S O F T W A R E

D E V E L O P M E N T

H A N D B O O K

Geoff Vincent

Jim Gill

Texas Instruments October 1981

IMPORTANT NOTICES

Texas Instruments reserves the right to make changes at any time to improve design and to supply the best possible product for the spectrum of users.

Information contained in this publication is believed to be accurate and reliable. However, responsibility is assumed neither for its use nor for any infringement of patents or rights of others that may result from its use. No license is granted by implication or otherwise under any patent or patent right of Texas Instruments or others.

Note "Texas Instruments" includes where the context permits Texas Instruments Incorporated, and any of its affiliated companies, including Texas Instruments Limited.

PREFACE

This Second Edition of the Software Development Handbook has been extensively revised and updated to incorporate new developments, and to improve and clarify the presentation.

As before, it is hoped that the book will appeal on several levels. The first three chapters are an introduction to the technology, and assume little or no technical knowledge. Chapter 1, which is introductory, describes the nature of software and the particular contribution of microsystems technology. Chapter 2 describes, step by step, the process of software development for microcomputers. Chapter 3 describes the tools of the software engineer. It is hoped that these chapters will appeal to those who have a peripheral interest in the technology, as well as to those who are or will become directly involved in software engineering.

Chapter 4 addresses the subject of software design, which we feel can and should be tackled separately from the discipline of programming in a particular language. The goal of appealing to a wide level of readership means that experienced software engineers will find some of the material familiar; however the approach may well be new, and some at least of the ideas will be novel, This chapter introduces suggested algorithmic and graphical notations for language independent software design, Those new to the technology are advised to read Chapter 4 in conjunction with some practical experience of programming in one of the languages available.

Chapter 5, Component Software, is the major new addition to the book. It describes a method of developing and packaging complex real time software functions. Such packages are available off the shelf from Texas Instruments for direct incorporation in application systems. Component Software is a significant step towards complete packaged functions, incorporating both hardware and software. These are likely to play an important part in microsystems technology in the future. Chapter 5 also includes a description of concurrency and the requirements of real time software.

Chapters 6, 7 and 8 describe in turn Microprocessor Pascal, Power BASIC, and 9900/99000 Assembly Language. These chapters are not intended to be complete language tutorials. Tutorials are available elsewhere; and it is felt that programming is best taught by a combination of

personal tuition and practical experience. Courses on programming are available from various sources, including Texas Instruments. Rather, these chapters are designed to give a feel for each language, its important features, and its areas of application. Microprocessor Pascal is a professional programmer's tool which permits the construction of reliable, real time software systems of any level of complexity. Power BASIC is a much simpler language that can be learned in a few hours, and can be used even by non software professionals to provide quick solutions to simple problems. Assembly language provides direct access to all the resources of the microcomputer, and can be used in critical areas of a system to "fine tune" for maximum performance. Naturally, effective use of assembly language requires a certain level of skill. Chapter 8 contains an extensive "Algorithms and Techniques" section, describing some commonly used solutions to specific problems. Each chapter includes, besides the language description, a Reference Section that tabulates the vital elements of each language.

This handbook is not intended as a complete course in software development for microcomputers. However, with appropriate additional material and combined with practical experience of one or more of the languages described, it could form the basis for such a course. The aim is to provide a Handbook for the emerging discipline of software engineering for microcomputers, and to begin the process of identifying and communicating those elements of the technology that will prove to be of lasting value. This book is a distillation of the practical experience of software engineers, and it is hoped that it will make some contribution to those entering on or already immersed in the technology.

The authors wish to thank all those who have contributed approaches, ideas, descriptions or actual software examples, and without whom this book could not have been written,

Geoff Vincent
Jim Gill

October 1981

We would appreciate your comments on the usefulness of this handbook. Please complete and return this form to the address overleaf.

Name: (last) _____ (first): _____
Company: _____ Position: _____
Address: _____
_____
_____
Country: _____

1. Is the handbook well organised? Yes ____ No _____
   Comments: _____
   _____
2. Is the text correctly presented and adequately illustrated? Yes ___ **No** ___
   Comments: _____
   _____
3. What subject matter could be expanded or clarified?
   _____
   _____
4. Are you directly involved in software development? Please indicate your main **area(s)** of interest.
   _____
   _____
5. Have you found this handbook useful
   (a) As an introduction to the field _____
   (b) As a source of **ideas/information** _____
   (c) As a reference book _____
   (d) In any other way (please specify) _____
   _____
6. Do you use any Texas Instruments software products? Is the information on these products useful to you?
   _____
   _____
7. Any other comments _____
   _____
   _____
   _____

Please mail this sheet to:

M/S 35
Microprocessor Group
TEXAS INSTRUMENTS Ltd
Manton Lane
Bedford
MK41 7PA
ENGLAND

TABLE OF CONTENTS

CHAPTER 4 SOFTWARE DESIGN

CHAPTER 5 COMPONENT SOFTWARE

CHAPTER 6 MICROPROCESSOR PASCAL

CHAPTER 7 POWER BASIC

CHAPTER 8 ASSEMBLY LANGUAGE

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 WHAT IS SOFTWARE?

Software is what makes microprocessor technology different from conventional engineering techniques. Fundamentally, software is a set of instructions that tells the hardware (the microprocessor, and any electrical or mechanical devices connected to it) what to do.

In a conventional machine, the physical layout of the parts determines what the machine will do:

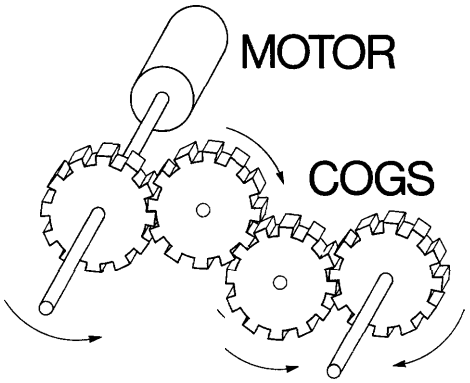


Figure 1-1 Conventional Machine

In a microprocessor machine, it is not always possible to tell from the physical arrangement exactly what the machine does:

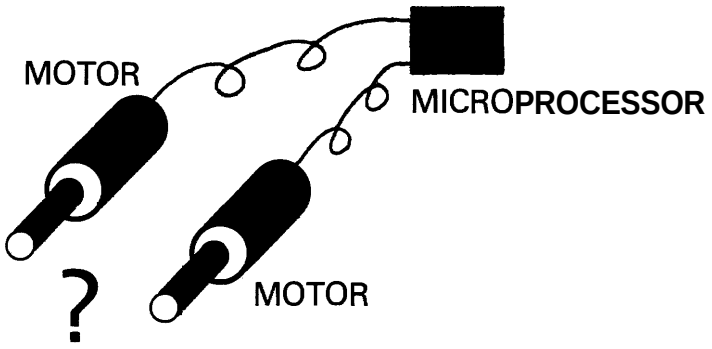


Figure 1-2 Microprocessor Machine

The function of the machine is determined by software.

The general layout of a microprocessor machine is shown in Figure 1-3.

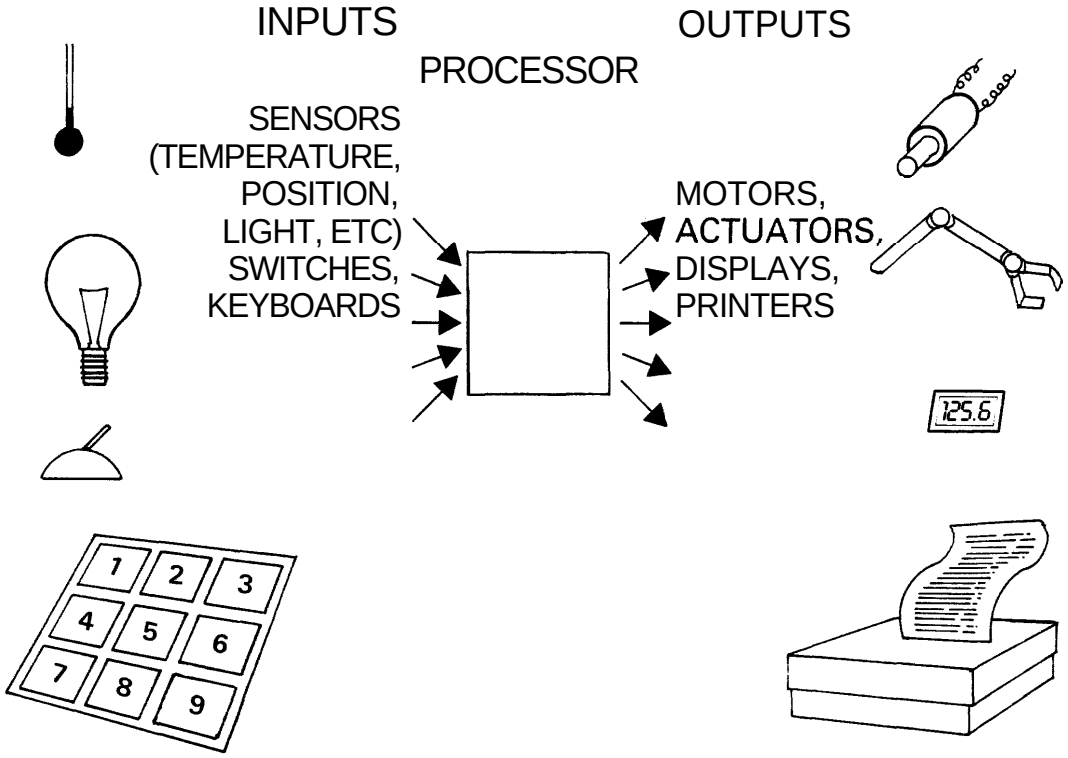


Figure 1-3 Layout of a Microprocessor Machine

In the centre is the microprocessor. To the processor are brought a series of inputs - which might come from temperature sensors, limit switches, operator keyboards and so on, All inputs must be converted to electrical signals before they reach the processor.

From the processor come a collection of outputs - again electrical signals, which can be used to operate motors, actuators, displays and so on, The processor itself has an extensive repertoire of operations it can perform, involving inputs, outputs and internal manipulations. However, by itself the processor is useless, It needs a program - a set of software instructions that specify exactly what operations to perform, and in what order. The program will determine when to take notice of (to read) the input signals, what to do with them, and what output signals to produce. It is the program that controls the machine.

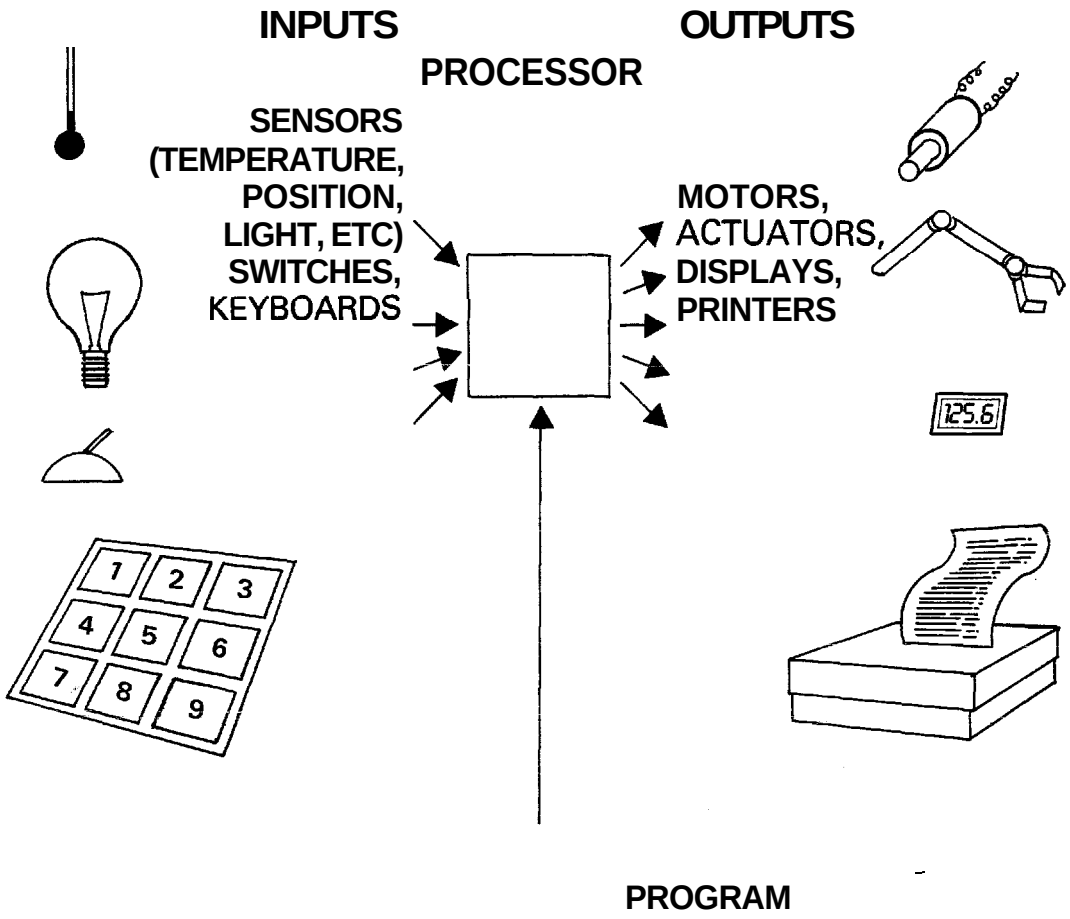


Figure 1-4 Program Control

One characteristic of microprocessor systems is that a different program placed in the 'same set of hardware will cause the machine to do different things. Of course, the scope of what can be done is determined by the hardware: if there is not a motor control circuit connected to a microprocessor, there is no way that the software will be able to turn a motor on and off. It is the hardware that determines what is possible; it is the software that determines what the machine actually does.

Software must have some ultimate physical reality in order to have any effect on the real world. However, it has two fundamental characteristics which distinguish it from hardware. First, it is at least an order of magnitude easier to manipulate than hardware: changing a piece of software usually involves no more than typing a few keys at a keyboard, while changing a hardware layout (say a printed circuit board) requires a lot of work and a lot of time. Second, software has a chameleon-like quality of being able to change its physical form without altering its essential nature. The same piece of software may exist on a magnetic disk, in semiconductor memory, as printed output or displayed on a screen.

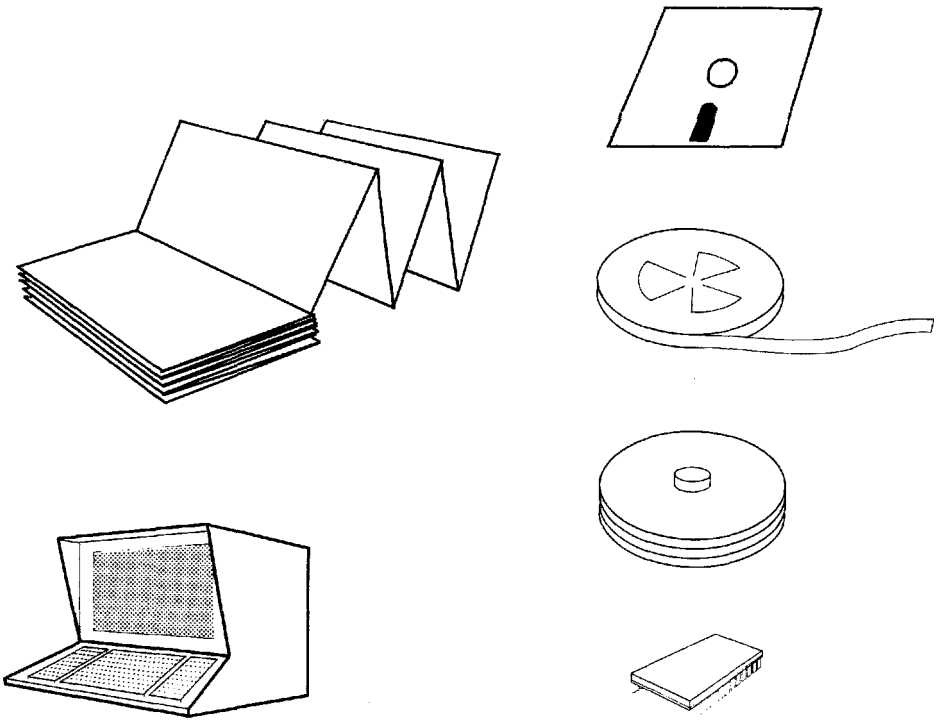


Figure 1-5 Software Has No Unique Physical Form

The problems which characterise software engineering are problems of management and organization rather than the problems of dealing with the physical world.

The way the traditional computer evolved was determined by the size and cost of available technology. These factors influenced how the different parts of the computer developed, how they were put together, and the kinds of applications where computers could be used. For reasons of cost and physical size it made no sense at all to consider placing a computer in a consumer product, or even in the average factory. Microprocessors are small and cheap enough to be placed in any piece of equipment. This, in turn, has revolutionised some aspects of computer technology: microcomputers are not just smaller copies of large computers, but have some significant new characteristics.

The major effort of design for a microcomputer application 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 potential of the microprocessor leads to more ambitious projects. With a new technology, 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 adapting, because of the special features and the different areas of application of microcomputers. This book describes the techniques of system and software design that are applicable to the new technology of microsystems (= microprocessor systems).

1.2 BLACK BOXES AND DIGITAL ELECTRONICS

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

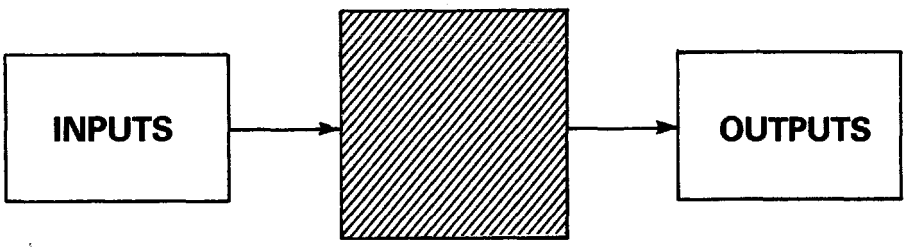


Figure 1-6 "Black Box"

"Inputs" might be switches, temperature sensors, flow. rate detectors, or keys pressed by a human operator. "Outputs" might control a motor, print text or figures, switch on a heater, and so on.

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. 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 black boxes 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. In general, mechanical systems are restricted to the kind of simple and direct response characterised by the typewriter. Electrical systems provide additional power, but in general do not permit much greater complexity.

Electronics introduced a whole new range of possibilities. 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, conventionally depicted as follows:

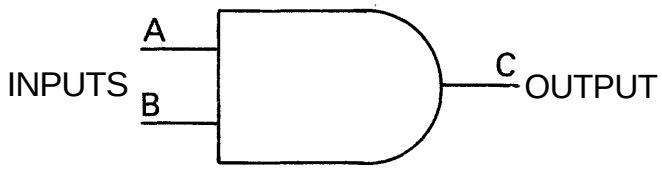


Figure 1-7 AND Gate

The basic AND gate has two inputs, here called A and B, and one output C. These are digital signals, each of which can take one of two possible values (conventionally represented as "0" and "1"). Each input and output line represents one bit of information. 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. This can be summarised in a truth table, which maps the value of the output C for all possible values of the inputs A and B:

		B	
		0	1
A	0	0	0
	1	0	1

Figure 1-8 AND Gate Truth Table

By combining logic elements such as the AND gate, electronic circuits can be constructed to take decisions and signal appropriate outputs depending on the state of any number of inputs. It is only necessary to arrange that the inputs represent the state of switches, sensors etc, and to connect the outputs to motor control circuits, actuators and displays, to construct very complex pieces of machinery.

Electronic systems can provide a limited kind of memory, counting operations, and simple arithmetic. Integrated

circuit technology allows many thousands of logic elements such as the AND gate to be implemented on a single chip of silicon 4 or 5 mm square. Electronics works very fast, too: many millions of decisions of the AND gate variety (determining the vaiue of C given the values of A and B) can be made per second, and many decisions can be made in parallel. However, the technology becomes very expensive for complex applications, and systems take a long time to develop.

Digital electronics is powerful because it permits any operation that can be conceived using bits; and any real world action that can be translated into electrical signals can he represented as bits. The techniques of digital electronics can be used for a vast range of different applications, where any kind of iogicai decision making or arithmetic processing is required.

Solving a real world problem, of course, depends on translating real inputs (such as mechanical movements, temperature readkngs, etc) into bits, and translating bits back into the rear world.

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

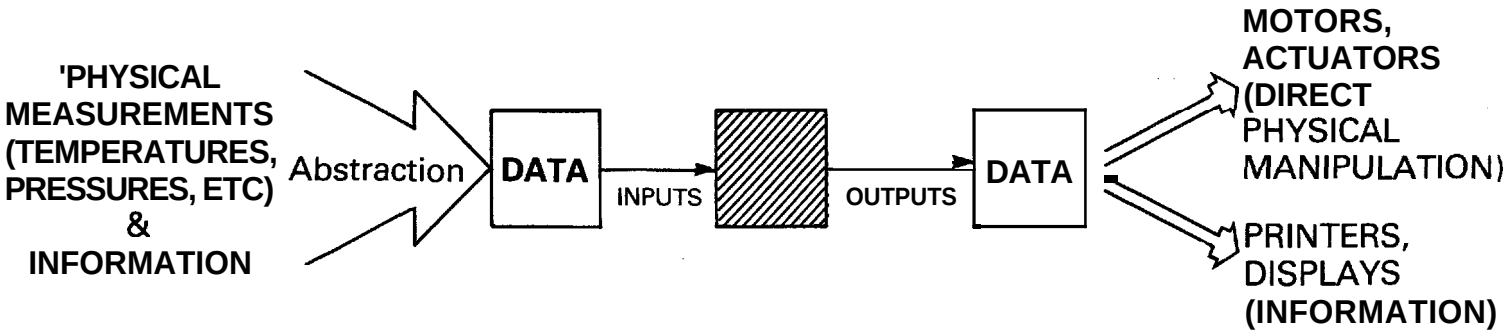


Figure 1-9 Data Translation

'Data' is a term used for coded information - that is, information translated into a pattern of bits for processing by a digital circuit. Data can be considered as an

abstracted represenpation of the real world.

