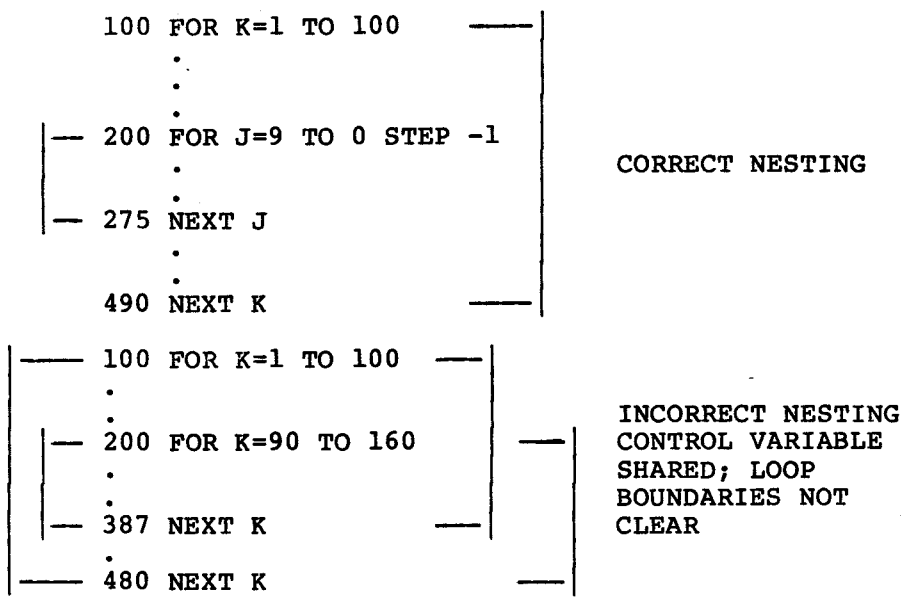
(step value)*(start value) > (step value)*(final value)

is true when the FOR statement is first encountered, the loop will not be executed. But if this condition is false, the FOR variable is set to the <start> value and the sequence of statements between the FOR and NEXT statements are executed. When the NEXT statement is encountered the FOR variable is incremented/decremented by the <step> value. Control is passed back to the FOR statement and while the condition :

(step value)*(FOR variable) <= (step value)*(final value)

remains true the loop will be executed. When execution of the loop is finished, control is transferred to the statement following the NEXT.

FOR NEXT loops can be nested (contained within one another). There is a maximum nesting depth of 5 for Evaluation Power BASIC, 10 for Development Power BASIC. No overlapping is allowed; inner loops must be closed before closing outer loops. Nested FOR NEXT loops must have different FOR variables; they can not share control variables. Otherwise, loop boundaries will not be clearly defined.



```
|---- 100 FOR K=1 TO 100 STEP 3
|     .
|     .                                  INCORRECT NESTING
|     200 FOR J=9 TO 0    ---|           OVERLAPPING LOOP
|     .                      |           BOUNDARIES
|     .                      |
|---- 300 NEXT K             |
|     .                      |
      400 NEXT J           ---|
```

Within the loop, the control variable can not be modified. It can however be used to access the elements of an array (for example).

While control can be transferred from within a loop to a statement outside, it is not possible to transfer control from outside to the inside.

A FOR NEXT loop can be written on a single line with '::' separating each statement:

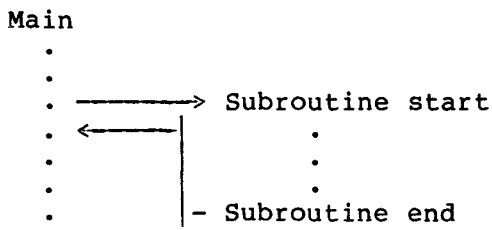
```
100 FOR I=0 TO 10 ::sequence::NEXT I
```

This disables the ESCape key on the terminal while the loop is being executed(until the loop has completed it is not possible to interrupt program execution and return Power BASIC to keyboard mode). This is because POWER BASIC will only recognize an interrupt at the end of the currently executing statement. Also, if the initial check indicates that the loop is not to be executed, error 31 (FOR W/O NEXT) will result because the NEXT statement will not be found.

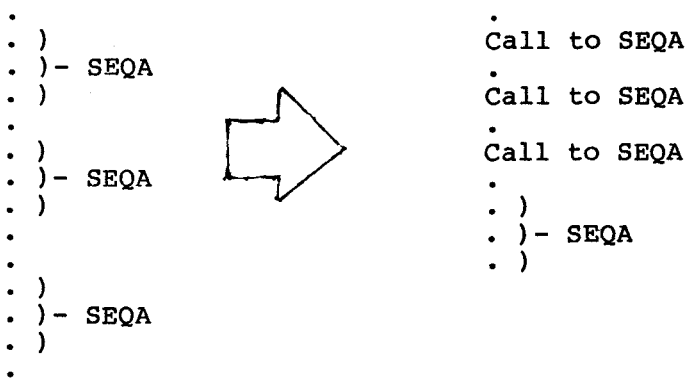
5.6.2 Subroutines

As previously stated, statements are normally executed in a straightforward sequential manner. A subroutine represents a method of executing a number of statements outside of the normal sequence.

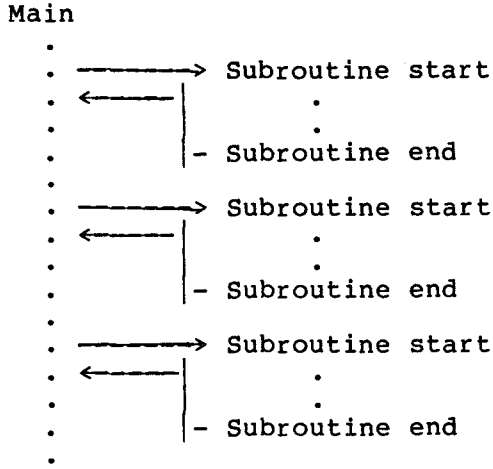
Pictorally, subroutine execution is:



If a subroutine is only used once, there is little point in separating the subroutine code from the calling routine. No benefit is derived apart from (perhaps) clarifying the program structure. However, there is a benefit when a subroutine is used to replace a number of statements that appear in several different places in a program. An example follows.



Program execution would become:



If the subroutine is large there can be a considerable saving realized in program storage (because of a small overhead) in calling and in returning from the subroutine.

A POWER BASIC subroutine is simply a sequence of statements that is entered using the GOSUB statement and exited via a RETURN statement. A subroutine can have multiple exit points (each distinguished by a RETURN statement), but this is usually considered bad programming practice. The syntax for the GOSUB and Return statements are:

```
GOSUB <ln>
RETURN
```

An example follows:

```
        .
        .
    100 GOSUB  2000                 ---|
--> 110 REM  RETURN TO HERE             |
        .                               |
        .                               |
        .                               |
    2000 REM  START OF SUBROUTINE  <----|
        .
        .
--- 2300 RETURN   ! EXIT SUBROUTINE
```

A GOSUB statement causes the address of the statement immediately following it to be pushed onto the GOSUB stack and then passes control to the specified line. In the above, the address of line 110 is pushed onto the top of the stack before control is passed over to line 2000.

The RETURN statement transfers program control back from a subroutine

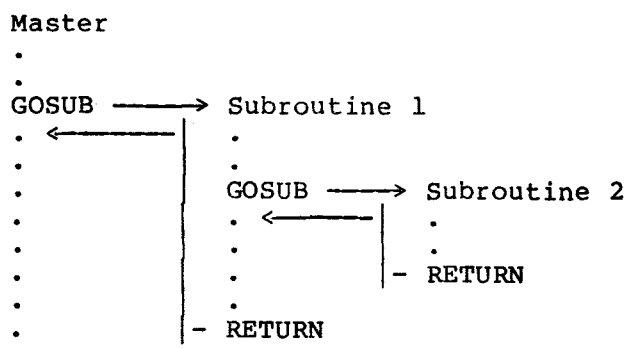
to the statement following the last GOSUB executed, by popping the top item off the GOSUB stack. In the above, the last entry to the stack (address of line 110) is popped thus allowing control to be passed back to line 110.

If a subroutine is exited by any way other than a RETURN statement, program flow can become unpredictable. POWER BASIC performs no check that a subroutine has been exited via a RETURN statement . Executing a RETURN statement when a subroutine has not been invoked will result in Error 12 (STACK UNDERFLOW).

Subroutine calls may be nested, (a subroutine may call another subroutine, upto a maximum of 10 levels for Evaluation Power BASIC and 20 levels for Development Power BASIC). In other words, there can be a maximum of 10 outstanding RETURNS at any one time. An attempt to exceed this number will result in error 11 (STACK OVERFLOW).

```
          .
    55 GOSUB  200
60 REM  RETURN TO HERE FROM S/R1
          .
          .
          .
   200 REM START OF S/R1
          .
          .
   270 GOSUB  1000
280 REM  RETURN TO HERE FROM S/R2
          .
          .
   400 RETURN    ! EXIT S/R1
          .
          .
  1000 REM  START S/R2
          .
          .
1200 RETURN    ! EXIT S/R2
```

Pictorally, program execution becomes :



As a POWER BASIC subroutine has complete access to all variables defined in a program, no parameter passing mechanism is supplied (nor is one really necessary). POWER BASIC is not a block structured language. Therefore, the programmer must make his own checks that variables are not accessed incorrectly (inadvertently modified by a subroutine). If a subroutine can write over critical data, it is necessary to use temporary variables for storage of this data. The programmer should then make sure that the subroutine can only access this data through the temporary variables.

5.6.3 ON Statement

The ON statement is a type of 'computed' GOTO. The syntax for this is:

```
ON <expression> THEN GOSUB/GOTO <l1>,<l2>,...,<ln>
```

A branch is made to line <li>, depending on the value of <expression>, via a GOTO or GOSUB statement. This is equivalent to:

```
     IF  <expression>=1  THEN  GOTO/GOSUB  <l1>
ELSE IF  <expression>=2  THEN  GOTO/GOSUB  <l2>
.
.
ELSE IF  <expression>=n  THEN  GOTO/GOSUB  <ln>
```

If a GOSUB is used on returning from the subroutine, control passes to the statement following the ON statement.

If the expression evaluates to less than one or greater than 'n', no transfer is made and execution continues from the statement following the ON.

5.6.4 ERROR Statement

The ERROR statement allows the user to specify a POWER BASIC routine that is to be executed when an error occurs. The syntax for this is:

```
ERROR <ln>
```

When an error condition arises, control is passed to line <ln> via a GOSUB statement; this preserves the address of the statement in which the error occurred on the GOSUB stack. If the error is recoverable, a return to the line with the error is made by a RETURN statement. However, if the error is unrecoverable, control will not be transferred back by the RETURN.

When the error-handling routine has been invoked, the system function SYS can be interrogated to find the cause of the error. SYS(1) will return the error code number, and SYS(2) the number of the statement in which the error occurred. Once the error has been trapped using this statement, future errors will not be trapped until another ERROR

statement is executed.

Use of the ERROR statement suppresses the automatic printing of error code/message.

```
   10 ERROR  1000
      .
      .
      .
 1000 REM  ERROR HANDLING ROUTINE
 1010 IF NOT(SYS(1)=23) THEN PRINT "ERROR=",SYS(1)::STOP
 1020 RESTOR
 1030 RETURN
```

When an error occurs, control is transferred to statement 1000. If the error was not due to "READ OUT OF DATA" (error 23) then the message "ERROR=" and the error code are printed to the terminal, and program execution STOPs. Otherwise the error is corrected by resetting the READ pointer to the first DATA statement in the program. The RETURN statement next passes control back to the line where the error occurred.

5.6.5 CRU Operations

The 9900 supplies a bit-oriented method of I/O called the Communications Register Unit (CRU). Under Power BASIC the CRU is accessed using the BASE statement and the CRB and CRF functions. For full details of the CRU and its operation refer to Section 6.8.

5.6.5.1 BASE Statement. CRU operations are performed on a signed displacement (in the range -128 to +127 bits) from a base address. This base address is set using the BASE statement. The syntax for this statement is :-

```
BASE   <exp>
```

where <exp> is any valid arithmetic expression.

Note: The base address is a 12 bit address that is stored in bits 3 to 14 of workspace register 12. Because of this, the value of <exp> must be twice that of the actual CRU base address desired. For example; to access a device that has a CRU base address of 32, <exp> must evaluate to 64.

5.6.5.2 CRB Function. Single-bit I/O is performed using the CRB function. Depending on the context in which it is used, this function either reads or writes to the specified bit.

When reading, the function returns a 1 if the specified bit is set, and a 0 if it is not set. For example;

```
IF   CRB(15)   THEN   <statement>
```

<Statement> is only executed if the 15th bit from the base address is set to 1.

When writing, the selected bit is set to 1 if the assigned value is non-zero, and to 0 if the assigned value is zero. For example;

```
CRB(100)=200
```

Sets the 100th bit from the base address to 1.

5.6.5.3 CRF Function. The specified number of bits are transferred to or read from the CRU starting at the address set by the BASE statement. The specified number of bits is in the range 0 to 15. If zero, all 16 bits are transferred. For example;

```
CRF(0)=-1
```

Transfers the 16 bit value (minus one - hex FFFF) to the CRU address specified by the BASE statement.

```
VAL=CRF(8)
```

Reads the 8 bit value from the CRU base address and stores the result in VAL. (VAL will be in integer format with the value occupying the least significant byte of the integer word.)

5.6.6 MEM Function.

The memory modification (MEM) function reads or modifies the specified byte memory location. The main use for this function is in performing memory mapped I/O. For example, if a peripheral device register is located at address >AE00, the character 'A' can be output by

```
MEM(0AE00H)=65            !DEC 65=ASCII 'A'
or   MEM(0AE00H)=ASC('A')
```

ASC returns the decimal ASCII code of the character argument.

A character can be read from the device by

```
$CIN=%MEM(0AE00H)
```

5.6.7 Interrupts

Development POWER BASIC allows the user to perform interrupt handling using Power BASIC statements. This is achieved through the IMASK, TRAP and IRTN statements.

When an interrupt occurs, the interpreter completes the POWER BASIC

statement it is executing and then executes the specified Power BASIC interrupt subroutine. On completion of the interrupt routine, execution continues from the statement following the last one executed before the interrupt was recognized.

With the TM 990/100M and /101M microcomputer modules, all interrupt lines are connected to the onboard TMS 9901 Programmable Systems Interface. It is this device that informs the 9900 microprocessor when an interrupt has been generated. The 9901 is accessed via CRU instructions using a hardware base address of >80; this address needs to be doubled when used in the BASE statement to set the base address of the 9901. For an interrupt to be recognized by the 9901 (and subsequently by the 9900), its level must be enabled. This is performed by setting the appropriate mask bit in the 9901's CRU address space to 1 (for details on the operation of this device refer to the TMS 9901 Programmable Systems Interface' Data Manual).

To program the 9901 to enable an interrupt level it is necessary to:

1) Select interrupt mode.
2) Write a 1 to the appropriate mask bit.

For example: to enable interrupt level 7:

```
BASE 100H       !set base address of 9901
CRB(0)=0        !set control bit=interrupt mode
CRB(7)=1        !enable mask 7
```

If a 0 is written (instead of 1) to the mask bit that interrupt level is disabled. For example, to disable interrupt level 12:

```
CRB(0)=0        !select control bit=interrupt mode
CRB(12)=0       !disable mask 12
```

The above example assumes that the base address of the 9901 has already been set.

Additional information on interrupts is contained in Section II of this manual.

5.6.7.1 IMASK Statement. The IMASK statement is used to control the interrupt mask (bits 12 to 15 of the Status Register) of the TMS9900 microprocessor. The 9900 recognizes 16 distinct interrupt levels, level 0 is the highest priority interrupt and level 15, the lowest.

Level 0 is reserved for the RESET function and level 3 for the real-time clock. Apart from these two, all other interrupt levels may be used by external devices. Several devices may share the same interrupt level (if system considerations require it). If this is the case, the programmer must ascertain which device caused the interrupt by polling the devices' status registers.

An interrupt can only be recognized by the TMS9900 when the incoming interrupt has an equal or higher priority (equal or lower numerical level value) than that specified in the interrupt mask of the status register. If, for example, the interrupt mask is set to 5, then only interrupt levels 0 to 5 will be recognized by the processor. The interrupt mask can be changed using the IMASK statement. The syntax for this statement is:

```
IMASK   <exp>
```

where <exp> is an arithmetic expression in the range 0 to 15.

Note : Setting the interrupt mask to 2, 1 or 0 disables the real time clock.

5.6.7.2 TRAP Statement. The TRAP statement is used to define a POWER BASIC subroutine that is to be executed when an interrupt of the specified level occurs. The syntax for this statement is:

```
TRAP   <exp>   TO   <ln>
```

where <exp> is the interrupt level and <ln> is the line number of the first statement of the interrupt routine.

5.6.7.3 IRTN Statement. The last statement of an interrupt subroutie must be an IRTN. When this statement is executed, the interpreter recognizes that the interrupt has been serviced and that it should continue program execution from where it left off. The syntax for this statement is:

```
IRTN
```

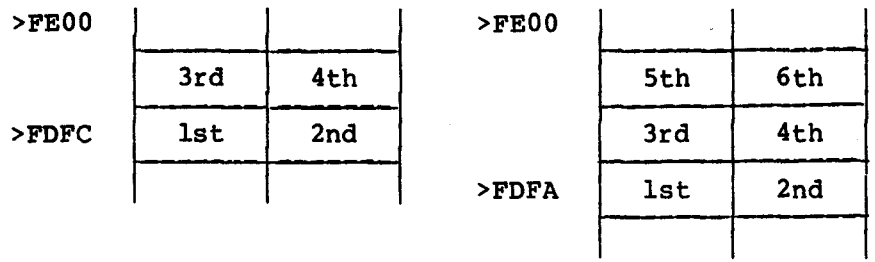
5.7. POWER BASIC STORAGE ALLOCATION

The paragraphs that follow discuss variable storage and the system memory map.

5.7.1 Variable Storage

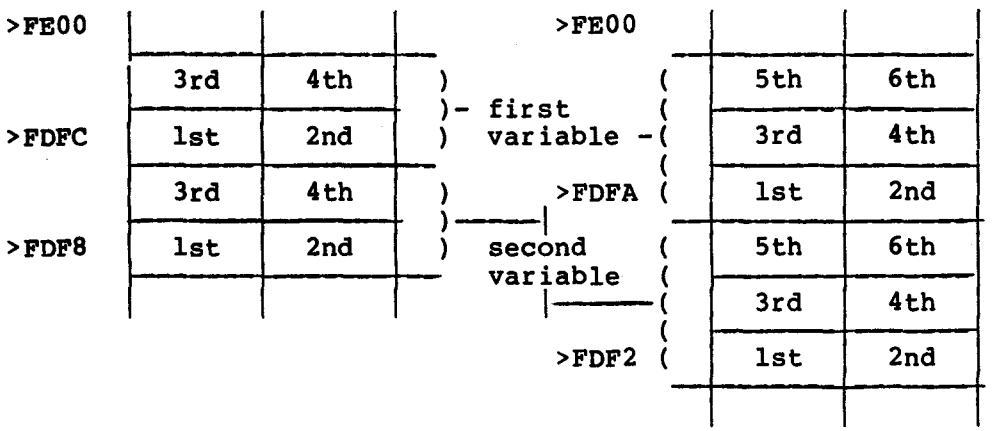
Variable storage starts in high memory and builds down toward low memory as each new variable is declared. In Development POWER BASIC a variable is allocated 6 consecutive bytes; while in Evaluation Power BASIC only 4 bytes are used.

Suppose variable storage starts at memory address >FE00, the first variable used will be allocated space as follows:



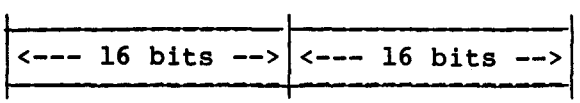
Evaluation POWER BASIC Development POWER BASIC

The next variable will be allocated space as follows:



Evaluation POWER BASIC Development POWER BASIC

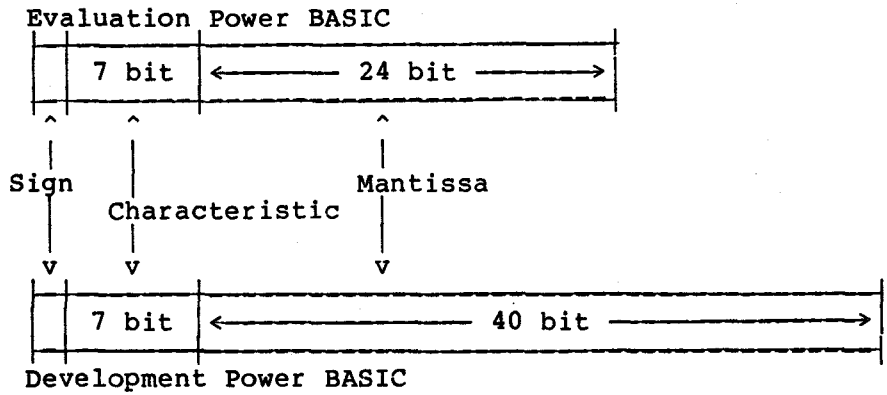
5.7.1.1 Integer Format. Integer numbers are stored in 32 bits.



The first word (bits 0 to 15) is set to zero indicating an integer number. The second word (bits 16 to 31) contains the two's complement integer value.

Although Development POWER BASIC only uses two words to store an integer number, three words are actually allocated. If three words were not allocated it would be extremely difficult for the interpreter to swap a variable's contents between integer, floating point or character string formats as the context required.

5.7.1.2 Floating Point Format. Floating point numbers are represented internally as a fraction multiplied by a power of 16 (this power is known as the characteristic) and are stored as:

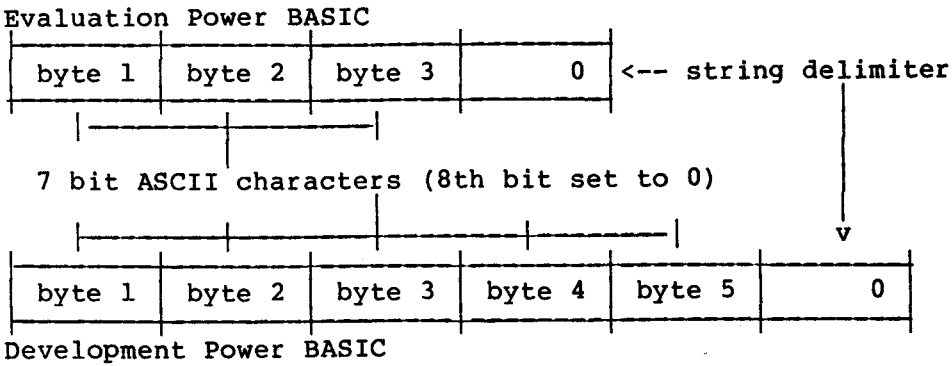


Bit 0 is the sign bit and represents the sign of the floating point number: 0 for positive, 1 for negative. Bits 1 to 7 hold the characteristic coded in Excess 64 notation (the characteristic is incremented by 64; this gives the characteristic a range of 0 to 127 representing a true exponent range of -64 to +63). The remaining 24 bits (40 for Development Power BASIC) contain the normalized mantissa (the mantissa is normalized if its first hex digit is non-zero). Negative fractions are stored in true form with the sign bit set to one and not in two's complement notation.

A notional point is understood to exist between bits 7 and 8 (between the characteristic and the mantissa).

5.7.1.3 Character String Format

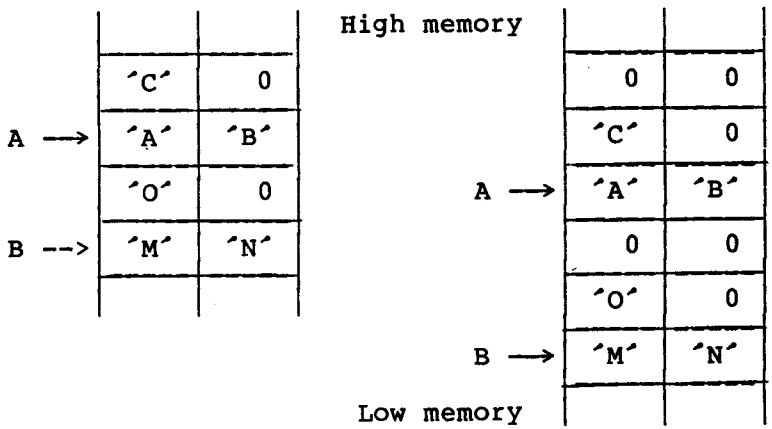
A character string is stored as follows:



Suppose the two variables A and B, defined in that order, occupy successive memory locations. The statements:

```
$A="ABC"
$B="MNO"
```

would cause these strings to be stored as follows:

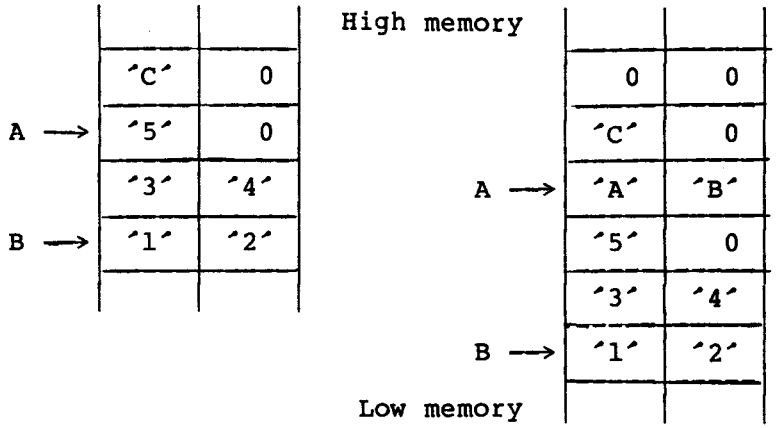


Evaluation Power BASIC Development Power BASIC

However, the statement:

```
$B="12345"
```

would result in:



Evaluation POWER BASIC Development POWER BASIC

With Evaluation Power BASIC, the statement:

```
PRINT $B
```

would output the string '12345', while the statement:

```
PRINT $A
```

would output the string '5'.

An effect similar to overwriting the contents of variable A can be produced with Development POWER BASIC statement :

```
$B='1234567'
```

When a character string is too long to be held in a variable, an array should be used.

5.7.1.4 Array Storage. An array is referenced by its array header. This contains information such as the size of each dimension and its stride. The stride is the number of bytes between successive elements of an array. For a one-dimensional array the stride is 6; 4 for Evaluation Power BASIC.

The memory address of any element in a one dimensional array is calculated (in bytes) as:

```
start address + n * subscript
```

where start address = address of array header + 4
n = 4 for Evaluation POWER BASIC
6 for Development POWER BASIC

If the array header is located at >EFF0, the 9th element, array name(8), starts at memory address:

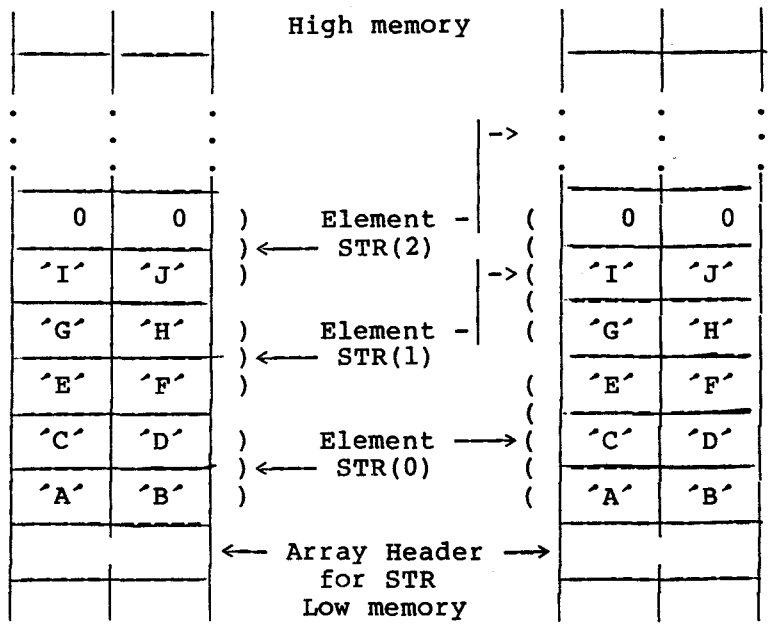
```
>EFF0 + 4 + n*8
```

For Evaluation POWER BASIC = >EFF4 + 4*8 = >F014
For Development POWER BASIC = >EFF4 + 6*8 = >F024

To allocate a ten-element array (STR) and store the character string 'ABCDEFGHIJ' into it, the following statements are required.

```
DIM  STR(9)
$STR(0)='ABCDEFGHIJ'
```

This string would be stored as :-



Evaluation POWER BASIC Development POWER BASIC

The statements:

```
PRINT   $STR(0)
PRINT   $STR(1)
PRINT   $STR(2)
```

would produce the following output:

```
ABCDEFGHIJ                      ABCDEFGHIJ
EFGHIJ                          GHIJ
IJ
Evaluation Power BASIC          Development Power BASIC
```

Individual bytes of an array containing a character string can be accessed by placing a semicolon (';') after the array subscript; and then writing the number of the required byte in that element. For example, $STR(1;3) references the letter 'G' (the letter 'I' in Development POWER BASIC).

The statement:

```
DIM  LST(25,9)
```

allocates space for 26 one-dimensional arrays each containing 10 elements. The stride for the first indice will be 60 (40 for Evaluation POWER BASIC); the stride for the second will be 6 (4 for Evaluation POWER BASIC).

The memory address of any element in a two-dimensional array is calculated (in bytes) as:

```
start address + n*(subscript1*multiplier + subscript2)
```

where start address = address of array header + 4*m
m = number of dimensions
multiplier = maximum value of subscript2 + 1
n = 4 for Evaluation Power BASIC
6 for Development Power BASIC

If the array header for LST is located at >E4DC then the element LST(16,4) is at memory address:

```
>E4DC + 4*2 + n*(16*10 + 4)      = >E4E4 + n*164
```

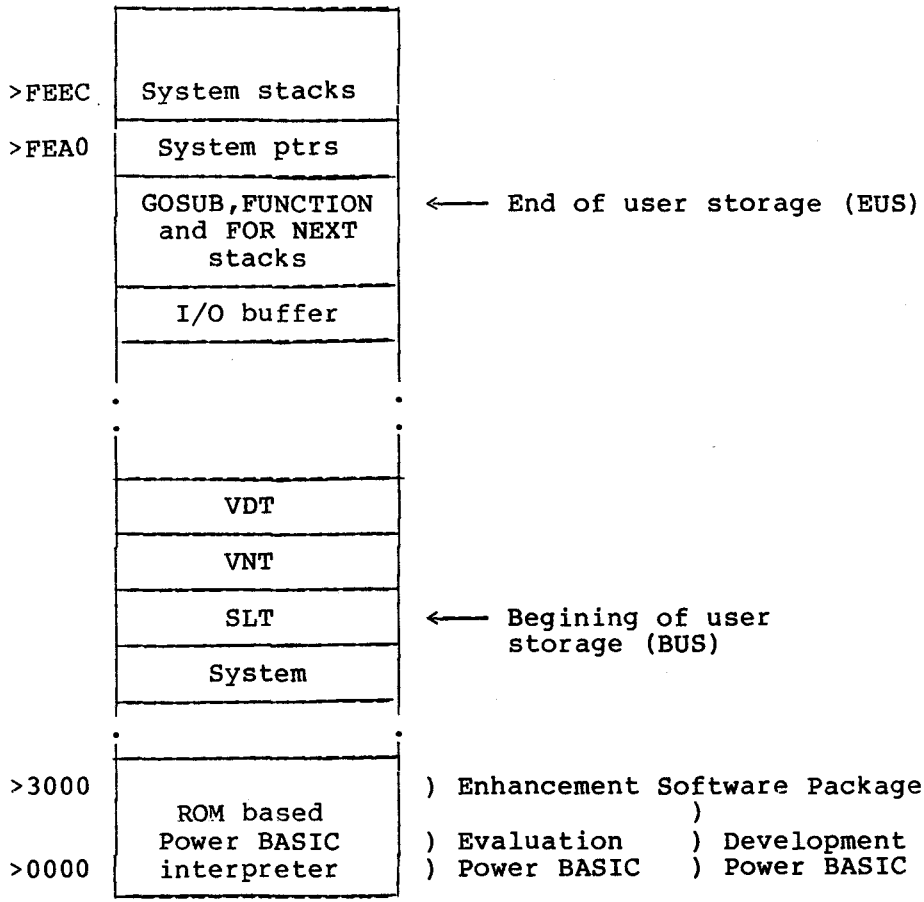
for Evaluation Power BASIC = >E4E4 + 4*164 = >E774
For Development Power BASIC = >E4E4 + 6*164 = >E8BC

5.7.2 System Memory Map

RAM, in addition to that supplied on board with the TM 990/101 M and /100 M CPU boards, must be configured to be contiguous and end at address >EFFF. For full details on how to do this, refer to Section 3 of the TM 990/201 and TM 990/206 Memory Expansion Boards Data Manual.

The lower limit of RAM is determined at system initialization time by autosizing. (This can be altered using the 'NEW <exp>' command, where <exp> is the address of the first byte of RAM to be used.) The first few bytes of RAM are reserved for sytem use.

Once the system has been initialized, the memory map will look something like:



Note : The actual addresses of the unspecified portions of memory are dependent on the amount of RAM configured into the system. Each of these portions is accessed via pointers contained in system pointers (memory addresses >FEA0 to >FEEB).

When a POWER BASIC statement is entered, it is checked for syntax errors. Syntactically correct statements are encoded to minimize storage space. The encoded statement is stored in the program space in ascending line number order. Program space starts at BUS and builds up in memory towards EUS. Line numbers are stripped off the statements as they are encoded and are stored in the Statement Location Table (SLT) along with the statement's position in the program space. (This allows statements that are entered out of sequence to be repositioned in their correct program space.) When a variable is first encountered, its name is encoded and entered into the Variable Name Table (VNT). Before a statement is encoded, all variable names present are compared with the variables in the VNT. The statement's variables are then replaced by the variable name's position (indicated by a location number) in that table. For example,

the statement:

```
LET   AJ=SIN (PI*RAD)
```

can initially be converted into:

```
LET   <77>=SIN(<76>,<75>)
```

Each two-digit hexadecimal number enclosed by angle brackets is incremented by >70 to indicate that an entry in the VNT is being referenced. <77> is the 8th entry in the VNT, <76> the 7th and <75> the 6th.

As the program grows, it may be necessary to move the system tables (VNT, VDT and SLT) up in memory in order to expand program space and increase the size of each table. At run time, space is allocated to each variable as they are defined; the address of this space is recorded in the Variable Definition Table (VDT). Variable storage space is allocated from below the I/O buffer down towards BUS. The Next Variable Storage Pointer (NVS) contains the address of the memory location that will be allocated to the next variable defined. If insufficient space exists, the run will terminate with error 10 (STORAGE overflow).

Note: All addresses refer to Development Power BASIC.

5.8. REFERENCE SECTION

+/-	Means plus or minus
/	Means not equal

An item preceded by an asterisk ('*') denotes a feature that is not supported by Evaluation Power BASIC.

5.8.1 Character Set

1) Upper and lower case alphabet.
2) Digits 0 to 9.
3) Special characters
   ! " # $ % ^ ' (! $) * : = - + ; , . ? / < >

Non-printable characters may be specified by enclosing the character's hex representation with angle brackets.

CHARACTER	USE
::	Statement separator
!	Tail remark indicator
;	Equivalent to PRINT

5.8.2 Hexadecimal Constants

A hexadecimal integer constant is one to four hex digits followed by the letter H. A hex constant begining with one of the letters A - F must be preceded by a zero.

5.8.3 Variable Names

A variable name starts with an alphabetic character optionally followed by up to two additional alphabetic characters or a number in the range 0 to 127. The variable name may not be the same as a POWER BASIC keyword; nor can it form the begining of a keyword.

5.8.4 String Variables

A varible is specified as being a sring variable variables may have a byte index following the subscript(s) to indicate a byte position within the specified string. To indicate the byte index, place a semicolon (;) after the last subscript; then insert the byte position.

5.8.5 POWER BASIC Commands

POWER BASIC commands may not appear in a program.

COMMAND	FUNCTION
CONtinue	Continue execution from last break
<ln> LISt	List current program from specified line <ln>=null, line=lowest line number present <ln>= /null, line=<ln>
LOAd <exp>	Load a BASIC program from specified device <exp>=null, device=733 digital cassette <exp>=0, device=733 digital cassette <exp>=1 or 2, device=audio cassette > <exp>=address, device=2716 eprom
NEW <exp>	Clear system for new program <exp>=null, RAM limit set by autosizing <exp>= /null, RAM limit=<exp>
PROgram	Burn current program into 2716 EPROM
RUN	Clears all variable space, pointers, and stacks and executes current program from lowest line number present
SAVe <exp>	Save current program on specified device <exp>=null, device=733 digital cassette <exp>=0, device=733 digital cassette <exp>=1 or 2, device=audio cassette >
SIZe	Display size of current program

> When using an audio cassette player all interrupts are disabled. The real time clock is stopped and zeroed.

5.8.6 Edit Commands

```
     CR          Enter line into program source
     LF          Enter line into program source.  Enable
                 auto-numbering facility
     ESCAPE        Cancel input line, return to keyboard
     DEL/RUBOUT  Backspace and delete character
     ctrl D<n>    Delete <n> characters
     ctrl I<n>    Insert <n> blanks
     ctrl H      Backspace 1 character
     ctrl F      Forwardspace 1 character
<ln> ctrl E      Display line <ln> for editing
```

5.8.7 POWER BASIC Statements

POWER BASIC program lines are of the form:

```
    (line number)- <statement> [ :: <statement> ] -!comment-
```

```
Where    (  )  Indicate optional items
         [  ]  Indicate item is repeated as many times
               as required - 0,1,....
```

Exceptions:
   NEXT should not be preceded by '::<statement(s)>'
   REM should not be followed by '::<statement(s)>'
   DA a should the only statement on a line

```
     BAUD <exp1>,<exp2>
```
Sets the baud rate of the serial I/O port(s) of the TMS 9902 Asynchronous Communications Controller.

```
          <exp1>=0, port=A (CRU address >80)
          <exp1>/0, port=B (CRU address >180)
          <exp2>=0, baud rate=19200
          <exp2>=1, baud rate=9600
          <exp2>=2, baud rate=4800
          <exp2>=3, baud rate=2400
          <exp2>=4, baud rate=1200
          <exp2>=5, baud rate=300
          <exp2>=6, baud rate=110
```

```
     BASE <exp>
```
Sets CRU base address to <exp> for subsequent CRU operations.

```
     CALL <name>,<add>- , <parms> )
```
Transfers control to an assembly language subroutine.

```
          <name>=IDenTity of subroutine in quotes
          <add>=hex address of subroutine
          <parms>=upto 4 parameters for subroutine, separated by
```
commas. If the parameter is contained in parenthesis, the address of the parameter is passed over. Parameters passed in R4, R5, R6, and R7. Return address is contained in R11.

DATA <item>[, <item>]
Defines internal data block for access by READ.
<item>=<exp> or <string>

DEF FN<i> ((<arg>))=<exp>
Defines a single line arithmetic statement. <i>=function identifier letter
<arg>=upto 3 single letter dummy variables, separated by commas. When calling FNi the dummy variables may be replaced by any valid Power BASIC variable/array.

DIM <var> (<dim> [, <dim>])
Allocates user space for dimensioned array. (The dimension starts at element 0.
<dim>=size of dimension

ELSE <statement> [:: <statement>]
When the most recently executed IF THEN statement is false, all subsequent ELSE statements are executed; otherwise they are ignored.

END
Terminates program execution and return to keyboard mode.

ERRO <ln>
Specifies a subroutine,starting at line <ln>, that is to be executed via a GOSUB statement when an error occurs.

ESCAPE
Enables the ESCape key to interrupt program execution.

for <var>=<expl> TO <exp2> (STEP <exp3>)
The FOR statement is used with the NEXT statement to open and close a program loop. Both identify the same variable <var>. If STEP is omitted, a stepsize of 1 is assumed.
<expl>=starting value
<exp2>=final value
<exp3>=step value, default value=1

GOSUB <ln>
Transfers control to an internal POWER BASIC subroutine starting at line <ln>. Stores the address of the statement following on the GOSUB stack.

GOTO <ln>
Transfers control to line <ln>.

IF <cond> THEN <statement> [:: <statement>]
The statement(s) following the THEN keyword are only executed if the condition <cond> is true.

IMASK <exp>
Sets the interrupt mask of the TMS 9900 microprocessor to allow interrupts of higher or equal priority (equal or lower numerical

value) to <exp>. <Exp> is valid over the range 0 to 15.

IRTN
Used to return from an interrupt routine. Restores the program environment existing prior to taking the interrupt.

(;)
INPUT <item> [(,)<item>]
Take input (numeric or string) from the terminal and store into variables <item> in the INPUT list. Input is prompted with a question mark ('?') for numeric data and a colon (':') for character data. A double question mark ('??') signifies an illegal number.

(LET) <var>=<exp>
Evaluate <exp> and store the result in the variable, string variable or array element <var>.

NEXT <var>
Delimits a FOR loop. The variable <var> must match the FOR variable.

NOESC
Disables ESCape key on the terminal.

(GOSUB)
ON <exp> THEN (GOTO) <ln> [, <ln>]
Transfer control, via a GOSUB or a GOTO statement, to the line specified by the value of the expression.
<exp>=n then nth <ln> in list
<exp> out of range then line following the ON

POP
Removes top item from the GOSUB stack.

PRINT <item> [, <item>]
Prints (without formatting) the contents or the evaluated expressions of the items in the PRINT list.

RANDOM <exp>
Sets the seed for the random number generator to the value of the expression <exp>.

READ <item> [, <item>]
Stores input from the internal DATA block into variables <item> in the READ list.

REM <text>
Inserts comment lines (REMarks) into a user program. The rest of the line regarded as a comment.

RESTOR <ln>

Resets the DATA pointer to the specified line <ln>. If <ln> not present, the pointer is set to the first DATA statement.

RETURN

Returns from a POWER BASIC subroutine and remove the last entry in the GOSUB stack.

STOP

Terminates program execution and returns to keyboard mode.

TIME <item>

Interrogate the 24 hour time of day clock.

<item>=null output time in HR:MN:SD format
<item>=$<var> store time in string variable
<item>=<expl>,<exp2>,<exp3> set clock to specified time. <expl>=hours;<exp2>=mins;<exp3>=secs

TRAP <exp> TO <ln>

Defines the entry point <ln> of a Power BASIC interrupt routine for the given interrupt level <exp>. <Exp> is valid over the range 0 to 15. Levels 0 (RESET) and 3 (CLOCK) are reserved and can not be serviced by the TRAP statement.

UNIT <exp>

Designates the device(s) to receive all printed output.

<exp>=1, I/O port=A
<exp>=2, I/O port=B
<exp>=3, I/O ports A and B

5.8.8 Operators

5.8.8.1 Arithmetic Operators.

A=B	Assignment
A-B	Subtraction
A+B	Addition
A*B	Multiplication
A/B	Division
A^B	Exponentiation
-A	Unary minus
+A	Unary plus

5.8.8.2 Relational Operators. Return values of 1 (TRUE) or 0 (FALSE).

A=B	TRUE if equal, else FALSE
A==B	TRUE if approximately equal (+/- 9.5E-7), else FALSE
A<B	TRUE if less than, else FALSE
A<=B	TRUE if less than or equal, else FALSE
A>B	TRUE if greater than, else FALSE
A>=B	TRUE if greater than or equal, else FALSE
A<>B	TRUE if not equal, else FALSE

5.8.8.3 Boolean Operators. Return values of 1 (TRUE) or 0 (FALSE). A non-zero value variable is considered TRUE; a zero-valued variable is considered FALSE.

NOT A	TRUE if FALSE (zero), else FALSE
A AND B	TRUE if both TRUE (non-zero), else FALSE
A OR B	TRUE if either TRUE (non-zero), else FALSE

5.8.8.4 Logical Operators. Perform 'bitwise' operations on the operand(s). Operand(s) are converted into 16 bit integers before the operation.

LNOT A	1's complement
A LAND B	Bitwise AND
A LOR B	Bitwise OR
A LXOR B	Bitwise exclusive OR

5.8.8.5 Operator Precedence

1) Expressions in parentheses
2) Exponentiation and negation
3) *,/
4) +,-
5) <=,<>
6) >=,<
7) =,>
8) ==,LXOR
9) NOT,LNOT
10) AND,LAND
11) OR,LOR
12) Assignment (=)

5.8.9 Arithmetic Functions

FUNCTION	EXPLANATION
ABS (<exp>)	Absolute value of <exp>
ATN (<exp>)	Arctangent of <exp>, <exp> in radians
COS (<exp>)	Cosine of <exp>, <exp> in radians
EXP (<exp>)	Raise e to the power of <exp>
INP (<exp>)	Signed integer part of <exp>
LOG (<exp>)	Natural logarithm of <exp>
RND (<exp>)	Random number between 0 and 1
SIN (<exp>)	Sine of <exp>, <exp> in radians
SQR (<exp>)	Square root of <exp>

5.8.10 CRU Operations

CRB (<exp>)
Read CRU bit selected by the CRU hardware base address plus <exp>. <Exp> is valid over the range -128 to |127.

CRB (<expl>)=<exp2>
Sets or resets CRU bit selected by CRU base address plus <expl>. If <exp2> is non-zero, the bit will be set, otherwise it will reset. <Expl> is valid over a range of -128 to |127.

CRF (<exp>)
Read <exp> CRU bits from the CRU hardware base address . <Exp> is valid over the range 0 to 15. If <exp>=0 then 16 bits will be read.

CRF (<expl>)=<exp2>
Output <expl> bits of <exp2> to CRU lines at the CRU hardware base address. <Expl> is valid over the range 0 to 15. If <expl>=0 then 16

bits will be output.

5.8.11 Memory Functions

BIT (<var>,<exp>)
Reads the bit, within the variable <var>, specified by <exp>. Returns a 1 if the bit is set and 0 if not set.

BIT (<var>,<expl>)=<exp2>
Modifies the bit, within the variable <var>, specified by <expl>. The selected bit is set to 1 if <exp2> is non-zero, otherwise it is set to 0.

MEM (<exp>)
Read the byte from user memory specified by <exp>.

MEM (<expl>)=<exp2>
Store byte <exp2> into the user memory specified by <expl>.

5.8.12 Miscellaneous Functions

NKY (<exp>)
Samples the keyboard in run-time mode. If <exp>=0 then return the decimal value of the last key struck and clear the key register. Zero is returned if no key was struck. If <exp>/0 then compare the last key struck with the decimal value of <exp>. If they are the same, a value of 1 is returned and the key register is reset, otherwise a 0 is returned.

SYS (<exp>)
Obtain system parameters generated during program execution.
<exp>=0, parameter=input control character
<exp>=1 parameter=error code number
<exp>=2, parameter=error line number

TIC (<exp>)
Samples the real time clock and returns the current TIC value minus the value of <exp>. One TIC equals 40 milliseconds. TIC (0) obtains the current value.

5.8.13 String Operations

<$var>	denotes either a literal string, enclosed in quotes, or a string variable
$<var>	denotes a string variable

CHARACTER ASSIGNMENT

characters are transferred one by one until a null character is found.

```
$<var>=<$var>
```

CHARACTER PICK

The number of characters to be transferred can be specified.

```
$<var>=<$var>,<exp>
```

When <exp> characters have been transferred the string will be terminated with a null character.

CHARACTER CONCATENATION

Strings are concatenated using the '+' operator. Concatenation operations may be chained together, the final string will automatically be terminated with a null character.

```
$<var>=<$var>+<$var> [ + <$var> ]
```

CHARACTER REPLACEMENT

Replacement is similar to character pick except that a null character is not placed at the end of the string.

```
$<var>=<$var>;<$var>
```

CHARACTER INSERTION

Characters can be inserted into a string using the slash ('/') operator.

```
$<var>=<$var>/<$var>
```

CHARACTER DELETION

<Exp> number of characters can be deleted from a string.

```
$<var>=<$var>/<exp>
```

BYTE REPLACEMENT

Individual bytes within a string can be altered using the decimal equivalent of an ASCII character along with the percent sign ('%').

```
$<var>=%<exp> [ % <exp> ]
```

STRING COMPARISON

Character strings may be compared using :

```
IF <$var><relop><$var>-,<exp>- THEN <sequence>
```

where <relop>=relational operator

If the second string is followed by a comma, the expression following indicates the number of characters to be compared.

CONVERT FROM ASCII TO BINARY

A character string may be converted to a number by using the assignment operator along with an error variable. The delimiting

character is placed in the first byte of the error variable.

```
<var>=<$var>,<var>
```

CONVERT FROM BINARY TO ASCII

A number is converted to a string simply by assigning the number to a string variable. The string is automatically terminated with a null character.

```
$<var>=<exp>
```

Formatted conversions can be made by preceding <exp> with the formatting operator ('#') and a string.

```
$<var>=#<$var>,<exp>
```

5.8.14 String Functions

ASC ($<var>)

Returns the ASCII decimal value of the first character in the specified string.

LEN ($<var>)

Returns the length of the specified string. Zero is returned if the string is the null string.

MCH ($<var1>,$<var2>)

Return the number of characters that are the same in the two strings. A zero is returned if no match is found.

SRH ($<var1>,$<var2>)

Return the character position of where the first string is located in the second. A zero is returned if the search is unsuccessful.