In extracting data from the real world for processing by a digital circuit, the designer 'selects only the aspects of the information? available that he wants, enumerates all possible values, and designs his system to cope with and respond predictably to every possible combination. The digital circuit does not know or care what the data represents; it simply processes bits according to the logic designed into it,

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 in a well thought-out manner is probably the most important step in designing any digital system.

In the last 20 years, advances in technology have vastly decreased the price and increased the capability of digital electronics. 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 COMPUTERS

The idea for the computer existed long before the implementation techniques that made it practically realisable. In the 19th Century, Charles Babbage conceived a 'difference engine" that would operate according to the instructions of a stored program. However, the techniques available to him (mechanical cogs and levers) were unequal to the task. Babbage never completed his project.

Practical realisation of the computer had to wait for electronics - first using valves (which were .notoriously unreliable, large, and power hungry), then transistors, and finally integrated circuits. What the computer does is to separate the device which carries out the work of decision making, calculation etc - the processor - from the set of instructions - the program - which tell the processor what to do. This separation allows specialist manufacturers to design and implement powerful and efficient processors for

the range of possible applications, while application engineers can take a standard processor and write a software program to tailor its operation to their specific need.

Like other digital devices, computers work with bits, In fact, they usually work with groups of bits. The Texas Instruments TMS 9900/99000 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:

```
firstword    0 1 0 1 1 0 1 1 1 0 0 1 0 1 1 0
secondword   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 hits, 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:

```
                          BINARY                          DECIMAL

firstword    0 1 0 1 1 0 1 1 1 0 0 1 0 1 1 0              23446
second word  0 1 0 1 0 1 0 1 1 0 1 0 1 1 0 1            + 21933