5.8.15 INPUT Options

INPUT <feature><item> [<del><feature><item>]

<item>	Either a variable, a string variable, or an array element

<del>	Explanation
,	Delimit items in INPUT list
;	Delimit items in INPUT list. Suppress <cr><lf> if at end of statement line

<feature>	Explanation
string	Prompt with string then get input
?<ln>	Upon an invalid input or control character, a GOSUB to the line <ln> is executed
%<exp>	Requires entry of exactly <exp> characters
#<exp>	A maximum of <exp> characters to be entered
;	Suppress prompting
null	Prompt (':' for numeric, '?' for character) and then get input

5.8.16 PRINT Options

```
PRINT  <feature><item> [<del><feature><item>]
```

<item>	Either a variable, an expression, a string variable, a string, or an array element

<del>	Explanation
,	Delimit items in PRINT list. TAB to next print field
;	Delimit items in PRINT list. Suppress <cr><lf> if at end of statement line

<feature>	Explanation
TAB (<exp>)	TAB to column specified by <exp>
#<exp>	Print <exp> in hex in free format
#,<exp>	Print <exp> in hex in word format
#;<exp>	Print <exp> in hex in byte format
#<string>	Decimal formatting - only available with the Enhancement Software Package. <string> in quotes, consisting of:
9	Digit holder
0	Digit holder or force 0
$	Digit holder and floats $
S	Digit holder and floats sign
<	Digit holder before decimal and floats on negative number
>	Appears after decimal if negative
E	Sign holder after decimal
.	Decimal point specifier
,	Comma in output - suppressed if before significant digit
^	Translated to decimal point on output

5.8.17 Floating Point XOP Package

For use with Assembly language routines.

```
FORMAT   XOP  <ga>,<op>
where    <ga> - general memory address operand
         <op> - XOP number
```

FPAC - Floating Point ACcumulator

XOP NO.	FUNCTION
0	LOAD FPAC with 6 byte number addressed by <ga>
1	STORE FPAC in 6 byte number addressed by <ga>
2	ADD 6 byte number addressed by <ga> to FPAC, store result in FPAC
3	SUBTRACT 6 byte number addressed by <ga> to FPAC, store result in FPAC
4	MULTIPLY FPAC by 6 byte number addressed by <ga>, store result in FPAC
5	DIVIDE FPAC by 6 byte number addressed by <ga>, store result in FPAC
6	SCALE adjusts FPAC's exponent to value of byte addressed by <ga>
7	NORMALISE FPAC - 1st hex digit of mantissa is non-zero. Operand not used
8	CLEAR FPAC. Operand not used
9	NEGATE FPAC - change 1st bit. If FPAC=0 then no change. Operand not used
10	FLOAT FPACs 2nd word - 16 bit two's complement number to floating point. Operand not used

Converting Integer to Floating Point

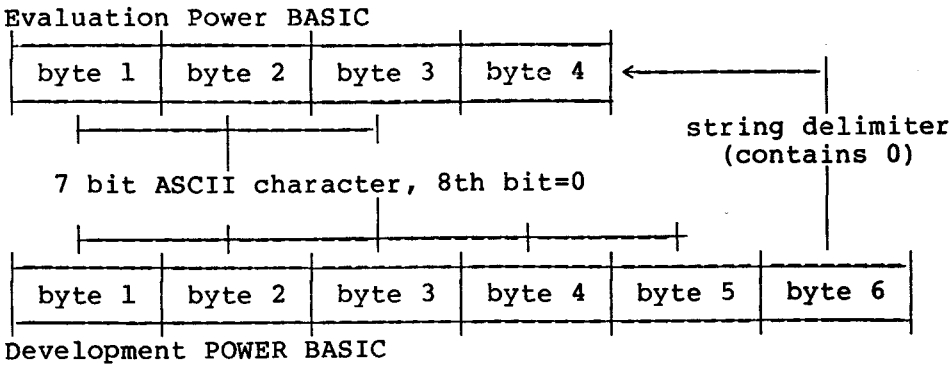
1) Set words 1 and 3 of 6-byte reserved area to zero.
2) Store integer number in 2nd word of area.
3) LOAD this 6-byte number into FPAC.
4) FLOAT FPAC.
5) STORE FPAC in 6 byte area.

```
DECNO BSS  6
FLPT  BSS  6
      .
      CLR  @DECNO
      CLR  @DECNO+4
      LI   R0,NUM
      MOV  R0,@DECNO+2
      XOP  @DECNO,0
      XOP  0,10
      XOP  @FLPT,1
```

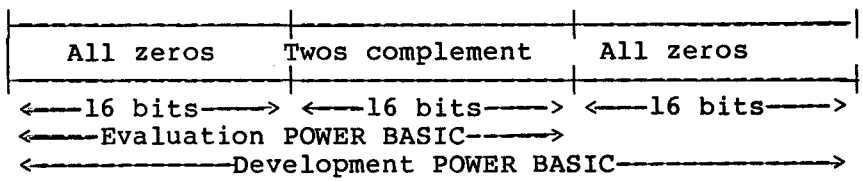
5.8.18 Variable Storage

A variable occupies 4 consecutive bytes in Evaluation Power BASIC and 6 in Development Power BASIC. Variable storage is allocated down through memory (from high memory to low). The variable is referenced by the address of the lowest byte it occupies.

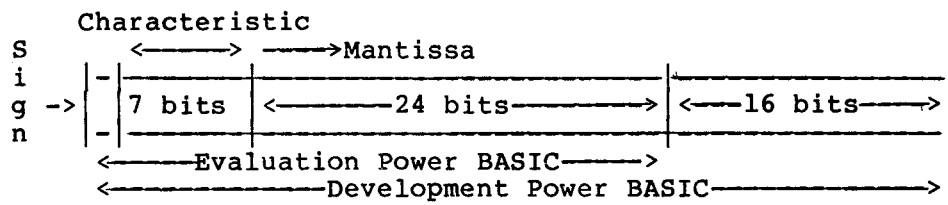
Character String Format



Integer Format



Floating Point Format



5.8.19 ASCII Character Set

CHAR	HEX	CHAR	HEX	CHAR	HEX
NUL	00	+	2B	V	56
SOH	01	,	2C	W	57
STX	02	-	2D	X	58
ETX	03	.	2E	Y	59
EOT	04	/	2F	Z	5A
ENQ	05	0	30	[	5B
ACK	06	1	31	\	5C
BEL	07	2	32	]	5D
BS	08	3	33	^	5E
HT	09	4	34	_	5F
LF	0A	5	35	`	60
VT	0B	6	36	A	61
FF	0C	7	37	B	62
CR	0D	8	38	C	63
S0	0E	9	39	D	64
S1	0F	:	3A	E	65
DLE	10	;	3B	F	66
DC1	11	<	3C	G	67
DC2	12	=	3D	H	68
DC3	13	>	3E	I	69
DC4	14	?	3F	J	6A
NAK	15	@	40	K	6B
SYN	16	A	41	L	6C
ETB	17	B	42	M	6D
CAN	18	C	43	N	6E
EM	19	D	44	O	6F
SUB	1A	E	45	P	70
ESC	1B	F	46	Q	71
FS	1C	G	47	R	72
GS	1D	H	48	S	73
RS	1E	I	49	T	74
US	1F	J	4A	U	75
SPACE	20	K	4B	V	76
!	21	L	4C	W	77
"	22	M	4D	X	78
#	23	N	4E	Y	79
$	24	O	4F	Z	7A
%	25	P	50	{	7B
&	26	Q	51	|	7C
'	27	R	52	}	7D
(	28	S	53	~	7E
)	29	T	54	DEL	7F
*	2A	U	55		

5.8.20 Hex-Decimal Table

EVEN BYTE				ODD BYTE			
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4,096	1	256	1	16	1	1
2	8,192	2	512	2	32	2	2
3	12,288	3	768	3	48	3	3
4	16,384	4	1,024	4	64	4	4
5	20,480	5	1,280	5	80	5	5
6	24,576	6	1,536	6	96	6	6
7	28,672	7	1,792	7	112	7	7
8	32,768	8	2,048	8	128	8	8
9	36,864	9	2,304	9	144	9	9
A	40,960	A	2,560	A	160	A	10
B	45,056	B	2,816	B	176	B	11
C	49,152	C	3,072	C	192	C	12
D	53,248	D	3,328	D	208	D	13
E	57,344	E	3,584	E	224	E	14
F	61,440	F	3,840	F	240	F	15

5.8.21 Error Codes

Code	Error message
1	Syntax error
2	Unmatched parenthesis
3	Invalid line number
4	Illegal variable name
5	Too many variables
6	Illegal character
7	Expecting operator
8	Illegal function name
9	Illegal function argument
10	Storage overflow
11	Stack overflow
12	Stack underflow
13	No such line number
14	Expecting string variable
15	Invalid screen command
16	Expecting dimensioned variable
17	Subscript out of range
18	Too few subscripts
19	Too many subscripts
20	Expecting simple variable
21	Digits out of range (0< no. digits >12)
22	Expecting variable
23	Read out of data
24	Read type differs from data type
25	Square root of negative number
26	Log of non-positive number
27	Expression too complex
28	Division by zero
29	Floating point overflow
30	Fix error
31	FOR without NEXT
32	NEXT without FOR
33	Exp function has invalid argument
34	Unnormalised number
35	Parameter error
36	Missing assignment operator
37	Illegal delimiter
38	Undefined function
39	Undimensioned variable
40	Undefined variable
41	Expansion EPROM not installed
42	Interrupt without TRAP
43	Invalid baud rate
44	Tape read error
45	EPROM verify error
46	Invalid device number

CHAPTER VI

ASSEMBLY LANGUAGE

6.1 INTRODUCTION

The relationship between assembly language and the computer it was designed to support is displayed below. Assembly language provides the interface between the hardware operation and the high-level language specifying the problem. Therefore, assembly language is machine dependent. As such, it has the capability to access the low-level features of the machine (memory, hardware registers, etc.)

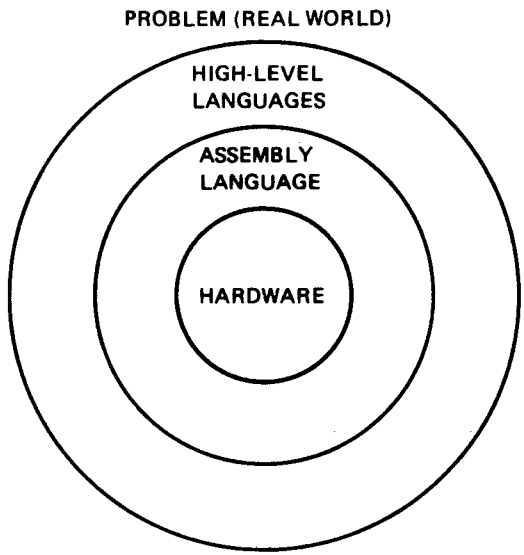


FIGURE 6-1. ASSEMBLY LANGUAGE AND COMPUTER

Due to its low-level nature, assembly language does not have the programming aids that are built into high-level languages. For example, high-level languages automatically provide the necessary data mappings and addressing mechanisms used to access declared variables, while the assembly language programmer must perform this housekeeping for himself.

Assembly language is useful when tight control must be maintained over the use of resources (for example where particularly compact or efficient code is required). A disadvantage of using assembly is that alot of programmer skill and time are required to realize compactness and efficiency. Using high-level languages can speed up program production considerably and the program will be less prone to errors.

Also, an assembly language program becomes more and more difficult to manage as its size increases.

However, assembly language is ideal for short, frequently executed program segments such as I/O routines and for high-volume applications where savings on code (and hardware) outweigh the extra development effort.

The machine instruction is a hardware-defined operation and is the basic unit of processing. The complete range of hardware instructions designed into a particular processor forms the instruction set (sixty-nine instructions make up the TMS9900 instruction set.)

Every program written for the 9900 (or any other processor) will eventually be broken down into a sequence of these basic instructions. Each instruction is actually stored in program memory as a number (a string of '0's and '1's). In this state the instruction is usually referred to as a machine code instruction.

While programming at the machine code level is possible, it is not very practical. Moreover, understanding the function of a machine code program is very difficult and requres careful study.

Assembly language allows programming directly in the machine's instruction set using mnemonics instead of numbers. Further, most assembly languages allow symbolic referencing; using a name to reference a data item or a code segment (the assembler translates these references into their actual memory addresses).

Consider the following example. A value is stored at address >4E70 (symbolic location START). This value is to be transferred to address >5630 (symbolic location NEW). The assembly language instruction

```
MOV  @START,@NEW
```

will do this. The machine code equivalent is:

```
>C820 4E70 5630
```

The symbol '>' indicates that the number that follows is a hexadecimal number (the hexadecimal number system is described in Subsection 6.11.2.

Before an assembly language program can be executed, it must first be converted into a form the processor can handle (machine code). This conversion is performed by an assembler on a one for one basis. (A single assembly language instruction generates one machine code instruction.)

Instructions can be one, two or three words long. The length of an instruction depends on the number of operands contained and the type of addressing allowed. The MOV instruction above has two memory address operands (START and NEW) and thus requires three words of storage. If one of these operands had been a register only two words would be needed. Had both operands been registers one word would be sufficient.

6.2 INSTRUCTION FORMAT

An instruction consists of four fields, each separated from the other by at least one space.

1) Label field - An optional field; when used the user-supplied name is assigned the current value of the location counter (the address in memory where the instruction will be stored). This field starts in column 1. An asterisk in column one indicates that the whole line is a comment.

2) Opcode field - The operation code, or mnemonic, specifies what the instruction does (e.g. MOV). Assembler directives, assembly language instructions and pseudo-instructions are covered by this term.

3) Operand field - This field specifies the argument(s) of the opcode; e.g., where the data is to be taken from (source) and/or where the the data is to be stored (destination).

4) Comment field - An optional field ignored by the assembler and used for documentation purposes. Although comments have no effect on the code produced, they are useful. They allow the programmer to describe exactly what is done at the point in the code where the action is performed. If used properly, comments can make a program completely self documenting.

The assembler places no restrictions on the position of any field in the line, except for the label field. However, it is advantageous for the programmer to adopt some convention. The recommended convention is as follows:

- label field starts in column 1
- opcode field starts in column 8
- operand field starts in column 13
- comment field starts in column 31

Several examples follow.

```
          OP
LABEL     CODE  OPERAND(S)          COMMENTS

RESET     CI    R4,>100             Contents of R4= >100?
*
* operands - 1 workspace register, 1 immediate value
*
          C     R2,R3               Contents of R2=R3?
*
* operands - both workspace registers
*
          B     @RESET              Branch to RESET
*
* operands - 1 symbolic memory location
*
          RSET                      Reset the 9900
*
* operands - none
*
```

From the above examples, it can be seen that the number of operands depends solely upon the instruction.

6.3 INSTRUCTION FORMAT RESTRICTIONS

Restrictions to instruction formats are listed below.

1) If a label is present, the instruction starts in column one; otherwise column one is left blank.

2) A label consists of up to six alphanumeric characters, the first of which must be alphabetic.

3) All fields are separated by one or more spaces.

4) Operands, if more than one is required, are separated by commas.

6.4 MEMORY ORGANIZATION

Computer memory is sequential and consists of a large number of storage cells or locations. Each location has a unique address. Using this address, the processor is able to directly reference a particular location.

Memory is used for storing patterns of bits that may be interpreted as either:

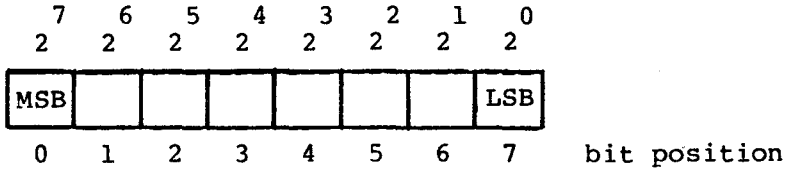
1) Programs - lists of instructions that tell the processor what to do.

2) Program Data - patterns of bits that can be used to

represent numbers, status of switches, etc (anything that the computer is programmed to deal with).

6.4.1 Byte

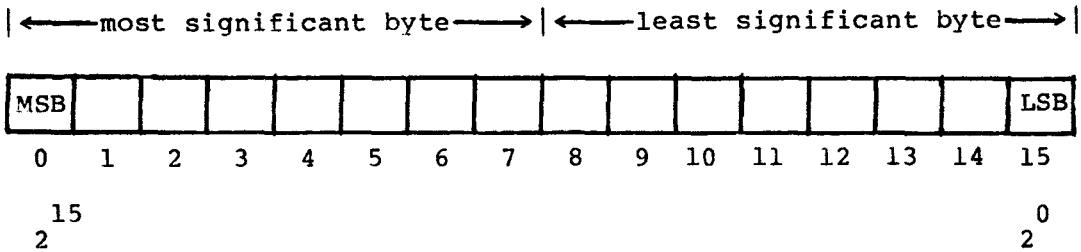
A byte is a group of eight binary digits (bits). The most significant bit (MSB) is designated bit zero and the least significant bit (LSB) as bit seven. The contents of a byte can be represented by two hex digits (>00 to >FF).



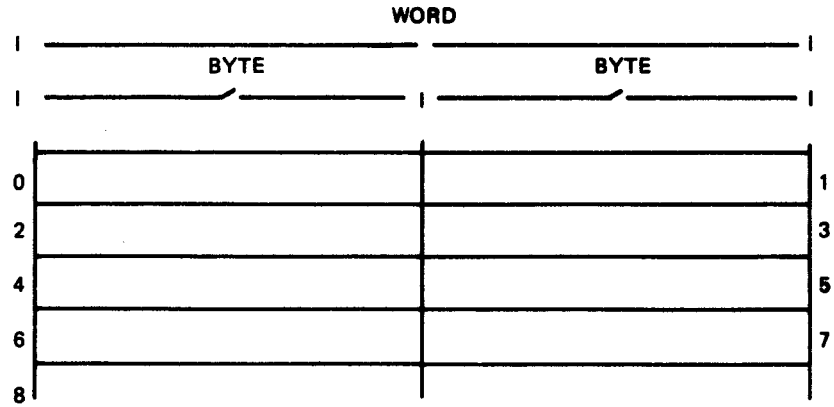
01101010 binary = 6A hex

6.4.2 Word

A memory word, on the 9900, occupies 16 bits (2 bytes). A word's MSB is designated bit 0 and its LSB as bit 15. The contents of a word can be represented by four hex digits (>0000 to >FFFF).



The architecture of the TMS9900 is based on words. However, semi-conductor memory is usually organized in bytes. Therefore, although the word is the basic unit, byte addressing is used. This means that the addresses of consecutive words in storage are n, n+2, n+4, etc. The first byte of a word (the most significant byte) must be on an even numbered address.



Storing a single byte's worth of data in a memory word is not very efficient. The 9900 instruction set provides a number of instructions for byte operations (e.g., MOVB, CB, AB, SB, etc). Using these instructions, it is possible to individually access/manipulate each of the bytes within a word.

6.4.3 Registers

Most computers provide a number of general purpose hardware registers that are accessible to the assembly language programmer. All operations are centred around these registers. To add the contents of two memory locations (A and B) together and store the result in the first location (A), these steps are necessary:

1) Load the contents of one of the locations into a register.

2) Add the contents of the other location into this register.

3) Store the contents of this register into memory location A.

The register-oriented instruction evolved because of the great differences in operation speeds between hardware registers and ferrite core memory.

The introduction of semi-conductor memory (considerably faster than ferrite core) into computer systems has eliminated the need for such registers. With the TMS9900 microprocessor, direct memory to memory operations are possible. The above example can now be performed in a single instruction.

The 9900 has only three dedicated hardware registers:

1) Program Counter (PC) - contains the address of the next instruction to be executed.

2) Workspace Pointer (WP) - contains the address of the first word of the current workspace.

3) Status Register (ST) - stores the processor's status flags (bits 0 to 6) and the current interrupt mask (bits 12 to 15). Bits 7 to 11 are reserved for future use.

6.4.4 Workspace Registers

The TMS 9900 does not provide one set of hardware implemented registers. Instead, any contiguous 16-word area of read/write memory (RAM) may be defined as the 16-word workspace. The 16 workspace registers (R0 to R15) may be used exactly as if they were implemented in hardware. However, the location of the workspace may be changed during program execution to give 16 completely new registers. This is called a context switch and occurs automatically during an interrupt or when the BLWP instruction is used to call a subroutine. The workspace can also be changed using the Load Workspace Pointer Immediate instruction (LWPI).

Although the registers can be located anywhere in memory, only 4 bits are needed to completely specify any word address within the workspace. This allows a register operand to be incorporated into the instruction word without having to set aside another word for the address.

The BSS (Block Starting with Symbol) assembler directive allows the user to reserve an area of data storage for use as a workspace. The following lines of code reserve a 16-word area starting at address >2000. The LWPI instruction causes this value to be loaded into the WP. When the instruction has been executed, R0 references address >2000, R1 references address >2002, etc.

```
        AORG   >2000
WKSP    BSS    32       Reserve 16 word area
        .
        .
        .
        LWPI   WKSP     Set WP= >2000
```

The benefit of this approach is realized when it is necessary to save the contents of the registers (for example, on interrupt). With the traditional approach, the content of every register has to be copied into reserved memory locations. With the 9900, only the three dedicated registers need to be saved and the WP loaded with the address of another workspace. This is handled automatically when an interrupt occurs.

6.4.5 Register Functions

In general, when a register is required as an operand for an instruction, any of the 16 workspace registers can be used. However, for certain operations (in particular the context switch) some of the registers have specially designated functions, as follows:

R0 — If the count operand to the shift instruction is zero, the shift count is taken from bits 12 to 15 of R0. If the 4 bits are all zeros, the shift count is set to 16.

R11 — Branch and Link instruction uses R11 to store its return address. R11 stores the effective address of the source operand for an XOP.

R12 — Bits 3 and 4 of R12 contain the hardware base for the CRU instructions.

R13 — When a context switch occurs, R14 is used to store the old PC.

R15 — When a context switch occurs, R15 is used to store the old ST.

Note: The MPY and DIV instructions use two consecutive registers, the first is supplied as an operand to the instruction (e.g.,if R2 is the register operand, R2 and R3 are both used). If R15 is the specified register, the word following the workspace is used to store either the remainder for DIV or the least significant half of the result for MPY.

6.4.6 Context Switch

When a context switch occurs, the WP and PC registers are loaded with new values. The old contents of the WP, PC and ST registers are then stored in the new workspace registers 13, 14 and 15 respectively. The old registers can be accessed using the indexed mode of addressing (see Addressing Modes, Section 6.4.7) on the new register 13.

Interrupts (both hardware and software) and the BLWP instruction cause a context switch to take place. For an interrupt, the WP and PC are taken from the interrupt's trap (or transfer) vector. The BLWP instruction requires the address of a two-word area, containing the new WP and PC, as its operand.

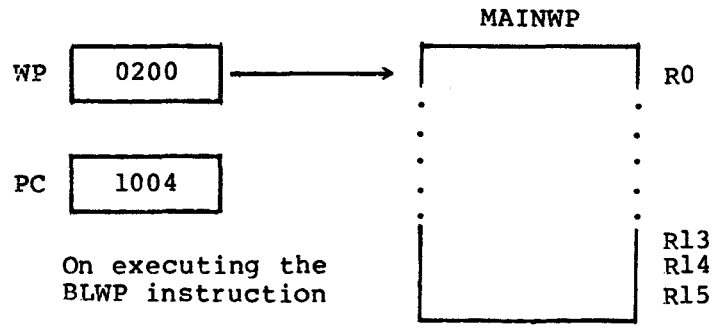
A context switch provides a completely fresh environment, or context, for program execution and results in program control being transferred to a new routine. The last instruction in this routine must be an RTWP. This restores the environment existing prior to the context switch.

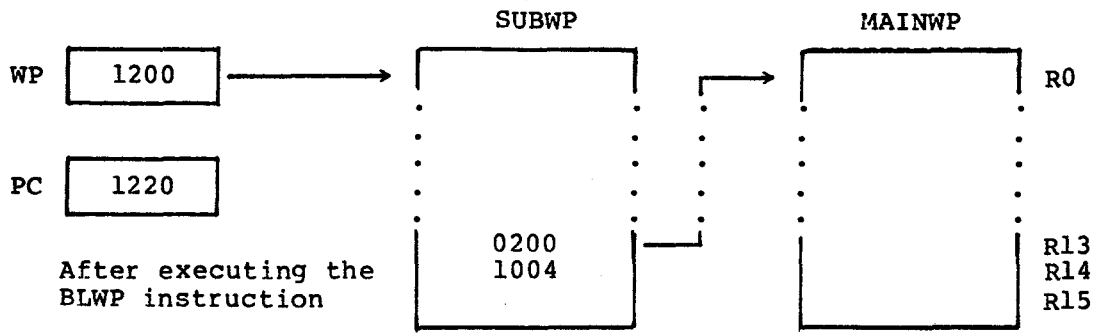
Consider the following code:

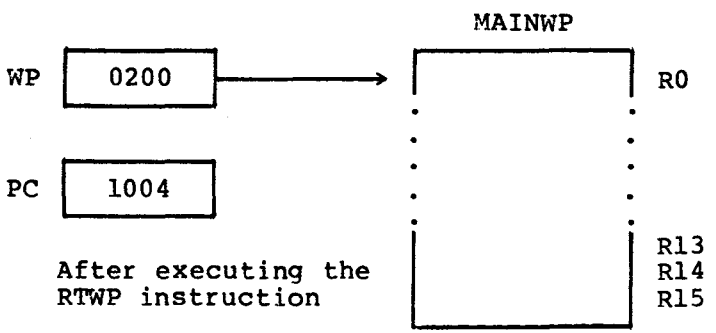
```
 Address Label    Instruction        Comment

                  AORG    >200
   0200  MAINWP   BSS     32         Define main's registers
   0220  SUBPTR   DATA    SUBWP      SUB's registers
   0222           DATA    SUB        SUB's entry point
                  .
                  .
         MAIN     EQU     $          Entry point for MAIN
                  LWPI    MAINWP     Load WP with >200
                  .
                  .
   1000           BLWP    @SUBPTR    Execute subroutine SUB
                  .
                  .
   1200  SUBWP    BSS     32         Define SUB's registers
   1220  SUB      EQU     $          Entry point for SUB
                  .
                  .
   1300           RTWP               Exit from SUB
```

The context switch is shown diagrammatically below.







6.4.7 Addressing Modes

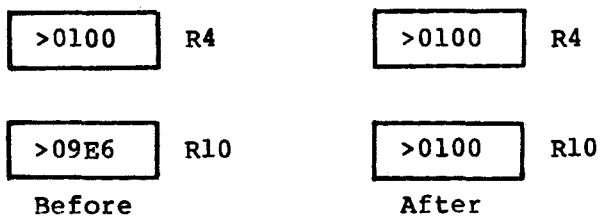
Often a programmer wants to use an instruction in slightly different ways. For example: At one point he may want an operand to be a workspace register. Later, he may want the operand to be a specified memory location, or he may want it to be a memory location the address of which is contained in a workspace register.

Implementing these different ways of accessing operands by way of a different instruction for each method is wasteful, and can easily lead to confusion. If, instead, a part of the instruction is reserved for specifying which method is to be used, a compact, but very powerful, instruction set is produced. (The method of accessing an operand is usually referred to as the addressing mode.)

The 9900 microprocessor provides five distinct addressing modes for instructions that specify a general address as an operand. Full details on these modes are available in Section 3 of the 'TMS9900 Assembly Language Programmer's Guide'. A simplified description of each of these modes is presented below.