result       1 0 1 1 0 0 0 1 0 1 0 0 0 0 1 1            = 45379
```

The instruction set for the TMS9900 and 99000 also includes operations on bytes (1 byte = 8 bits) of data,

In addition there are instructions to read input signals from the outside world and to write outputs, and to move data around within the computer.

* The binary number system is described in Chapter 8, section 8.13.2.1

A program is a list of these instructions stored in the computer's memory. A computer, then, looks like Figure 1-10.

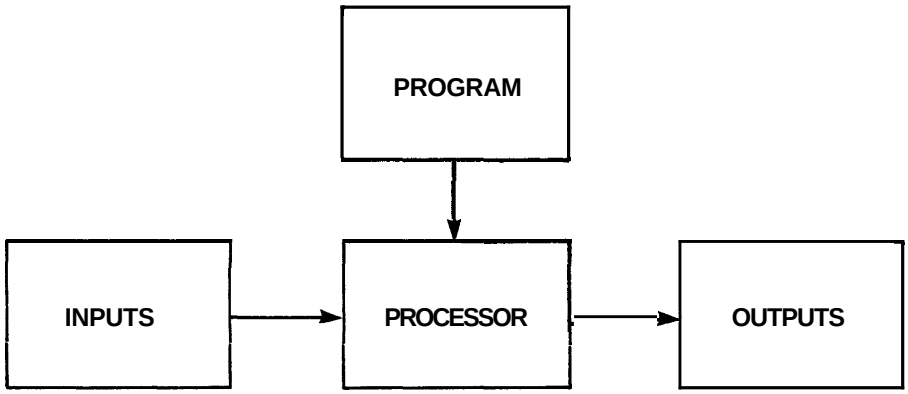


Figure 1-10 Computer

The stored program controls the operation of the computer. The processor fetches the program instructions one at a time. Instructions are normally executed in sequence, one after another. However, the computer has the capability to change this. It can make simple decisions about whether to execute one set of instructions or another. The decisions might depend on the value of some data word stored in memory, or the state of some input, or on a more complex condition.

For example,

"IF temperature LESS THAN set value AND heater is off THEN switch heater on"

The primitive control instructions, which can change program flow and make pre-programmed decisions, are the final group of operations that make up the computer's instruction set. With these five basic groups of instructions - logical, arithmetic, input/output (I/O), data transfer, and control - a computer can perform any task that can be precisely and unambiguously specified. The task of software design is to carry out this specification and, ultimately, to produce the program in a form that the computer can implement it.

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 he entirely predictable. Thus a computer is a black box, but one whose operation is determined not by the physical arrangement of its parts, but by a software program. Computer hardware can

be regarded as a pool of resources, which are organized by the software. By placing the burden of organization on software, many of the problems of designing a digital system are solved.

Figure 1-11 shows the structure of a computer in more detail.

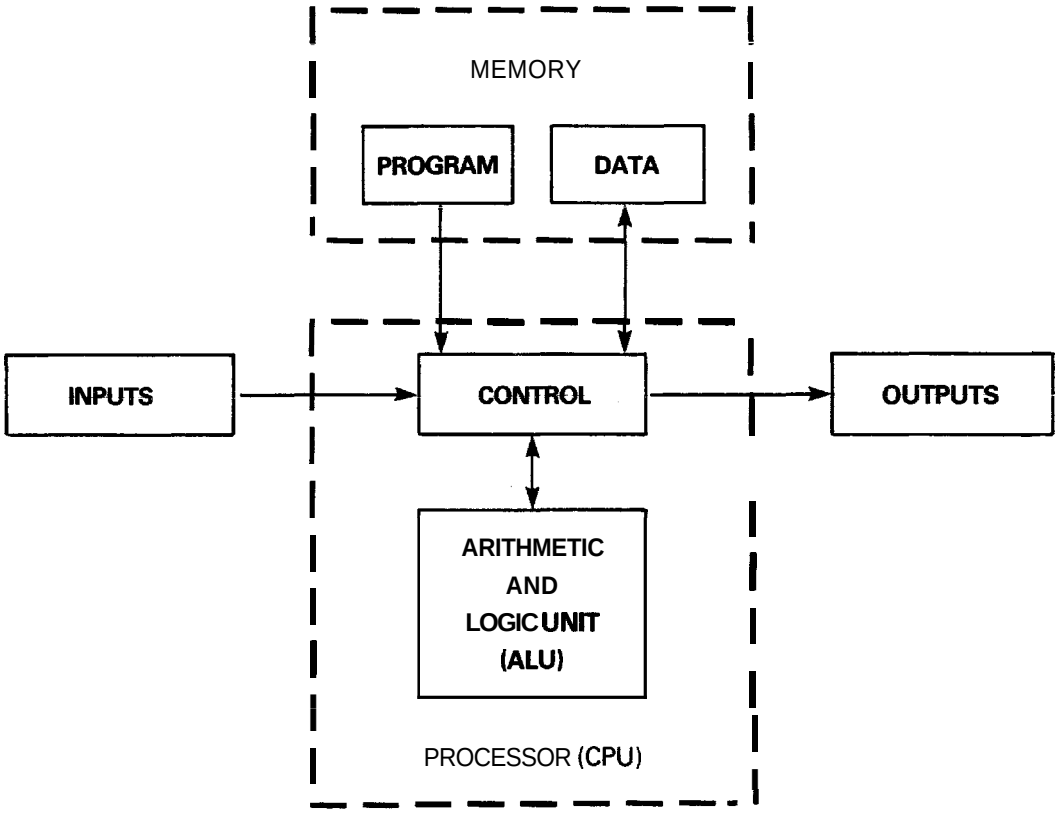


Figure 1-11 Structure of a Computer

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 1/0 (Input/Output), 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 Microprocessor System, or Microcomputer. A microcomputer may be implemented as a single chip (eg the Texas Instruments TMS9940) or as several chips.

Besides inputs and outputs, a computer will need a place in which to store intermediate data (a scratchpad or filing system). Therefore a computer will generally have data memory as well as program memory.

The inputs and outputs, more than anything else, determine what a computer system "looks like[1]' to the user, When the usual peripherals (card reader, visual display unit (VDU), line printer, magnetic tapes, etc) are connected, the system looks like the traditional idea of a computer. But connect motors, actuators, lights, switches, displays and it could be a part of anything from a washing machine to a car, 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 SOFTWARE DEVELOPMENT

Because there is typically a large gap between the task to be performed by the system (eg "control a factory production line") and the instruction set of the computer ("ADD two numbers"), various techniques have been evolved to bridge the gap and make the task of software design and development simpler and faster. Most of these make use of development tools and utilities that are themselves implemented in software. In fact, one of the major advantages gained in moving from a digital electronic to a software implementation is that the design information itself can be manipulated by computer, allowing much of the design and development process to he automated.

The tools of the software engineer are rather more abstract than the screwdriver and the soldering iron. A software engineer will spend much of his time typing information at a keyboard, and looking at results displayed on a screen. However, the keyboard and screen will take on different roles depending on which utility program (which "software tool") is being used at the time. Chapters 2 and 3 of this book describe what is involved in the process of designing and developing software for a microprocessor system, and the tools and procedures used. Chapter 4 describes some of the principles of software design, and the modern techniques of software engineering which have been developed to make complex software systems manageable.

A high level language (see Sections 2.6 and 3.5) allows the software designer to make strategic decisions about what the system will do, while the compiler determines the tactics to be employed by the computer (memory addresses, storage allocation and other "housekeeping" functions that have to be performed thousands of times a second). The compiler is a software utility that translates high level language programs into the detailed machine instructions required by the computer.

In effect, a high level language provides a more powerful

computer that can deal with most of its internal functions automatically, allowing the software designer to concentrate on the application problem to be solved,

Component Software supplies further assistance by permitting complete pre-written software packages, designed to implement whole areas of an application, Chapter 5 describes Component Software in detail. This chapter also describes concurrency, which is a powerful technique for designing software systems which have to perform a number of different tasks simultaneously(as is often required in real systems).

Early programming languages performed their task imperfectly, and were often designed simply as extended versions of the instruction set of a particular computer, Modern languages, with the benefit of twa decades of research on the requirements for specifying and solving application problems, come much closer to the ideal of requiring nothing more than a complete and unambiguous specification of what is to be done (an algorithm) in order to produce an executable program. One of the best and most successful of the modern languages is Pascal. Chapter 6 describes the Microprocessor Pascal language.

Pascal is a professional programmer's tool, designed to produce reliable systems and yet to give full flexibility for implementing complex applications, For users who do not wish to become professional programmers, but who need to write occasional programs in the course of their work, BASIC may be an acceptable alternative, BASIC is a simple language that can be learned in a few hours and is exceptionally easy to use. Chapter 7 describes Texas Instruments' implementation of Power BASIC.

For those who wish to understand the machine architecture of the TMS 9900/99000 family, or to program directly in the instruction set of the microprocessor, Chapter 8 describes 9900/99000 assembly language, Assembly language programming requires more detailed knowledge and there is more risk of error than when using a high level language, However, assembly language programming allows the designer to squeeze the last ounce of performance out of the machine, and may be especially useful in critical areas of a software design.

1.5 GENERAL PURPOSE COMPUTERS

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 set of standard peripherals - devices which enable data to be input to and output from the computer. A typical configuration might look something like this:

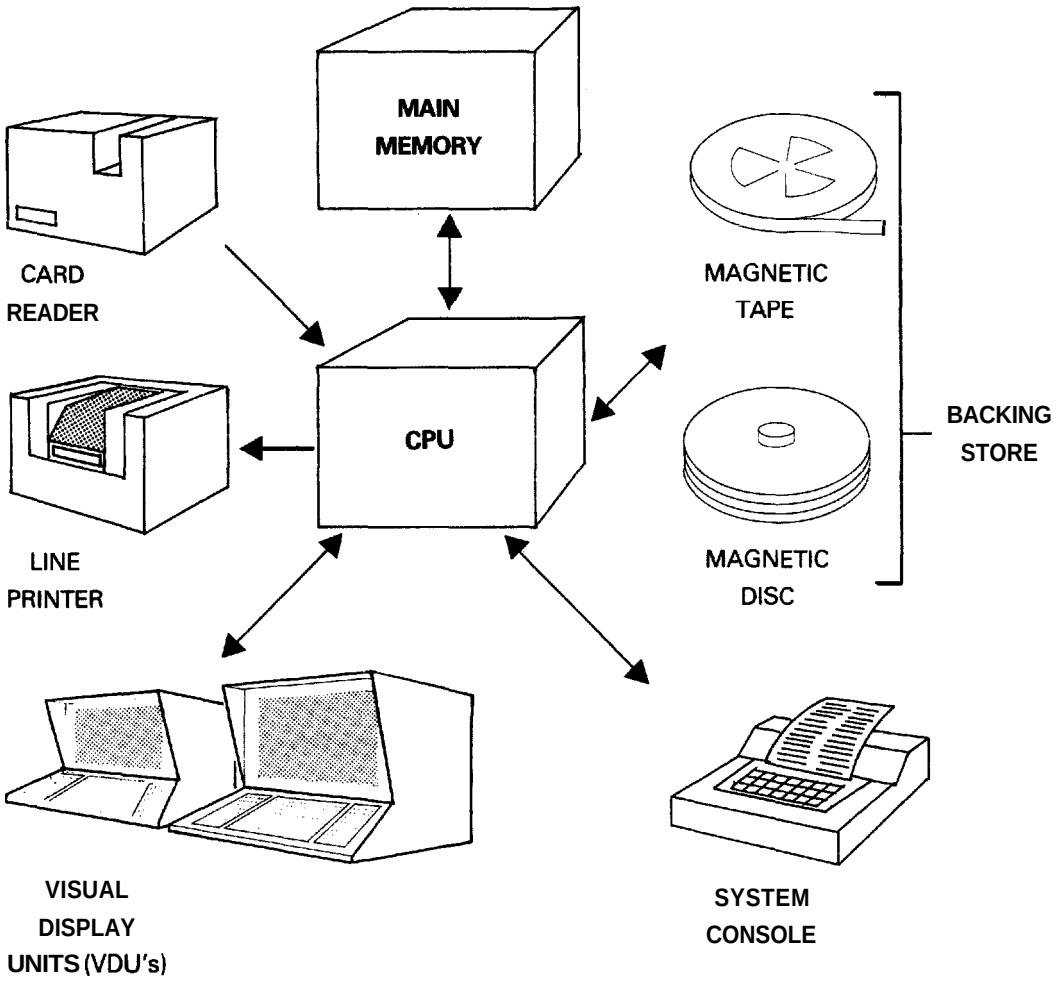


Figure 1-12 A General Purpose Computer

The input and output to a computer of this type is likely to be entirely textual or numeric information (customer files, order details, scientific results etc), and the work that it does is entirely information processing or data processing (DP for short). Human beings always act as buffers to this kind of system - preparing textual or numeric input data in the form of punched cards or keyboard input, and interpreting or acting on printed results or reports.

One of the most important peripherals is the backing store, This is a memory device that is slower acting 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 generai 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. Others are application programs developed by the user for his particular needs.

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 operator, This program is called the Operating System (OS) 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 application programs. They can be called in as required by the Operating System.

A general purpose computer is, therefore, a chameleon-like device which can perform any processing function depending on the application program which is loaded into it. However, the range of things it can do is limited by its input and output devices. 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.

1.6 DEDICATED COMPUTERS

A microcomputer can be constructed as a general purpose computer. But the microcomputer has brought a new possibility: the dedicated system. A dedicated microcomputer might look like this:

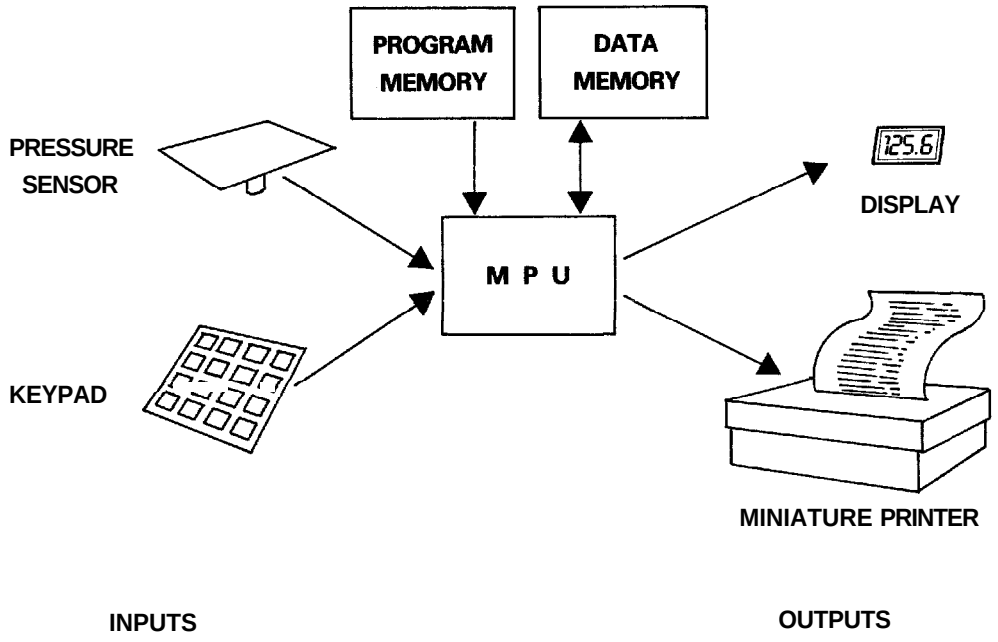


Figure **1–13** A Dedicated Microcomputer

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 term that is often applied to dedicated computer applications is real time. "Real time" means that the computer is responding to and controlling events as they are happening. Unlike a DP system, which provides huge processing power but at a considerable remove from real physical events, a real time system must respond immediately. It will often need to respond within milliseconds or less.

Dedicated microcomputers often have an executive rather than an Operating System. While an Operating System is likely to be a large, all-inclusive piece of software, an executive is more likely to be a set of service functions selected for the particular application, and occupying very little memory space. The program for a dedicated system may well be

permanently and ineradicably stored in read only memory *(see* below), and the microcomputer may only execute one small set of programs all its life. A dedicated microcomputer may we11 have no backing store from which to load alternative programs.

In the example pictured above, the program would repeatedly check whether or not there was any input 'from the pressure sensor or the keypad. If there was, the portion of the program written to deal with that input would execute.

1.7 ROM AND RAM - SEMICONDUCTOR MEMORY

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

Semiconductor memory systems are typically organized in bytes (1 byte = 8 bits). The TMS 9900/99000 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 special type of program memory, called Read Only Memory (ROM) is commonly used for dedicated microcomputer systems. A ROM memory chip is programmed (ie, loaded with a program) once, during production of 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 often be no backing store from which to load the program when the device is switched on.

1.7.1 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 cheaply. Where large quantities (hundreds of 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, with appropriate development tools, 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 (thousands), 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; the static charge will be retained for a period of 20 years or more. But by exposing it to ultraviolet light for a period of about 20 minutes, the EPROM becomes erased and can be programmed with something different. EPROMs are now commonly used in all medium volume applications, except for very high performance applications where the superior speed of bipolar PROMs is required.

1.7.2 RAM Types

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 a slightly misleading term, since ROM can also be accessed randomly. (Read/Write Memory would be more descriptive, but 'RAM' is at least easier to say.) 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.

Semiconductor RAM is volatile; that is, 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.

1.7.3 ROM/RAM Summary

The characteristics of semiconductor memory are summarised in Table 1-1 below.

	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

Table 1-1. Semiconductor Memory Characteristics

1.8 APPLICATIONS

The microcomputer has accomplished three things:

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

2) It has changed the nature of conventional computer systems

3) It has made possible a completely new range of applications, for which the new technology of microsystems is uniquely suited.

There is virtually no electrical device within which a microcomputer cannot be incorporated, providing cheap but sophisticated control, and powerful processing capability.

Many applications previously performed by large general purpose computers ('mainframes') can now be carried out more effectively by microprocessor systems, located at the point where they are needed rather than isolated in a remote data processing department.

With the arrival of the minicomputer several years ago, the death of the mainframe was predicted. That death sentence was premature. Rut a 'mainframe' is no longer likely to be a solitary monolith, isolated within a data processing department. It is more likely to fulfil a specialised need for central data storage or massive processing power, within a network incoporating microcomputers, minicomputers and possibly other mainframes too.

Computer power now comes in sufficient shapes and sizes (and prices) that it can be distributed anywhere that there is a need for it. Large computer systems look less and less like traditional computers and more like communications networks, with processors judiciously placed at appropriate points in the network. The microcomputer allows the distribution of computing power to the place where it is needed - the office, the factory floor, or 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 already exists. Such devices can do away with the tedious manual process of data preparation.

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, and can be iinked to a network for reference to central files, and for handling large processing tasks.

The development of 'personal' computers and small business systems allows a further stage of development. 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 network.

In future, there are likely to be a number of imaginative applications linking the power of the microprocessor with rapidly deveioping communications technology. Viewdata is an example that makes use of television, telecommunications and processor technology. This is a public computer network which can be accessed by anyone with the right equipment (an adapted TV set) via the telephone network. It provides information and services, and can even be used to transmit software to a subscriber's computer.

The development of local area networks will allow separate computing devices to be connected together simply and straightforwardly, to build distributed systems for office, factory and even home environments. Fibre optics technology promises a cheap, reliable and interference-free communication medium.

The automation of industrial processes was first made possible by minicomputers, which were general purpose computers small and cheap enough that they could be placed in a factory or chemical plant and used to provide some degree of automatic control. However, such computers still typically required a room to themselves.

Microcomputers are small and cheap enough to be incorporated in individual machines, and to be distributed across the factory floor wherever control functions or processing power are required. Cheap, fast microprocessors make robots of all kinds technically and economically feasible. Robots can be used to construct flexible manufacturing systems, which can provide the advantages of mass production in the manufacture of small quantities of diverse products.

Microcomputer applications range from simple real time control functions (such as a weighing scale) to production control systems and sophisticated computer networks. In 'real-time' applications the computer is in direct control of a process, event, or phenomenon such as engine control - monitoring electronic ignition timing and fuel mixing, for example, and modifying the physical parameters while the process is taking place. Real time applications can be on a small scale, or could involve control of (say) a complete

chemical plant. The TMS 9900/99000 family is particularly suited for real time and control applications. It has a fast context switch to implement multiprocessing and modular programs, and a flexible bit-oriented method of input and output (the architecture of the 9900/99000 family is described in Chapter 8).

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.

1.9 FUTURE DEVELOPMENTS

With microcomputers cheap and readily available, there is no need for systems to be restricted to a single processor. Groups of cooperating processors, each with its own software and possibly local input and output, can implement powerful and reliable systems.

A significant development in this regard is the Electronic Function Package (EFP).

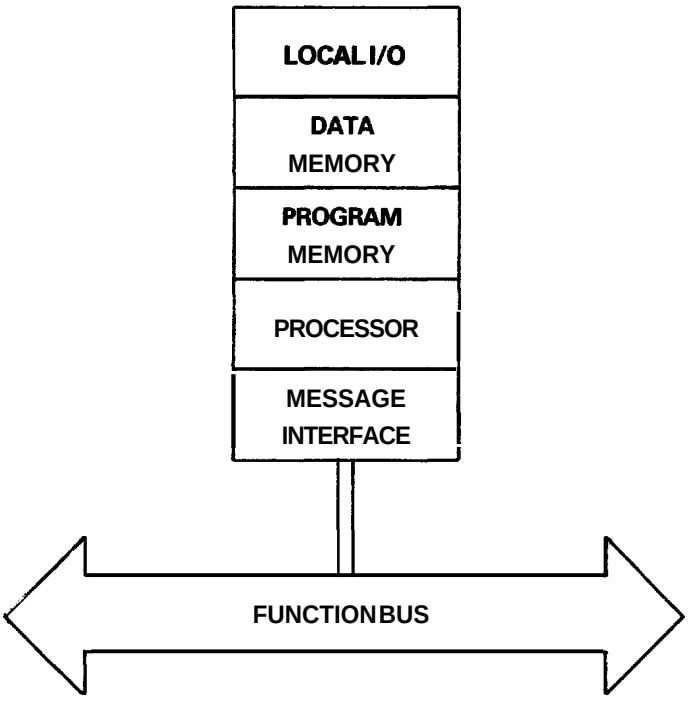


Figure 1-14 Electronic Function Package

Each package encapsulates a local processor with program and data memory, I/O, and a standard functional interface to

other packages. The first implementation of such a package will be as a complete circuit board; but miniaturisation will quickly reduce the size and cost of such packages. Developments in hardware and software will make such packages easy to construct, and easy to connect together into appiication systems. Such packages are likely to be common components in tomorrow's systems.

Speeds of microcomputer devices are likely to increase significantly over the next decade, so that many new applications, including real time signal processing, wiii become possible. Among other things, real time processing and storage of speech, audio and even video signals is likely to become a reality, all at reasonable cost. The scope for new products and applications is considerable.

CHAPTER 2

SOFTWARE DEVELOPMENT

2.1 THE SOFTWARE DEVELOPMENT PROCESS

This chapter gives an overview of the steps required to design and implement software for a microprocessor system.

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 requires several stages of development:

(1) Functional. Specification

(2) System Design

(3) Software Design (and, in parallel, hardware design)

(4) Programming (ie entering the software design in precisely coded source program statements on a development computer system)

(5) Translation of the source program (in a human-readable programming language) into binary machine code

(6) Configuration and linking of the software

(7) Debugging the software

(8) Integration and testing of hardware and software

(9) Evaluation of 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 more like Figure 2-1:

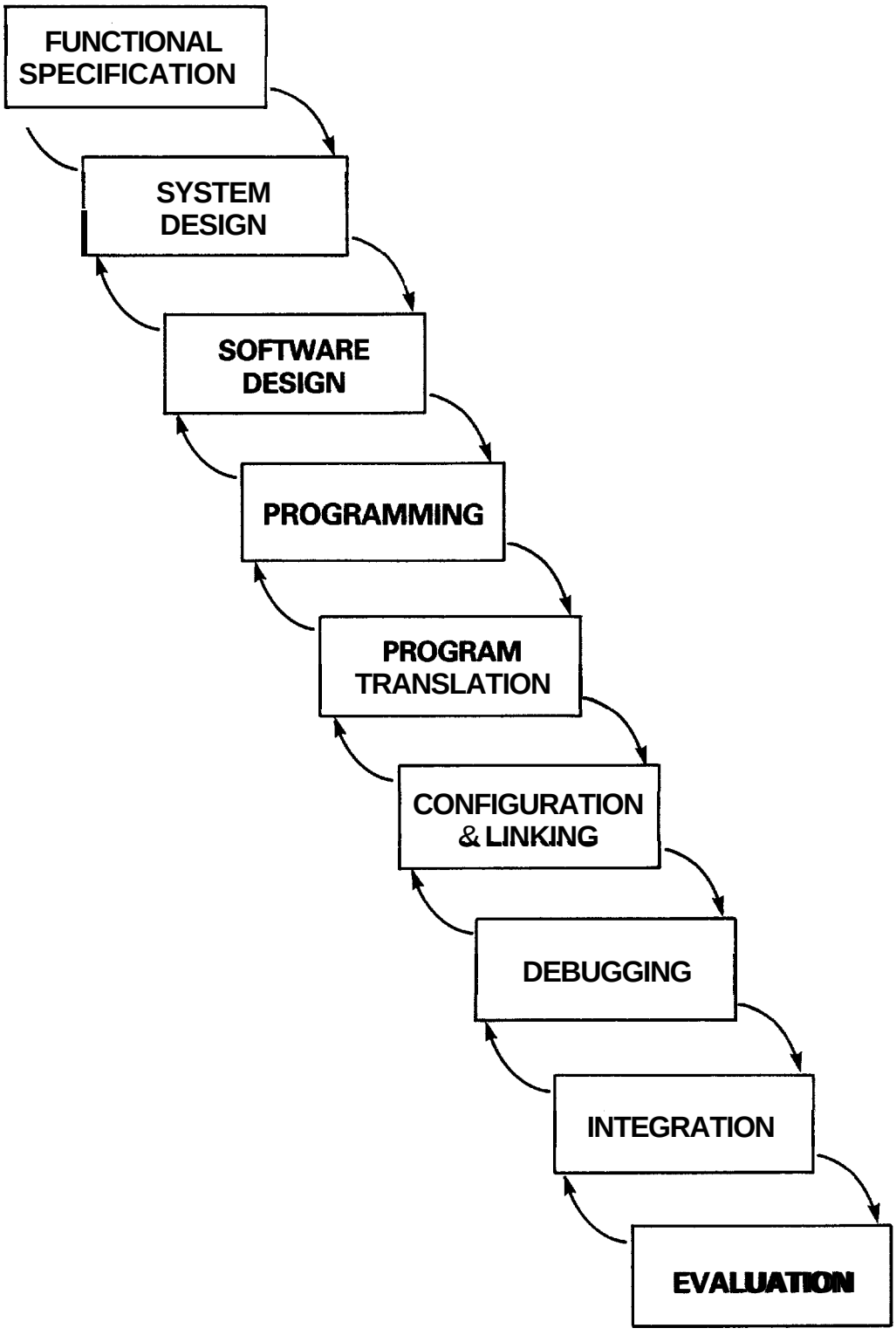


Figure 2-1 The Software Development Process

2.2 FUNCTIONAL SPECIFICATION

Functional specification is where product requirements and implementation technology meet, It is the first, and most important, stage in developing any system.

A good functional specification will take account of the spectrum of possible market requirements, and the range of possible implementation techniques, and derive a "best fit" solution, Characteristic of a good functional specification is that it can accomodate a degree of change both in product requirements and in implementation technology,

As both types of change are likely to happen during the development phase of a product, it is worth spending a good deal of time (perhaps 30 per cent of the total project effort) to derive the best possible functional specification. Microprocessor technology, software and hardware, means that implementation from a well defined functional specification is fast and straightforward. Surprisingly, the major cause of delays, problems, and ultimately project failure is inadequate specification,

The task of specification is to isolate and identify, from a general appreciation of what is required, precise definitions of the functions to be performed, Fast developing technology, and rapidly changing markets and user requirements, dictates collaboration between experts in the area of application and engineers with knowledge of the technology (software and hardware),

Microprocessors can replace more conventional technology - for example digital logic - in existing applications, but there are other possibilities, Software is a medium that can be engineered in the same way as hardware, If it is managed correctly, software development can be done much more cheaply, more quickly and more flexibly than developing custom hardware, Software functions can provide "intelligent" control, information processing, and flexible operator interaction, With software it is possible to construct "working models" that can be tried out, adapted, tested and finally "frozen" in silicon memory chips for use in a production system.

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 can be given some degree of programmability, to allow the user to set up a series of operations to be performed unattended. New possibilities are introduced simply by using a microprocessor.

A full functional specification 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. Usually, this can be done by defining a number of distinct functions that the product is required to perform - operator interface, data storage, machine control, report generation etc. The major functions must be identified, their operation specified and their interaction detailed. If the different functions are clearly isolated and well defined, they can be implemented straightforwardly as separate "packages". Some functions may be implemented directly using standard hardware and software components.

Writing the functional specification requires some understanding of what is possible with microprocessor systems, as well as what is required by users. Functional specification cannot be completely isolated from system design, which considers some of the "how" of implementation. Several passes through the functional specification/system design cycle may be needed before an acceptable solution is produced.

Nevertheless, the functional specification should be maintained as a separate document, which does not describe any of the "how". The functional specification is the interface between market (or user) requirements and implementation technology; changes in either can be incorporated in the functional specification and their implications worked through. Functional specifications can be written in a language that both engineers and marketing executives (or users) can understand. Other types of

specification may be incomprehensible to one or the other. With both market requirements and technology changing month to month, this channel of communication is essential.

2.3 SYSTEM DESIGN

The purpose of system design is to derive from the what of specification, a how that describes an implementation strategy. The system designer must decide how to integrate hardware and software, whether any special interfaces are required, if any special hardware is needed (for analog to digital conversion, for instance), and so on. System design must specify how each function is to be performed - in software, hardware or a combination of both, and with what mix of standard and custom-developed components.

The first step is to identify whether standard hardware or software packages can be used for any of the functions identified. An existing custom IC designed for a particular function (eg control of a floppy disc) brings increased performance and, usually, cheapness. A standard Component Software package gives tremendous savings in development cost and time, plus reliability. Unlike hardware components, Component Software can also be tailored to meet very precise application needs (see Chapter 5).

Having eliminated those parts of the system to be implemented with standard components, attention can be turned to the other functions required. System design requires an appreciation of the characteristics of hardware and software, and how they fit together. Often a function (say, signal averaging) can he performed in either hardware or software. Strictly, the comparison is between :dedicated hardware, and general purpose hardware (eg a microprocessor) plus custom software. The advantages of a software implementation are flexibility, fast development time and low development cost. The general equation governing microsystems production is:

$$\text{unit cost} = \text{material, labour, overheads} + \frac{\text{development cost}}{\text{no of items}}$$

For products which will be produced in large quantities, development cost is of no importance: where a product is to be mass produced in tens or hundreds of thousands, development of a custom integrated circuit is justified. As the number of products to be produced falls, development cost becomes more and more important. For systems to be produced in small quantities (say 1 - 100 per year) the cost of development dominates all consideration of material costs. Microsystems technology (in particular software

technology) allows the tremendous advances in integrated circuit technology to be applied to areas where a custom chip design could not be justified. It does so by dramatically lowering the cost of development for a product.

Other considerations may apply: a microprocessor is already present in a product and has spare capacity, it makes sense to use it to "mop up" as much as possible of the logic. Some functions may require custom hardware for speed reasons. Again, there are functions, such as complex calculations, that simply cannot be performed economically in hardware.

However, software is not just directed to solving problems of cost. Software also gives flexibility that, in some applications, can be crucial. Whereas changing a hardware design requires, probably, manufacture of a new printed circuit board, a software program can be changed by typing the modification at a keyboard, executing one or two automatic software utilities (a matter of minutes), and programming a new EPROM. Engineering changes can be made in days rather than weeks or months (assuming the use of PROMs or EPROMs rather than mask ROMs).

Modern techniques are integrating software and hardware in new ways, and giving the system designer an expanding range of choices. TI's Function to Function Architecture (FFA) is directed to defining a common set of rules for the interaction of complex functions, whether implemented in hardware, software or a combination. In future systems, it will be possible to choose the appropriate mix of hardware and software (and a wide range of corresponding standard components) for every function in a system.

A well thought out system design, with adequate appreciation of functional divisions, will make possible relatively painless evolution of today's systems to make use of advanced functional components. Functions can be replaced incrementally, to incorporate new components and new application requirements, without requiring major redesign of the whole system. Chapter 5, Component Software, gives more details of the functional approach to system design.

The end result of system design should be a specification of how each function is to be implemented, and a precise definition of the interface between functions. System design should specify all hardware/software interaction (eg the configuration of all I/O devices), so that hardware and software design can proceed independently.

2.3.1 Documentation

It is important to keep a record of the design process. Notes, and formal documents such as specifications, can be collected together to form a project notebook. Some part of this can usefully be an "electronic notebook". Documents stored in files on a development system computer (see Section 3.3) can easily be kept up to date, and printed copies can be obtained when required. This is an ideal medium for specifications.

The project notebook should record design decisions taken. For example, an analog input (a voltage, for example) may be required. Decisions to be taken include:

(1) How much precision (ie, how many bits) is required

(2) Bow 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

Hardware/software trade-offs can also be recorded in the notebook. 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?

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.

If the situation changes (eg new technology becomes available), a comprehensive project notebook makes it much easier to backtrack and discover for what reasons the original decisions were made, and whether they are still valid.

2.4 HARDWARE DESIGN

This section describes some aspects of hardware design which affect and are affected by software.

In many applications, it makes sense to regard the hardware of a system as resources, to be controlled by the software. This implies an approach that is different from designing a purely hardware system.

Much of the design effort consists simply of interfacing the outside world (the inputs and outputs) to the microprocessor system bus.

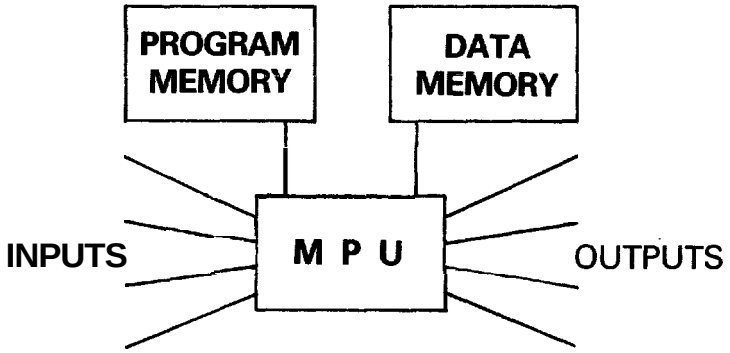


Figure 2-2 Hardware Design for a Microprocessor System

What must be presented to the bus is a control interface. The software will only have access to those signals which are connected to the bus.

The design decision which must be taken when constructing each 1/0 interface is "how much control and information is to be given to the software?". The answer will be based on

(a) the decisions taken at the system design stage on what is to be implemented in software and what in hardware
(b) how much flexibility is required in the design.

Where software access is provided, design changes can be made simply by reprogramming rather than redesigning the hardware. Extra software control signals may be provided for this reason, particularly at an early stage of the design.

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 and TM990/Euroboard 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 additional functions.

2.4.1 Estimating System Load

A single microprocessor can do only one thing at a time. If it is required to perform several functions in parallel (as a real time system usuaiiy is) it must do so by tackiing each one in turn, sufficiently fast that every one is performed within the required time. An important part of specification is defining how fast and how often the microprocessor needs to perform each function. (For example: an anaiog 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 hardware design is to determine that the processor can meet these specifications.

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

$$\frac{\text{Processor Time}}{\text{Rea1-Time}}$$

For a given task, the load on the system is the processor time needed to perform the task, divided by how often the task must be performed. 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.

An estimate of 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 (naively) that processor 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 system

(3) Use a more powerful processor

(4) Add another processor

If the system load comes out very low (less than 1 per cent, for example) this need not be a bad thing, if 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 in many applications to have them lying idle for much 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: unless carefully controlled, designs expand to fill 150 percent of the resources available. To avoid this, the designer needs to evaluate carefully the effect of proposed enhancements, and consider them in relation to his loading estimates - which can be checked experimentally once the design is built.

2.4.2 Memory Size

Naturally, one important consideration when designing the hardware for a system is how much memory space to allow, The only way to estimate memory size is to break a system down into software packages and estimate the size for each, based on existing packages, If the software designer making the estimate lacks confidence in his figures, then the packages should be broken down still further and, perhaps, parts of them trial coded,

Whatever the figure arrived at, the hardware designer should allow a sizeable margin for expansion; first, because no-one has yet found a completely reliable method for estimating the final size of a software package, and second because of the previously mentioned tendency for 'creeping enhancements'. It is usually much easier to cut down an over-designed prototype version when producing a production model, than to add significant memory space not foreseen in the original design. The size of each software package can be monitored as it is produced and compared with the original estimate, to give a progressively better picture of the final memory size,

2.5 SOFTWARE DESIGN

Software design consists of turning the specification of each function the processor is to perform into precise software algorithms (ie step by step procedures for performing the desired function) and data structures. This is not yet programming, which occurs at a more detailed level. Starting to program too early, before a software design strategy has been worked out, will lead to a design that is incoherent and badly structured, At least a third of the software development effort should be spent on design, to establish the overall structure of the software before starting on the details,

Software design should identify:

(1) The data structures to be used

(2) The routines and algorithms to be written

(3) How the different parts of the software will work together.

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 or 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 concepts internal to the system of which a record must be kept.

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.

Older 'high level' languages such as FORTRAN, and low level assembly language, provide no means of grouping and structuring basic items of data to form more complex entities. Any such grouping that is done must be done inside the programmer's head. Newer languages such as Pascal provide, within the language itself, powerful means of building complex data entities out of simple ones. This means that complex software systems can be built up that model the outside world, and real operations, with surprising accuracy. A single data structure, for example, referred to by a single name, may contain all the information that needs to be known about a chemical process, or the operation of a machine. This data structure may be passed as a single item to a routine that performs a complex operation - say, shutting down the chemical reaction or using the machine to manufacture a part for a motor, The data structures establish a basis - an abstract model of the "real world" - from which program algorithms can be developed to perform various useful tasks. The real time structure of Microprocessor Pascal and Component Software also makes it possible to define and group complex operations, "packaging" a group of concurrent, closely interacting operations, together with the data they operate on as a single, higher level function.

The process of software design is considered in detail in Chapter 4.

2.6 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 a Pascal program 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 automatically by a software utility - 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 was designed for simplicity and is not as powerful as Pascal. It does not provide all the constructs required for reliable software design 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 7. 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, in small, high volume applications). 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. Later chapters of this book describe each language in more detail.

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.

Programs may be written on paper and then entered into the development system, or they may be written directly at the computer. The second method offers many advantages - no duplication of effort, easy modification of the program, and an immediate printed record if required. The development system acts, in effect, as an electronic notebook - faithfully 'recording the program as it develops, and also checking periodically that the programmer has followed the rules sf the programming language.

The programmer uses a software tool called an editor (see Section 3.4) to enter and modify his program on the development system. A structured high level language like Pascal makes it easy to build up a program as it develops in the mind of the programmer. The Microprocessor Pascal System (Chapter 6) includes a syntax-checking editor, which will point out language errors for immediate correction on the screen, during an edit session.

2.7 PROGRAM TRANSLATION

The source program, which is in a programming language, must be translated into machine executable form - that is, a pattern of binary 0's and 1's corresponding to the microprocessor's instruction set.

This is done by software tools called compilers and linkers (see Sections 3.5.5, 3.6). The process of translation from human-readable to machine-executable form is almost entirely automatic, and takes only a few minutes. It will usually need to be done several times, as the programmer corrects errors in his program by changing the source program code and re-translating.

Two types of error can arise:

(1) Language errors. If what the programmer writes does not conform to the rules of the programming language, the compiler or assembler will give an appropriate error

message, and the error can be corrected immediately.

(2) Logical errors. If there is an error in the logic of the program, this may not be found until the software is tested.

To minimise frustration and development bottlenecks, it is important that compilers and assemblers can be called up simply and directly from the development system keyboard, and that they execute quickly.

2.8 CONFIGURATION AND LINKING

Most software systems are written not as one large piece of software, but as several smaller packages. Smaller programs are much easier to manage, and take less time to translate.

This means that the pieces must be welded together into one complete system before they can be used. Configuration is the process of selecting the pieces of software required for an application (perhaps from a "library" of software parts), taking care of any system-wide considerations (such as how to allocate memory, and what will be the hardware addresses of I/O devices), and linking the pieces together, Configuration is particularly relevant to Component Software systems - see Chapter 5.

The actual forging of the links between software packages is carried out automatically by a software tool called a link editor or a linker (see Section 3.6).

2.9 DEBUGGING

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 development system is often referred as the host system; the final application system is called the target system). 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.

2.9.1 Simulation

To overcome this problem, some means of simulating the target system environment on the development system is required. The Texas Instruments Microprocessor Pascal

System provides a host debugger that permits target system programs to be executed and monitored interactively on the host development system. The debugger builds a "software model" of the target system on the development system. Inputs and outputs can be simulated via operator commands. Program flow can be traced, and data items examined. Using the debugger, the user can examine exactly what goes on when the program is running. A 9900 Simulator is also available to test assembly language programs.

Testing should exercise every possible path through the software, and every possible condition, A good test strategy is to test each software module separately, simulating its interaction with the rest of the system (perhaps writing a test program to provide. suitable inputs and outputs). Modules can then be placed together with confidence that they work in themselves, and the interaction between modules ean then be tested, Without a test plan like this, it is almost impossible to carry out a thorough test.

2.10 HARDWARE INTEGRATION AND EVALUATION

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 by emulation.

2.10.1 Emulation

Using emulation, the software can 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. The microprocessor is removed from the target system and the cable plugged in in its place.

Part way along the cable is a "buffer module" containing a microprocessor and interface circuitry. This microprocessor can execute a program contained in "emulation memory" on the development system. Emulation 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. The development

system can monitor program execution, trace the program flow and stop execution if specified conditions (breakpoints) occur.

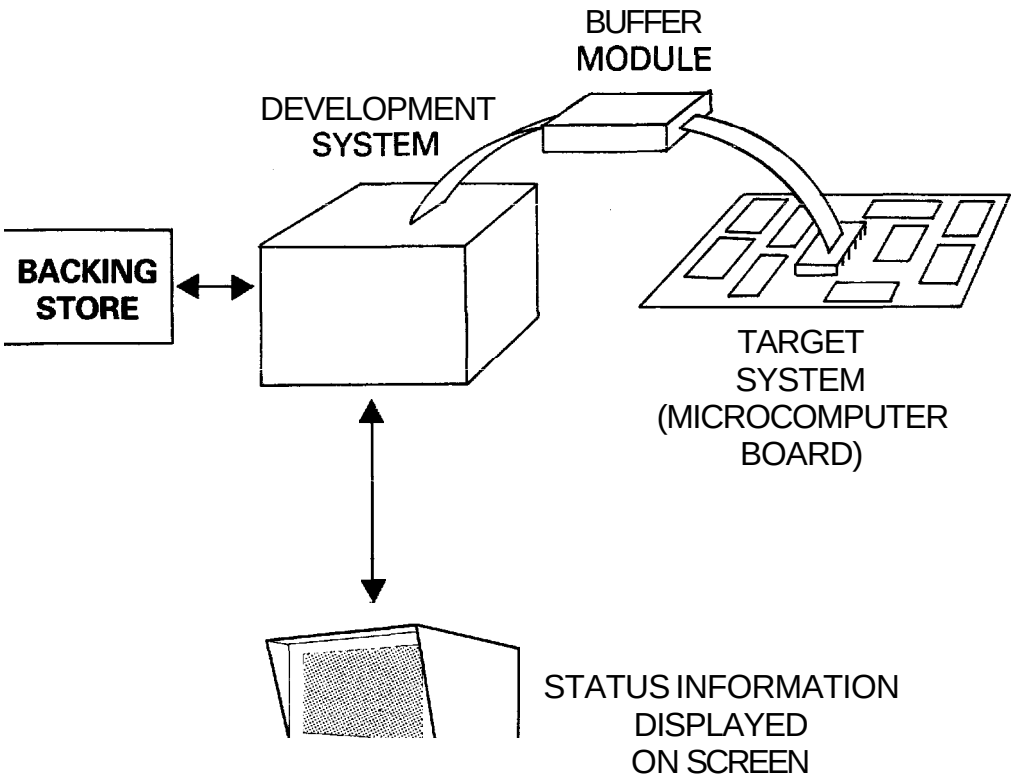


Figure 2-3 Emulation

For Texas Instruments microprocessors, emulation is provided by the AMPL (Advanced Microprocessor Prototyping Laboratory) module. Emulation is controlled by a structured high-level language, in which sophisticated test procedures can be written.

2.10.2 Evaluation

Once the system is working in emulation, the software can be programmed into PROMs and the "umbilical cord" to the development system can be severed. At this stage the device should undergo a thorough evaluation and audit by someone not involved in its development. The designer will have tested the device to the best of his ability, knowing its internal structure and what might be likely to go wrong.

The independent auditor will test without knowledge of the internal workings, according to how the device is likely to be used. This audit should be performed against the original statement of requirements; and it should use (and criticize) the documentation (User's Guide, etc) that is to be provided to the end user.

2.11 PRODUCTION

When 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, this 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.

The software should not be frozen until it has been tested with production hardware. It may be possible to fix minor problems introduced by the move from prototype to production by modifying the software. This will usually be much easier than changing the hardware at this stage.

CHAPTER 3

DEVELOPMENT TOOLS

3.1 OVERVIEW

This chapter describes the hardware and software tools used in software development for microprocessors, and some of the mechanics of software development.

3.2 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. Many dedicated systems will not provide 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 (or host 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 family, 990 minicomputers have the same basic instruction set as the TMS 9900 family of microprocessors, which makes software development much easier. However, it is possible to develop software for a microcomputer on a large mainframe computer, such as an IRM 370.

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 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 development system, it is possible to write a microcomputer program, translate it into machine understandable form (ie 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.

3.3 FILES

Much of the mechanics of program development consists of creating and manipulating files on a development system. A file is a sequential list of information held on a backing storage device (disc, magnetic tape, etc). This information may be text, numbers or binary digits. Files are used to store the source program code that a programmer writes, and to store the machine code that can be executed by the microcomputer. Files can also be used to store documentation, user's guides etc - in fact anything that can he reduced to words, numbers or bits.

Once a design has passed the paper stage, it will consist entirely of files stored on the development system. This medium may be unfamiliar to those used to working with circuit diagrams, printed circuit boards and soldering irons. However, once the basic techniques have been mastered, it is an easy and natural medium to work in. Software tools can manipulate the "stuff" of the design directly, and hence a large part of the design and development process is automated, eliminating repetitive work and enhancing productivity.

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,

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 object code that can be understood by the microprocessor. The source code is read from a file held on backing storage; the object code is written to another file.

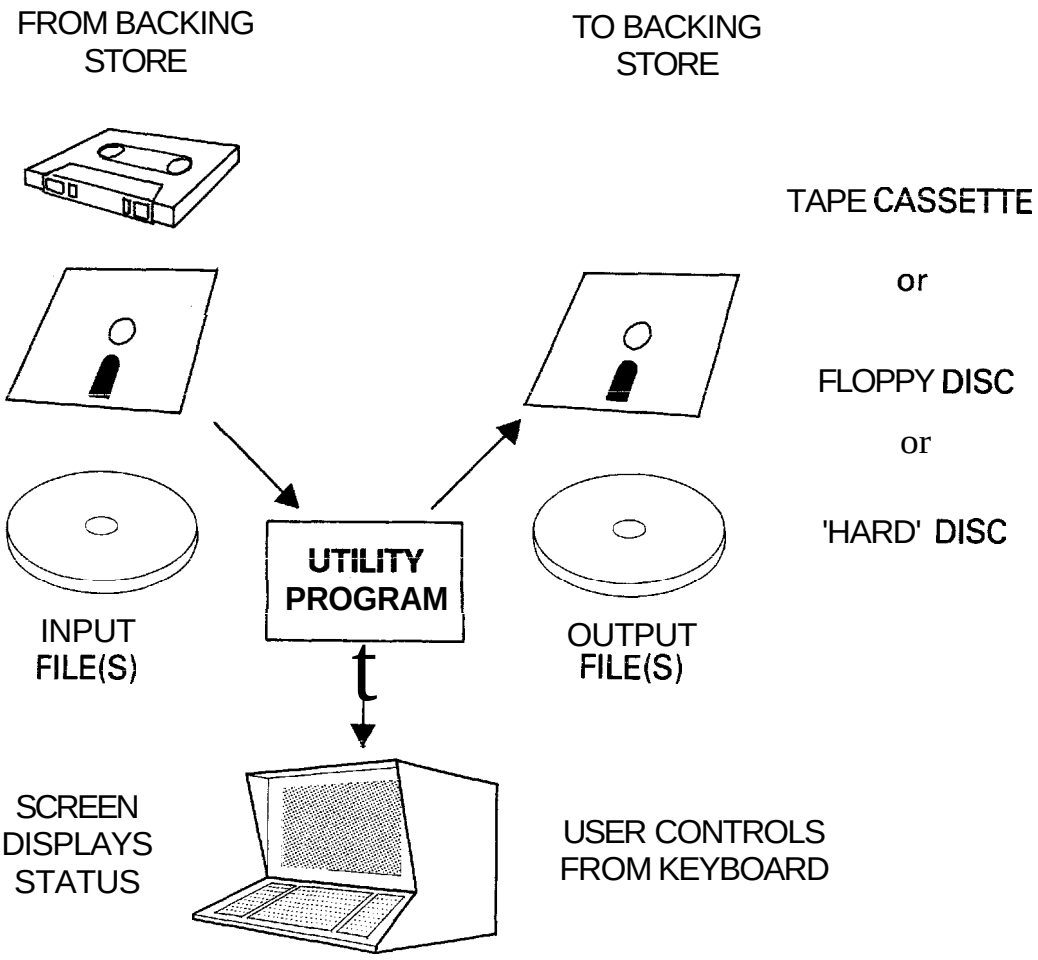


Figure 3–1 Software Tools

These utility programs are the tools of the software engineer; they are what he or she uses to create and manipulate software. A utility program (a 'software tool') 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 typed in at a keyboard.

Files which contain readable text - that is, information that can be understood and manipulated by a programmer - are known as text files. Binary codes are used to represent the individual text characters (see section 3.8).

3.3.1 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:

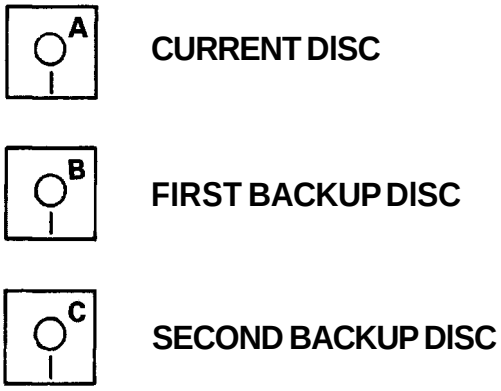


Figure 3-2 Backup Cycle - 1

At regular intervals - say once a week, but depending on how much work 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.

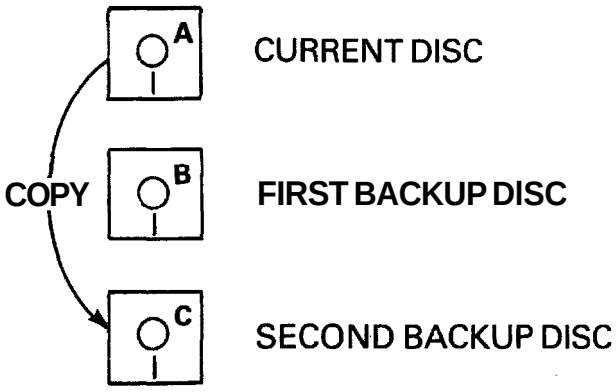


Figure 3-3 Backup Cycle - 2

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:

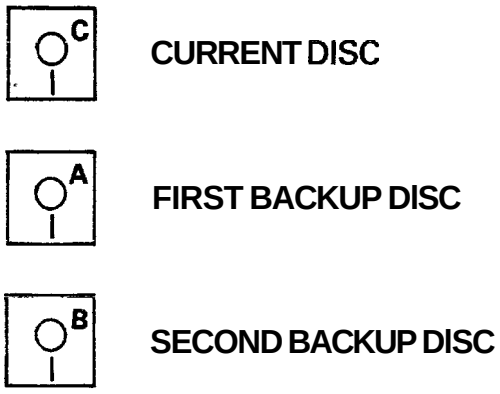


Figure 3–4 Backup Cycle - 3

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.

3.4 Text Editing

The text editor is a program which allows the user to create and manipulate text files. The editor is perhaps the most important tool on the development system. It is the tool which a programmer will spend more time using than any other. So it is important that an editor is well designed, easy to use and has a good set of facilities.

New text is entered at a keyboard, and saved in a file on backup storage (cassette, floppy or hard disc). The text will usually consist of source program code in assembly or high level language; however most editors will allow any

kind of textual information to be entered. The text (whether newly entered or recalled from backing storage) can be modified by entering commands at the keyboard (Figure **3–5).**

Generally the editors which are easiest to use are those which are screen based: that is, the text is displayed on a visual display screen and can be modified by moving a cursor and using simple key strokes to change, insert-or delete characters at appropriate positions (Figure **3–6).**

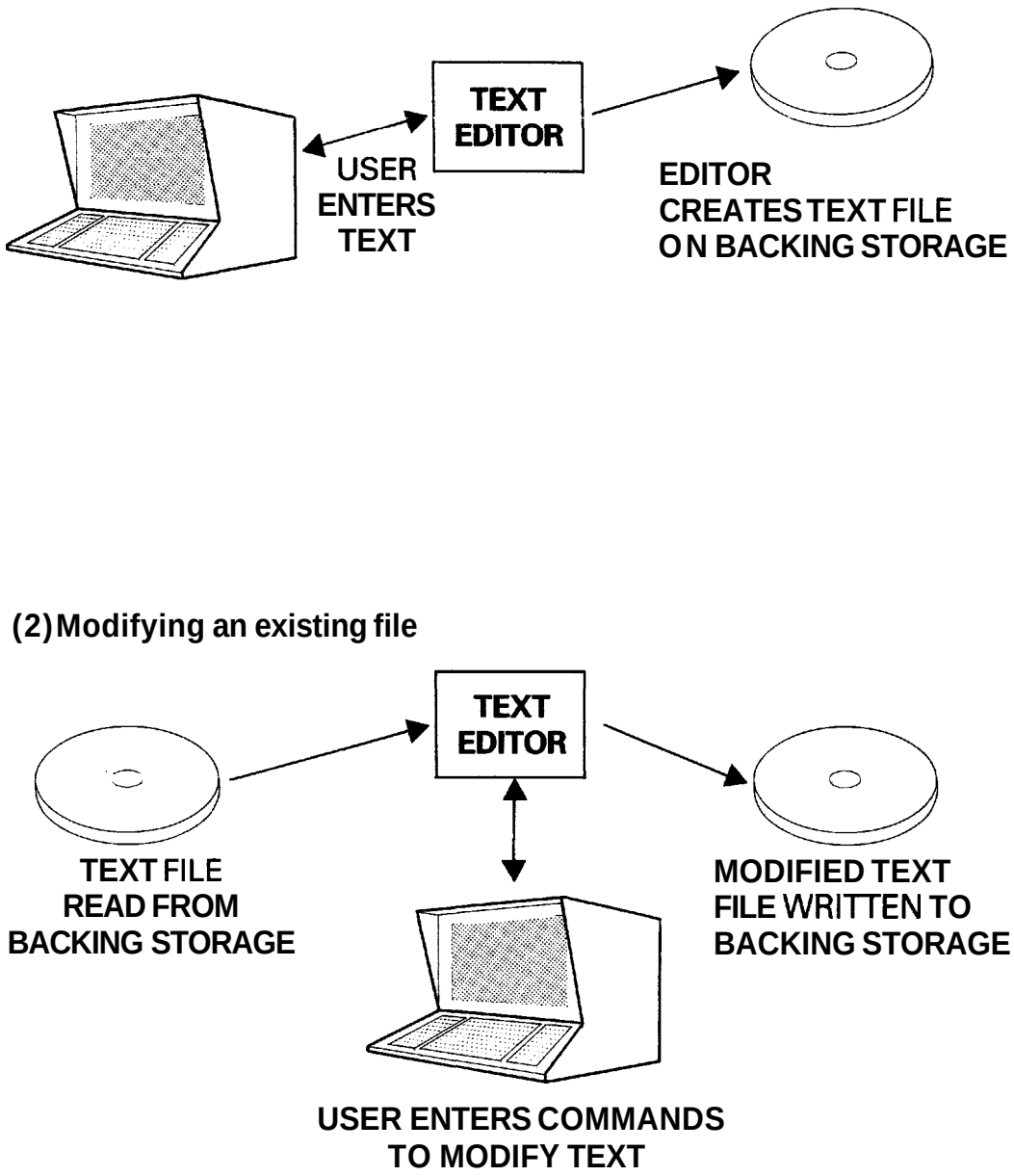


Figure **3–5** Editor Function

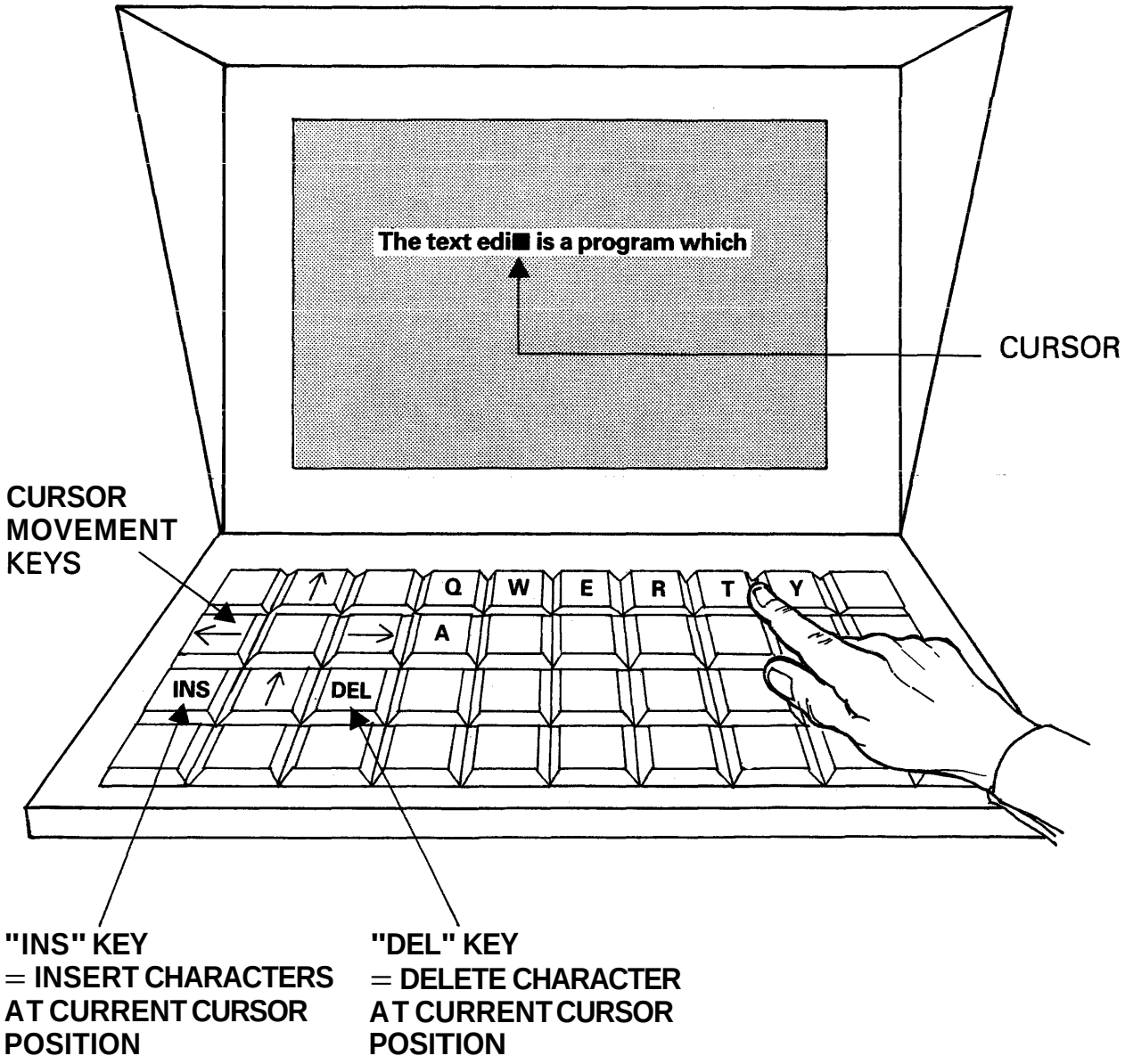


Figure 3-6 Use of a Screen Based Editor

Most editors also provide a repertoire of commands that allow such functions as searching for and replacing specified strings of characters.