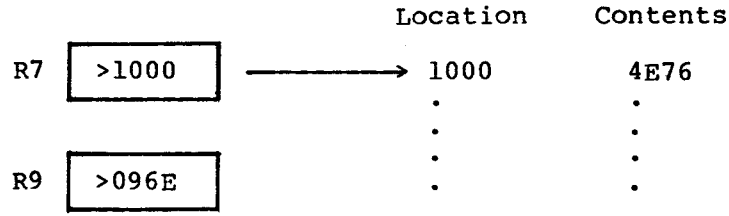
6.4.7.1 Workspace Register Addressing. This mode specifies a workspace register that contains the operand.

```
MOV   R4,R10     Copy contents of R4 into R10
```

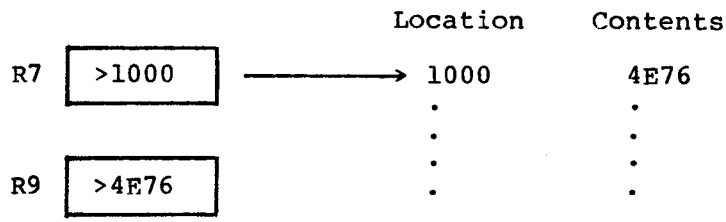


6.4.7.2 Workspace Register Indirect Addressing. This mode specifies a workspace register that contains the address of the operand. To identify this mode the workspace register is preceeded by an asterisk (*).

```
MOV    *R7,R9      Copy contents of address in R7 to R9
```



Before



After

6.4.7.3 Symbolic Memory Addressing. This mode specifies a memory address that contains the operand. To identify this mode, the memory address is preceeded by an at sign (@). (If a symbolic name such as TABLE is used, the name must be defined somewhere in the program.)

```
MOV    @TABLE,@>7C   Copy contents of the word at
                     symbolic address TABLE into
                     address >7C
```

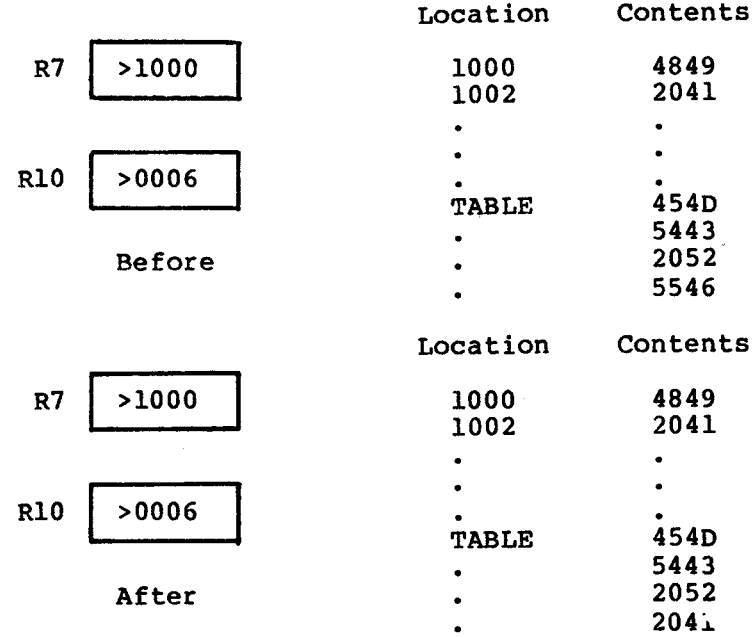
	Location	Contents
Before	007C	0471
	.	.
	.	.
	TABLE	6483

	Location	Contents
After	007C	6483
	.	.
	.	.
	TABLE	6483

6.4.7.4 Indexed Memory Addressing. This mode specifies a memory address that contains the operand. This address is the sum of the contents of a workspace register and a symbolic address. This mode is written as an address preceeded by a ' sign and followed by a

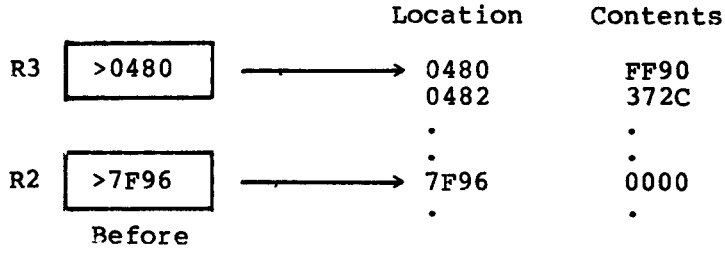
workspace register enclosed in parentheses (the index register). Register 0 can not be used as an index register.

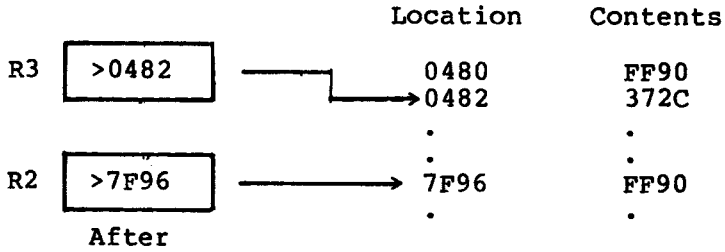
```
MOV   @2(R7),@TABLE(R10)   Copy contents of word at
                           location (2+contents of R7)
                           into location (address of
                           TABLE + contents of R10)
```



6.4.7.5 Workspace Register Indirect Autoincrement Addressing. This mode is similar to workspace register indirect addressing mode except that after obtaining the address from the workspace register, the register is incremented (by one for byte operations and two for word operations). To identify this mode, the register is preceeded by an asterisk(*) and followed by a plus sign (+).

```
MOV   *R3+,*R2   Copy contents of the word at the
                 address in R3 into the word at the
                 address in R2.  Increment R3 by 2
```





This mode is very useful for indexing through structures such as tables, where a succession of memory locations must be accessed in sequence.

6.4.8 Specialized Addressing Modes

The preceding addressing modes are all used to address variables (data) and can be used with any instruction that specifies a general memory address as its operand(s). The following three modes have more specialized applications.

6.4.8.1 Immediate Addressing. This mode is used by immediate instructions. The word immediately following the instruction contains the operand (the operand is contained in the program code). Immediate instructions that require two operands have a workspace register preceding the immediate value.

```
LWPI  >FE70     Place >FE70 in the WP
LI    R5,1000   Place 1000 in R5
```

6.4.8.2 CRU Bit Addressing. This mode is used by CRU bit instructions for performing bit I/O. The operand is a signed displacement in the range -128 to +127 bits from the CRU base address which is stored in workspace register 12. (Only bits 3 to 14 are actually used.) When the CRU is addressed the least significant bit (bit 15) of this register is not transferred onto the address bus. Because of this it is necessary to store the doubled base address in the register. Thus, if register 12 contains >80, the actual base address of the hardware accessed is only >40.

SBO 8 Sets the CRU bit, 8 greater then the base address, to one. If R12 contains >20 then CRU bit 24 will be set to one by this instruction

SBZ DTR Sets the CRU bit to zero. If DTR has the value 10, and R12 contains >40, then this instruction sets CRU bit 42 to zero

6.4.8.3 Program Counter Relative Addressing. This mode is used by the jump instructions. The operand for this mode is a symbolic address (not preceded by an 'at' sign) or a signed displacement. This addressing mode can only be used to transfer control to a location within the range of -128 to +127 words from the current location. For jumps outside this range, the branch instruction must be used (B @location).

When a symbolic address is given, the assembler performs the following:

1) Subtracts the value of the incremented PC (address of the current instruction + 2) from the symbolic address.

2) Halve the difference to arrive at the displacement in words.

To jump to symbolic location THERE, the instruction

```
JMP   THERE
```

is required. If THERE was at location >2090 and the jump instruction is at location >2060, then

```
JMP   $+>30      30 byte jump from here
```

would perform the same operation. The symbol '$' is used to represent the current value of the location counter (the address at which the instruction will be stored in memory).

6.5 SUBROUTINES

In a low-level language a subroutine, or procedure, is simply a sequence of assembly language instructions preceded by a symbolic name (a label) and terminated by a return statement.

The subroutine CLOSE can be defined by:

```
CLOSE  ....        1st instruction
```

It is in fact better to use the EQUate directive and set the subroutine name equal to the address of the subroutine (= the current

value of the location counter) in the previous line:

```
CLOSE  EQU   $
       ....        1st instruction
```

Although both approaches produce the same machine code, the second clearly indicates a subroutine's entry point and thus aids program documentation.

The Branch and Link instruction (BL) causes the address of the next instruction to be stored in workspace register 11, and then passes control to the specified routine. The operand for this instruction is the address (or the name if the symbolic memory addressing mode is used) of the required subroutine. For example, if subroutine RESET is located at memory address >2000, then either of the following may be used. (The first is much clearer.)

```
       BL   @RESET
or     BL   @>2000
```

The BL instruction provides a 'short linkage' which is best used for a small subroutine that is local to the area of the program from which it is called. A subroutine called with a BL uses the same workspace as the calling program, and so the subroutine is able to directly access the calling program's registers.

The Branch and Load Workspace Pointer instruction (BLWP) causes a context switch to take place and then transfers control to the specified subroutine. The operand for this instruction is the address of a two-word area that contains the addresses of the new workspace and of the subroutine to be executed. (When a context switch takes place the incremented PC--the address of the instruction following the BLWP--IS stored in register 14 of the new workspace.)

```
SUB    DATA    SUBWP   SUB's workspace
       DATA    SUBPC   SUB's entry point
       .
       .
       .
       BLWP    @SUB
```

If SUB is at memory address >1000, then

```
       BLWP   @>1000
```

can also be used.

A BLWP establishes a completely new context that is separate from the calling program, thus, a BLWP subroutine can be written separately from the calling program without any danger that it will inadvertently corrupt the caller's registers. The registers of the calling program can be accessed using the indexed addressing mode, because when a context switch is performed, register 13 of the new workspace automatically contains the address of the old workspace. For example,

register 5 of the old workspace can be referenced by writing:

```
@10(R13)
```

as the operand of an instruction. The indexed address is obtained by adding 10 to the contents of register 13, to arrive at an address 10 bytes offset from the start of the old workspace (in other words register 5, because each register occupies 2 bytes). BLWP is a very useful instruction for implementing modular software in assembly language (see Chapter II).

Control is returned from a subroutine either by a RTWP instruction (if the subroutine was invoked by a BLWP instruction) or the RT pseudo-instruction (if the subroutine was invoked by the BL instruction). The RTWP instruction restores the context (PC. WP and ST) of the calling program from registers 13, 14 and 15 of the new workspace. The RT pseudo-instruction translates into:

```
B   *R11
```

which is an indirect jump to the address contained in register 11 (location used by BL instruction to store old PC).

6.6 PARAMETER PASSING

All high-level languages have a built in parameter passing mechanism. When using subroutines (or procedures, in the more modern languages) the programmer must conform to their conventions.

Low-level languages, on the other hand, impose no such restrictions as all parameter passing mechanisms must be explicitly implemented by the programmer. To avoid confusion, it is important that the programmer chooses his own convention and sticks to it.

However, when low-level language routines are to be incorporated into a high-level language program, it is necessary that these routines use the conventions of the host language.

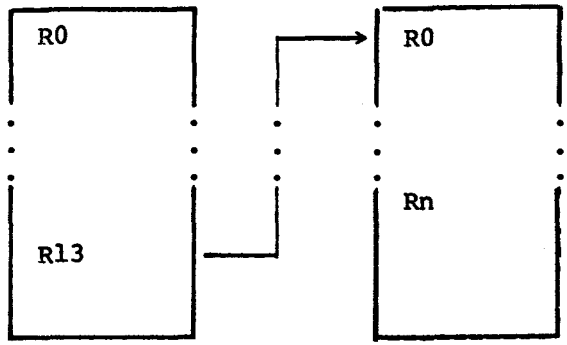
The main methods of parmeter passing and their implementation in 9900 assembly language are given below.

1) The paremeter is stored in a register

   a) Subroutine invoked by BL instruction has direct access to all the calling routine's registers.

   b) Subroutine invoked by BLWP instruction:

```
MOV   @2*n(R13),TEMP   Copy contents of calling
                       routine's workspace
                       register N into TEMP
```



Subroutine's workspace

Calling routine's workspace

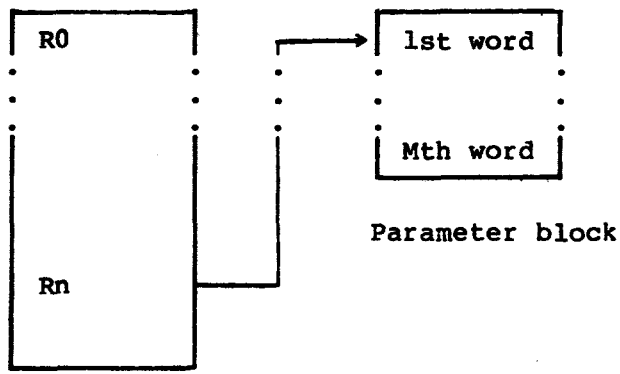
Note: The register number is doubled as byte addressing is used on the 9900.

2) The parameter is stored in an area of memory that is referenced by a register:

a) Subroutine invoked by BL instruction:

```
MOV   @2*m(Rn),TEMP
```

Copy contents of Mth word (Mth parameter) of the parameter block into TEMP

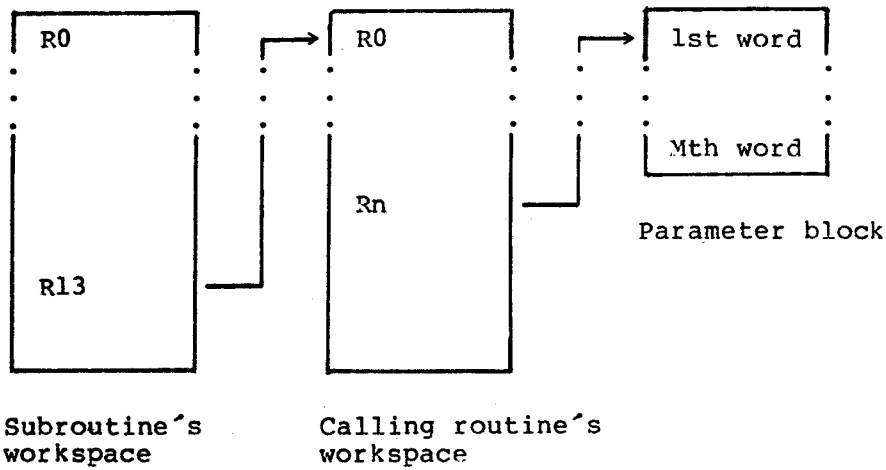


Parameter block

Calling routine's workspace

b) Subroutine invoked by BLWP instruction:

```
MOV   @2*n(R13),Rs     Copy address in calling
                       routine's workspace register
                       N into register S
MOV   @2*m(Rs),TEMP    Copy contents of Mth word of
                       parameter block into TEMP
```



3) This is a variation on the previous method. In this method, the parameter block appears in-line (it immediately follows the call). With this approach the subroutine must ensure that the return address (where control is transferred when the subroutine is exited) is updated to skip over the parameter block and pick up the instruction after the call. This can be done using the indirect autoincrement addressing mode on R11 for the BL instruction and R14 for the BLWP instruction. This approach can only be used when the data to be passed to the subroutine is constant (its value is known when the program is assembled).

a) Subroutine invoked by BL instruction:

```
        BL     @SUBR            Call SUBR
        DATA   ....             Parameter block
        .
        .
        .
SUBR    MOV    *R11+,TEMP       Store 1st parameter in TEMP,
                                update the return address in
                                register 11
        .
        .
        .
        RT                      Return
```

b) Subroutine invoked by BLWP instruction:

```
SUBADD  DATA   SUBWP            SUB's workspace
        DATA   SUBPC            SUB's entry pointer
        .
        .
        .
        BLWP   @SUBADD          Call SUB
        DATA   .....            Parameter block
        .
        .
        .
        .
SUB     MOV    *R14+,TEMP       Store 1st parameter in TEMP,
                                update the return address in
                                register 14
        .
        .
        .
        RTWP                    Return
```

Note : Invoking a subroutine is faster using the BL instruction as no context switch takes place, but there is a grave risk that data might be inadvertently lost when any of the calling routine's registers are used for temporary storage purposes.

6.7 STRUCTURING

With a high-level language, structuring presents no problem. High-level languages were designed with this in mind; structuring constructs are an integral part of the language.

However, assembly (or low-level) languages are designed around the hardware and are not considered to be problem-oriented languages. The programmer must provide the necessary structures. Turning a software design into an executable program is considerably more difficult in assembly language because problem-oriented design constructs must be translated accurately into groups of low-level machine instructions. The information that follows describes assembly language implementation of the basic sequence, and selection and iteration constructs used in software design.

In writing an assembly language program, it is effective to produce a software design before writing the code; this enables the programmer to design the application's logic before worrying about the implementation details (which, in assembly language, are considerable). This approach has been shown to lead to better and more correct software, and has been used very successfully for internal TI projects. The sequence, selection and iteration constructs (and the notation used here) are described in Chapter II.

6.7.1 Selection

Normally the action taken at a specific point in a program depends on a number of factors or conditions. If one of the conditions changes, the action to be performed changes. This choice of action is represented by the selection construct displayed below.

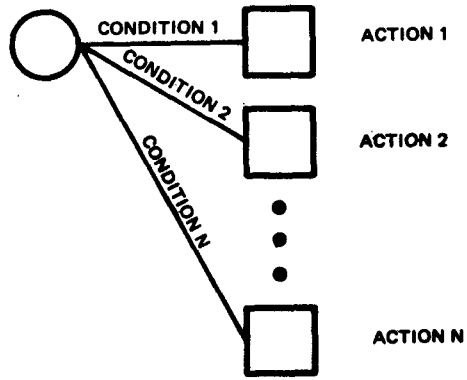
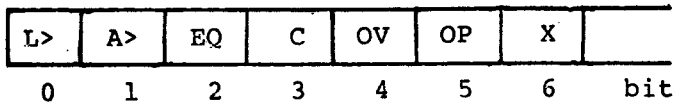


FIGURE 6-2. SELECTION CONSTRUCT

Implementing this construct at the assembly language level requires an understanding of the condition codes (or status flags). These are stored in the processor status word (on the 9900 this is a special hardware register called the status register - ST), with each flag occupying one bit.

6.7.1.1 Condition Codes.



Condition Codes for the TMS 9900 status register

- Logical Greater Than (L>) contains the result of a comparison of words or bytes as unsigned binary numbers; the sign bit is interpreted as part of the number. Thus a negative number is logically greater than a positive one. (See Paragraph 6.13.2.2 for the binary representation of negative numbers.)

- Arithmetic Greater Than (A>) contains the result of a comparison of words or bytes as twos complement numbers.

- Equal (EQ) is set when the words or bytes being compared are equal.

- Carry (C) is set by a carry out of the most significant bit of a word (or byte) during arithmetic operations. The carry bit is used by the shift operations to store the last bit shifted out of the specified workspace register.

- Overflow (OV) is set when the result of an arithmetic operation is too large or too small to be correctly stored in 16 bits. (Refer to section 2.4.5 of the 'TMS 9900 Assembly Language Programmer's Guide' for full details.)

- Odd Parity (OP) bit is set in byte operations when the parity of the result is odd, and reset when the parity is even. The parity of a byte is odd when the number of bits having a value of one is odd, and even when this number is even.

- Extended operation (X) is set when a software implemented extended operation (XOP) is initiated.

  The processor automatically sets (or resets) the appropriate status flags once it has executed an instruction. Only certain instructions affect certain flags, for example, the 'X' flag is only set by an extended operation instruction.

6.7.1.2 Jump Instructions. Perhaps the most important members of a machine's instruction set are the jump instructions. These transfer control (unconditonally or conditionally according to the state of one or more status flags) from one point in a program to another, without affecting the flags. The jump instructions available are listed below:

JMP	JOC	JEQ	JGT
JHE	JLT	JH	JL
JNE	JLE	JNC	JNO
JOP			

The conditional jump instructions (all those listed above except JMP) can be used to implement the selection construct. For example the contents of R2 (>10, =10, or <10) determine which sequence will execute (ACT1, ACT2, or ACT3 respectively). The execution of sequence ACT4 follows.

The structure diagram for this is:

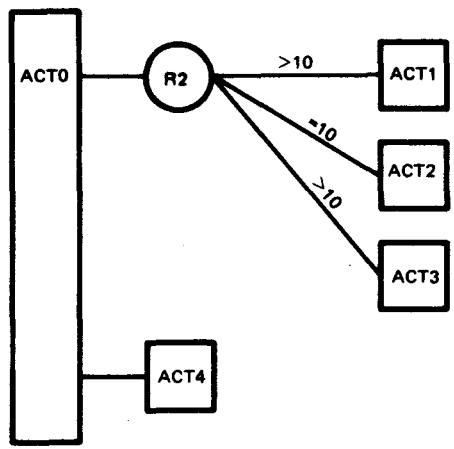


FIGURE 6-3. JUMP INSTRUCTION

In 9900 assembly language this can be coded as:

```
ACT0    EQU   $
        CI    R2,10     Compare R2 with 10
        JGT   ACT1      To ACT1 if R2 > 10
        JEQ   ACT2      To ACT2 if R2 = 10
ACT3    EQU   $         To here if R2 < 10
        .
        Code for ACT3
        .
        JMP   ACT4      To ACT4
ACT1    EQU   $
        .
        Code for ACT1
        .
        JMP   ACT4      To ACT4
ACT2    EQU   $
        .
        Code for ACT2
        .
ACT4    EQU   $
        .
        Code for ACT4
        .
```

Note : If R2 contains 10 then after executing the code for ACT2, program control drops through to the code for ACT4.

For a simple two-way selection:

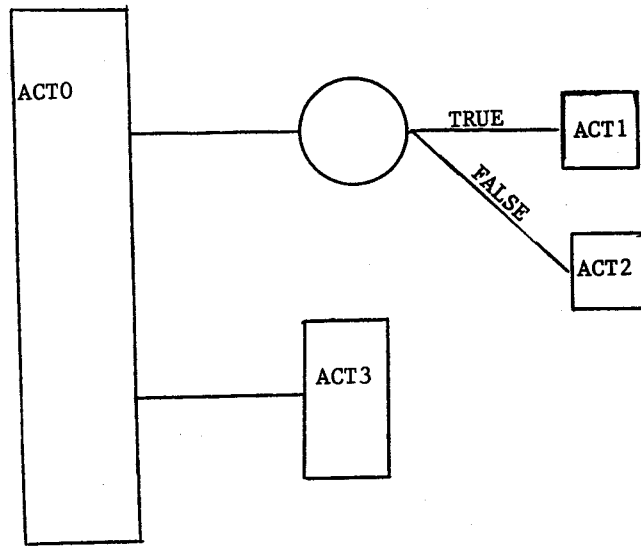


FIGURE 6-4. TWO-WAY SELECTION

The code structure for this may appear as:

```
ACT0    EQU    $
        'test'
        JNE    ACT2      To ACT2 if condition false
ACT1    EQU    $
        .
        Code for ACT1
        .
        JMP    ACT3      To ACT3
ACT2    EQU    $
        .
        Code for ACT2
        .
ACT3    EQU    $
        .
        Code for ACT3
        .
```

6.7.2 Iteration

Quite often it is necessary for a sequence of instructions to be executed a number of times. One way of implementing this repitition is to code the sequence the required number of times. However, if either the sequence to be coded and/or the repetition number is large, a large amount of memory will be used. Further, if the sequence is to be repeated until a particular condition arises, the repetition number is unknown. The use of the iteration construct overcomes these problems.

Example: a sequence (SEQ1) must be repeated N times (where N is a variable supplied by a previous stage) followed by the execution of SEQ2.

The structure diagram illustrating this follows:

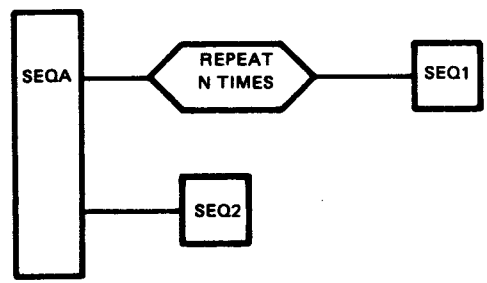


FIGURE 6-5. ITERATION

This can be coded in 9900 assembly language as:

```
SEQA    EQU   $
        MOV   @N,R0     Copy count into R0,sets flags
SEQAST  JEQ   SEQ2      To SEQ2 if R0 = 0
SEQ1    EQU   $
        .
        Code for SEQ1
        .
        DEC   R0        Decrement repetition count
        JMP   SEQAST    To SEQAST
SEQ2    EQU   $
        .
        Code for SEQ2
        .
```

If N is a constant (e.g. 20), the following is applicable:

```
        LI    R0,20     Set R0 to 20
SEQ1    EQU   $
        .
        Code for SEQ1
        .
        DEC   R0        Decrement repetition count
        JNE   SEQ1      To SEQ1 if R0 > 0
SEQ2    EQU   $         To here if R0 = 0
        .
        Code for SEQ2
        .
```

Example: While KEY=0 perform SEQ1. When KEY is changed perform SEQ2.

The structure diagram for this task follows:

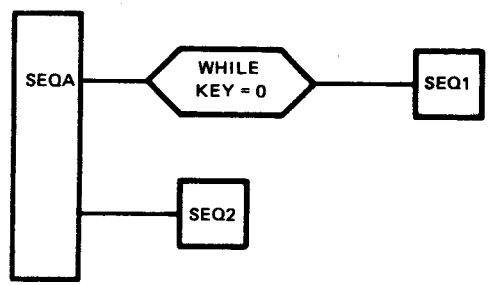


FIGURE 6-6. ITERATION AGAIN

The code representing this may be:

```
SEQA    EQU   $
        CI    KEY,0      Compare KEY with 0
        JNE   SEQ2       To SEQ2 if KEY/0
SEQ1    EQU   $          To here if KEY = 0
        .
        Code for SEQ1
        .
        JMP   SEQA       To SEQA
SEQ2    EQU   $
        .
        Code for SEQ2
        .
```

6.7.3 Sequence

On the surface, the sequence is the simplest construct to implement. The sequence represents a number of elements that are executed one after the other. At the single instruction level, assembly language programs are naturally sequential. However, when writing a program with a complex structure, some additional thought is needed to ensure that the logical flow of the program is always sequential and from top to bottom. Probably the best way to do this is to exactly follow the order in which blocks of code appear on the structure diagram.

If the program flow is not sequential but jumps backwards and forwards in an irregular manner, the program will be be difficult to follow and modify. It is important that a single block on the structure diagram be implemented as a single block of code.

This is, in fact, the easiest and the most natural way to write programs; it is certainly the easiest to follow.