```
Commands  *-*-*-*-*-*-*-*-*-*-*-*-*  Commands  *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

ABORT     Exit the editor             BOTTOM    Position cursor at end-of-file
INPUT     Edit another file           TOP       Position cursor at top-of-file
QUIT      Save file & ABORT           +/- int   Position cursor up or down "int"
SAVE      Save file & INPUT
                                      INSERT    Insert a file
CHECK     Check syntax of file        SHOW      Display a file

COPY      Copy the specified block after the current line
DELETE    Delete the specified block
MOVE      Move the specified block after the current line
PUT       Put the specified block into the specified file

FIND(tok,n)              Find the Nth occurrence of tok
REPLACE(tok1,tok2,n)     Replace tok1 with tok2 for n occurrences
TAB(increment)           Set tab increment

*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-* Function Keys *-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
  F1         F2          F4          F5          F6          F7          F8
Roll Up   Roll Down   Duplicate   Start Block   End Block   Edit/Compose   Split

File = INPUT.FILE                                                    Tab = 2
<>
```

Figure 3-7 Microprocessor Pascal Editor 'Menu' of Commands

3.5 PROGRAMMING LANGUAGES

As far as a prqgrammer is concerned, software development consists mainly of manipulating text files stored on a development system. These text files will probably be written in some programming language. A programming language is a precise form of notation that a programmer uses to specify what he requires the microprocessor to do. Software tools are used to translate the program in this form (in which it can be created and worked on by a software engineer) into a form that can be understood and executed by the microprocessor. Together, the language and the software tools form a design system for programming electronic parts.

3.5.1 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,

1010XXXXXXXXXXXX means "add"

The X's carry other information (where the elements to be added can be found, and where to store the result) and may 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 mnemonfcs (e.g., MPP for multiply) instead of binary code for instructions, and symbols instead of numeric addresses for memory locations:

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

3.5.2 Assemblers

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 carried out automatically by a software tool (a computer program) called an assembler.

An assembler converts assembly language source code, which is produced by a programmer, into object code, which can be understood 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.

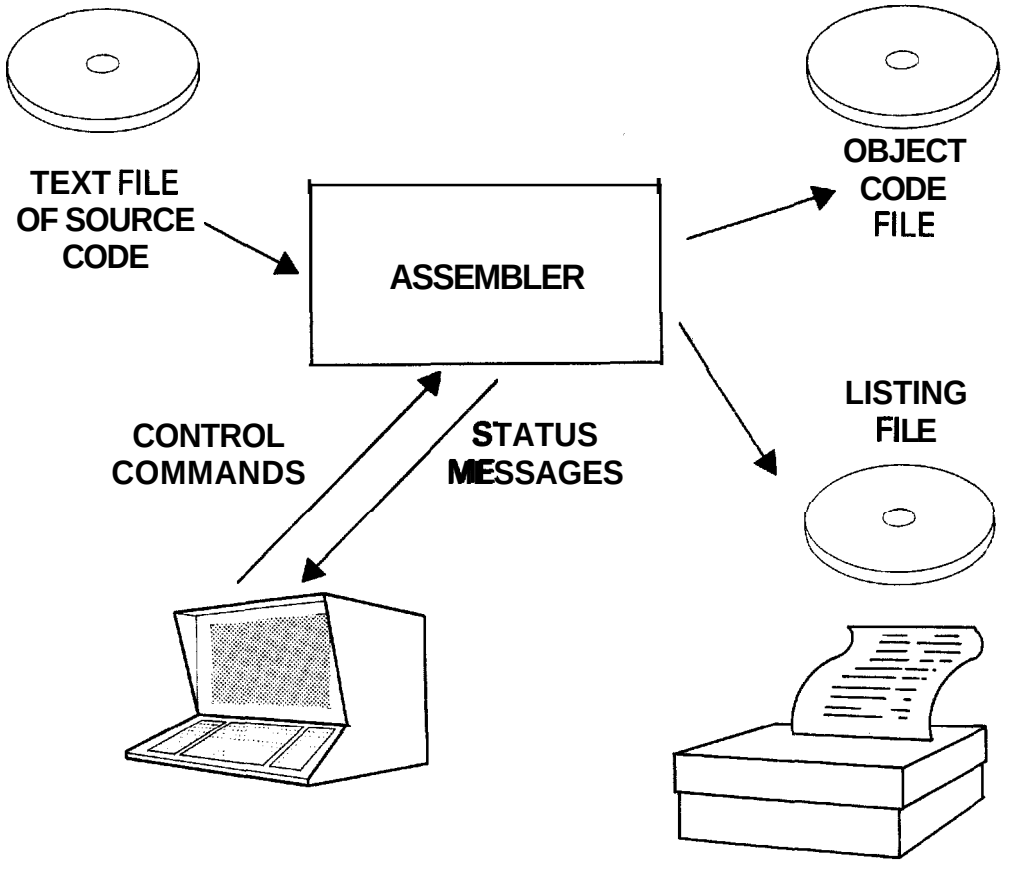


Figure 3-8 Assembler

One of the advantages of using an assembler (instead of programming directly in machine code) 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 (eg) jump addresses, because the position of all the code after the insertion would have changed.

3.5.3 High-Level Languages

Assembly language, though a great improvement on machine code, still requires a problem to be translated into machine terms before it can be programmed. Each assembly language instruction corresponds to one machine instruction.

The programmer must write a statement like

```
IF temperature less than 70 degrees AND
   pressure sensor is off THEN
   notify operator
```

in terms of the low-level tests and conditional jumps that are the only things the computer understands:

```
        CI   @TEMP,70
        JNE  NEXT
        CI   @PRESS,OFF
        JNE  NEXT
        BLWP @NTFYOP
NEXT    .
        .
```

In addition, the programmer must manage all the resources of the computer, such as which memory locations are to be used to store each item of data, himself.

High level languages were introduced to allow 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:

```
10 J = 4
   I = 5*J + 7
   IF (I.EQ.27) THEN GOTO 100
```

The programmer can set up storage locations with names like "I" and "J". I and J are called variables because they can be assigned any value. The first line of the program (labelled 10) sets J to the value of 4. The second line takes the value stored in J (which we know to be 4), multiplies it by 5, adds 7 and assigns the resulting value to I. Line 30 then tests I to see if it has the value 27; if so, the next line to be executed will be the one labelled 100. Otherwise the program continues with the next line in the sequence.

I and J represent memory locations. But the programmer does not have to worry about where in memory they are.

It is much easier to write programs in FORTRAN than in assembly language. However, in some respects FORTRAN is still closer to the way machines operate 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 human, or logical, one.

Implementation of conditional statements, for example, requires GOTO statements and labels. To program "If I is equal to 5 then do X else do Y", it is necessary to write:

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

Not only are the statement numbers an additional confusion and a source of error, but the order is inverted: the then action comes second. FORTRAN was designed simply as an easier and quicker way of writing assembly language programs.

More recently, high-level languages have been designed with the intention of getting as close to the problem as possible. The ideal is that writing a program should require no more than a precise and unambiguous statement of what to do. Everything else (translating this precise statement into code to be understood by a machine, and allocating machine resources) should be done automatically by software tools.

A precise and unambiguous statement of what to do is known as an algorithm, One advantage of this approach is that the algorithms derived are independent of a particular machine architecture, and can survive changes in hardware technology, Many of the newer languages are based on ALGOL (ALGOrithmic Language), which was designed in the 1960s as a natural language for writing algorithms.

3.5.4 Pascal

Pascal is acknowledged as one of the best modern high-level languages, Developed principally by one man, PASCAL has a coherence which some committee-designed languages lack. It implements most of the generally accepted good programming practices, Besides providing the fundamental constructs needed to write algorithms, in a much more natural way than in FORTRAN (say), Pascal also has powerful methods of organizing and structuring data.

Algorithms can be turned directly into Pascal programs with very little effort.

A Pascal program is easy to read, and is almost self-documenting:

```
IF input_value = 5 THEN
  BEGIN
  perform_test_procedure;
  print_results
  END
ELSE
  record_value;
```

perform_test_procedure, print results and record_value will be precisely defined elsewhere-in the program.

3.5.5 Compilers

A compiler performs the same function as an assembler (see section 3.5.2 above), but its input will be a program 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 input to a compiler or assembler is called source code; the output is object code.

Execution of a compiler or an assembler is completely separate from execution of the resulting program. A compiler or assembler is a software tool used during 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.

3.5.6 Interpreted Languages

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. At execution time, the interpreter, a program which resides with the intermediate code 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 examines it to determine what it means, then calls an appropriate piece of assembly language code, contained within the interpreter, to perform the operation.

Intermediate code is much more compact than machine code; however, the interpreter must always he there, whatever the size of the intermediate code, so that there is a minimum overhead in an interpretive system. 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 (typically 5 to 10 times) due to the extra work that must be done at execution time in interpreting the intermediate code.

3.5.6.1 BASIC

BASIC is a simple language which is very easy to learn. BASIC systems also use a very simple set of software tools.

BASIC is especially suited to systems where development and execution are carried out on the same hardware. BASIC systems usually have a special editor, which converts input programs to intermediate code, a line at a time, as they are entered. The BASIC editor checks each line for syntax errors as it is entered, and signals any errors for immediate correction. There is no separate compilation or assembly step; programs can be executed simply by typing "RUN". Programs can be halted and changed, then run again, which makes for very quick, interactive development.

Texas Instruments' Power BASIC (see Chapter 7) is designed to run on the TM990 range of microcomputer boards. A RASIC program can be developed and executed using, at minimum, one TM990 board and a terminal. BASIC provides an inexpensive microcomputer system which is ideal for small applications and experimental work, and can be used by people without computer experience.

However, BASIC does have limitations. Its "line at a time" nature means that there is no adequate program or data structuring, and very limited checks on program correctness. RASIC is not recommended for the development of complex systems.

3.5.6.1 Interpreted Pascal

Microprocessor Pascal programs (see chapter 6) will normally be executed in machine code ("native" code). This gives maximum execution speed. However, they can optionally be executed interpretively. This allows the user to trade-off execution speed against memory size, and to select which is

more important for his particular application. Interpretive execution is slower, but takes less memory.

3.5.7 High-Level vs Low-Level

Faced with the choice of which language is best, some general observations can be made,

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 well as a good assembly language 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. (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,

3.6 Linker

A linker, or link editor, is a program which will combine separately compiled or assembled object code 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,

3.6.1 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 in memory is specified at the time of assembly, and cannot subsequently be changed,

However, most assemblers 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 (Figure 3-9).

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 nodule can be assembled separately without needing to calculate exactly where it will fit in memory - which would involve knowing the lengths of all the ether 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,

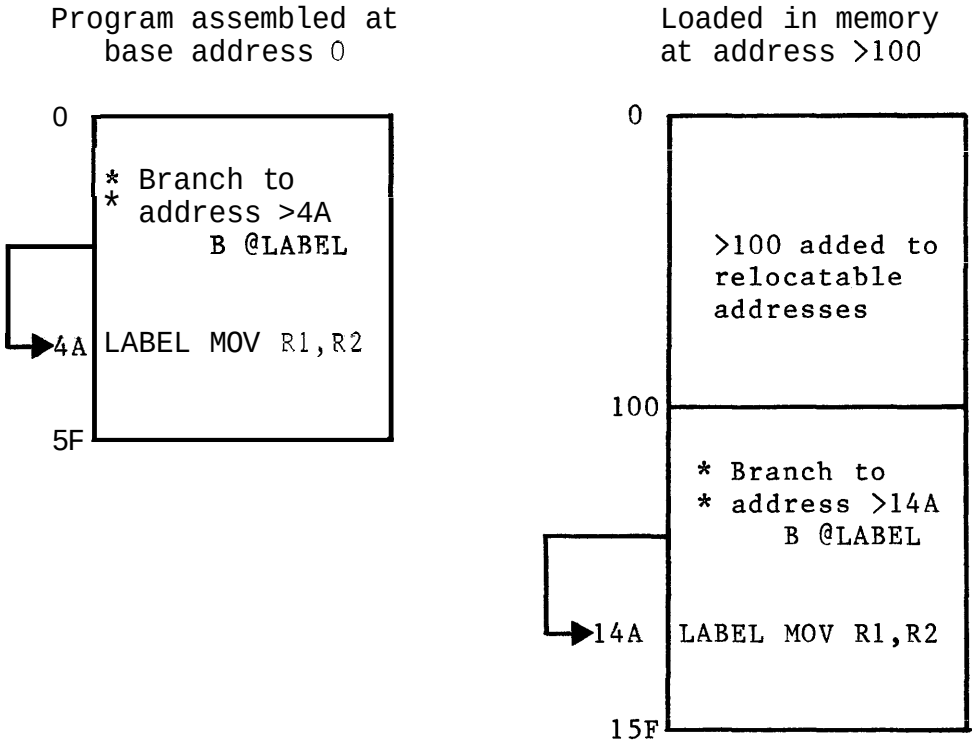


Figure 3-9 Relocatable Code

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 recompiling or reassembling 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. Libraries 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. See Chapter 5, Component Software, for further information on the use of software libraries.

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

3.7 TARGET SYSTEM EXECUTION

Having produced an executable program using the software tools of a development system, there are two ways of transferring the program for execution in the intended target system (a third method, emulation, is described in Chapter 2, section 2.10.1).

3.7.1 Loader

A loader is a software utility that loads an executable program from some form of backing storage into read/write (RAM) memory, for execution by the processor. A loader will therefore be used in a target system which has been designed to execute more than one program, and which has a backing store of some kind (magnetic disc, tape etc) available, However, a loader may also be used in a target system without backing storage, to load a program into RAM memory for test execution. Here, the "backing store" is likely to be a host development system, or a terminal with some form of storage.

Any computer system requires some form of program stored in read only memory that will be executed immediately when the system powers up. In a general purpose computer, this program may do nothing more than load in the Operating System or Control Program from backing store, and then relinquish control. Such a program is called a "bootstrap loader".

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

execute correctly. Other loaders require program code in image format - that is, absolute binary code that can be placed directly in the computer's RAM memory.

3.7.2 PROM Programmer

A dedicated microcomputer is likely to have its program code already stored in read only memory when the system powers up, so that no ioader is required. A utility called a PROM Programmer is used to permanently fix the program into a PROM memory chip which can be plugged into the target system. (In the case of EPROM, the program can be erased again by exposure to ultraviolet light - see Section 1.7, Semiconductor Memory). A PROM Programmer is a peripheral device attached to a microcomputer development system, together with a software utility which takes program files from disc on the development system and feeds them to the peripheral device.

For systems produced in large quantities, mask ROM (Section 1.7) may be used. In this case the developed program will be incorporated into the ROM device during manufacture. However, PROM (Programmable ROM) is likely to be used to prove the final program before it is committed to mask.

3.8 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).

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

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

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.)

Numbers (decimal, hexadecimal or otherwise) can be represented in text as a string of ASCII 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,

* For the hexadecimal number system, see Section 8.13.2.1

CHAPTER 4

SOFTWARE DESIGN

4.1 OVERVIEW

This book cannot present a full description of the software designer's craft. However, the aim of this chapter is to suggest directions and provide a starting point for further investigation. The science of software - particularly real time software - is inexhaustible.

New tools and procedures are gradually automating the "lower levels" of software development and pushing the area where creative engineering is most needed back towards system design and requirements specification. New requirements will always provide scope for innovative and practical engineering solutions.

This chapter is concerned with the design and structuring of software for microcomputer applications. What is presented here is independent of any particular programming language - though much of it is quite close to Pascal, which was designed with the explicit goal of implementing the "universal" elements of a programming language.

Producing an initial language-independent software design has a number of advantages. It allows the overall strategy of the design to be worked out before it becomes cluttered with implementation detail; and it provides a common point of reference that can be returned to when making changes to the system, or if it is desired to implement the sane application using different techniques. For a large project, the initial design can be kept sufficiently simple to be manageable by one man, or a small team. This design specification can then be used to coordinate the efforts of a larger group.

Some languages (eg assembly language, FORTRAN, BASIC) offer no means of developing a high level design strategy without descending to the details of implementation. Here a stylized design language must be used in the initial stages. Using more modern, application-oriented languages such as Pascal, it is possible to develop a high level design in the language itself. Some users may still prefer to use a design language to produce a separately documented design.

4.2 SOFTWARE STRUCTURE

Good structure, both of program and data, makes the difference between a well-ordered, reliable program that is easy to maintain and upgrade, and untidy ("spaghetti") code, with hidden bugs that may not be found until it is too late. Establishing a good structure may mean spending some time on system and software design before going near a keyboard or coding pad, but the time spent is well worth while. Errors not caught at the design stage become ten times more expensive to correct at the programming stage, a hundred times more expensive at final test, and, potentially, thousands of times more expensive when the product is in the field.

Structure is equally important for high level and for assembly language programs, although a good high level language gives much more assistance by supplying pre-defined structural constructs.

This chapter describes the basic principles of modular software design (ie structuring at the level of software/hardware packages and modules), and also some of the 'fine detail' of data structure and program algorithms. An algorithmic design language and a structured graphical notation that can be used for design are introduced. This chapter owes much to the pioneers of modern software engineering techniques, in particular Dahl, Dijkstra, Hoare and Wirth. The graphical notation used in this book was developed by Eric Richards * from a notation devised by Michael Jackson. The references at the end of this chapter provide material for further research.

No accepted standard for a design language yet exists. A suggested notation and standard is introduced in this chapter. Designers who wish to adopt a strict formal notation for software design are recommended to use Pascal. Designs can then be checked automatically for consistency by a suitable Pascal compiler. This approach has been very successfully adopted within the experience of the authors.

The present chapter describes in some detail the basic structuring techniques that are fundamental to modern high level languages. Chapter 5 describes how these have been extended in the Component Software environment to apply to real time microprocessor systems. Chapter 6 describes Texas Instruments' Microprocessor Pascal System.

* Described in an article in the British journal Computing, May 19 1977

4.3 SOFTWARE PACKAGES

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

When adopting this approach, two things must be considered:

(1) The detailed nature of each package

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

To simplify the task of interfacing, packages should be selected to be as self-contained as possible. In other words, the package boundaries should be drawn so that relatively little information needs to be exchanged between packages, compared with the work done within each package.

"Mature" systems, where significant thought and experience has been put into the design, and where the implementation medium is flexible enough not to dictate the system structure, tend to migrate to this condition. However, for a new system, the designer may have to put in considerable thought to ensure that the system structure is appropriate from the start. Where the designer is implementing an existing system in a new way (ie where the application is mature), much of this thought may have been done for him.

Packages should be logically self-contained, each performing a well-defined set of functions. The ways in which each package interfaces with the rest of the system must be clearly defined.

A designer implementing a factory control system, for example, might identify the following packages:

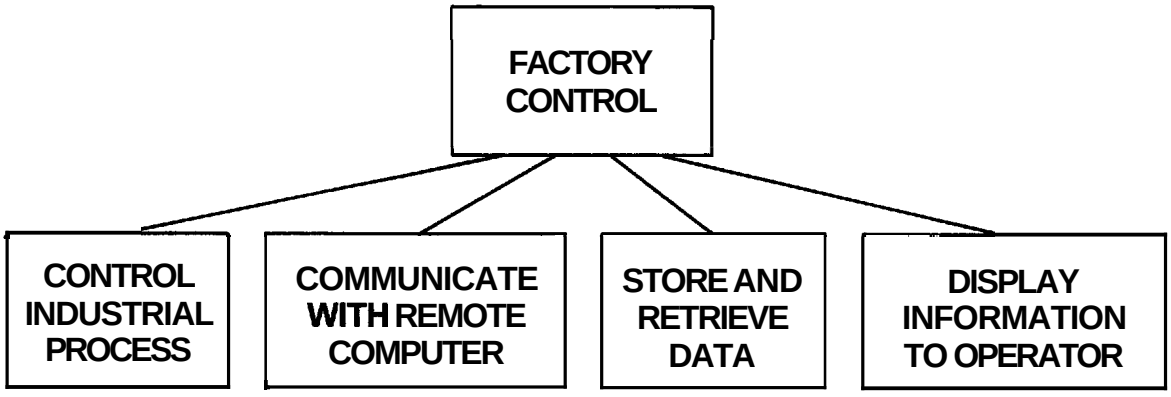


Figure 4-1 Component Packages of a Factory Control System

Each of these packages is still a fairly complex entity, but the problem is beginning to look more manageable.

This analysis identifies the logical components of the system. At this point, it is important to determine the physical distribution - where will each function need to be performed, and what communication paths are necessary? The physical analysis will determine the likely hardware components of the system - where processing capability is required, where physical operations have to be performed, at what points interaction with a human operator is required, and where the communication paths will run. Microsystems technology allows information processing capability (which includes the ability to control things, and the rudiments of an "intelligent" response) to be located wherever it is required.

Although the example described is a factory control system, the same considerations, on an appropriate scale, apply to systems of all types and sizes.

A software package encapsulates a particular type of "intelligence", a control function, or a data processing operation. Many such packages can be specified independently from the hardware environment where they will he used, and some may be available as standard software (see Chapter 5, Component Software). A standard package will usually need to be "configured" into the particular application (analagous to providing a standard socket and circuit elements to interface to an integrated circuit).

Some applications may require little more than selecting standard software packages and configuring them into a final system. However, most applications will require some custom software to be developed.

Each package can in turn be split into successively smaller packages, until the complete problem has been broken down into manageable blocks. At every level in the structure, the packages can be regarded as 'black boxes' that perform clearly specified functions and combine in clearly defined ways. The programmer can focus on a particular part of the design, knowing that he can concentrate on the other parts of the structure at other times.

4.4 DESIGN LANGUAGE

Design language can he compared to the logic diagrams used by circuit designers. As yet there is no universal standard for software design languages, but there are some generally agreed "good practices". The notations used in this and the following sections incorporate the features generally

regarded as useful in software design.

A design language can be regarded as a generalised programming language, with the following characteristics:

(1) Syntax need not be completely rigid, as long as the logic is clearly defined and unambiguous

(2) Operations can be identified by verbal description to start with, and later described precisely - eg "calculate mean"

(3) Only standard, "universal" constructs - sequence, selection, iteration (see below) and standard data structures - are used. Language-dependent constructs are not included.

The aim of the design language is to establish the precise logical structure of the application before proceeding to implementation. In fact the notation described here is very close to the Pascal programming language (see Chapter 6). Pascal was developed as a language that would implement, more or less directly, the features required for software design. It was not designed for any particular machine architecture and hence has a "universal" structure.

It is possible to use Pascal itself as a design language. The advantage of this is that a design can be checked automatically for logical correctness, even if parts of the design are incomplete.

The graphic notation described below provides an alternative notation that implements the same constructs. Either or both can be used during design; sometimes a graphic notation provides a clearer picture, especially in the early stages.

4.5 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. However, writing an algorithm rather than a program liberates the designer from concern with the syntax and details of a particular programming language. An algorithm should be understood by people who are not programming specialists; hence it is very useful when specifying a project.

An algorithm for making tea might be as follows:

```
begin
fill kettle;
put kettle on;
put tea in teapot;
wait for kettle to boil;
fill teapot;
delay 5 minutes;
for number of cups required do
    pour cup
end
```

Figure 4-2 Tea Making Algorithm

Two things can be identified in this (or any) algorithm. First, there are the fundamental operations (fill kettle, pour cup etc). Second, there are the control structures which dictate if and when these operations are to be performed. These control structures are identified by underlined keywords:

```
begin . . end
if ... then ... else
for ... do ...
while ... do ...

       etc
```

It is the control structures that provide the power of an algorithm, and of a computer program. Algorithms can specify alternative or repeated operations, provided the conditions that determine the different actions are specified completely and precisely. The algorithm enumerates all possible options, and specifies exactly how to take every decision. This is what is required to write a computer program.

The individual operations described in Figure 4-2 can themselves be analyzed into algorithms. For example, 'pour cup' :

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

Figure 4-3 "Pour cup" Algorithm

By combining the control structures shown here, extremely powerful algorithms can be developed to control, for example, a complex scientific instrument or an industrial process.

It is possible to define many different control structures, 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 examples.

4.5.1 Sequence

A sequence is simply a list of operations carried out one after the other, in order:

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

The keywords "begin" and "end" bracket the sequence, so that it can be treated as one logical entity. The general form of a sequence is:

```
begin
<statement>;
    .
    .
    .
<statement>
end
```

<statement> defines a single operation, Individual statements are separated by semicolons. In the design language, a statement can be a verbal description that will later be expanded into a precise definition (as in the example above, which could be expanded into a precise program for a tea making robot),

It is impossible to start the sequence anywhere other than at the begin, or finish it anywhere other than at the end. This property of having a single entry and a single exit point is shared by all of the basic constructs,

A sequence can also be represented graphically, as follows:

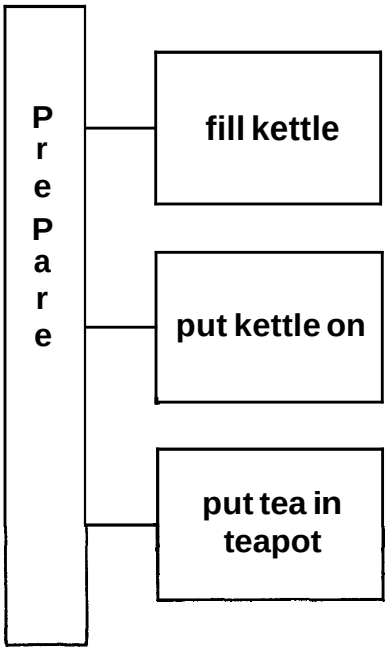


Figure 4-4 Sequence Structure Diagram

The long vertical box represents the sequence as a whole, The other boxes are the elements of which it is composed. It is often useful to give a sequence a name, because it can then be referred to as a single operation in a 'higher-level' algorithm. The elements of the sequence are carried out in order, from top to bottom,

This is a structure diagram. The connecting lines show that the elements belong to the sequence. (The lines do not indicate logic flow, as in a flowchart). The logic flow is obtained simply by proceeding from top to bottom, performing each operation in turn.

The elements of a sequence might be simple operations, or they can themselves be any of the three basic constructs (sequence, selection or iteration),

A complete program will usually be a sequence. In the design language, the semicolons are an important part of the sequence construct. They are not part of the individual statements; rather they separate (or delimit) the statements, and should more properly be regarded as belonging to the begin ... end construct. Note that there is no semicolon following the last statement; there is no need for one, as the end serves as a delimiter instead.

4.5.2 Selection

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 open ventilators
  else switch on heaters
```

Graphically, this is represented as:

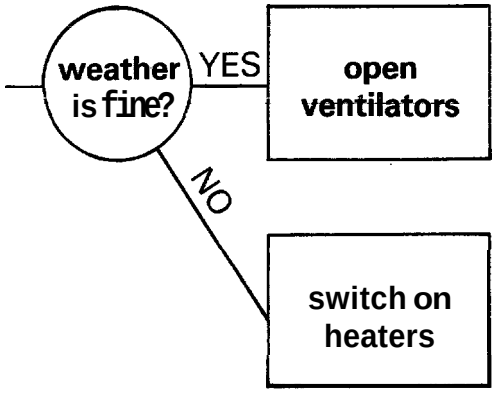


Figure 4-5 Selection Structure Diagram

The circle represents the selection as a whole: that is, a single component which can be either of two things* The boxes are the elements of the selection. For each execution of the selection, one and only one of the elements is executed. Once again, the connecting lines express that the components are members of the selection (they are subordinate to it). The logic flow through a selection consists of testing the condition, and executing one only of the elements.