Consider this structure diagram:

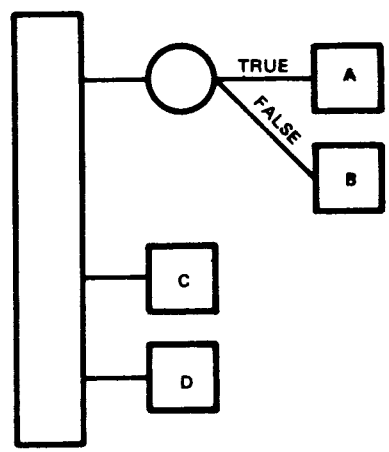


FIGURE 6-7. SEQUENCE

This may be represented in code as:

```
        'test'
        JNE   B
A       EQU   $
        .
        Code for A
        .
        JMP   C
B       EQU   $
        .
        Code for B
        .
C       EQU   $
        .
        Code for C
        .
D       EQU   $
        .
        Code for D
        .
```

```
        'test'
        JNE   B
A       EQU   $
        .
        Code for A
        .
C       EQU   $
        .
        Code for C
        .
D       EQU   $
        .
        Code for D
        .

B       EQU   $
        .
        Code for B
        .
        JMP   C
```

```
        'test'
        JNE   B
A       EQU   $
        .
        Code for A
        .
C       EQU   $
        .
        Code for C
        .
        Jmp   D
B       EQU   $
        .
        Code for B
        .
        JMP   C
D       EQU   $
        .
        Code for D
        .
```

Of the three sets of code listed above, only the first is structured according to the diagram. The other two are both less clear and less compact than the first.

If the program is not sequential, it is easy to omit a branch instruction, or even branch to the wrong location.

With a more complex structure diagram (see below), the probability of producing an incorrect program increases dramatically. This can be reduced by exactly following the diagram when writing the code.

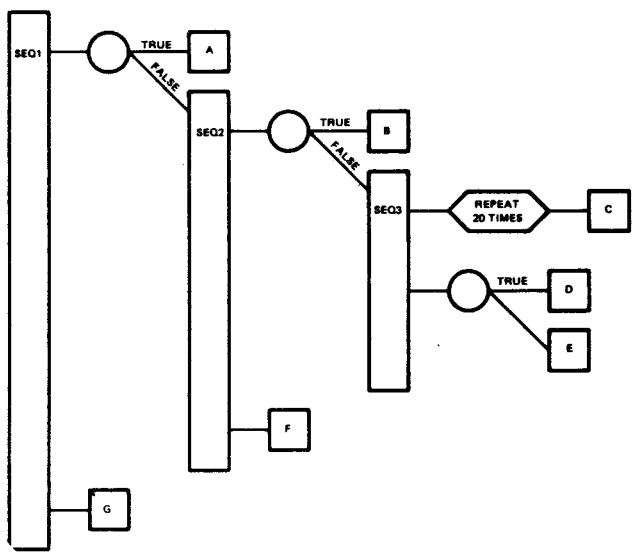
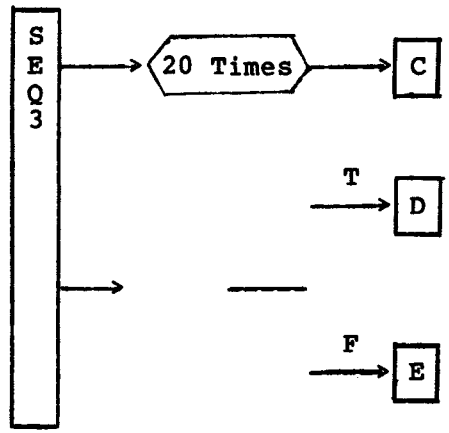


FIGURE 6-8. COMPLEX STRUCTURE

With SEQ3 defined as:



The 9900 assembly language code for this is:

```
SEQ1     'test'
         JNE   SEQ2      To SEQ2 if false
         .
         Code for A
         .
         JMP   G         To G
SEQ2     EQU   $
          'test'
         JNE   SEQ3      To SEQ3 if false
         .
         Code for B
         .
         JMP   F         To F
SEQ3     EQU   $
         LI    R0,20     Set loop count to 20
C        EQU   $
         .
         Code for C
         .
         DEC   R0        Decrement loop count
         JNE   C         To C if count > 0
          'test'         To here if count = 0
         JNE   E         To E if false
D        EQU   $
         .
         Code for D
         .
         JMP   F         To F
E        EQU   $
         .
         Code for E
         .
F        EQU   $
         .
         Code for F
         .
G        EQU   $
         .
         Code for G
```

6.8 COMMUNICATIONS REGISTER UNIT

The 9900 supplies a bit-oriented method of I/O called the Communications Register Unit (CRU). This provides a maximum of 4096 bits of read space and 4096 bits of write space. Each bit (or line) is individually addressable. Although the CRU uses the address bus to access its read and write spaces, these are totally independent from the memory address space.

The CRU transfers data along a separate three-wire bus (the wires are known as CRUIN, CRUOUT and CRUCLK).

Using the CRU, it is possible to test, set or reset a single bit anywhere in the 4096-bit address space, using a single instruction. Instructions are also provided to read and write to any group of from 1 to 16 bits.

This 'bit-picking' I/O is particularly useful for control applications, where input and output is typically single bits (sensors, switches, warning lights, relays, valves, etc.) all of which are either on or off.

The CRU was developed from Texas Instruments' experience in designing minicomputers for process control applications. It grew out of the method of I/O used, with great success, on the 960 minicomputer. As the majority of microprocessor applications involve some kind of control, this feature is very valuable.

The 9900 is the only major microprocessor to have a bit-oriented I/O structure, as well as the byte and word-oriented techniques such as memory mapping.

The five CRU instructions operate from a base address, which must be stored in workspace register 12 (R12). The contents of this register are known as the software base address. (In fact only bits 3 to 14 of this register are used to generate the address, the other bits are ignored. The value of these 12 bits is referred to as the hardware base address. The keywords 'hardware' and 'software' are used to avoid confusion when specifying the base address. The software base address is twice the hardware base address.)

The three single-bit CRU instructions use a signed displacement, from the base address, to reference a particular line. This displacement allows the instructions to access any CRU bit within a range of -128 to +127 bits from the base address.

Suppose a number of CRU operations are required around CRU line >100 and a particular instruction needs to access CRU line >120. To do this, set the hardware base address to >100 (a software base address of >200) and use a signed displacement of +32 (>20).

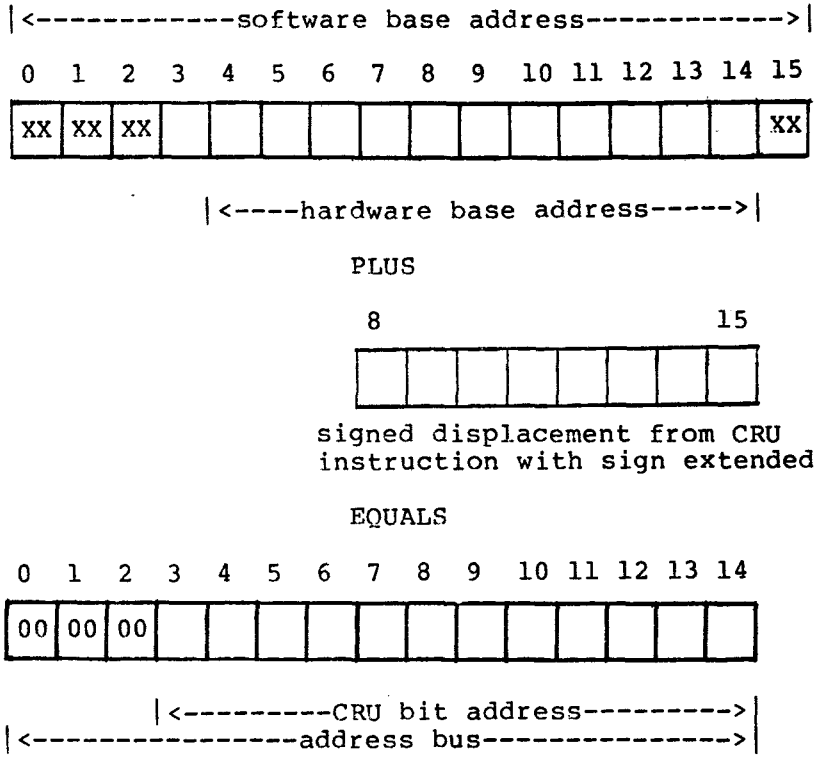
With the two multiple-bit CRU instructions, the base address must reference the first CRU line that the instruction is to access. For example, if the transfer is to start at CRU line >50 then the hardware

base address must be >50. (This is equivalent to a software base address of >A0.)

6.8.1 Single-Bit CRU Instructions

The operand of a single-bit CRU instruction is a signed displacement (in the range -128 to +127) from the base address. This specifies the particular line to be accessed.

This is displayed in the following diagram:



XX indicates that the bit is ignored
00 indicates that the bit is set to 0

SBO : Set Bit to One. This sets the specified CRU output line to a logical one.

Assume a control device is connected to CRU output line >10F. This device turns on a motor when its CRU line is set to a one. If the hardware base address is set to >100 (this corresponds to a software base address of >200) then a displacement of +15 is required. The instructions to active this motor are:

```
LI    R12,>200    Set software base address
SBO   15          Set >10F to 1
```

SBZ : Set Bit to Zero. This sets the specified CRU output line to a logical zero.

Assume that a control device is connected to CRU output line >80. This device closes a valve when its CRU line is set to zero. Also assume that workspace register 12 contains >140. To access CRU output line >80 a displacement of ->20 is required. The instruction to close the valve is:

```
        SBZ   ->20        Set >80 to 0
```

TB : Test Bit. This instruction reads the digital input and sets the equal status flag (bit 2) to the value of the bit.

Assume that workspace register 12 contains >140 (this is a hardware base address of >A0). The following lines wll test the input on CRU input line >A4 and either execute the code at location RUN (if input is a '1') or WAIT (if input is a '0').

```
        TB    4           Test CRU input line >A4
        JEQ   RUN         If on, go to RUN
WAIT    .                 If off, contine
        .
        .
        .
RUN     EQU   $
        .
        .
        .
```

6.8.2 Multiple-Bit CRU Instructions

The operands of a multiple-bit CRU operation are:

1) A general memory address. For a 'read' operation this address specifies where the input is to be stored, and for a 'write' operation from where the output is to be taken.

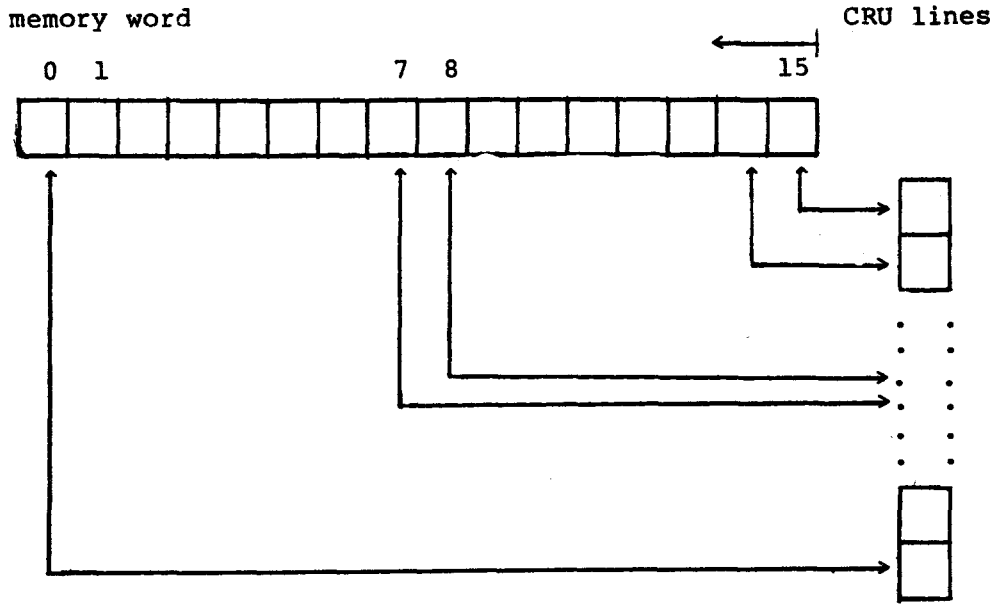
2) A count of the number of bits (in the range 0 to 15) to be transferred.

These instructions transfer from 1 to 16 bits. A 16-bit transfer is specified by setting the count to zero.

Unless otherwise explicitly stated, when less than nine bits of data is being transferred, the processor uses the most significant byte of a word for the operation. (This can be overridden by using the indirect addressing mode to reference the required byte.)

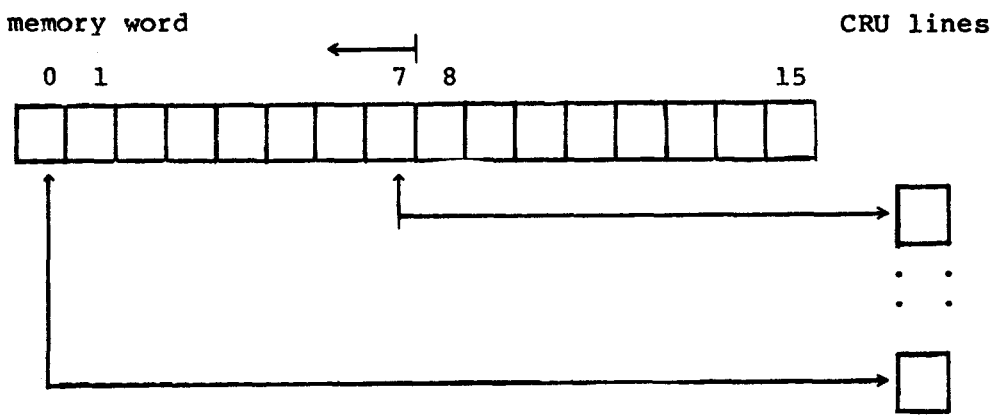
The base address for the operation is the CRU address of the first CRU line to be accessed.

For a transfer of more than 8 bits:



For example, In a transfer involving 10 bits, the data is taken from or stored in bits 15 to 6.

For a transfer of less than 9 bits:

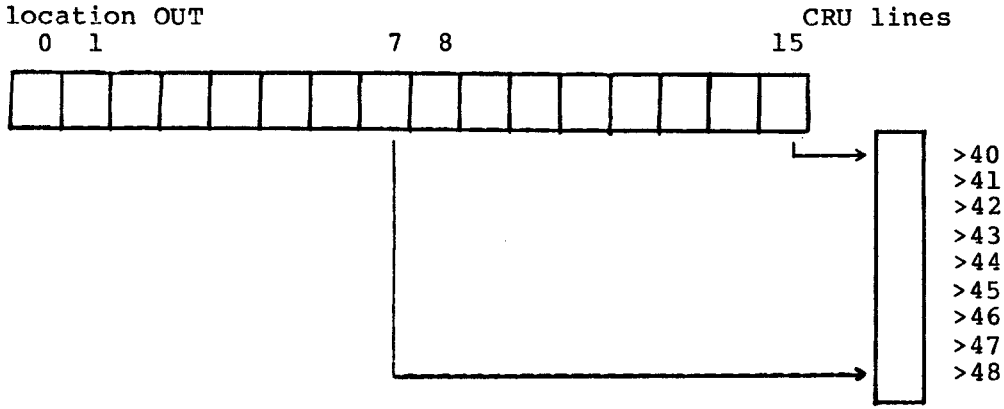


For example, In a transfer involving only 5 bits, the data is taken from, or stored in bits 7 to 3.

LDCR : LoaD Communications Register. This instruction transfers ('writes') the specified number of bits from the source operand into the CRU.

To write 9 bits from symbolic location OUT to the CRU starting at CRU output line >40, the necessary instructions are:

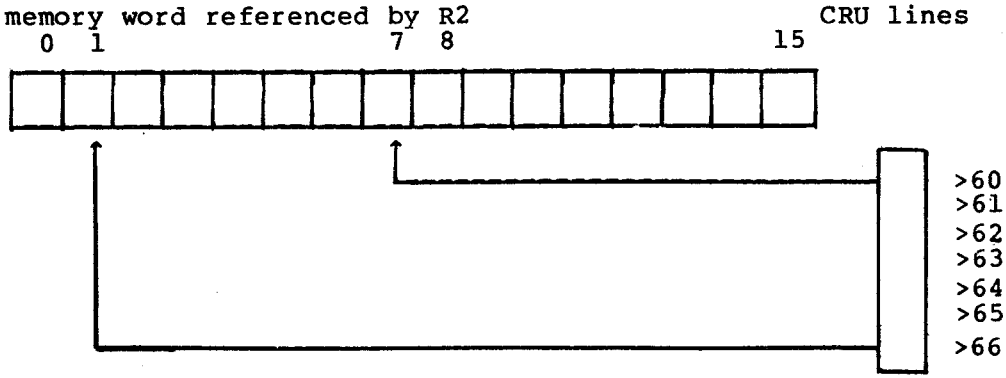
```
        LI    R12,>80       Set software base address
        LDCR  OUT,9         Output 9 bits
```



STCR : STore Communications Register. This instruction transfers ('reads') the specified number of bits from the CRU input lines into the specified memory location.

To read 7 bits, starting from CRU input line >60, into the memory location addressed by workspace register 2, the necessary instructions are:

```
        LI    R12,>C0       Set software base address
        STCR  *R2,7         Read in 7 bits
```



Note: If workspace register 2 had contained an odd address (it referenced a word's least significant byte) then the input would have been stored in bits 15 to 9.

6.9 INTERRUPTS

In a real-time system, there are two methods of determining when an external event has occured (for example, when a device connected to the computer needs to be serviced).

1) Polling - In this mode, the program polls, or tests every device known to it in a cyclic fashion. When a ready device is found, the device is immediately serviced, and the program continues its polling cycle.

   Although the program immediately services a device when it is found to be ready, there can be a delay between the time when the device generatesa ready signal and the time when the program reads the ready signal. Because of this, polling is only practical on a simple system, or when response time is not critical.

2) Interrupts - In this mode, the device signals the processor when it is ready to perform the next operation. This signal is known as an interrupt.

   With a more complex system (one that contains a number of devices) the processor is able to perform another action while waiting for an interrupt. As soon as an interrupt occurs, the processor stops what it was doing and services the device that caused the interrupt. When the device has been serviced, the processor continues the action it was performing prior to the interrupt.

6.9.1 Interrupt Structure

The 9900 supports up to 16 interrupt levels, numbered from 0 to 15. Level 0 has the highest priority; 15 the lowest. The interrupt mask, bits 12 to 15, determine which interrupts are recognized.

A device with a lower priority (higher numerical level number) than that contained in the interrupt mask is not allowed to interrupt the processor.

For example, if the interrupt mask contains '0011',only devices with an interrupt level of 0 to 3 are allowed to interrupt the processor. An interrupt from a device with a lower priority is ignored until the interrupt mask is reset to a value that is greater or equal to the device's interrupt level.

Often, instead of being coupled directly to the 9900 microprocessor, interrupt lines are connected to a TMS 9901 Programmable Systems Interface. The 9901 decides whether the interrupting device is allowed to generate interrupts; and, if so, passes the interrupt to the 9900. A device that is allowed to generate interrupts is said to be enabled. An interrupt is enabled by setting the the 9901's control bit to 0 (select interrupt mode) and then writing a 1 to the appropriate mask bit. Full details of the operation of this device are given in the TMS 9901 Programmable Systems Interface data manual.

Note : The 9901 is a CRU-DRIVEN device; before it can be accessed (using CRU instructions) its base address must be stored in workspace register 12. Further, this base address is dependent on the hardware configuration.

TABLE 6-1. INTERRUPT MASK TABLE

INTERRUPT MASK BITS				INTERRUPT LEVELS ALLOWED	MASK SET BY INTERRUPT LEVEL	
12	13	14	15			
0	0	0	0	0	0,1	HIGH PRIORITY
0	0	0	1	0,1	2	
0	0	1	0	0 -- 2	3	
0	0	1	1	0 -- 3	4	
0	1	0	0	0 -- 4	5	
0	1	0	1	0 -- 5	6	
0	1	1	0	0 -- 6	7	
0	1	1	1	0 -- 7	8	
1	0	0	0	0 -- 8	9	
1	0	0	1	0 -- 9	10	
1	0	1	0	0 -- 10	11	
1	0	1	1	0 -- 11	12	
1	1	0	0	0 -- 12	13	
1	1	0	1	0 -- 13	14	
1	1	1	0	0 -- 14	15	LOW PRIORITY
1	1	1	1	0 -- 15	--	

6.9.2 Interrupt Transfer Vector

Every interrupt level has a two-word dedicated location known as the interrupt transfer vector. A transfer vector contains:

1) The address of the workspace that is to be used by the interrupt service routine.

2) The address of the service routine's entry point.

Low-order memory, address >00 to >3F, is reserved for these transfer vectors.

TABLE 6-2. INTERRUPT TRANSFER

ADDRESS	INTERRUPT VECTOR	VECTOR CONTENTS
0000	0	WP ADDRESS FOR LEVEL 0
0002	0	PC ADDRESS FOR LEVEL 0
0004	1	WP ADDRESS FOR LEVEL 1
0006	1	PC ADDRESS FOR LEVEL 1
0008	2	WP ADDRESS FOR LEVEL 2
000A	2	PC ADDRESS FOR LEVEL 2
000C	3	WP ADDRESS FOR LEVEL 3
000E	3	PC ADDRESS FOR LEVEL 3
0010	4	WP ADDRESS FOR LEVEL 4
0012	4	PC ADDRESS FOR LEVEL 4
0014	5	WP ADDRESS FOR LEVEL 5
0016	5	PC ADDRESS FOR LEVEL 5
0018	6	WP ADDRESS FOR LEVEL 6
001A	6	PC ADDRESS FOR LEVEL 6
001C	7	WP ADDRESS FOR LEVEL 7
001E	7	PC ADDRESS FOR LEVEL 7
0020	8	WP ADDRESS FOR LEVEL 8
0022	8	PC ADDRESS FOR LEVEL 8
0024	9	WP ADDRESS FOR LEVEL 9
0026	9	PC ADDRESS FOR LEVEL 9
0028	10	WP ADDRESS FOR LEVEL 10
002A	10	PC ADDRESS FOR LEVEL 10
002C	11	WP ADDRESS FOR LEVEL 11
002E	11	PC ADDRESS FOR LEVEL 11
0030	12	WP ADDRESS FOR LEVEL 12
0032	12	PC ADDRESS FOR LEVEL 12
0034	13	WP ADDRESS FOR LEVEL 13
0036	13	PC ADDRESS FOR LEVEL 13
0038	14	WP ADDRESS FOR LEVEL 14
003A	14	PC ADDRESS FOR LEVEL 14
003C	15	WP ADDRESS FOR LEVEL 15
003E	15	PC ADDRESS FOR LEVEL 15

6.9.3 Interrupt Sequence

The level of the highest priority pending interrupt request is continually compared with the contents of the interrupt mask. When the interrupt level of the pending request is equal to or less than the mask contents, the interrupt is taken after the currently executing instruction has completed.

For example, if the processor is servicing a level 4 interrupt, only interrupts of level 3 and higher will be recognized. (This does not hold for level 0 interrupts. This is predefined as the RESET interrupt.)

To process an interrupt, a context switch takes place. The contents of the transfer vector's first word is stored in the WP register; those of the second word in the PC register. The old contents of the WP, PC and ST registers are stored in the new workspace registers 13, 14 and 15 respectively.

No additional interrupt is taken until the first instruction of the service routine has been executed. If the first instruction sets the interrupt mask to zero using

```
LIMI   0
```

(Load Interrupt Mask Immediate with zero) then further interrupts will be inhibited.

After storing the contents of the ST register, the processor decrements the incoming interrupt level by one and stores the result in the PT pt increment mask. This disables the current interrupt level leaving only higher levels enabled. (This does not happen with level 0 interrupts.)

The last instruction in the service routine must be a 'RTWP'. This causes the processor to restore the contents from workspace registers 13, 14 and 15 into the WP, PC and ST registers respectively (it restores the original environment). Control then returns to the point where the interrupt was taken.

Several interrupt lines may be combined at one level; it becomes the programmer's responsibility to determine which device generated the interrupt by polling the devices and then executing the appropriate service routine.

Any interrupt request must remain active until the interrupt is taken, and must be reset before the service routine is completed.