There is a selection in the example algorithm:

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

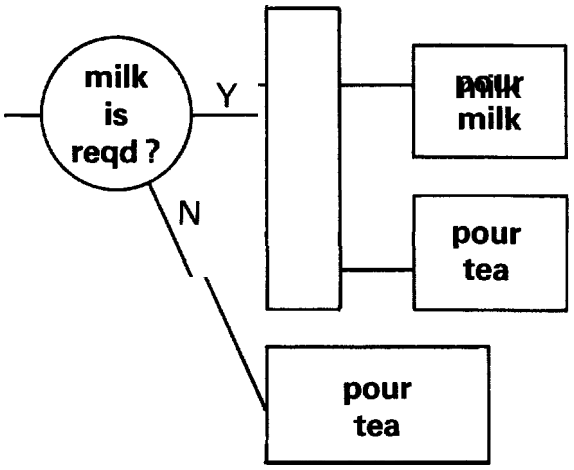


Figure 4-6 "Pour cup" Structure Diagram

Here, the first alternative is a sequence of operations. The begin and end indicate clearly that, as far as the selection is concerned, the sequence is a single element that can be regarded as one statement. The single entry/exit property of the sequence makes this possible. Each of the three basic constructs "packages" a complex operation, so that from outside it can be regarded as a single, indivisible statement.

The keywords begin ..... end can be regarded as "bracketing" a sequence of statements in the same way that parentheses are used to bracket numerical expressions:

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

The general form of a selection in the design language is:

```
if <condition> then <statement>
   else <statement>
```

<condition> is any expression which evaluates to one of the values TRUE or FALSE. Such an expression is called a Boolean expression, and the most common way to arrive at it is by the use of comparison operators such as =, <, >:

```
if temperature > 70 then ...
```

A special case of a selection 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 design language, this corresponds to omitting the else clause:

```
if <condition> then <statement>
```

In the example, 'pour cup' can be written another way:

```
begin

if milk is required
      then pour milk;


pour tea

end
```

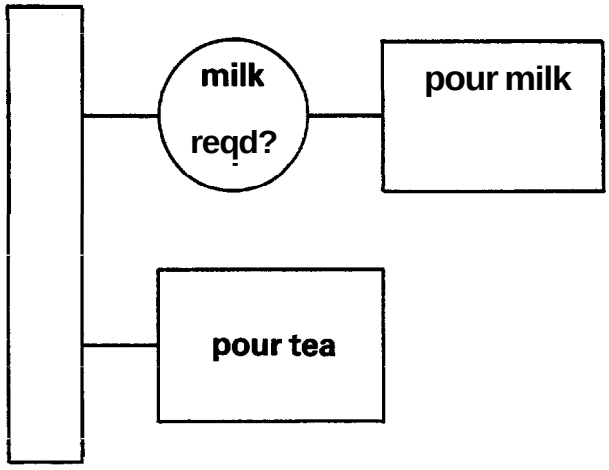


Figure 4-7 Alternative Algorithm for "pour cup"

Here, 'pour cup' is a sequence consisting of two elements: an if construct (with only one alternative), and a simple statement. 'Pour tea' is always executed;. 'pour milk' is executed only if milk is required. The effect is exactly the same as before.

The semicolon (which, as indicated in section 4.5.1, is part of the begin ... end construct) separates the two elements of the sequence, and makes clear where the end of the if statement occurs. 'Pour tea' is not a part of the if statement, and hence is not dependent on the condition; it is the next item in the begin ... end sequence, and is executed in all circumstances. If 'pour tea' was to become part of the if statement, begin ... end brackets would be used as in Figure 4-6. The indentation of the text makes the relationship clearer. The structure diagram shows without doubt that "pour tea" is an element of the sequence and not of the selection. The strong visual resemblance of the diagram to the indented text makes comparison of the two notations easy.

4.5.3 Algorithm Design

It is common in software design to start with a vague formulation of the problem (if weather is fine ...) and gradually home in on a precisely defined, deterministic solution that specifies every measurement and calculation. Although a precise solution is finally needed (or it will never get past a compiler or assembler), a degree of vagueness (or "controlled imprecision") is actually beneficial in the early stages, even though it may go against the grain. A precise formulation too early on may exclude some vital elements, particularly if the software designer does not have direct knowledge or experience of the application, The design language helps here by permitting partial solutions to be tried out on paper before they become cast in silicon, The logic of the application can be precisely formulated before considering in detail how the individual operations required are to be implemented, The design language allows the designer to identify and precisely specify each operation required (reading a temperature, controlling motors and heaters etc) before an attempt is made to implement them,

The software design can be compared to the architect's plans for a building. Although some of the details may be changed during implementation, plans for the foundations and overall structure must be established before starting to build individual rooms.

The algorithm of Figure 4-5 might be part of a system controlling the environment in a greenhouse (say), The next stage in design might be to consider whether it is the inside or outside temperature (or both) that is significant, whether the temperature should vary according to the time of day, and what effect other parameters such as humidity might have.

There are often several alternative ways of writing an algorithm to perform a particular function. The first solution hit upon may not always be the best,

Just as a good data structure (see section 4.6) 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 design.

The best algorithms are usually those that clearly reflect some underlying structure of the application itself, rather than imposing some new structure invented by the system designer. It's quite easy to see why, Unless the specification for a piece of software is perfect the first

time, changes are likely to occur. Perfect specifications are almost unheard of. If the software is structured along the same lines as the application, the software will be able to follow changes in specification quite easily. It will have some "resilience" in the face of changing requirements.

A software design that is structured in a significantly different way to the application is likely to be "brittle", and to break under the strain rather than adapt gracefully to new requirements. Changes in requirements may have unpredictable consequences in different areas of the design, which will either make adaptation impossible, or will reduce confidence in the reliability of the final system.

The nature of software aggravates the problem. Software tends to be applied to complex problems, so that changes are likely to be complex. It's very easy to actually make a software change - simply type in something new. It is much more difficult to ensure that the change is correct.

At first sight it may he very hard to tell the difference between a change that has only limited effect in an isolated software function, and a change that can have ramifications throughout the design.

For this reason it's necessary to pay a lot of attention to software design, Programming is only a part (a relatively small part) of the story. Software needs to be designed and engineered for resilience and reliability, rather than stacked up like a house of cards.

In fact, there are two types of resilience, Software should be able to cope with and recover from unexpected conditions and, ideally, minor hardware faults. Secondly, the system should maintain its integrity in the face of changes to parts of the software itself - perhaps in response to new requirements. A structured design methodology, such as is presented here, assists greatly. The framework of Component Software (Chapter 5) and Microprocessor Pascal (Chapter 6) was designed to the same purpose.

However, a good set of tools is not enough. The system designer needs to spend a good deal of time understanding the application he is designing for, and the ways in which it is likely to change over the lifetime of the system. In this way, likely changes can actually be anticipated and the system can be designed to make them possible.

4.5.4 The CASE Construct

There is a version of the selection which permits more than two choices. 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
```

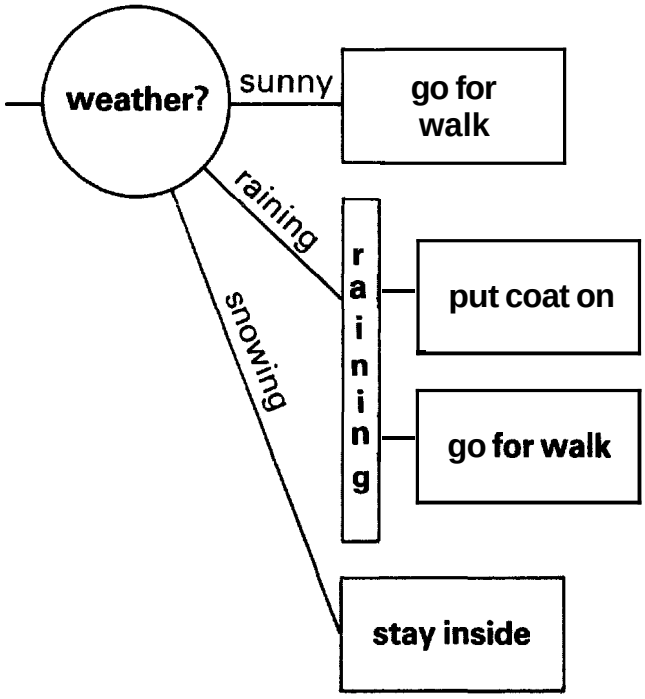


Figure 4-8 The CASE Construct

The case labels "sunny", "raining", "snowing" specify the possible values of the case expression "weather", and the actions to be performed for each ("weather" will have been declared as a variable of type (sunny, raining, snowing)). When executing the selection, the case expression is tested and, according to its value, only one of the operations will be performed. (Note that the operation for "raining" is a sequence, enclosed within a begin ... end bracket.)

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
```

Graphically, this is represented as:

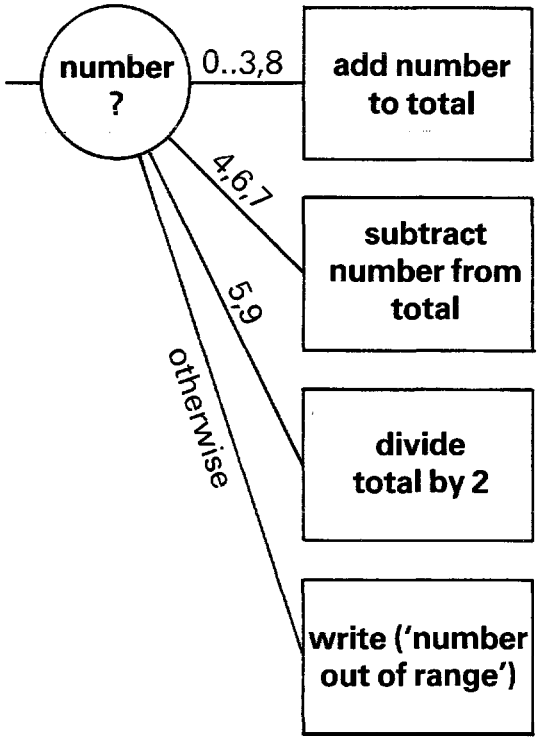


Figure 4-9 CASE Construct with OTHERWISE Clause

The general syntax of the case statement is:

```
case <expression> of

    <case label> : <statement>;
        .
        .
        .
    <case label> : <statement>
    otherwise <statement>

end;
```

The otherwise clause is optional.

4.5.5 Iteration

The third and final algorithmic construct is the iteration, or loop. The iteration allows an operation to be repeated either a specified number of times, or while some condition remains true. There is an example of the first kind of iteration in the algorithm of Figure 4-2.

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

Graphically, an iteration can be represented by a lozenge-shaped box:

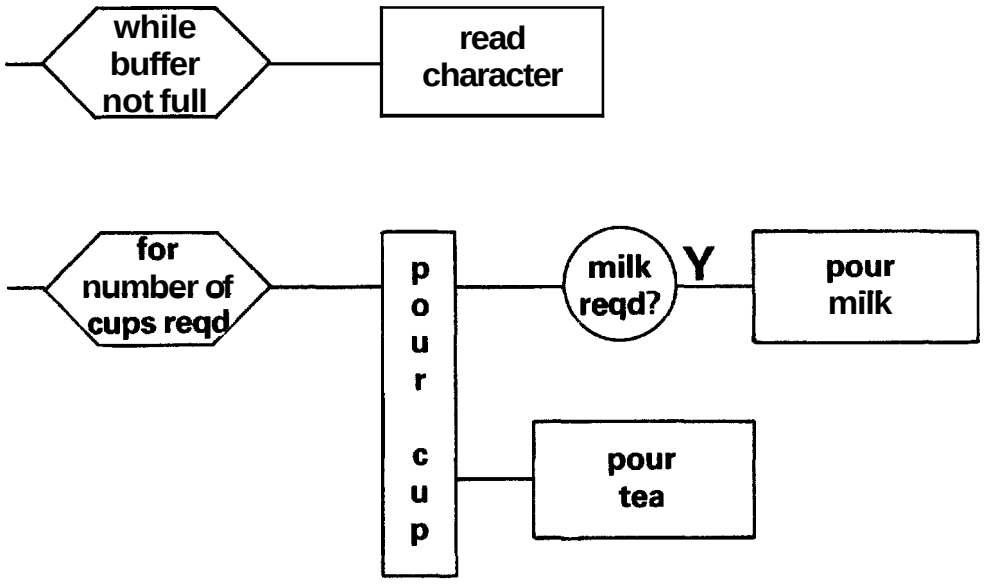


Figure 4-10 Iteration Structure Diagrams

Once again, the left hand box represents the iteration as a whole, which can form a single element in another algorithm. This single element consists of a (possibly zero) number of executions of the right hand box. The right hand box represents an individual execution of the **operation** to be performed. The distinction may appear subtle at first, but it is important. It allows a repeated operation to be included as a single element of, say, a selection construct. Like the sequence and selection, the iteration packages a complex operation as one element with a single **entry** and **exit point,**

Usually, it is a sequence of operations that will be repeated. 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 very useful** construct.

In many iterated operations, it is useful to know which iteration is currently being performed. Most programming languages that implement the for construct therefore specify a for-loop variable:

```
FOR I := 1 TO 10 DO
    BEGIN
    START_MACHINE(I);
    DISPLAY(START_MESSAGE, I)
    END
```

The variable I keeps a count of the repeated execution, and can be referred to within the code of the for-loop. This feature is often required, and this convention will be adopted in the design language. The general form of the for-loop, **then,** is:

```
for <variable> := <initial expression) to
      <final expression) do
         <statement>
```

<statement> is executed for all possible values of (variable), in order, starting at <initial expression) and ending with <final expression>. <statement> will usually be a sequence, enclosed within begin ... end brackets. <initial expression) and <final expression) must be compatible with the type of <variable>, which can be any enumeration type (see section 4.6). <initial expression) and <final expression) are only evaluated once, on entry to the for loop (so it is not possible to change the value of <final expression>, for example, within the loop). If <initial expression) is greater than <final expression) to begin with, <statement> is not executed at all. *

* Some programming languages differ slightly from these conventions. However, some **standards** must be specified to maintain consistency in the design language. These standards represent generally agreed opinion on language

A variant is:

```
for <variable> := <initial expression) downto
      <final expression) do
         <statement>
```

Here <variable> is decremented from <initial expression), which should be the larger of the two, down to <final expression>. This may be more useful in some applications.

The alternative form of the iteration construct is:

```
while buffer is not full do
    read character
```

The while construct is used where it is not possible, or not convenient, to find out in advance how many times the loop must be executed. The general form is

```
while <condition> do <statement>
```

The condition is checked before each execution of the loop; as long as it remains TRUE, the loop is executed one more time.

4.5.6 Structured Programming

Although many programming languages provide additional control structures, programs written using only the three constructs described above have been shown to be easily understood, easily amended, and above all likely to be correct. This discipline is known as structured programming.

The three constructs sequence, selection, and iteration are basic mental structures, representing very closely the way the human mind analyzes a problem. Consequently they are very easy and natural to "think in[1]", once the notation has become familiar. The single entry and exit properties of each construct mean that "high level[1]", application-oriented algorithms can be developed without worrying (yet) about what happens at the detailed level of the operations described. It is known that the effect of each operation is

---

design, and most modern languages (including Pascal) behave exactly like this. When translating a software design into a particular programming language, it is important to determine how the language implements the standard programming constructs - eg does the iteration construct allow for the special case of zero iterations? Pascal directly implements all the constructs of the design language; implementation of these constructs in Power BASIC and Assembly Language is discussed in Chapters 7 and 8.

localised, and that the operation will complete and return control to the high level algorithm without (say) jumping unexpectedly to another part of the program.

Other notations, such as flowcharts, have sometimes been used for designing computer programs. Flowcharts may be useful at the lowest levels of implementation, when coding in Assembly Language for instance (see Chapter 8). However, flowcharts 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, and the application, to work in the way machines do. This may be necessary at some point, but it is not advisable in the earlier stages of a design. Flowcharts concentrate on the details of implementation, and have no way of representing structure.

4.6 DATA

Data elements, which are implemented in the computer simply as a collection of bits, can he used to represent any kind of information. Often the information represented will be numeric, but this need not be the case. A single bit may signal the state of a digital input or output line; or a group of bits may be coded to represent text or any other information.

Most programming languages provide some pre-defined data types (eg FORTRAN defines integers and real numbers) that can be used directly in a program. A data type definition can be regarded as a code that translates some kind of information into an internal representation in the computer. Some languages allow users to define new data types, either by combining already existing data types into new structures, or by specifying the characteristics of a new data type from scratch. These capabilities are very useful when developing software designs.

Structured data types allow related data items to be grouped together and referred to as a single entity. This is much easier than remembering that the information about (say) a piece of production machinery is contained in several different integer and real variables, all with different names. Programs with well thought out data structures are likely to be more reliable and much easier to maintain.

Even where the programming language chosen for implementation does not support flexible data structures, such structures can be worked out by developing a paper design using a design language. This can then be translated into the implementation language. This method, which seems roundabout, will often result in a faster development

turnround than coding directly in the implementation language. Certainly, it will produce a more reliable system.

Effective use of data depends on identifying the essential elements of what is to be represented, and choosing the most appropriate representation in terms of numbers or binary digits. For example, if a temperature is to be input from the outside world to a microprocessor system, how should it be represented? Does the system need to know the actual temperature value? To what precision? Or is a single bit, indicating that the temperature is above or below some threshold, sufficient?

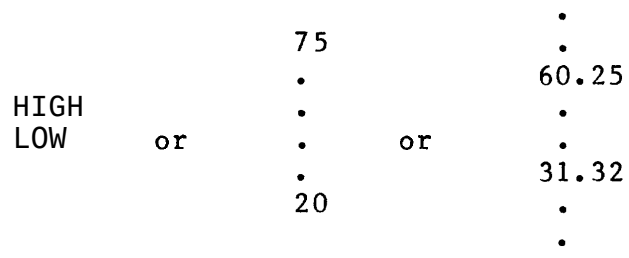


Figure 4-11 Data Representation of a Temperature

This decision will, of course, dictate the choice of sensor used to measure the temperature.

Data items can also represent things that are much more abstract than a temperature - for example the root mean square of a collection of statistical figures, It is this ability to represent and manipulate anything that can be defined exactly that gives software its power. Data items can represent things which only have meaning within a particular piece of software - intermediate results in a calculation, for example, or codes representing which of a number of possible operations should be performed.

How the data types are chosen defines the environment within which software algorithms can work. A program can only manipulate things which have previously been defined as data items. Hence, data design is the key to any piece of software.

4.6.1 Data Types

The first step in building a software design is to identify the different kinds of information that need to be dealt with, and to define appropriate data types, A type declaration identifies a particular type of variable that will be dealt with in the program, and the range of

values that variables of this type might have. For example, a particular system might need to make decisions according to what day of the week it is, It makes sense to define a data type called "day":

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

The items in brackets identify the values that variables of type "day" might have. Note that this declaration does not actually specify any variables of type "day". It simply introduces the notion that variables of this type can exist. After this declaration, we can talk about "days" in the software design and know exactly what is meant. (In ordinary conversation we think we know what days are, but in software it's necessary to be more precise. The definition makes clear that we're talking about days of the week, not days of the month, and in particular that we're talking about workdays: Saturday and Sunday aren't included,)

At this stage it is neither necessary nor desirable to consider 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 necessary simply to understand what's needed to satisfy the application.

From the computer's point of view, what has been said so far is:

(1) There will be data items that can take one out of five possible values

(2) The designer is going to refer to these as "day"s

(3) The designer is going to refer to the different values of these "day"s as Monday, Tuesday, Wednesday, Thursday, Friday.

The general form of a type declaration is:

```
type <name> = <type definition);
```

The angle brackets indicate a generic name; in an actual type statement, "<name>" will be replaced by an actual type name. The form "(<value list>)", as in the "day" declaration, is one kind of type definition. Other kinds of type definition are presented below.

For the purpose of a software design, the following data types can be regarded as predefined:

```
integer     (-32768..32767)
real        (= floating point)
char        (= ASCII character set)
boolean     (= TRUE or FALSE)
```

4.6.2 Variables

Type declarations simply specify a kind of information that is to be represented, To define actual data storage items, or variables, of a particular type, a variable declaration is needed:

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

This statement declares two variables, which will ultimately be storage locations within a computer. These variables are called "startday" and "endday". They are of type "day", which means that the values they can take are Monday, Tuesday etc. Whatever implementation is later decided on for "day", that amount of storage and that representation will be assigned to "startday" and "endday".