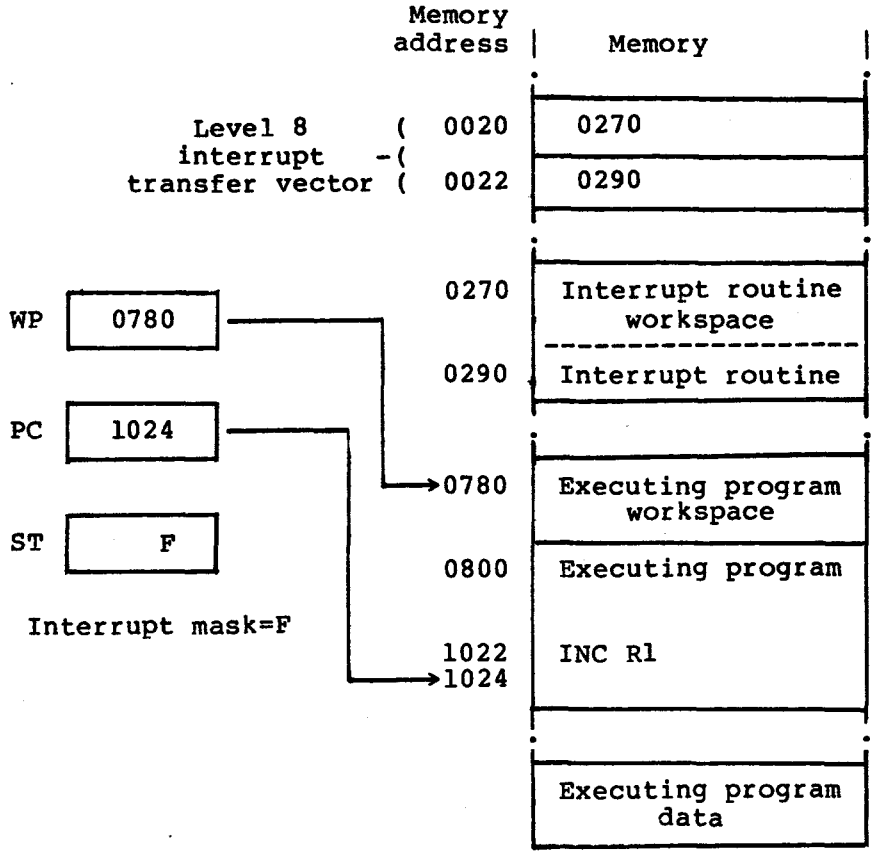


FIGURE 6-9. STATE PRIOR TO A LEVEL 8 INTERRUPT

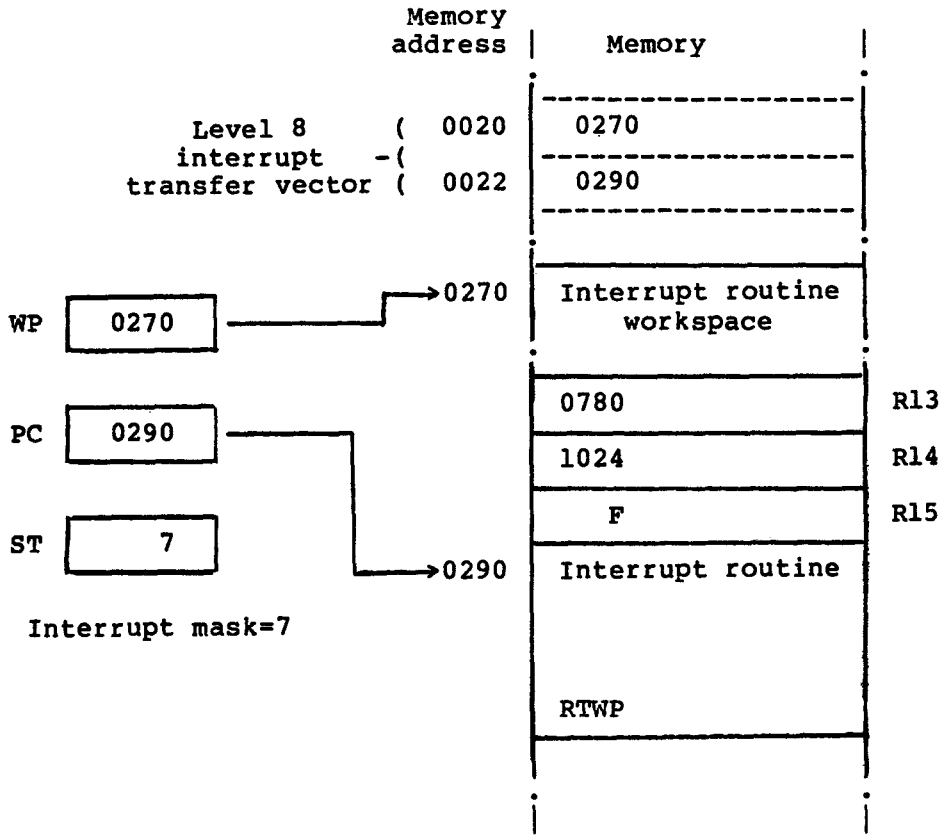


FIGURE 6-10. STATE AFTER A LEVEL 8 INTERRUPT

6.10 EXTENDED OPERATION INSTRUCTIONS

Extended operation instructions (XOPs) enable the user to extend the existing instruction set by defining instructions that are implemented by software routines.

The 9900 supports 16 extended operation instructions, numbered 0 to 15.

If the program is running under an executive, extended operation instructions are used in a slightly different way. They are used as a method of entering operating system routines that perform specific functions. These functions, in particular input/output operations, are provided by the system because it is not safe to allow their implementation by the user as the functions affect other users too easily. Extended operation instructions, used in this manner are also known as extracodes or supervisor calls (SVCS).

This type of instruction is usually referred to as a software interrupt. Software interrupts differ from hardware generated interrupts in that software interrupts have no priority sequencing. (There is no waiting to be recognized by the processor, an extended operation instruction is taken as soon as it is issued). Also, the extended operation instruction requires an operand that allows the programmer to pass a parameter over to the service routine.

6.10.1 Defining Extended Operation Instructions

The Extended Operation Instruction (XOP) is a valid assembly language mnemonic; unfortunately, it does not convey any detail about the operation a particular XOP performs. However, it is possible to assign a more meaningful mnemonic to an extended operation instruction using the Define Extended Operation (DXOP) directive. Its operands are:

1) The mnemonic by which the XOP is to be known

2) The number of the XOP involved

This directive associates the mnemonic to a particular XOP; when the mnemonic appears an an intruction's opcode, the XOP's routine is invoked. For example:

```
DXOP   CALL,4
.
.
.
CALL   @FRED
```

The first instruction associates the mnemonic CALL to XOP 4. The second instruction is an example of an extended operation instruction. The effect of this is to invoke the code for XOP 4 with

the symbolic address FRED as its parameter.

6.10.2 Extended Operation Instructions Trap Vectors

Like the hardware interrupt, the extended operation instruction has a two-word dedicated area known as a trap vector and containing:

1) The address of the workspace to be used by the XOP

2) The address of the XOP routine's entry point

These trap vectors are located at memory addresses >40 to >7F.

TABLE 6-3. XOP TRAP VECTOR TABLE

ADDRESS	XOP NUMBER	VECTOR CONTENTS
0040	0	WP ADDRESS FOR XOP 0
0042	0	PC ADDRESS FOR XOP 0
0044	1	WP ADDRESS FOR XOP 1
0046	1	PC ADDRESS FOR XOP 1
0048	2	WP ADDRESS FOR XOP 2
004A	2	PC ADDRESS FOR XOP 2
004C	3	WP ADDRESS FOR XOP 3
004E	3	PC ADDRESS FOR XOP 3
0050	4	WP ADDRESS FOR XOP 4
0052	4	PC ADDRESS FOR XOP 4
0054	5	WP ADDRESS FOR XOP 5
0056	5	PC ADDRESS FOR XOP 5
0058	6	WP ADDRESS FOR XOP 6
005A	6	PC ADDRESS FOR XOP 6
005C	7	WP ADDRESS FOR XOP 7
005E	7	PC ADDRESS FOR XOP 7
0060	8	WP ADDRESS FOR XOP 8
0062	8	PC ADDRESS FOR XOP 8
0064	9	WP ADDRESS FOR XOP 9
0066	9	PC ADDRESS FOR XOP 9
0068	10	WP ADDRESS FOR XOP 10
006A	10	PC ADDRESS FOR XOP 10
006C	11	WP ADDRESS FOR XOP 11
006E	11	PC ADDRESS FOR XOP 11
0070	12	WP ADDRESS FOR XOP 12
0072	12	PC ADDRESS FOR XOP 12
0074	13	WP ADDRESS FOR XOP 13
0076	13	PC ADDRESS FOR XOP 13
0078	14	WP ADDRESS FOR XOP 14
007A	14	PC ADDRESS FOR XOP 14
007C	15	WP ADDRESS FOR XOP 15
007E	15	PC ADDRESS FOR XOP 15

Before an extended operation instruction is executed, its trap vector must contain the appropriate values. For the CALL extended operation above:

```
AORG    >50       CALL's trap vector at >50
DATA    CALLWP    Workspace for CALL
DATA    CALLPC    Entry point for CALL
```

6.10.3 Extended Operation Instruction Execution

When an extended operation instruction is executed, the processor

1) Locates the XOP's trap vector (4 times the XOP number plus >40) and then loads the WP and PC registers with the values contained there.

2) Performs a context switch.

3) Sets bit 6 of the status register to 1 (this indicates that an extended operation instruction is being executed) if it is implemented in software.

4) Places the effective address of the instructions's operand into the new workspace register 11.

5) Passes control to the routine's entry point.

Return from an extended operation instruction is via the RTWP instruction. This restores the program environment existing before the instruction was executed.

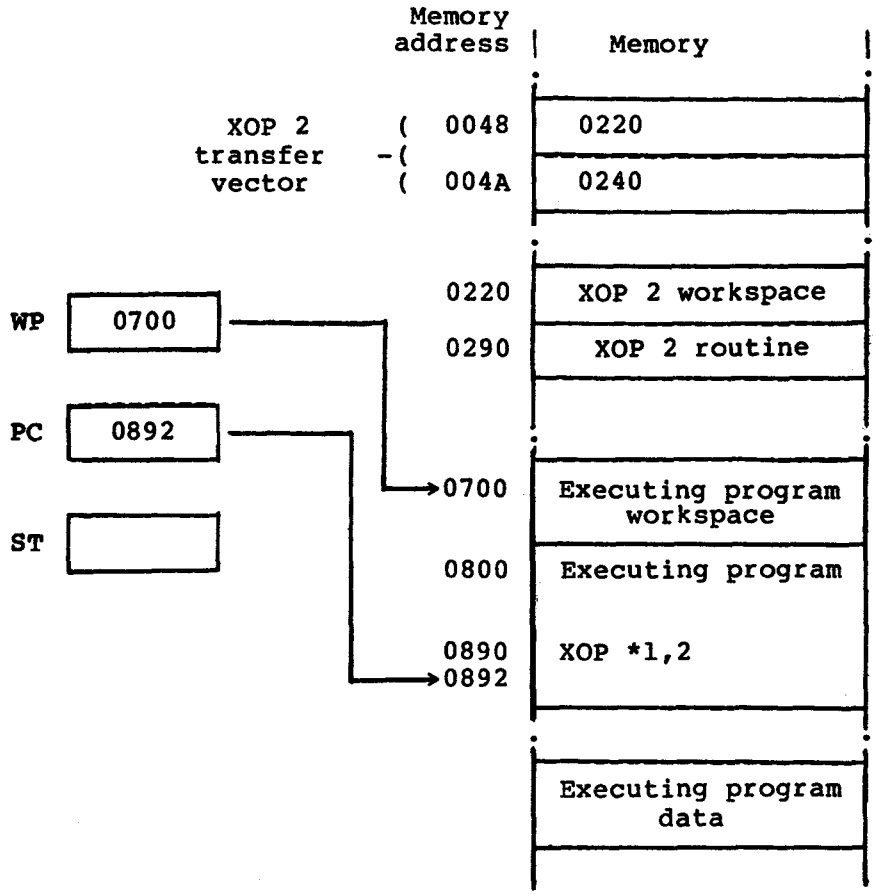


FIGURE 6-11. ISSUING AN EXTENDED OPERATION INSTRUCTION

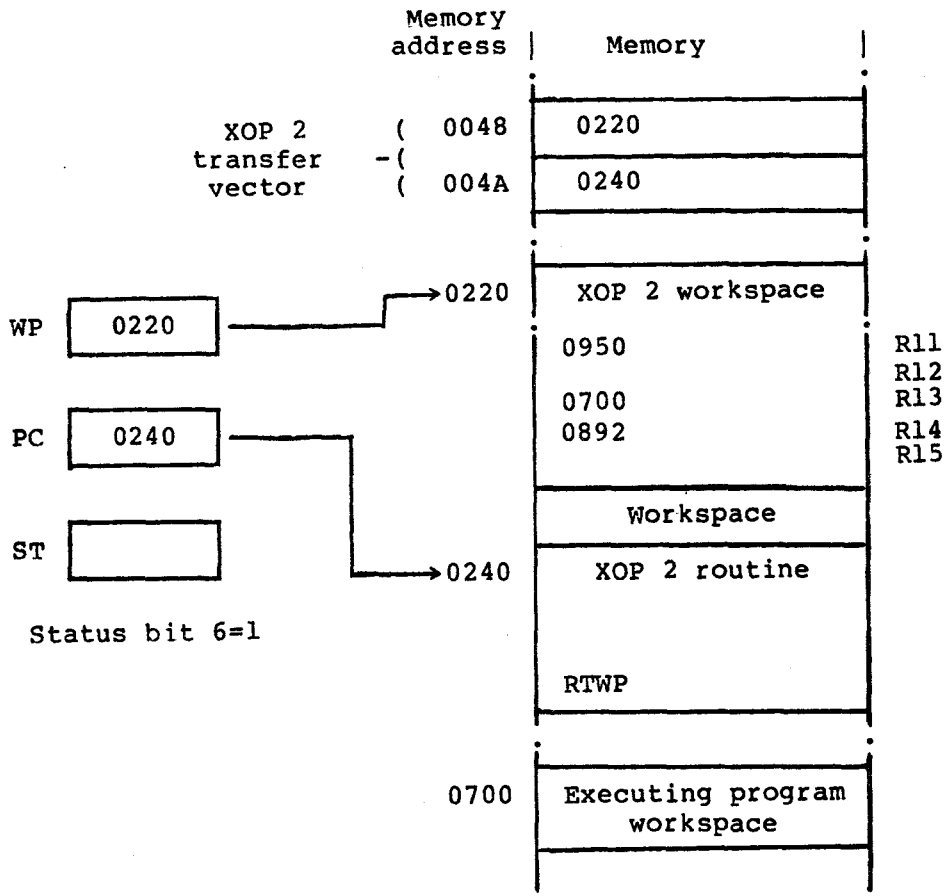


FIGURE 6-12. EXTENDED OPERATION INSTRUCTION EXECUTION

Note: Extended operation instructions can also be called using the XOP instruction. This requires two operands:

1) Source operand, as above for CALL

2) XOP number

The extended operation instruction above

```
CALL   @FRED
```

can be written as

```
XOP    @FRED,4
```

The latter code doe not require the DXOP directive to be used.

However, it is recommended that the first approach be adopted as the mnemonic indicates what the routine actually does and thus aids program readability.

6.11 ALGORITHMS AND TECHNIQUES

The paragraphs that follow provide information about algorithms and techniques that are applicable to 9900 assembley language.

6.11.1 Invoking the 9900 Family of Assemblers

Although the 9900 family of assemblers are upward compatable, there are restrictions on the use of certain instructions.

Long Distance Destination	LDD
Long Distance Source	LDS
Load Memory Map File	LMF

The above instructions are only valid on the 990/10 minicomputer with map option. The following instructions are valid for the /10 and /4 computers. Although they are not illegal for the TMS 9900 microprocessor, they do not necessarily operate as described in the Reference Section.

Clock Off	CKOF
Clock On	CKON
Idle	IDLE
Load Rom and Execute	LRE
Reset I/O	RSET

Note: A two-pass assembler reads the source program twice, maintaining a location counter as it reads the source lines. On the first pass it builds a symbol table containing the name of every symbol used in the program and the address where it was defined. During the second pass the machine code is produced using the operation codes and the completed symbol table.

6.11.1.1 LBLA. The Line-By-Line Assembler is a two-EPROM package that is used in conjunction with the TIBUG monitor supplied with the TM 990/101 and /100 microprocessor boards. With these two additional EPROMs correctly installed, the Line-By-Line assembler is entered by the following sequence:

TIBUG MONITOR PROMPTS AND REPLIES	USER REPLIES
?	R
W=XXXX	space
P=XXXX	9E8 return
?	E

Note: In some versions the address maybe 9E6.

This initializes the workspace, sets the program counter to the entry point of the assembler and begins execution.

The assembler prints the address of the first word of memory into

which the subsequent program will be stored and waits for instructions to be entered. To exit from the assembler and return to TIBUG press the escape key (ESC).

Once the program has been entered, it can be executed by performing the same sequence of commands used for entering the assembler. However, P should be set to the program's entry point instead of 9E8.

For further details refer to the TM 990/402 Line-By-Line Assembler User's Guide.

6.11.1.2 SYMBOLIC. Symbolic is a ROM resident two-pass assembler that is supplied with the TM 990/302 Software Development Board. It interprets source statements stored on audio cassette that have been created via the resident Text Editor and produces absolute (not relocatable) machine code. The first instruction in the program should be an AORG directive that sets the location counter to the absolute start address of the program. Before executing the symbolic assembler, the cassette containing the source statements must be positioned to the begining of the program. The assembler is invoked by:

```
.SA  <dev1>,<dev2>,<dev3>   return
```

where DEV1 is the device number of the cassette containing the source statements. DEV2 is the device number of the cassette where the object code is to be stored; and DEV3 is the device number of the listing device.

After the first pass, the assembler responds with:

```
** REWIND TAPE

** HIT 'CR' TO GO
```

If DEV1 and DEV2 are the same, the assembler responds with these messages following the second pass:

```
** SWAP TAPES

** HIT 'CR' TO GO
```

If the program is too large to fit into the assembler's buffer at one time, more steps will be involved.

Having stored the object code on cassette, the next step is to invoke the Relocatable Loader to load the absolute program into the board's user memory. This is performed by:

```
.RL  <dev>   return
```

where DEV is the device number of the cassette containing the object code.

The loader requires information to determine where the program is to be loaded into memory, how much of the program is to be loaded, etc. When the loader is ready for this information, it informs the user by prompting '?'.

Once loaded, the assembled program is executed by invoking the Debugger Utility and issuing the EX command (after the debugger has prompted for input):

```
.DR  return
?EX  return
```

See the TM 990/302 Software Develpoment Board User's Guide for further details.

6.11.1.3 TXMIRA. TXMIRA is a two-pass assembler that runs on a 990/4 microcomputer under the floppy disc based TXDS Control Program. The assembler is invoked by replying to the Control Program prompts as follows:

```
PROGRAM:    DSCX:TXMIRA/SYS                 return
  INPUT:    DSCX:NAME/ASM                   return
 OUTPUT:    DSCX:NAME/OBJ,DSCX:NAME/LST     return
OPTIONS:                                    return

 TXDS CONTROL   USER REPLIES
PROGRAM PROMPTS
```

DSCX:NAME/EXT is the full pathname of the file (or device) containing the program to be assembled.

During output, if a file does not exist, it will be created on the specified floppy disc with the name given. The second parameter specifies where the listing is to be sent. This is usually a device such as the line printer (LP). If this parameter is missing, the system default printer will be used.

For a full list of the available options refer to Section 5.4 of the Model 990 Computer Terminal Executive Development System (TXDS) Programmer's Guide.

The TXDS Linking Utility Program (TXLINK) must be used to resolve any external references (REFs) contained by the program.

Execution of an assembled and linked (if necessary) program is via the EX or RU commands of the TXDS Standalone Debug Monitor (TXDBUG).

6.11.1.4 SDSMAC. SDSMAC (Software Development System Macro Assembler) is a multipass macro assembler that runs on a 990/10 minicomputer under the hard disc based DX10 operating system. This assembler is invoked by issuing a XMA command to the SCI (System

Command Interpreter) prompt and then supplying the relevant information to the XMA prompts.

```
[] XMA  return

SCI PROMPT

   EXECUTE MACRO ASSEMBLER
          SOURCE ACCESS NAME:     DISC.SOURCENAME   return
          OBJECT ACCESS NAME:     DISC.OBJECTNAME   return
        LISTING ACCESS NAME:      DISC.LISTNAME     return
          ERROR ACCESS NAME:      DISC.ERRORNAME    return
                        OPTIONS:                    return
     MACRO LIBRARY PATHNAME:      DISC.LIBRARYNAME  return

      XMA COMMAND PROMPTS         USER REPLIES
```

DISC specifies the name of the (installed) disc on which the file resides. If the file does not exist prior to the command for the listing, object, and error access name prompts, it will be created on the specified disc with the name given.

DISC.xxxxNAME is the full pathname of the file (or device) to be used.

When creating a program on the /10 it is good practice to create a directory (using the CFDIR command) through which all files related to that particular program are referenced. This allows the replies to the XMA prompts to be of the form:

```
DISC.PROGNAME.EXT
```

where PROGNAME is the directory name for the program files, and EXT is one of ASM, OBJ, LST, ERR, MACRO.

When the assembly is complete it may be necessary to execute the Link Editor (XLE command) to resolve all external references to the assembled program. The assembled and linked (if necessary) program must then be installed as either a procedure, task or overlay (using the IP, IT or IO commands). This can then be executed by the XT command.

6.11.2 Number Representations

The information that follows discusses how numbers are internally treated by the computer.

6.11.2.1 Number Systems

A number in the decimal, base 10, system is composed of the digits 0 - 9. Numbers greater than 9 are represented using the decimal place convention. The value of each place is ten times that of the place to its immediate right.

For example, the decimal number 2976 means

$$2*10^3 + 9*10^2 + 7*10^1 + 6*10^0$$

Note : 10^0 is equal to 1

While the decimal system is the most frequently used number system it is not suitable for use on a computer.

The smallest unit of storage in a computer is the bit (from Binary digIT). The bit can be thought of as a single wire that can only be in one of two states: on or off, 'high' or 'low', '1' or '0'. The binary system automatically lends itself to this.

A number in the binary, base 2, system uses only the digits 0 and 1. The value of each place, in the binary place convention, is twice that of the place to its immediate right (as opposed to 10 in the decimal system).

For example, the binary number 1011101 (93 decimal) means

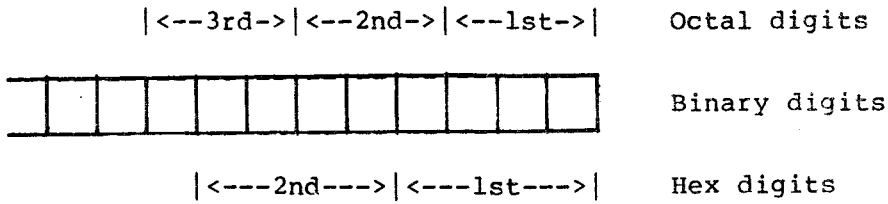
$$1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$$

Note: 2^0 is equal to 1

Writing large numbers in their binary representation is too cumbersome for most applications. However, it is possible to group bits together and represent each group by a single digit. This gives rise to the octal and hexidecimal number systems.

Octal, base 8, representation uses the digits 0 - 7. An octal digit corresponds exactly to 3 bits.

Hexadecimal (or hex for short) notation, base 16, uses the digits 0 - 9 plus A - F to represent the decimal values 10 - 15. Each hex digit corresponds to exactly 4 bits.



1001111111011010

1001 1111 1101 1010

9 f d a

BINARY	OCTAL	DECIMAL	HEX
10	2	2	2
1000	10	8	8
1010	12	10	a
10000	20	16	10
11111111	377	255	ff

Note: Ten does not correspond to an integral power of two. Therefore conversion from decimal to binary (and vice versa) is more difficult.

6.11.2.2 Representation of Negative Numbers. Negative numbers are stored in two's complement form. In this form, the most significant bit of a word (bit 0) indicates the sign of the number. If it contains a 0, the number is positive; if it contains a 1, its negative. The remaining 15 bits (bits 1 - 15) hold the two's complement value of the number. For a positive number this is simply the binary representation of that number.

The representation of a negative number however, (for example 1096) is derived as follows:

1) Take the magnitude of the number, in this case 1096, and write it in binary, using the full word length of the machine. (For the TMS 9900 microprocessor this is 16 bits.)

1096 ⟶ 0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 0

2) Take the one's complement of this number (change the state of each bit; replace 0's with 1's and 1's with 0's).

1 1 1 1 1 0 1 1 1 0 1 1 0 1 1 1

3) Add 1 to the least significant bit.

```
1 1 1 1 1 0 1 1 1 0 1 1 0 1 1 1
                                +1
-------------------------------
1 1 1 1 1 0 1 1 1 0 1 1 1 0 0 0
```

The positive number 1096 is stored as >0448 while the negative number -1096 is stored as >FBB8.

6.11.2.3 Representation of Fractions. The general equation to convert a binary fraction into its decimal equivalent is:

$$0.d1\ d2\ \ldots\ dn = d1*2^{-1} + d2*2^{-2} + \ldots + dn*2^{-n}$$

where d1 ..... dn represent binary digits

for example, the binary fraction 0.1001 is equivalent to

$$1*2^{-1} + 0*2^{-2} + 0*2^{-3} + 1*2^{-4}$$

$$= 0.5 + 0 + 0 + 0.0625$$

$$= 0.5625$$

To convert a decimal fraction to its approximate binary equivalent, multiply the decimal fraction continually by 2, saving the integer part of the result (either '0' or '1') until the result is zero. Unfortunately it is not always possible to produce an exact binary representation.

Consider the number 0.8125.

```
0.8125      0.6250      0.2500      0.5000
   *2          *2          *2          *2
---------   ---------   ---------   ---------
1.6250      1.2500      0.5000      1.0000
```

This number can be accurately expressed as 0.1101.

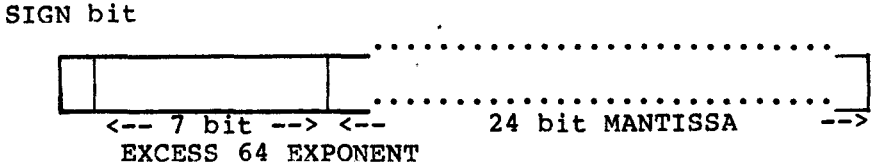
Now consider the number 0.9725.

```
0.9725      0.9450      0.8900      0.7800      0.5600
   *2          *2          *2          *2          *2
---------   ---------   ---------   ---------   ---------
1.9450      1.8900      1.7800      1.5600      1.1200

0.1200      0.2400      0.4800      0.9600
   *2          *2          *2          *2
---------   ---------   ---------   ---------
0.2400      0.4800      0.9600      1.9200
```

We could continue this process indefinately, but there is little point to it as the number 0.9725 can not be accurately represented in binary. After 9 iterations the binary approximation to the number is 0.111110001. This yields the number 0.970703125; an error of 0.001796875. Obviously the error can be reduced further by performing several more iterations. However, there are practical limitations to how far this can be taken.

6.11.2.4 Representation of Floating Point Numbers. Floating point numbers can be stored in two consecutive 9900 memory words using Excess 64 notation. The 32-bit real word is formed as follows:



A real number is converted into the form 'fraction*exponent'. The fraction is stored in the 24-bit mantissa in true form and not two's complement. The sign bit is used to indicate whether the number is positive or negative. The most significant hex digit of the mantissa must be normalized (it must contain a value other than zero). This is performed simply by shifting the four places to the left (one hex digit) and decrementing the exponent value by one until the mantissa is normalized.

Excess 64 notation means that when the value of the exponent lies between 127 and 64 the value is decremented by 64 to give a true exponent range of +63 to 0. The values 0 to 63 are used to represent the true exponent range of -1 to -64.

Consider the number -107.5

In true binary this is
01101011.1000 — 107.5
In 'fraction*exponent' form this is
$0.0110101110000 * 16^{2}$ — $0.41992188 * 16^{2}$
The fraction is already normalised
In floating point format this is
1 1000010 0110101110000....0
(Sign=negative,Exponent=+2)

The number -107.5 would be stored as >C26B8000

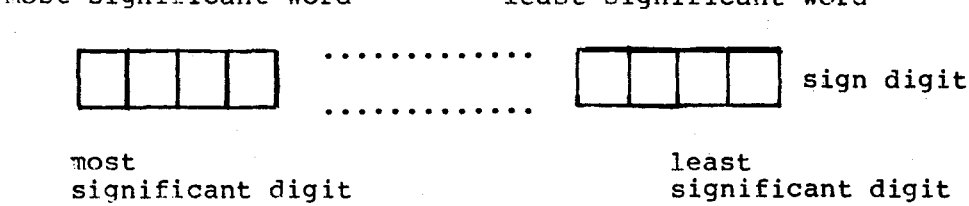
Now consider the number 0.03125

```
In true binary this is
0.00001000  — 0.03125
In 'fraction*exponent' form this is
                0                   0
0.000010 * 16    — 0.03125 * 16
Normalizing the fraction gives
0.1000 * 16**-1  — 0.5 * 16**-1
In floating point format this is
0 0000000 1000000000000....0
(Sign=positive,Exponent=-1)
```

The number 0.03125 would be stored as >00800000

6.11.2.5 Binary Coded Decimal. A number that is stored in a decimal form is said to be in Binary Coded Decimal notation (BCD). In this form. a word holds four decimal digits with each digit occupying four bits. For numbers greater than 9999, more than one word is required to store the BCD value.

If signed numbers are allowed, the user must decide on some convention for indicating whether a number is positive or negative (such as using the least significant four bits of the least significant word to contain the sign).



BCD is not supported by the 9900 instruction set (all arithmetic operations are performed on two's complement numbers). Thus it is necessary for the user to supply the appropriate BCD operations, such as add and subtract, as well as the routines to convert a number from two's complement to BCD notation and vice versa.

6.11.3 Position Independent Code

A program is normally assembled and linked to produce an executable object module that is designed to reside at a particular position in memory. Jump addresses, data references, etc. are directed to this portion of code, and normally the program will not execute correctly in any other position. However, it is possible to write a module so that it will execute at any position in memory. (This is different from relocatable code, which is not directly executable until it has gone through a location step to resolve all addresses tagged relocatable into absolute form. It is then no longer relocatable.)

Writing a program in Position Independent Code makes the program fully portable and at the same time directly executable. This is achieved by referencing relocatable addresses. For example:

```
        BL   @SUB
```

(where SUB is in a relocatable code segment) using the indexed mode of addressing on a displacement from the program's entry point. The first instructions cause the actual address of the entry point to be stored in the indexing register.

```
ENTRY EQU   $           ENTRY EQU   $
      .                       .
      .                       .
      .                       .
      BL    @SUB              BL    @SUB-ENTRY(R4)
      .                       .
      .                       .
SUB   EQU   $           SUB   EQU   $
      .                       .

RELOCATABLE CODE        POSITION INDEPENDENT CODE
```

In the above example, workspace register 4 (R4) contains the actual address of ENTRY. Obtaining this is performed by:

```
START EQU   $
      LI    R10,>045B   Load R10 with RT instruction
      BL    R10          Execute instruction in R10
ENTRY EQU   $            R11 contains add of entry
      MOV   R11,R4      R4  contains add of entry
```

6.11.4 ROM/RAM Systems

Before burning a program into ROM (the usual course of events for a microprocessor based application/control program), it is necessary to separate the variable data and temporary storage locations from the constant data and program instructions, and then add instructions to the program to ensure that all the variable data is correctly initialized.

The simplest way of initializing data is by using the DATA, BYTE, and TEXT assembler directives:

```
TEMP1   DATA    100
TEMP2   DATA    25
        .
        .
MSG     TEXT    'READY'
        BYTE     >D,>A,0
```

While this will work in a RAM environment such as a development system where the program is loaded prior to each execution, it will not work in a dedicated microcomputer. There will be no operating system to load the progam and initialize the data. If the data is placed in RAM, it will never be initialized; if in ROM, it cannot be changed by the program (this is perfectly all right for constants). Even in a RAM environment, if the program is restarted without reloading, the data will not be reinitialized.

The only way of ensuring variables are correctly initialized is to include instructions in the program code to do the initialization. This can be performed by:

- Data storage allocation in RAM

```
TEMP1   BSS     2
        .
        .
MSG     BSS     8
        .
VAREND  BSS     0
```

- Initial variable values in ROM

```
VALUES  DATA    100
        DATA    25
        .
        .
        TEXT    'READY'
```

- Inialization loop

```
ENTRY   EQU   $
        LI    R1,TEMP1       R1 pt TEMP1
        LI    R2,VALUES      R2 pt VALUE
INIT    MOV   *R2+,*R1+      Load initial values
        CI    R1,VAREND      Done?
        JNE   @INIT          To INIT if no
        .
        .
```

The label VAREND (no storage space is allocated to it) is used to delimit the block of data; its address is used to terminate the initialization loop INIT.

The initialization can also be performed by:

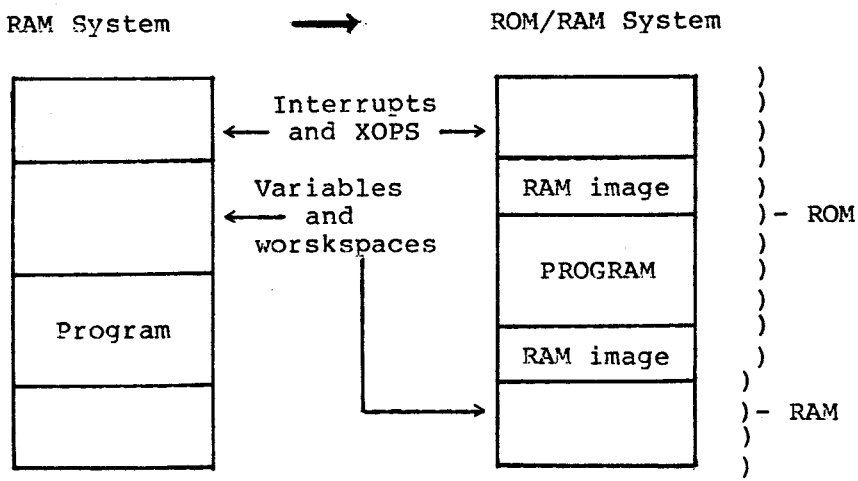
```
        LI    R1,100         Set TEMP1=100
        MOV   R1,@TEMP1
        LI    R1,25          Set TEMP2=25
        MOV   R1,@TEMP2
        .
        .
```

The above does not make use of the table of values (VALUES).

```
        MOV   @VALUES,@TEMP1    Set TEMP1=100
        MOV   @VALUES+2,@TEMP2  Set TEMP2=25
        .
        .
```

Although both of these methods are simple and straightforward, they can be more costly in memory space (they both require 4 words of ROM for each variable) for programs with a number of variables to be initialized.

Pictorally, the process can be represented as:



As shown in the diagram, at run time, the RAM image (held in ROM) is copied into the appropriate RAM storage area.

Note : A complete ROM/RAM system must contain

1) All interrupt vectors. If any interrupt level is not used then a spurious interrupt handler should be written and included in the system. All unused interrupt levels should set their WP and PC to access this routine.

2) If any XOPs are used then the appropriate XOP trap vectors must be included.

3) LOAD vector at >FFFC.

6.11.5 Macro Processing

Suppose a sequence of source lines will be used often in a program. There are several methods to accomplish this:

1) Explicitly write the sequence wherever it is to appear.

2) Make a subroutine out of the sequence and code subroutine calls wherever the sequence should appear.

3) Write the sequence at the begining of the program, associating a name with it. Insert this name wherever the sequence is to appear in the program and pass the program through a special program called a macro processor. The output from this is a program in which every occurrence of the sequence name is replaced by the sequence of source lines.

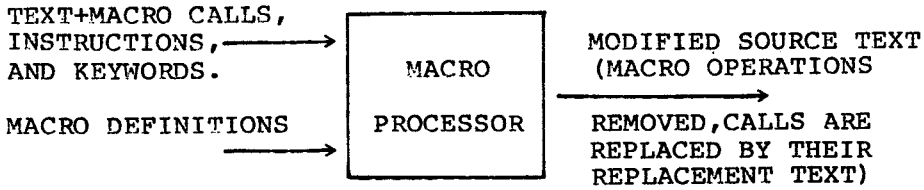
The following text is only concerned with the last method described above. The sequence of source lines is a macro. Associating a name to a macro is called macro definition and writing this name in a source line is known as a macro call.

Like the subroutine, macros can have parameters. Macro calls may require text that has approximately the same form as subroutine calls. For example, some instructions may use different operands. This can be handled by defining parameters for the macro. The actual operands required are then specified in the macro call (an example is presented below).

A macro processor processes text. This text may in fact be a program; to the macro processor however it is simply text. The macro processor is only concerned with macro related operations. Source lines that do not contain such operations are unchanged as output. Input to a macro processor is text containing macro definitions, macro calls, macro instructions and macro keywords. Output is text that has had all the macro calls replaced by the appropriate replacement text; all other

macro operations are removed).

Diagrammatically, this can be expressed as:



A macro processor has two phases:

1) Macro definition - A macro is defined and subsequently included into its macro library.

2) Macro expansion - A macro operation is found in the source text. A macro call causes the input to be 'switched' to the macro's replacement text. Processing continues from there until this text is exhausted. Other macro operations cause the macro processor to perform the necessary, inbuilt, operation.

The benefit of using a macro processor is that, once defined, a macro can be called from anywhere within the source (or replacement) text, with each call having specific arguments. Obviously, it is a good idea to build up a macro library (containing both special and general purpose macros). This can then be either automatically accessed when the macro processor is used or actually included into the macro processor itself.

Although a macro is only written once, the output from a macro processor will contain the replacement text wherever a macro was called in the source text. Note that although a macro call and a subroutine call look similar when written in a source program, a subroutine call is implemented in the object module by a short calling sequence to the subroutine, which only appears once. Wherever a macro call is written, the complete code sequence specified in the macro definition will be placed in the object module at the point of the call.

The SDSMAC assembler supports a macro language (therefore it's a macro assembler). A short description of defining and calling a macro under this assembler follows. Full details of the SDSMAC assembler capabilities are available in Section 7 of the TMS9900 Assembly Language Programmer's Guide.

6.11.5.1 Macro Definition. Macro definition is performed by the $MACRO instruction. All source lines following this instruction up to but excluding the definition terminator ($END macro instruction) constitute a macro.

```
Mname   $MACRO   parm
        .              )
        .              )-  Macro
        .              )
        $END
```

where: MNAME is the name of the macro PARM is the list of parameters, separated by commas, that the macro uses

$MACRO causes MNAME and its attributes to be stored in the assembler's symbol table. A similar table, the parameter table, is used to hold the names of the individual parameters and their attributes. (Information about any macro variables used within a program is also stored in this table.) $END informs the assembler that the definition is complete. All the source lines between these two macro instructions are stored in an encoded form in a macro file.

6.11.5.2 Macro Call. A macro is called by writing its name in the opcode field of an instruction, with the actual parameters written in the operand field.

When this is done, the actual parameters are linked to the dummy ones (those supplied at definition time) in the parameter table and then macro expansion takes place. The lines output from the macro expander are then passed straight to the assembler.

For example, to define a macro (AGAIN) with dummy parameters AD and NOW, the following lines are required:

```
AGAIN   $MACRO    AD,NOW
        .     )
        .     )-  Macro's replacement lines
        .     )
        $END
```

To call this with real parameters R4, *6 the following code is required:

```
        AGAIN     R4,*6
```

SDSMAC supports conditional assembly through the $IF, $ELSE and $ENDI macro instructions. The general form for conditional assembly is:

```
$IF     expression
.
.   Block A
.
$ELSE
.
.   Block B
.
$ENDIF
```

If the expression in the above example is true, Block A is included in the program; if not, Block B is included.

A simplified form of this follows:

```
$IF     expression
.
.   Block A
.
$ENDIF
```

Unlike most macro processors, SDSMAC allows the programmer to directly access and modify the individual components of each entry in the parameter table. Thus 'expression' can be:

P2.S='WORD'	Is the string component of variable P2 equal to the string WORD
T.L=5	Is the length component of variable T equal to 5

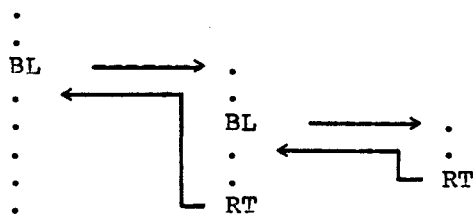
SDSMAC also supplies a number of keywords such as $PCALL (paremeter appears as a macro instruction operand) and $PIND (parameter is an indirect workspace register address) that enable the programmer to test a variable's attribute component. These keywords are used with the logical operators AND ('&'), OR ('++'), Exclusive OR ('&&') and NOT ('#'). For example:

P2.A & $PCALL	This expression has a non zero value when the variable P2 is a parameter supplied in a macro instruction. Otherwise the value is zero.

6.11.6 Nested Subroutines

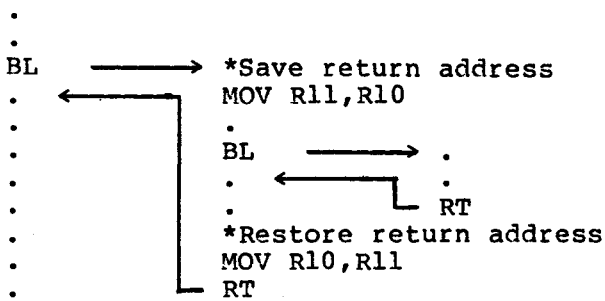
A subroutine is nested when it is invoked by another subroutine. The only problem with nested subroutine calls is that of ensuring that a subroutine's return address is not lost or overwritten. This is particularly troublesome if the subroutines are called via a BL instruction (the return address is stored in workspace register 11).

Conceptually the flow of control is as follows:



Executing the second BL instruction results in the loss of the first return address. Exiting the inner routine causes the continuous execution of the code located between the BL and RT instructions.

One approach to resolve this is:



The two instructions:

```
   MOV    R10,R11
   RT
```

can be replaced by:

```
   BL     *R10
```

6.11.7 Stacks

Another way of performing this saving and restoring of return addresses is by implementing a stack mechanism. In this, an area of memory is set aside as a stack. A stack usually starts at a high address and builds down towards low memory as items are added, (pushed onto the stack).

A register is reserved to point to the current top of stack; it points to the last item added to the stack. This register is usually referred to as the stack pointer.

The first instruction in a subroutine pushes the return address onto the stack and decrements the stack pointer. The last instruction, prior to a return, pops (or removes) the last entry from the stack, updating the stack pointer in the process.

```
SUB   PUSH  R11
      .
      .
      .
      POP   R11
      RT
```

PUSH and POP are not recognized assembly language instructions. If SDSMAC is available,these operations can be implemented by macros.

The reason for giving both PUSH and POP arguments (R11) is to make the stack operations general purpose, thus allowing data other than return addresses to be stored on the stack. However, if the stack is used in this way, care must be taken to ensure that all such items are removed before popping the return address.

PUSH and POP may be defined as macros as follows:

```
PUSH  $MACRO  OP            ;Define macro PUSH
        DECT   R10           ;Decrement stack pointer
        MOV    :OP.S:,*R10   ;Move data onto stack
       $END    PUSH
```

```
POP    $MACRO  SO              ;Define macro POP
        MOV     *R10+,:SO.S:  ;Move data from stack
       $END     POP
```

Workspace register 10 (R10) is used above as the stack pointer; the macro operands may be any valid operand for a MOV instruction.

Before the stack can be used, the stack pointer must be initialized to the address of the top of the stack 55us two; otherwise the first word in the stack will not be used.

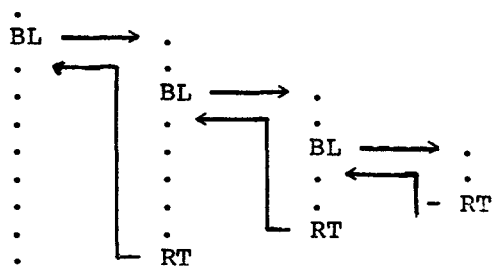
6.11.8 Automatic Workspace Allocation

Transparent stacking of workspaces is achieved by calling all subroutines through an XOP named CALL. Return from any subroutine is via a normal RTWP instruction. Arguments may be passed by standard register conventions. The stack builds down through memory and will be N*32 bytes deep, where N is the nesting level. An example follows.

```
* EXAMPLE OF USE
XOPWP   EQU  >FF00            ;ASSIGN WSP
TPSTCK  EQU  >FEC0            ;ASSIGN TOP OF STACK
        .
        .
        AORG >78              ;XOP VECTOR
        DATA XOPWP            ;XOP WORKSPACE
        DATA CALLPC           ;XOP ENTRY POINT
        .
        .
        AORG >80              ;ARBITRARY START
MAIN    LWPI TPSTCK           ;SET TOP OF STACK
        DXOP CALL,14          ;DEFINE XOP CALL
        .
        .
        CALL @SUBR            ;CALLS SUBR
        .
        .
        .
SUBR    EQU  $                ;SUB'S ENTRY POINT
        .
        .
        .
        RTWP                  ;NORMAL RETURN
        .
        .
* CALL XOP
* THIS ROUTINE AUTOMATICALLY STACKS WORKSPACES DOWN
* THROUGH MEMORY. RTWP WILL RETURN TO THE CALLER
* WITH THE OLD WORKSPACE, EFFECTIVELY POPPING THE
* STACK
CALLPC  LIMI 0                ;NON INTERRUPTABLE
        LI   R1,-6            ;OFFSET TO NEW WSP'S R13
        A    R13,R1           ;PT TO NEW WSP'S R13
        MOV  R13,*R1+         ;MOVE RETURN WP
        MOV  R14,*R1+         ;MOVE RETURN PC
        MOV  R15,*R1+         ;MOVE RETURN ST
        MOV  R11,R14          ;SUBROUTINE'S ENTRY POINT
        AI   R13,-32          ;HIT NEXT WORKSPACE
        RTWP                  ;CALL SUBROUTINE
* THIS XOP REQUIRES 148 CYCLES TO EXECUTE
* AT 3MHZ THIS IS 48.84 MICROSECONDS
```

6.11.9 Recursion

A nested subroutine has been defined as one that is called by another subroutine:



Recursion is not unusual; however, care must be taken to ensure that no return addresses are lost; otherwise the flow of control will not be as expected.

In the definition above there is nothing to stop the nested subroutine from being the same as the calling subroutine. If this is the case, the subroutine is known as a recursive subroutine (a subroutine that calls itself) and the mechanism is known as recursion. Care must be taken to ensure that a recursive subroutine does get caught up into an endless recursion loop.

Recursion presents problems. For example, how is a subroutine's return address to be saved? Obviously, simply copying it into another workspace register will not work, as on the next recursive call the value will be overwritten by the new return address. Here a stack mechanism is essential. By pushing the return addresses onto a stack the problem is solved, as long as the storage space allocated to the stack is not exceeded.

Suppose, in a multiple-user environment, a number of programs need to perform the same operation. The code performing this can be included in each program, or it could be written in such a way that it is possible for the programs to share a single copy of the code and execute it (simultaneously, if necessary) as though each program had its own copy. Code written to allow this is known as re-entrant code.

A recursive subroutine must be written in this way as, in effect, it shares the code with itself.

6.11.10 Re-entrancy

There are two problems associated with re-entrancy:

1) The subroutine code must not modify itself. Modifying code is dangerous, is difficult to debug and is discouraged. Storing the code in ROM elimiantes re-entrancy. If self modifying code is include, the program will not work as expected.

2) On entry to the subroutine, the data local to the subroutine must be correctly initialized. This implies that the data local to previous invocations must be preserved, and restored on exiting the routine. The simplest way of performing this is using a stack:

```
ENTRY  EQU  $
       PUSH R11              Save return address
       PUSH @ARG1            Save ARG1
       PUSH @ARG2            Save ARG2
       .
       .
       PUSH R0               Save R0
       LI   R0,...
       MOV  R0,@ARG1         Reset ARG1
       LI   R0,...
       MOV  R0,@ARG2         Reset ARG2
       .
       .
       .
       POP  R0               Restore R0
       .
       POP  @ARG2            Restore ARG2
       POP  @ARG1            Restore ARG1
       POP  R11              Restore return address
       RT
```

Note : The stacked items are popped in reverse order.

6.11.11 Jump Table

Suppose it is necessary to branch to a label (Li) depending on the value of a key (i); if i=1, then l1, if i=2 then L2, etc. Assume that R0 contains the key. This can be written as:

```
          CI    R0,1
          JEQ   L1
          CI    R0,2
          .
          .
          .
          JEQ   Ln
          JGT   OVER
UNDER     EQU   $                 Under range
          .
          .
OVER      EQU   $                 Over range
          .
          .
L1        EQU   $                 KEY=1
          .
          .
```

A more efficient method would be to replace each

```
CI   R0,i   with a   DEC   R0
```

This saves one word for each comparison.

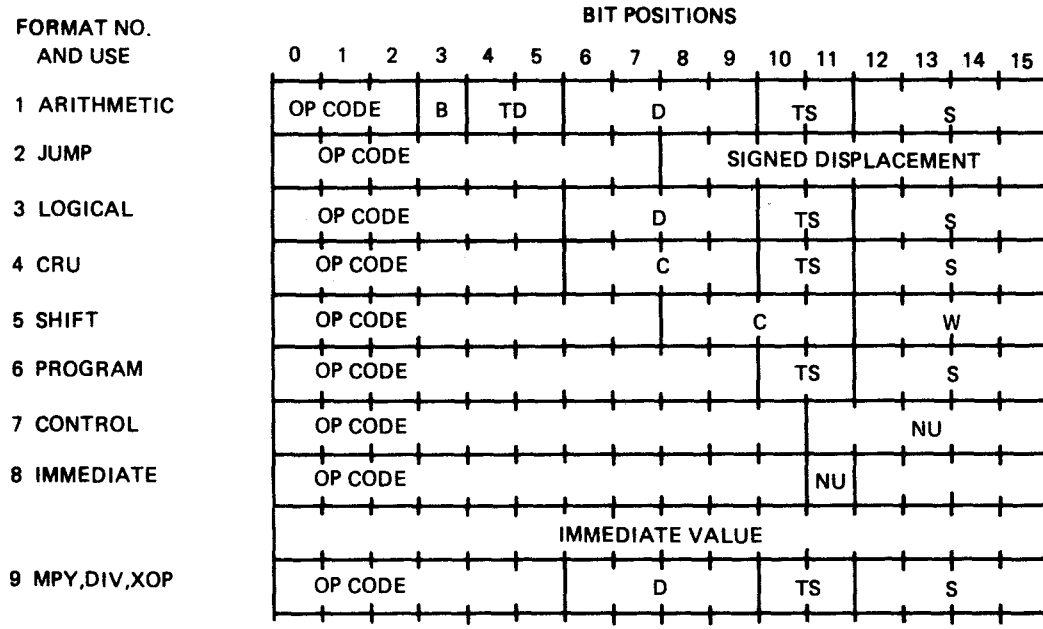
Probably the best method of implementing this would be to create a table of addresses, in ascending key order, of the labels. Using the index mode of addressing on the key, the following code is utilized:

```
TABLE  DATA L1,L2,....,Ln   Table of addresses
          .
          .
          .
* Key in range
          A     R0,R0           KEY->word offset,set code
          JLE   UNDER            KEY<=0?
          CI    R0,2*n
          JGT   OVER             KEY>n?
* Yes then branch to L1
```

6.12 REFERENCE SECTION

The paragraphs that follows provides additional explanation to the inforamtion on 9900 assembley language presented in this chapter.

6.12.1 Instruction Formats



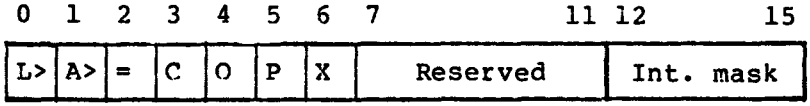
OPCODE - Assembly language mnemonic
B - Byte indicator (1=byte, 0=word)
Td - Destination address mode
D - Destination address
Ts - Source address mode
S - Source address
C - Shift or cru transfer count
W - Workspace register number
NU - Not used
SIGNED - Signed displacement of -128 to +127 words

TD/TS FIELD:

CODE	MODE		EFFECTIVE ADDRESS
00	WORKSPACE REGISTER	Rx	WP+2*[S OR D]
01	INDIRECT	*Rx	(WP+2*[S OR D])
10	INDEXED (S OR D/0)	@LABEL(Rx)	(WP+2*[S OR D])+(PC+2)
10	SYMBOLIC (S OR D=0)	@LABEL	(PC+2)
11	INDIRECT WITH AUTO INCREMENT	*Rx+	(WP+2*[S OR D]); Increment eff. address

An extra word is required for each operand code of 2.