The general form of a variable declaration is:

```
var (variable list> = <type>;
```

Separating out the type declaration from the var declaration means that the decision on how to represent "day"s is taken once and once only. There's no need to take this decision again (perhaps differently - particularly if more than one designer is working on the same system) every time a variable of this type is needed, Also, if the requirements change and it's necessary (say) to include Saturday and Sunday, this can be done simply and reliably throughout the system simply by changing the one type declaration.

This is a relatively trivial example; but multiplied by the thousands of decisions required during implementation, clearly thought out data typing can make the difference between manageable programs and intractable ones,

<type> in the var declaration need not be a type name, but can be an explicit type definition:

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

However, if more than one var declaration uses the same right hand side defintion, it is preferable to define a type, and then use the type name in the var declaration.

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

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

Suck types are called subrange types because they are defined as a specific subrange of an already defined type. The above declaration works because the type "integer", consisting of the values -32768, -32767,.....-1, 0, 1,....32766, 32767 (for a 16-bit processor) is predefined. "Weeknumber" is a subrange of integer.

It is also possible to define subranges of type "day":

```
type first_half_week = (Monday..Wednesday);
```

4.6.3 Operators

Having defined data items, it's necessary to do something with them. In a program, variables of particular types can be combined using operators. In the statement

```
a = b + c
```

"+" is an operator. "+" means "add the values of b and c to give a third value".

In ordinary mathematical language, the above formula is simply a statement of fact: "a is equal to b plus c". In computer language, it's more likely to signify an operation: "make a equal to the value of b plus c", or, to put it another way, "a becomes equal to b plus c". This is one of the most common of algorithm statements, namely the assignment statement. Here "=" is an operator too - the assignment operator, whose effect is to assign the value of whatever expression is on its right to the variable on its left.

To avoid confusion between the assignment operator and the mathematical "=", which mean quite different things, modern languages such as Pascal use a special symbol, ":-", for assignment:

```
a := b + c
```

read, "a becomes equal to b plus c". This convention will also be used in the design language. The left hand side of an assignment statement must always be a variable, because a value will be assigned to it. However, the right hand side can be an expression: that is, any combination of variables, operators and constant values that can be evaluated:

```
5*a + b - c/2
```

The general from of the assignment statement is

```
(variable > := <expression>;
```

The expression should evaluate to a type that is compatible with the variable on the left hand side. It makes no sense to assign a temperature value to a day of the week.

Some programming languages make no check that the type of the expression is compatible with the type of the variable: they simply assign the bit code representing the value of the expression to the storage location for the variable.

While this can be made use of in special cases, ninety per cent of the time an unmatched statement indicates that the programmer has made an error. Programming languages that check for exact compatibility of types in assignment and other statements are said to implement strong data typing.

Even when an unmatched statement is written deliberately *, it is a rather risky operation: it depends on a certain relationship between the internal bit representations of the two data types (some examples of internal representations are given in Chapter 8). If the software is transported to another machine, or even if the compiler is changed, this relationship may no longer hold. In developing a software design, it is wise not to make use of such relationships; or if they are used, to isolate them to certain routines which are known to be machine dependent.

In general, an operator will apply only to certain data types. In developing a software design, all the standard mathematical operations (+ - * /) (* = multipy, / = divide) can be regarded as pre-existing for numeric data types. But multiplying days of the week makes no obvious sense, either in the real world or in a software design. Any operations to he performed on non-numeric data types must be defined, perhaps as separate procedures (see section 4.10 below).

Types such as "day" and "weeknumber" (and "integer") are called enumeration types, because their possible values are specified by enumerating them, in sequence. The order of values in the sequence is significant. The operators PREC (preceding) and SUCC (succeeding) can be regarded as pre-defined for all sequenced data types:

```
eg    PREC(Wednesday) is Tuesday
```

* Microprocessor Pascal, which is a strongly typed language, provides a type transfer operator which can be used to override type checking. However, the programmer must explicitly tell the compiler that he is doing something out of the ordinary, and exactly what he is doing (Section 6.6.14).

SUCC(Thursday) is Friday

The assignment operator can also be applied to all data types. More complex operations can, of course, be devised, but they must be specified precisely.

Suhrange types can be used to specify the range and precision of numbers that will be used in calculations:

```
type temperature = (-50..+100);
     pressure    = (0..900);
```

(Note that the keywords type, var etc need not be repeated for multiple declarations. The declarations are separated by semicolons.) For Pascal designs, the compiler can optionally perform automatic checks to ensure that variables never exceed the bounds specified.

In addition to the type "integer" the numeric type "longinteger" (-2147483648..+2147483647, ie 32 bit signed) is often useful, and is directly implemented in Microprocessor Pascal and in some other languages.

Obviously, use of certain facilities of the design language will he conditioned by what is expected to be available in the final implementation language - for example, is a floating point package available? Nevertheless, the freedom of the design language is useful at least in the early stages of working out what is needed to implement the application.

Note that "real" is not an enumeration type. With enumeration types, it is always possible to identify a unique predecessor and/or successor for any value (eg with integers, 5 is preceded by 4 and succeeded by 6). However, what is the successor of the real number 2.414? Is it 2.415? 2.4141? or 2.41401? Given any two real numbers, it is possible to define a third real number that lies between them in value (up to the limit of precision of the computer). The representation of real numbers follows a completely different principle from the representation of integers. Real numbers are stored differently within the computer,* and cannot, for example, be used as an index to an array (see below, section 4.7.2).

The discipline of data typing makes it much harder to make mistakes - such as using variables in the wrong place - and much easier to find mistakes if they are made. Data types, and variables, can also be given meaningful names (in the design language at least, and in some implementation languages). With variables called I, J, K, or even K2BCPLZ, and all implemented as (say) integers, it's quite easy to mistake a variable representing a day of the week for one

* The representation of real and other numbers is discussed in Section 8.13.2

representing (say) the mean of 25 temperature values, and hence to perform a completely inappropriate operation. Such errors can easily propogate right through to implementation, and may only be discovered when the system doesn't work. For software designs executed in Pascal, the compiler will automatically check compatibility of data types.

4.6.4 Data Design

Designing good data types and data structures is not easy, and there is no standard way to go about it. It is perhaps the biggest challenge of software design.

Some languages (eg Pascal) implement the data type constructs described here directly. Others implement only a small range of data types (such as INTEGER and REAL). Whichever language is to be used for the final implementation, the software design can be developed using a design language, as described here. When the design is complete, each data type can be "mapped" onto a suitable implementation in the programming language to be used.

One advantage of this approach is that much of the design work is done in a medium that is not tied to any particular hardware implementation. This means that the design will be much more transportable. It also means that details of the implementation which might sidetrack design thinking at this stage (such as precise syntax and punctuation, and the idiosyncracies of a particular programming language) can be left until a later stage.

Besides documenting the system and the design process, the software design can be referred to when making changes to the system. It contains relevant information that may be lost or obscured in implementation. The design is also a starting point for implementation using different programming languages.

4.7 DATA STRUCTURES

Single data items, of whatever type, are of little use in real applications. Usually, the data required to describe anything in the real world is much more complex than this. It is useful to group single data items together into data structures. As with program algorithms, there is a set of simple constructs which can be used in a variety of combinations to represent data structures of any complexity. The principle data constructs are the record and the array,.

4.7.1 Records

The record enables data items that are associated in some way to he grouped together, and referred to by a single name. A record is simply a collection of (probably dissimilar) data types.

Consider an application that controls a number of pumps at a self-service filling station. A record can be defined to contain 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 the newly defined type "pump_record". The record construct is another form of <type definition>, as described in section 4.6.1. "end" closes the record definition. "_" is used to make pump_record into one word.

The record in this example contains three fields (status, grade and gallons), each of which has a unique name. The record groups, in one place, the status of operations at a particular pump (whether the pump is off, in the process of filling, or has completed); the grade delivered; and the number of gallons delivered.

The status information for the first pump can be referred to unambiguously as "pump1.status". ".status" is called the field qualifier. 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 allows the building of very powerful data structures.

Types of fields in a record can be predefined, eg:

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

type pump_record =
     record
       status  : status_values;
          .
          .
          .
     end;
```

The algorithm for the filling station 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, displays it at the cash desk and resets the pump:

```
if pump1.status = completed then
     begin
     calculate_cost;
     display_cost;
     reset_pump_1
     end
```

calculate—cost, display_cost and resetpump—1 are all operations that are expanded elsewhere in the software design.

The cost calculation is based on the "grade" and "gallons" fields of the pump record and a table of prices. "Calculate_cost" can be expanded as follows:

```
cost := pump1.gallons * cost_table[pump1.grade]
```

"cost_table" is an example of another structured data type called the array.

4.7.2 Arrays

An array is an ordered list of data items of identical type. The whole array is given one name; an individual element of the array is referred to (referenced) by giving the array name and an index or subscript, which identifies which element in the array is required.

```
type buffer = array [1..80] of char;

var bufl : buffer;
```

or, equivalently

```
var buf1 : array [1..80] of char;
```

"char" is a pre-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 (ie, the fourth character in the buffer) can then he referred to as "buf1[4]"; this element is of type "char".

In the design language (and in Pascal), any enumeration type can be used to index an array, So "cost_table" (above) is declared:

```
var cost_table : array [regular, premium, unleaded]
                 of price;
```

The reference cost_table[premium] will then give the price of premium grade ("price" is a type defined elsewhere).

To gain a feel for the notation, and its practical application, it's worthwhile constructing a few trial examples. For example: design a record type named "call_record" to contain all the essential information about an individual telephone call (originating number, destination, distance etc). Declare two or three record variables of this type, Declare an array to hold the tariff information, and write the algorithm to calculate the cost of the call. Declare another array to hold, for every subscriber, the current bill. Write the algorithm statement to add the cost of a new call, to the bill for the appropriate subscriber.

What is inside the square brackets of an 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, An array containing the daily receipts of a store can he declared:

```
var daily_takings : array [day] of money;
```

(assuming the previous declaration of type "day", as in section 4.6.1). The receipts for Tuesday can then be referenced by

```
daily_takings [Tuesday]
```

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

It is convenient to use the same type to declare an array and any variable used to index it:

```
type buf_size = 1..80;

var bufl : array [buf_size] of character;

var index : buf_size;
```

This makes changes to the buffer size much easier, and also aids documentation. With an appropriate choice of names, designs such as this can be largely self-documenting. If this design is turned into Pascal, compiler checks can be used to ensure that the array index never exceeds the specified bounds in execution.

With an index variable, the same portion of a 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 pumpl, 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 statements 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. The notation works like this:

```
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 this record, and is of
                    type: (regular, premium, unleaded)
```

Any array can be indexed by adding "[index]"; any record can be qualified by adding ".field". By nesting definitions in this way, data structures provide powerful tools 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 the techniques described here, if each level of the structure is correct and well understood, the designer 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,

4.7.3 Dynamic Data Structures

Returning to the filling station example, one problem appears in the original design. In order to save the cost information, a new customer cannot use a pump until its previous customer has paid his bill. Several solutions, however, are possible, For example, an array of pump_records could be defined 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 != 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 could be declared. These records could be assigned to customers as they arrive. However, customers leaving would create "holes" in the array. This problem can be solved (eg, by a "tidying up" algorithm). Such a solution, however, is 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, 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, consist of data elements (usually records) which are dynamically allocated 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 will be returned to the heap. Thus, customers can be added to the list when they arrive, and deleted when they leave, The data structures change dynamically to reflect the real situation.

Lists, and other useful data structures such as trees, are described in more detail in the references given at the end of this chapter (in particular see reference [1] in the Bibliography, section 4.13). Lists, and other dynamic data structures, are generally managed through another data type called the pointer. Pointers and the structures they can he used to implement are described in reference [1], and in the Microprocessor Pascal System User's Manual.

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 basic elements with which the system will work and affects both algorithms and input/output. The best way to arrive at an optimum solution is to be aware of the choices that can be made.

4.7.4 Data Diagrams

The graphical notation described above for algorithms can also be used for data structures. The sequence notation can be used to represent records, and the iteration construct to represent arrays. Thus, the array 'pump' of 'pump_records' in section 4.7.2 can be drawn:

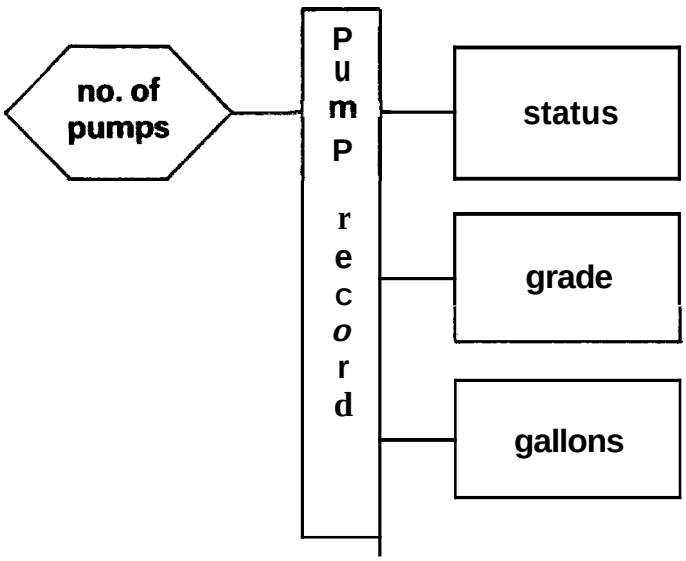


Figure 4–12 Data Diagram for an Array of Records

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 4–13).

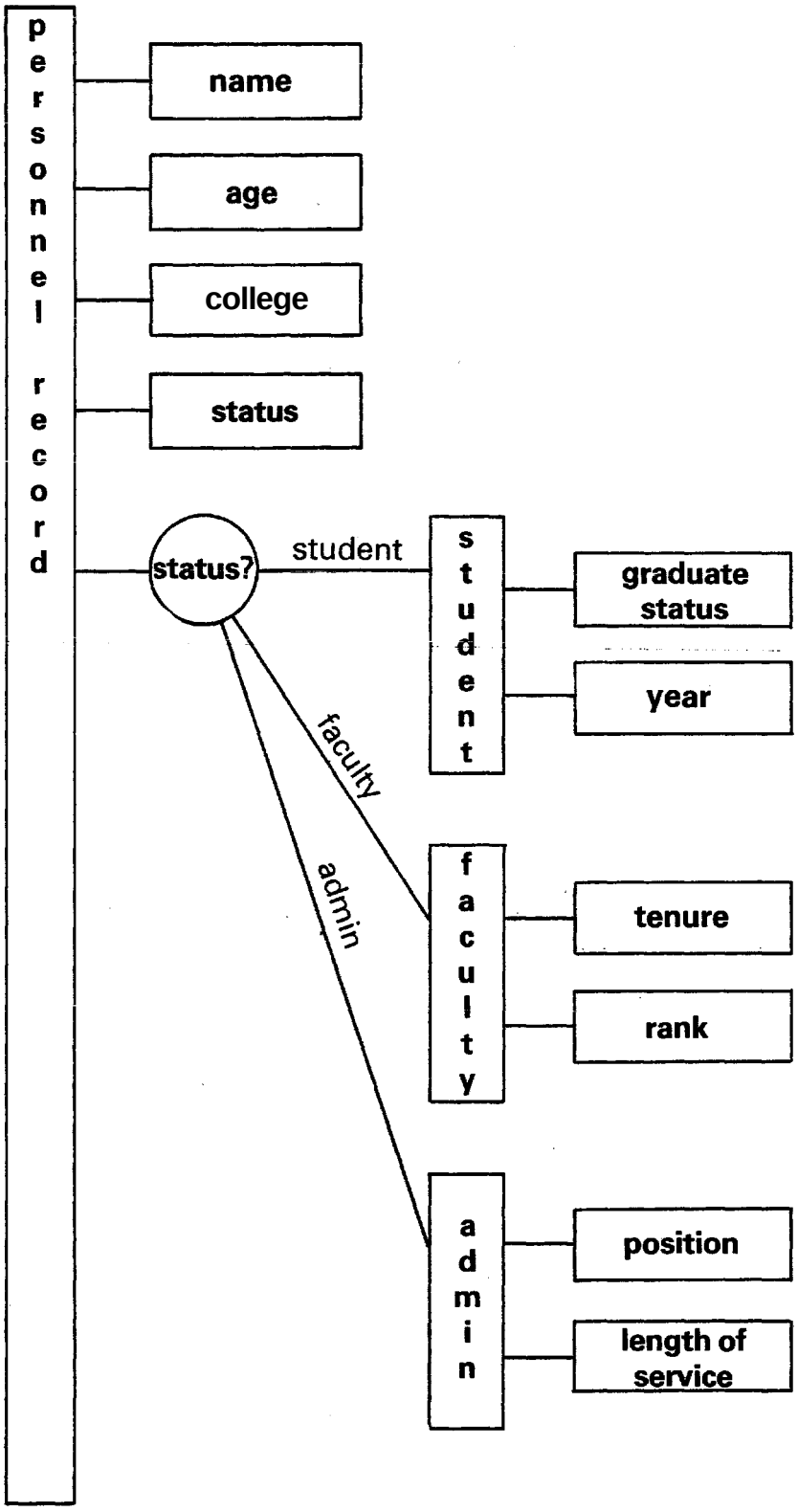


Figure 4-13 The Record Variant

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   : status_type;
                     year              : 1..7);
          faculty : (tenure            : boolean;
                     rank              : rank_type);
          admin   : (position          : position_type;
                     length_of_service : 1..50)
       end
     end
```

assuming the previous definition of:

```
type status_type   = (graduate, undergraduate);
     rank_type     = (inst, asst, assoc, prof);
     position_type = (asstdean, dean, chairman, other);
```

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.

Examples of further constructs which can be used (including the pointer type and dynamic data structures) are given in the Microprocessor Pascal System User's Manual. The constructs of Pascal are designed to be "universal", and many of them can be adapted for direct use in the design language.

4.8 DESIGN APPROACHES

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 that 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 processor instructions, 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 was thought that a system required a keyboard input routine and a display routine, these functions 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 very much thought is given to the precise requirements of each function, and the ways in which functions will fit together, before they are implemented. Therefore the designer ends up with blocks that are of incompatible size or shape, and he either has to reconstruct the blocks, or make the best of what he has and design some special pieces of software to overcome the problems of incompatibility. This does not lead to very robust systems.

The major problem of software, unlike other technologies, is not in the actual construction of functions. Once a requirement has been precisely identified, implementing a stand alone piece of software to perform it is fairly straightforward. The problem lies in organizing a collection of functions so that they will cooperate to perform a complex task. This is the problem that is addressed by top-down design. The requirement and the interface for each function is identified before it is implemented.

Actually, pure bottom-up design is not possible. The designer must have given the problem some 'top-down' thought or he would have no idea what building blocks to construct. What top-down design does is to make this thought much more systematic. It provides the designer with some tools to attack the problem (such as the design language), which are better than his bare hands. Traditionally, the only

languages available for design were programming languages, which typically required so much attention to machine detail that the major issues were obscured. Also, early programming languages were unstructured, so that it was difficult to isolate and focus on particular design issues or to look at the system as a whole without becoming involved in a mass of detail.

Design languages and notations like those introduced above have largely solved this problem.

A design might be conceived initially like this:

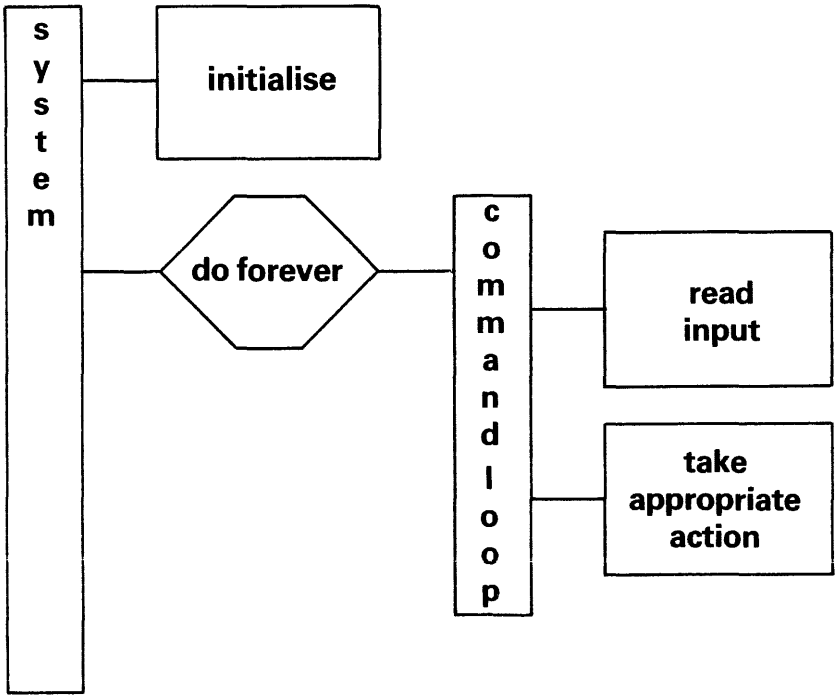


Figure 4-14 Initial Design Algorithm

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:

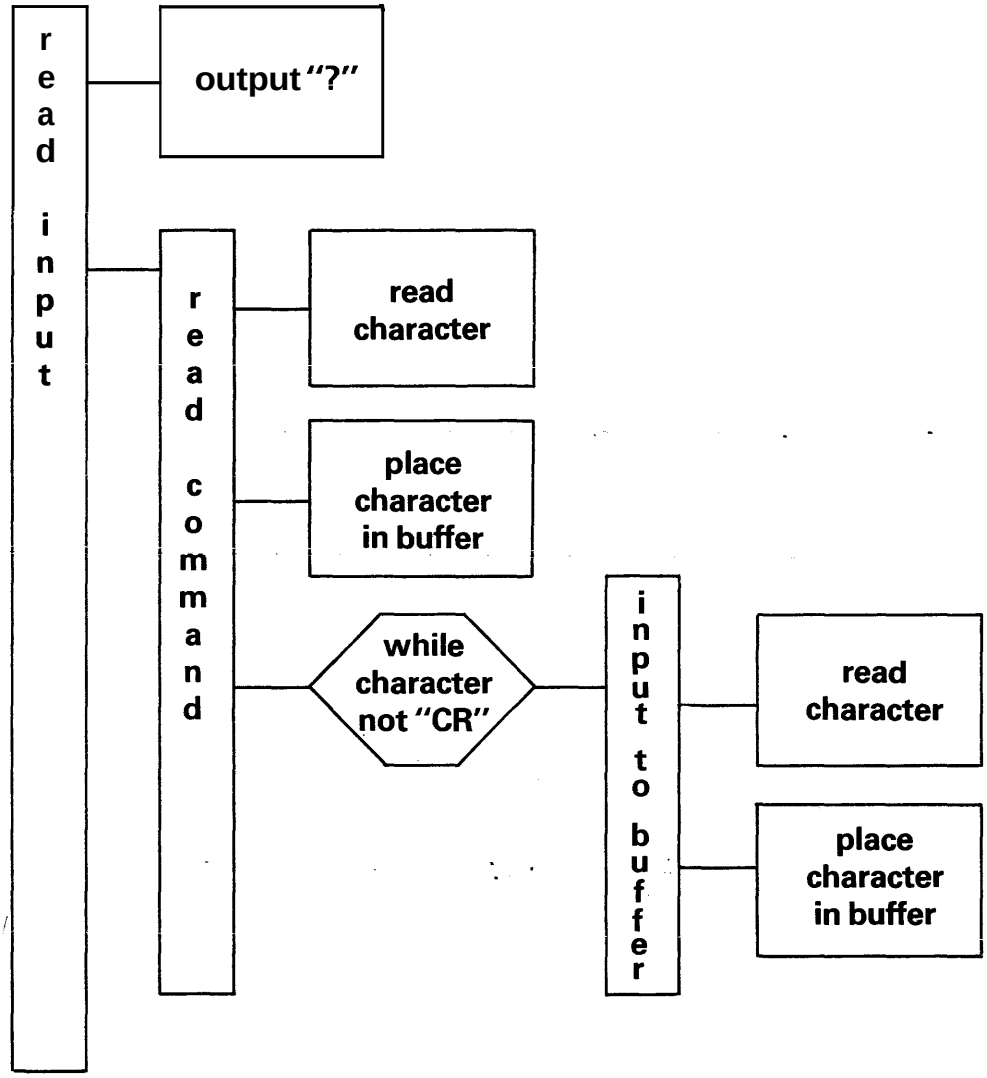


Figure 4-15 "Read Input" Algorithm expansion

The terminal boxes of this diagram can be further expanded until a complete solution is derived,

Because of the single entry and exit properties of the structured programming 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,

It is this property of structured notation which makes it possible to hold off consideration of details and to design from the top downwards (or, more accurately, from application towards implementation).