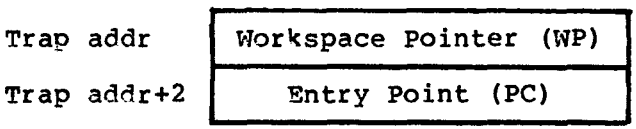
6.12.2 Status Register



0 - Logical greater (L>)
1 - Arithmetic greater (A>)
2 - Equal (=)
3 - Carry from msb (C)
4 - Overflow (O)
5 - Parity (P)
6 - XOP in progress (X)

Interrupt mask : F - All interrupts enabled
0 - Only interrupt level 0 enabled

6.12.3 Interrupts

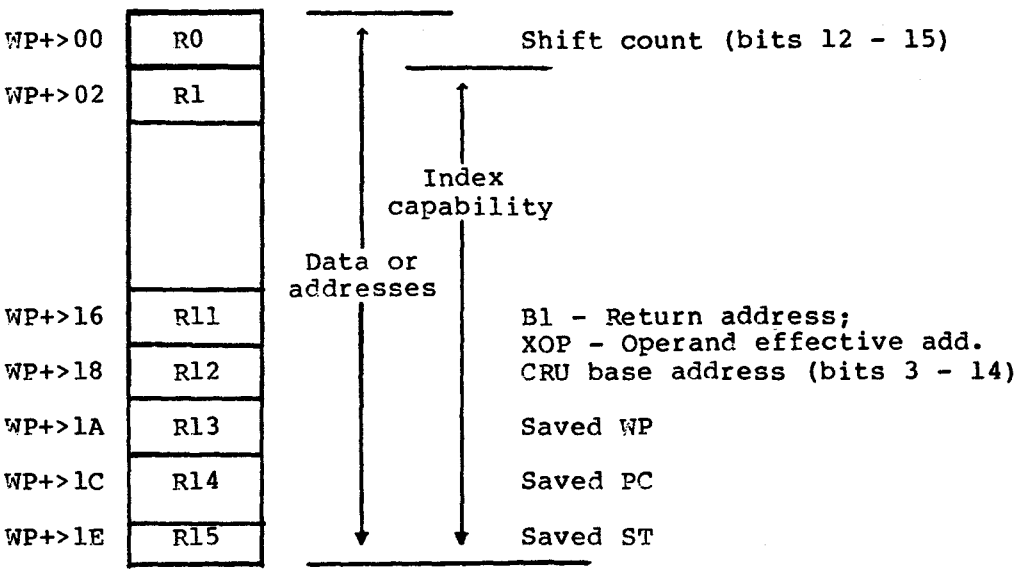


Note: 1) Interrupt vectors 0-15 from 0 to >3C
2) XOP vectors from >40 to >7C
3) LOAD vector at >FFFC
4) Interrupt 0 is the RESET interrupt

6.12.4 CRU

All CRU instructions use bits 3 - 14 of workspace register 12 (R12) as the base address for CRU operations. For CRU operations involving less than 9 bits (LDCR and STCR instructions) the most significant byte (bits 0 - 7) of a word is used.

6.12.5 Register Restrictions



Note: The MPY and DIV instructions use two consecutive registers, the first of which is supplied as an operand to the instruction (if R2 is the register operand then R2 and R3 are both used). If R15 is the specified register then the word following the workspace is used to store the remainder for DIV or the least significant half of the result for MPY.

6.12.6 Assembly Language Instructions

Symbols Used

G,G1,G2 - General memory addresses
R - Workspace register address
S - Symbolic memory address
E - Expression (all symbols previously defined)
I - Immediate value
T - Term (range 0 - 15)
( ) - Contents of the address within parenthesis
-> - 'Replaces'
: - 'Is compared to'
C - Count (0 to 15)
* - Result is compared to zero

INSTRUCTION	OPCODE	FORMAT TYPE	STATUS BITS AFFECTED	FORMAT		EFFECT
ABSOLUTE VALUE	0740	6	*0-2,4	ABS	G	ABSOLUTE(G)->(G)
ADD BYTES	B000	1	*0 -- 5	AB	G1,G2	(G1)+(G2)->(G2)
ADD IMMEDIATE	0220	8	*0 -- 4	AI	R.I	(R)+I->(R)
ADD WORDS	A000	1	*0 -- 4	A	G1,G2	(G1)+(G2)->(G2)
AND IMMEDIATE	0240	8	*0 -- 2	ANDI	R,I	(R) AND I->(R)
BRANCH	0440	6		B	G	G->(PC)
BRANCH AND LINK	0680	6		BL	G	G->(PC) (PC)->(R11)
BRANCH AND LOAD WP	0400	6		BLWP	G	(G)->(WP) (G+2)->(PC) (OLD WP)->(R13) (OLD PC)->(R14) (OLD ST)->(R15)
CLEAR	04C0	6		CLR	G	0->(G)
CLOCK OFF	03C0	7		CKOF		DISABLES CLOCK
CLOCK ON	03A0	7		CKON		ENABLES CLOCK
COMPARE BYTES	9000	1	0-2,5	CB	G1,G2	(G1):(G2)
COMPARE IMMEDIATE	0280	8	0 -- 2	CI	R,I	(R):I
COMPARE WORDS	8000	1	0 -- 2	C	G1,G2	(G1):(G2)
COMPARE ONES CORRESPONDING	2000	3	2	COC	G,R	ST2=AND OF RBITS CORRES. TO GBITS=
COMPARE ZEROS CORRESPONDING	2400	3	2	CZC	G,R	ST2=NAND OF RBITS CORRES. TO GBITS=
DECREMENT BY ONE	0600	6	*0 -- 4	DEC	(G)	(G)-1->(G)
DECREMENT BY TWO	0640	6	*0 -- 4	DECT	(G)	(G)-2->(G)
DIVIDE	3C00	9	4	DIV	G,R	INT (R)/(G)->(R) REM (R)/(G)->(R+1
EXECUTE INSTRUCTION	0480	6		X	G	EXECUTE INSTRUCTI AT ADDRESS G
EXTENDED OPERATION	2C00	9	6	XOP	G,T	(>40+4*T)->(WP) (>42+4*T)->(PC) EFF ADD OF G->(R1 (OLD WP)->(R13) (OLD PC)->(R14) (OLD ST)->(R15) 1->ST6
EXCLUSIVE OR	2800	3	*0 -- 2	XOR	G,R	(G) XOR (R)->(R)
IDLE	0340	7		IDLE		
INCREMENT BY ONE	0580	6	*0 -- 4	INC	G	(G)+1->(G)
INCREMENT BY TWO	05C0	6	*0 -- 4	INCT	G	(G)+2->(G)
INVERT BITS	0540	6	*0 -- 2	INV	G	1'S COMP (G)->(G)
JUMP (UNCONDITIONAL)	1000	2		JMP	S	S->(PC)
JUMP IF CARRY	1800	2		JOC	S	S->(PC) IF ST3=1
JUMP IF EQUAL	1300	2		JEQ	S	S->(PC) IF ST2=1
JUMP IF GREATER THAN	1500	2		JGT	S	S->(PC) IF ST1=1
JUMP IF HIGH OR EQUAL	1400	2		JHE	S	S->(PC) IF ST0=1 OR ST2=1
JUMP IF LESS THAN	1100	2		JLT	S	S->(PC) IF ST1=0 AND ST2=0

JUMP IF LOGICAL HIGH	1B00	2		JH	S	S->(PC) IF ST0=1 AND ST2=0
JUMP IF LOGICAL LOW	1A00	2		JL	S	S->(PC) IF ST0=0 AND ST2=0
JUMP IF LOW OR EQUAL	1200	2		JLE	S	S->(PC) IF ST0=0 OR ST2=1
JUMP IF NO CARRY	1700	2		JNC	S	S->(PC) IF ST3=0
JUMP IF NO OVERFLOW	1900	2		JNO	S	S->(PC) IF ST4=0
JUMP IF NOT EQUAL	1600	2		JNE	S	S->(PC) IF ST2=0
JUMP IF ODD PARITY	1C00	2		JOP	S	S->(PC) IF ST5=1
LOAD COMMUNICATIONS REGISTER	3000	4	*0-2,5	LDCR	G,T	TRANSFER T BITS FROM (G) TO CRU
LOAD IMMEDIATE	0200	8	*0 -- 2	LI	R,I	I->(R)
LOAD INTERRUPT MASK	0300	8	12-15	LIMI	I	I->(INT. MASK)
LOAD ROM AND EXECUTE	03E0	7	12-15	LREX		(>FFFC)->(WP) (>FFFE)->(PC) (OLD WP)->(R13) (OLD PC)->(R14) (ST)->(R15) 0->(INT. MASK)
MOVE BYTE	D000	1	*0-2,5	MOVB	G1,G2	(G1)->(G2)
MOVE WORD	C000	1	*0 -- 2	MOV	G1,G2	(G1)->(G2)
MULTIPLY	3800	9		MPY	G,R	MSW((G)*(R))->(R) LSW((G)*(R))->(R+1)
NEGATE	0500	6	*0 -- 4	NEG	G	-(G)->(G)
OR IMMEDIATE	0260	8	*0 -- 2	ORI	R,I	(R) OR I ->(R)
RESET I/O	0360	7		RSET		DISABLES INTERRUPTS RESETS I/O DEVICES BITS 12 -15 =0
RETURN WORKSPACE POINTER	0380	7	0 -- 6 12-15	RTWP		(R13)->(WP) (R14)->(PC) (R15)->(ST)
SET BIT TO ONE	1D00	2		SB0	E	1->(E+(R12))
SET BIT TO ZERO	1E00	2		SBZ	E	0->(E+(R12))
SET TO ONES	0700	6		SETO	G	>FFFF->(G)
SET ONES CORRESPONDING BYTE	F000	1	*0-2,5	SOCB	G1,G2	(G1) OR (G2) ->(G2)
SET ONES CORRESPONDING WORD	E000	1	*0 -- 2	SOC	G1,G2	(G1) OR (G2) ->(G2)
SHIFT LEFT ARITHMETIC	0A00	5	0 -- 4	SLA	R,C	) IF C=0 THEN 4
SHIFT RIGHT ARITHMETIC	0800	5	0 -- 3	SRA	R,C	) LSBS OF R0 USED.
SHIFT RIGHT CIRCULAR	0B00	5	0 -- 3	SRC	R,C	) IF THESE =0 THEN
SHIFT RIGHT LOGICAL	0900	5	0 -- 3	SRL	R,C	) C=16.
STORE COMMUNICATIONS REGISTER	3400	4	*0-2,5	STCR	G,T	T BITS FROM CRU LINES TO G
STORE STATUS REGISTER	02C0	8		STST	R	(ST)->(R)
STORE WORKSPACE POINTER	02A0	8		STWP	R	(WP)->(R)
SUBTRACT BYTE	7000	1	*0 -- 5	SB	G1,G2	(G2)-(G1)->(G2)
SUBTRACT WORD	6000	1	*0 -- 4	S	G1,G2	(G2)-(G1)->(G2)
SWAP BYTES	06C0	6		SWPB	G	INTERCHANGE BITS 0-7 WITH BITS 8-15 OF WORD G
SET ZEROES CORRESPONDING BYTE	5000	1	*0-2,5	SZCB	G1,G2	(INV(G1)) AND (G2) ->(G2)

SET ZEROES	4000	1	*0 -- 2	SZC	G1,G2	(INV(G1)) AND (G2)
CORRESPONDING WORD						->(G2)
TEST BIT	1F00	2	2	TB	E	(R12)+E->ST2

6.12.7 Pseudo-Instructions

INSTRUCTION	FORMAT	EFFECT
NO OPERATION	NOP	JMP $+2
RETURN	RT	B *R11

TRANSFER VECTOR for a 'BLWP @label' (SDSMAC only)

```
        label XVEC  wpadd,pcadd      label  DATA  wpadd
                                            DATA  pcadd
                                            WPNT  wpadd
```

6.12.8 Assembler Directives

( ) - The item in parenthesis is optional
(,x) - Any number of 'x's (each preceded by a comma)

All of these directives (except OPTION) may be preceded by a label and followed by a comment.

ABSOLUTE ORIGIN - AORG exp

AORG places the value of EXP (an absolute expression) in the location counter and defines the succeeding locations as absolute.

RELOCATABLE ORIGIN - RORG (exp)

RORG places the value of EXP (an absolute or relocatable expression) in the location counter; if encoutered in absolute code it also defines the succeeding locations as relocatable. If EXP is not used then the location counter is replaced by :
Current length of program segment for absolute code
Length of data segment for data relocatable code
Length of common segment for common relocatable code

DUMMY ORIGIN - DORG exp

DORG places the value of EXP (a relocatable or absolute expression) in the location counter and defines the succeeding locations as a dummy block. No object code is generated for the dummy block, but the module is allowed to access the symbols of another module.

DATA SEGMENT - DSEG

DSEG places a value in the location counter and defines the succeeding locations as data relocatable. Either of the following

values is placed in the location counter:
Maximum value location counter has ever attained as a result of assembling any preceding blocks of data relocatable code
Zero, if no data relocatable code has been assembled

DATA SEGMENT END - DEND

DEND terminates a DSEG by placing a value in the location counter and defines succeeding locations as program relocatable. Either of the following values is placed in the location counter:
Maximum value location counter has ever attained as a result of assembling any preceding blocks of program relocatable code
Zero, if no program relocatable code has been assembled

COMMON SEGMENT - CSEG (string)

CSEG places a value in the location counter and defines the succeeding locations as common relocatable code. STRING is used to define the begining (or continuation) of the named common segment. (If STRING blank then it refers to the BLANK common segment.) If the string has not previously been used in a CSEQ directive, it sets the location counter to zero and defines the succeeding locations as relocatable to the new segment. Otherwise it is a continuation and the location counter is set to the maximum value it attained when previously assembling the segment.

COMMON SEGMENT END - CEND

CEND terminates a CSEG by placing a value in the location counter and defines succeeding locations as program relocatable. The location counter value is the same as for DEND.

PROGRAM SEGMENT - PSEG

PSEG places a value in the location counter and defines the succeeding locations as program relocatable. Either of the following values is placed in the location counter :
Maximum value location counter has ever attained as a result of assembling any preceding blocks of program relocatable code
Zero, if no program relocatable code has been assembled

PROGRAM SEGMENT END - PEND

PEND terminates a PSEG by placing a value in the location counter and defines succeeding locations as program relocatable. The location counter value is the same as for DEND.

BLOCK STARTING WITH SYMBOL - BSS exp

BSS reserves EXP number of consecutive bytes. When a label precedes BSS it is assigned the address of the first byte of the block.

BLOCK ENDING WITH SYMBOL - BES exp

BES reserves EXP number of consecutive bytes. When a label precedes BES it is assigned the address of the first byte immediately following the block.

INITIALIZE BYTE - BYTE exp(,exp)

BYTE reserves successive bytes of memory and initializes them to

their respective values of EXP.

INITIALIZE WORD - WORD exp(,exp)
WORD reserves successive words of memory and initializes them to their respective values of EXP.

INITIALIZE TEXT - TEXT (-)string
TEXT reserves successive bytes of memory and initializes them to the appropriate character in the string. The string is delimited by single quotes and can be up to 52 characters long. If the optional minus sign is present then the last character in the string is negated.

WORD BOUNDARY ALIGN - EVEN
EVEN aligns the location counter to a word boundary if it contains an odd value. otherwise it is unchanged.

DEFINE ASSEMBLY TIME CONSTANTS - label EQU exp
EQU assigns the value of EXP to LABEL.

EXTERNAL DEFINITION - DEF symbol(,symbol)
DEF allows other programs to access a program's SYMBOLs.

EXTERNAL REFERENCE - REF symbol(,symbol)
REF provides access to SYMBOLs defined in other programs.

SECONDARY EXTERNAL REFERENCE - SREF symbol(,symbol)
SREF provides access to one or more SYMBOLs defined in other programs.

FORCE LOAD - LOAD symbol(,symbol)
LOAD causes a special object tag to be generated that acts as a Link Editor control command. SYMBOL is treated as if it were a value in an INCLUDE statement. This command is used in conjunction with SREF

DEFINE EXTENDED OPERATION - DXOP sym,num
DXOP assigns SYM to be used in the operator field as an extended operation. NUM, in the range 0 - 15, specifies the extended operation number.

PROGRAM END - END (symbol)
END terminates the assembly. Source lines following this directive are ignored. SYMBOL, if present, specifies the program's entry point.

OUTPUT OPTIONS - OPTION key(,key)
OPTION specifies the output and listing options to the assembler. A label is not allowed with this directive. KEY can be any of the following:

XREF - Print cross reference table.
OBJ - Print listing of the object code.
SYMT - Print symbol table.

Additional key words for SDSMAC only:

NOLIST - Suppress printing of source listing.
TUNLIST - Limit listing for text directives (1 line)
DUNLIST - Limit listing for data directives (1 line)
BUNLIST - Limit listing for cycle directives (1 line)
MUNLIST - Limit listing for macro expansion (1 line)

PROGRAM IDENTIFIER - IDT string

IDT assigns a name to the program. This directive must precede any assembly language instructions or assembler directives that produce object code. Only the first 8 characters of STRING (delimited by single quotes) are used.

PAGE TITLE - TITL string

TITL supplies the title to be printed as the heading for the source listing. If a heading is required on the first page, a TITL directive must be the first source statement. STRING is delimited by single quotes and can be up to 50 characters in length.

LIST SOURCE - LIST

LIST restores printing of the source listing and is only required when a no source list directive is in effect. The directive in not printed in the listing.

NO SOURCE LISTING - UNL

UNL inhibits the printing of the source listing. The directive is not printed in the listing.

PAGE EJECT - PAGE

PAGE causes the assembler to continue the source listing on a new page. The directive is not printed in the listing.

WORKSPACE POINTER - WPNT label SDSMAC only

WPNT defines the current workspace (referenced by LABEL) to the assembler but produces no object code.

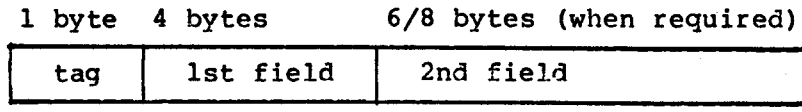
COPY SOURCE FILES - COPY file SDSMAC only

COPY causes input to the assembler to be taken from FILE. On end-of-file, input is resumed from the original file.

DEFINE OPERATION - DFOP sym,op SDSMAC only

DFOP defines a synonym (SYM) for an operation (OP). OP may be a mnemonic, a macro name, or the SYM of a previous DFOP or DXOP directive.

6.12.9 Object Record Format and Code



TAG	1st Field	2nd Field	Meaning
0	Length of all relocatable code	Program ID (8 chars)	Program start
1	Address	Not used	Absolute entry point
2	Address	Not used	Relocatable entry point
3	Location of last appearance of symbol	6 char symbol	External reference last Used in relocatable code
4	Location of last appearance of symbol	6 char symbol	External reference last Used in absolute code
5	Location	6 char symbol	Relocatable external Definition
6	Location	6 char symbol	Absolute external Definition
7	Checksum for current record	Not used	Checksum
8	Any value	Not used	Ignore checksum value
9	Load address	Not used	Absolute load address
A	Load address	Not used	Relocatable load address
B	Data	Not used	Absolute data
C	Data	Not used	Relocatable data
D	Load bias	Not used	Load bias or offset
E			Illegal
F	Not used	Not used	End of record

6.12.10 TMS 9900 Instruction Execution Times

INSTRUCTION	CLOCK CYCLES	MEMORY ACCESS	ADD. MOD TABLE SOURCE	DEST
A	14	4	A	A
AB	14	4	B	B
ABS (MSB=0)	12	2	A	-
(MSB=1)	14	3	A	-
AI	14	4	-	-
ANDI	14	4	-	-
B	8	2	A	-
BL	12	3	A	-
BLWP	26	6	A	-
C	14	3	A	A
CB	14	3	B	B
CI	14	3	-	-
CKOF	12	1	-	-
CKON	12	1	-	-
CLR	10	3	A	-
COC	14	3	A	-
CZC	14	3	A	-
DEC	10	3	A	-
DECT	10	3	A	-
DIV ST4 IS SET	16	3	A	-
ST4 IS RESET *	92-124	6	A	-
IDLE	12	1	-	-
INC	10	3	A	-
INCT	10	3	A	-
INV	10	3	A	-
JUMP PC CHANGED	10	1	-	-
PC UNCHANGED	8	1	-	-
LDCR C=0	52	3	A	-
1<=C<=8	20-2C	3	B	-
9<=C<=15	20-2C	3	A	-
LI	12	3	-	-
LIMI	16	2	-	-
LREX	12	1	-	-
~RESET FUNCTION	26	5	-	-
~LOAD FUNCTION	22	5	-	-
INTERRUPT CONTEXT SWITCH	22	5	-	-

* Execution time is dependent upon the partial quotient after each clock cycle during execution

INSTRUCTION	CLOCK CYCLES	MEMORY ACCESS	ADD. MOD TABLE SOURCE	DEST
LWPI	10	2	-	-
MOV	14	4	A	A
MOVB	14	4	B	B
MPY	52	5	A	-
NEG	12	3	A	-
ORI	14	4	-	-
RSET	12	1	-	-
RTWP	14	4	-	-
S	14	4	A	A
SB	14	4	B	B
SBO	12	2	-	-
SBZ	12	2	-	-
SETO	10	3	A	-
SHIFT C≠0	12-2C	3	-	-
C=0,R0=0	52	4	-	-
C=0,R0=N=0	20-2N	4	-	-
SOC	14	4	A	A
SOCB	14	4	B	B
STCR C=0	60	4	A	-
1<=C<=7	42	4	B	-
C=8	44	4	B	-
9<=C<=15	58	4	A	-
STST	8	2	-	-
STWP	8	2	-	-
SWPB	10	3	A	-
SZC	14	4	A	A
SZCB	14	4	B	B
TB	12	2	-	-
X *	8	2	A	-
XOP	36	8	A	-
XOR	14	4	A	-
UNDEFINED OPCODES	6	1	-	-

* Execution time is added to that of the instruction located at the source address minus 4 clock cycles and 1 memory access time

ADDRESS MOD TABLE A

MODE	CLOCK CYCLES	MEMORY ACCESS
00	0	0
01	4	1
10*	8	1
11	8	2

ADDRESS MOD TABLE B

MODE	CLOCK CYCLES	MEMORY ACCESS
00	0	0
01	4	1
10*	8	1
11	6	2

* Indexed addressing requires 1 more memory access than that shown for symbolic addressing

T=tc(0)(C+(W*M))

T - Total instruction execution time
tc(0) - Clock cycle time
C - Number of clock cycles for instruction execution plus address modification
W - Number of required wait states per memory access for instruction execution plus address modification
M - Number of memory accesses

6.12.11 TMS 9900 Pin Assignments

PIN	FUNCTION	PIN	FUNCTION	PIN	FUNCTION
1	Vbb	23	A1	45	D4
2	Vcc	24	A0	46	D5
3	WAIT	25	Ø4	47	D6
4	~LOAD	26	Vss	48	D7
5	HOLDA	27	Vdd	49	D8
6	~RESET	28	Ø3	50	D9
7	IAQ	29	DBIN	51	D10
8	Ø1	30	CRUOUT	52	D11
9	Ø2	31	CRUIN	53	D12
10	A14	32	~INTREQ	54	D13
11	A13	33	IC3	55	D14
12	A12	34	IC2	56	D15
13	A11	35	IC1	57	NC
14	A10	36	IC0	58	NC
15	A9	37	NC	59	Vcc
16	A8	38	NC	60	CRUCLK
17	A7	39	NC	61	~WE
18	A6	40	Vss	62	READY
19	A5	41	D0	63	~MEMEN
20	A4	42	D1	64	~HOLD
21	A3	43	D2		
22	A4	44	D3		

NC - No internal connection
Vss - Pins 26,40 must be connected in parallel
Vcc - Pins 2,59 must be connected in parallel

6.12.12 ASCII Character Set

CHAR	HEX	CHAR	HEX	CHAR	HEX
NUL	00	+	2B	V	56
SOH	01	,	2C	W	57
STX	02	-	2D	X	58
ETX	03	.	2E	Y	59
EOT	04	/	2F	Z	5A
ENQ	05	0	30	[	5B
ACK	06	1	31	\	5C
BEL	07	2	32	]	5D
BS	08	3	33	^	5E
HT	09	4	34	_	5F
LF	0A	5	35	`	60
VT	0B	6	36	a	61
FF	0C	7	37	b	62
CR	0D	8	38	c	63
S0	0E	9	39	d	64
S1	0F	:	3A	e	65
DLE	10	;	3B	f	66
DC1	11	<	3C	g	67
DC2	12	=	3D	h	68
DC3	13	>	3E	i	69
DC4	14	?	3F	j	6A
NAK	15	@	40	k	6B
SYN	16	A	41	l	6C
ETB	17	B	42	m	6D
CAN	18	C	43	n	6E
EM	19	D	44	o	6F
SUB	1A	E	45	p	70
ESC	1B	F	46	q	71
FS	1C	G	47	r	72
GS	1D	H	48	s	73
RS	1E	I	49	t	74
US	1F	J	4A	u	75
SPACE	20	K	4B	v	76
!	21	L	4C	w	77
"	22	M	4D	x	78
#	23	N	4E	y	79
$	24	O	4F	z	7A
%	25	P	50	{	7B
&	26	Q	51	|	7C
'	27	R	52	}	7D
(	28	S	53	~	7E
)	29	T	54	DEL	7F
*	2A	U	55		

6.12.13 Hex-Decimal Table

EVEN BYTE				ODD BYTE			
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4,096	1	256	1	16	1	1
2	8,192	2	512	2	32	2	2
3	12,288	3	768	3	48	3	3
4	16,384	4	1,024	4	64	4	4
5	20,480	5	1,280	5	80	5	5
6	24,576	6	1,536	6	96	6	6
7	28,672	7	1,792	7	112	7	7
8	32,768	8	2,048	8	128	8	8
9	36,864	9	2,304	9	144	9	9
A	40,960	A	2,560	A	160	A	10
B	45,056	B	2,816	B	176	B	11
C	49,152	C	3,072	C	192	C	12
D	53,248	D	3,328	D	208	D	13
E	57,344	E	3,584	E	224	E	14
F	61,440	F	3,840	F	240	F	15

INDEX