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 1/0 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 remains from problem towards implementation.

4.9 BLOCK STRUCTURE

In a software design, the general form of any programming unit can be expressed as follows:

TYPE DECLARATIONS

VARIABLE DECLARATIONS

PROCEDURE STATEMENTS

Such a program unit is called a block. The type declarations specify the types of data that will be used in the program (in addition to predefined types); the variable declarations specify actual data items of these types; and the procedure statements define what the program will do with these data items.

Most modern programming languages are block-structured - that is they make use of the block construct to modularise programs.

The advantages of blocks become apparent when considering how a large software design can be broken down into smaller parts for separate implementation (by the same programmer or by others). Each part can be implemented as a separate block, with its own types, variables and procedure statements.

A block encapsulates the complete programming environment for a particular program unit. The declarations made within a block apply only to that block. They constitute a local "language" invented and spoken (or rather written) by the programmer of that block. This language (the types of data permitted, the actual data items declared, and the procedures available for doing things) is designed to be

appropriate to the specific problem to be solved by that block, and is unknown outside the block,

Thus different parts of the same software design can be developed separately with no possibility of interference or confusion. It's even possible for two programmers to use the same name for two completely different variables. ("TEMP", for example, could be chosen to represent a temperature by one programmer, and to represent a temporary variable by another. While such name duplication should not be encouraged, it's difficult to ensure that it doesn't happen among the many separate decisions that are made in developing a software design,) There are standard and controlled means by which information is exchanged between different blocks,

The block construct can be used wherever a self-contained programming unit is to be defined. A complete program is a block; so is a subprogram, Blocks can be nested one within another.

A smaller block nested within a larger can be regarded as existing within the environment (or context) of the outer block, Thus, type and variable declarations in the outer block apply in the inner block, However, local declarations override global ones: if by chance a variable is declared in an inner block with the same name as one already declared in an outer block, the local declaration applies in the inner block, This is shown in Figure 6-2, Section 6.3.6.

The block structure defines a hierarchy, or tree, of relationships between programming units. These are called lexical relationships, In Figures 6-2 and 6-3, the lexical parent of PROCEDURE P is PROGRAM A (both PROCEDURE P and PROGRAM A are blocks), PROCEDUREs P and Q are lexical brothers; P, Q and A, as well as B and R, have SYSTEM X as a common lexical ancestor. This lexical relationship simply describes the (static) context in which the individual blocks are declared, and the data items, types etc which they share. It does not determine the (dynamic) order in which blocks will be executed when the system is running,

Block structure is a way of managing complex logical entities by splitting them into smaller entities with clearly defined relationships. From experience, this kind of structure is required to manage all but the smallest software systems,

4-10 PROCEDURES AND FUNCTIONS

The most common way of implementing a smaller block within a larger program is as a procedure or function. A procedure

(sometimes known as a subroutine) is a separate block 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 activated 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 is to operate:

```
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;
```

Figure 4-16a Procedure Declaration

Procedure call:

```
seq (x, y, z)
```

Figure 4-16b Procedure Call

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 special type of procedure that returns a single value of a particular type. ("function" underlined has a specific technical meaning, as described here. Elsewhere in this book, "function" is used in a more general sense.) A function can be treated as a variable and included in an expression, even though calculation of the value to be returned involves some algorithmic process. The type of the function 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')
```

Figure 4-17 Function Declaration and Reference

Besides variables, values or expressions can usually be passed as parameters, provided they are of the right type. Procedures can declare local variables which are only used within the procedure. In a block structured language the procedure also has access to the variables of the program in which it is declared, In Pascal, 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 he used in several places in the program only occurs once in the object code.

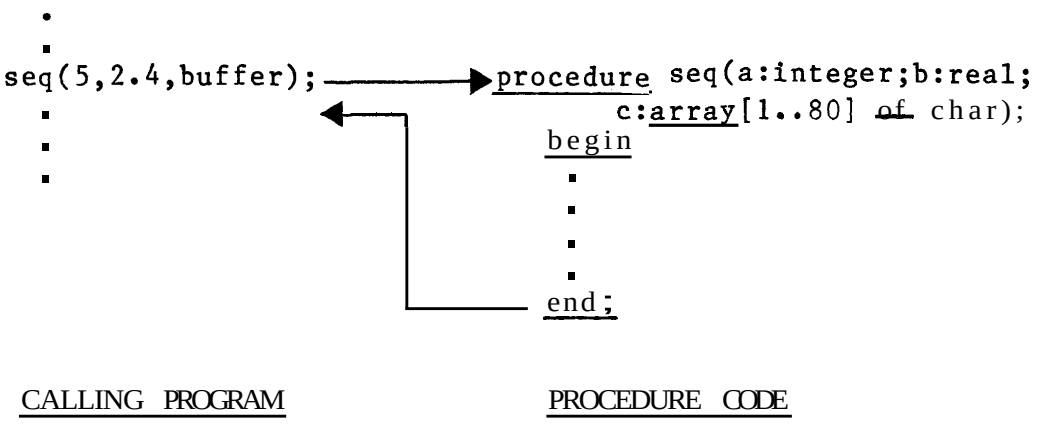


Figure 4-18 Procedure Call Mechanism

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 processor returns to the statement in the main program following the procedure call and resumes processing of the main program.

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*)

Most programming languages allow a program to make use of procedures defined elsewhere in the system, perhaps in another program module. Such procedures are declared within the program block which is to use them by some form of EXTERNAL declaration:

```
procedure select (a : integer; b : real); external;
```

The standard model for a program block (section 4.9) should therefore be expanded as follows:

TYPE DECLARATIONS

VARIABLE DECLARATIONS

EXTERNAL DECLARATIONS

PROCEDURE STATEMENTS

4.10.1 Parameter Passing

There are two distinct ways of passing parameters to a procedure or function. Passing by value will simply cause the value of the actual parameter to be found and assigned to a new storage location in the procedure or function. Any changes made to the formal parameter variable in the procedure will have no effect on the actual parameter variable in the calling program. In fact, actual parameters passed by value can be arbitrary expressions (of appropriate type):

```
test (5*x + 2)
```

Passing by variable reference (sometimes called "passing by location") transfers not a value. but the address of the actual parameter variable in the calling program. Operations in the procedure are performed using the actual variable in the calling program, not a local copy. Results can therefore be returned from the procedure to the calling program (by assigning a new value to a parameter). However, the call to "test" above would be illegal in this case as the actual parameter must be a variable.

A simple procedure will illustrate the difference:

Declaration:

```
procedure modify (x : integer),;
    begin
    x := 2 * x
    end;
```

Call:

```
modify (a)
```

If "x" is passed by value, there will be no effect on "a". If "x" is passed by variable reference, "a" will be doubled by the call to modify. However, a call such as "modify (5*a)" would be illegal. The differences are summarised in Table 4-1.

	METHOD OF PARAMETER PASSING	
	VALUE	VARIABLE REFERENCE
Allows expression as actual parameter	Y	N
Allows variable as actual parameter	Y	Y
Modifies value of actual parameter variable in calling program (ie returns results)	N	Y

Table *4-1* Methods of Parameter Passing

When writing a procedure or function, it is important to be clear about the method of parameter passing. If a value is to be returned, variable reference must be used. If not, value passing gives additional security against accidental modification of the calling program's data.

Some programming languages provide only one method of parameter passing, or determine the method required from the context. But problems can arise: in some versions of FORTRAN it's possible to change the value of a constant by a call such as "modify (5)". Strongly typed languages avoid such anomalies by checking the correspondence of parameter declarations and calls,

Most modern languages allow the programmer to choose the method of passing for each individual parameter. In the design language, parameters to be passed by variable reference should be identified in the procedure declaration by the prefix "**var**":

```
procedure example(var x : integer; y : real);
```

All other parameters are assumed to be passed by value, In the above, "x" is passed by variable reference and "y" by value.

4.11 REAL TIME SOFTWARE

Much of what has been described so far applies to sequential software. An algorithm is a sequential construct, representing a single thread of logic designed to perform a

particular function.

But purely sequential systems are of limited use in a parallel world. In real life, many things are happening simultaneously. Microprocessor applications in particular often need to be aware of, and to control, several things that don't have a simple, one-after-the-other relationship in time. A system controlling an industrial process nay need to monitor several different temperatures, pressures and flow rates, and take appropriate action to control the process. It may need to open and close valves and start pumps in a predetermined sequence. And it may need to respond to commands from an operator, which can come at any time.

A microprocessor will probably have the capacity to do all this. The problem lies in organizing its time and other resources so that everything gets done when it is required. A general solution to this problem requires something more than the sequential modularity described above. What is required is a modularity based on application function, that comprehends both the sequential and parallel nature of the world.

A procedure call is a sequential mechanism: the calling program suspends execution until the procedure has completed. But real time applications do not split easily into PROCEDUREs and FUNCTIONs with a simple sequential relationship. Squeezing such applications into a sequential package means a departure from natural program modularity, and usually results in "brittle" designs which are difficult to test and may be unreliable in operation.

It would be much easier to define individual tasks to be performed as separate program blocks, which could be considered to be executing at the same time. Concurrency permits this. Separate tasks can be written as individual processes. When the system is executing, processor time and other resources will be shared out automatically between the processes according to demand and priorities set by the designer. This sharing out of processor time is known as scheduling.

Each process is a separate sequential block which can be written separately from the other processes. Processes can signal to each other and exchange messages to coordinate the operation of the system.

A brief description of semaphores, executives and interrupts is given here. Concurrency and its implementation is described in more detail in the following chapter.

4.11.1 Semaphores

A semaphore is a signalling mechanism that represents an explicit event, It can be used for signalling between individual processes, and between processes and the external world.

Semaphores can indicate the occurrence of any kind of event that is of importance to more than one process in a system, A semaphore may indicate an external event - eg "character_received" from a terminal device - or an event purely internal to the software of the system - eg "text_buffer_full".

There are two primitive operations that can be performed by a process on a semaphore - signal and wait. A process that completes an event signals the appropriate semaphore; the semaphore "remembers" that the event has taken place. Another process can execute a wait operation on the semaphore, which means that it will be suspended until the semaphore is signalled from somewhere else. (If the semaphore has already been signalled, the waiting process will be released immediately and can continue.) Thus a semaphore is a simple signalling mechanism, mutually understood by two or more processes:

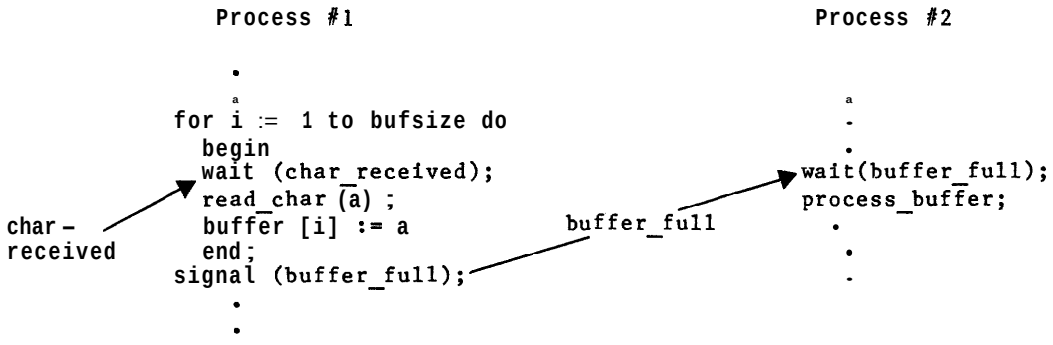


Figure 4-19 Semaphore Signalling

A process can synchronize its operation on an event taking place anywhere else in the system. A semaphore is a very simple signalling mechanism that conveys only that some event (mutually understood by signaller and waiter) has taken place,

The most useful type of semaphore is a counting semaphore, which will count and store the number of times it has been signalled if several signals have been received without a wait, A counting semaphore will also establish a queue of waiting processes if more than one wait is received without a **signal.** Thus semaphores can provide a degree of flexibility in a system, to cope with temporary **"peaks"** and **"troughs".**

The implementation of a semaphore must ensure that a process can complete its signal or wait operation without being interrupted by another process, so that the semaphore does not become corrupted,

Semaphores can be used to construct more powerful communication and synchronization mechanisms between processes, that allow for the exchange of messages as well as signalling the occurence of an event, Such mechanisms are discussed in Chapter 5, Component Software, and in the Microprocessor Pascal System User's Manual.

4.11.2 Executives

Because the processor instruction set does not directly implement concurrency and semaphores, a set of software routines executing on top of the bare machine are required to provide these facilities, This set of routines is known as an **executive.**

A **"bare"** software system can be written to run on a processor without an executive. This was often done in the early days of microprocessors, However, a standard executive makes things considerably easier and can provide services such as concurrency and standard management of interrupts and 1/0 (see below), An executive tailored to the needs of a microprocessor need not be large: Texas Instruments' **Realtime** Executive can be configured down to a size of 3K bytes,

4.11.3 Interrupts

There are two ways that a processor can become aware of something that is happening in the external world, One is to execute a software instruction at a particular point in a software algorithm to read or test an external input, This

technique is called polling. Until the appropriate instruction is executed, the software is completely unaware of the current value of that input (it may have stored the value read last time that input was polled).

The other technique is to connect a signal in hardware so that it immediately interrupts the processor when a certain condition occurs (defined by external hardware). When the processor receives an interrupt, it will carry out a context switch to completely save whatever it was doing at the time the interrupt was received, and will then execute an interrupt service routine. (The hardware mechanism implemented on the 9900 and 99000 microprocessors for interrupts and context switches is described in Chapter 8). In a system containing an executive, the interrupt service routine will probably signal a semaphore associated with the interrupt received, and cause a rescheduling operation. TI's Realtime Executive is event driven: that is, occurrence of an external event (an interrupt) will cause the processor to immediately reschedule its operations to deal with the event. The event may cause a process that has been suspended on the interrupt semaphore to reactivate, and this in turn may signal other processes, so that an external event may propagate a chain of activity throughout the system.

Event driven scheduling is what is required in real time and control situations, as it provides immediate response to external happenings. The hardware interrupt priority scheme may be used to prioritise the response to different external events, if more than one occurs at once, The executive provides a standard means of managing and controlling interrupts, so that synchronization with external events is handled in the same standard way as synchronization with internal processes, It is also possible to write interrupt service routines that execute outside the executive environment, so that very fast response can be provided for those signals which require it, without involving the executive or other processes.

4.12 MAKING TEA

The tea making algorithm (Figure 4-2) can be updated to run in a real time environment:

```
begin
fill_kettle;
put_kettle_on;
put tea_in_teapot;
wait (kettle_boiling);
fill_teapot;
delay (5*60*1000);
for number := 1 to cups_required do
   pour_cup
end
```

Figure 4-20 Real Time Algorithm

"kettle_boiling" is now a semaphore, and the process containing this algorithm performs a "wait" on it. The semaphore will be signalled, and the process will be revived, by the external event of the kettle boiling. (A steam sensor will probably be wired up to generate an interrupt to the processor, which will signal the semaphore). While this process is suspended, other processes can be executed. If this is really a domestic robot, it might have a table laying or washing up algorithm which could be carried out. Similarly, a concurrent system is likely to include a standard delay routine which will suspend the process for the required time. The parameter for this routine is assumed to be the number of milliseconds delay required. The other operations (eg fill_kettle) can be declared as procedures.

This algorithm now conforms to standard Pascal syntax and can actually be compiled (omitting the underlines, which Pascal does not require), Figure 4-21 shows the compilation listing which was obtained from the Microprocessor Pascal System. "fill_kettle" etc are declared as EXTERNAL procedures, to be defined elsewhere.

```
DX Microprocessor Pascal System Compiler  3.0  10/23/81 11:41:52

   0      PROGRAM make_tea;
   0
   0          VAR number, cups_required : integer;
   4              kettle_boiling : semaphore;
   6
   0          PROCEDURE fill_kettle; EXTERNAL;
   0          PROCEDURE put_kettle_on; EXTERNAL;
   0          PROCEDURE put_tea_in_teapot; EXTERNAL;
   0          PROCEDURE fill teapot; EXTERNAL;
   0          PROCEDURE wait—(sema : semaphore); EXTERNAL;
   0          PROCEDURE delay (milliseconds : INTEGER); EXTERNAL;
   0          PROCEDURE pour_cup; EXTERNAL;
   0
   1          BEGIN
   1          fill_kettle;
   2          put_kettle_on;
   3          put_tee_in_teapot;
****                          !104
   4          wait (kettle_boiling);
   5          fill_teapot;
   6          delay (5*60*1000);
   7          FOR number := 1 TO cups_required DO
   8              pour_cup
   8          END;
```

Figure 4–21 Compilation Listing for the Tea Making Algorithm

Error 104 is described in the Microprocessor Pascal System User's Manual as ''identifier not declared". The compiler is pointing out that "put_tee_in_teapot" is misspelled. This must be corrected in the final software design. A corrected compilation, with the "(* MAP *)" option set to show the actual variable storage allocated for the module, is displayed in Figure 4–22,

Figure 4–23 shows the reverse assembled TMS9900 object code that was output from the compiler, With a little more work, this module could form part of a real system,

```
DX Microprocessor Pascal System Compiler  3.0  10/23/81 11:31: 7
   0 (* MAP *)
   0     PROGRAM make_tea;
   0
   0         VAR number, cups_required : integer;
   4             kettle_boiling : semaphore;
   6
   0         PROCEDURE fill kettle; EXTERNAL;
   0         PROCEDURE put_Fettle_on; EXTERNAL;
   0         PROCEDURE put_tea_in_teapot; EXTERNAL;
   0         PROCEDURE fill teapot; EXTERNAL;
   0         PROCEDURE wait— (sema : semaphore); EXTERNAL;
   0         PROCEDURE delay (milliseconds : INTEGER); EXTERNAL;
   0         PROCEDURE pour_cup; EXTERNAL;
   0
   1         BEGIN
   1         fill_kettle;
   2         put_kettle_on;
   3 .       put tea_in teapot;
   4         wait (kettie_boiling);
   5         fill teapot;
   6         delay (5*60*1000);
   7         FOR number := 1 TO cups_required DO
   8             pour_cup
   8         END.

PROGRAM MAKE_TEA;
    STACK SIZE = 0006

          VARIABLE  DISP      TYPE       SIZE
          NUMBER    0000      INTEGER    2
          CUPS_REQ  0002      INTEGER    2
          KETTLE_B  0004      SEMAPHORE  2

PROCEDURE FILL_KET; EXTERNAL;

PROCEDURE PUT_KETT; EXTERNAL;

PROCEDURE PUT_TEA_; EXTERNAL;

PROCEDURE FILL_TEA; EXTERNAL;

PROCEDURE WAIT     ( SEMA    :SEMAPHORE); EXTERNAL;

PROCEDURE DELAY    ( MILLISEC:INTEGER); EXTERNAL;

PROCEDURE POUR_CUP; EXTERNAL;

MODULE - MAKE_TEA
   R15 - CONTAINS VALUE OF LOCAL VARIABLE AT DISPLACEMENT 0006
   R14 - CONTAINS VALUE OF LOCAL VARIABLE AT DISPLACEMENT 0008
* LITERAL CODE LENGTH = 000E,   TOTAL CODE LENGTH = 0060
```

Figure 4–22 Corrected Compilation Listing

```
*
       IDT  'MAKE_TEA'                    10/23/81  11:31:32
*
*
       DEF  SYSTM$
       REF  FILL_K
       REF  PUT_KE
       REF  PUT-TE
       REF  FILL_T
       REF  WAIT-
       REF  DELAY
       REF  POUR_C
       REF  S$PRCS
       REF  E$PRCS
       REF  CALL$
       REF  EXIT$P
*                                          LC     HEX    CHAR
       PSEG
SYSTM$ EQU  $
PR     EQU  7  R7
CODE   EQU  8  R8
LF     EQU  9  R9
SP     EQU  10 R10
LO     EQU  $
       DATA LOOOE-LO
       DATA L0054-LO
       DATA >0000                          0004   0000     ..
       DATA >0000                          0006   0000     ..
D0008  DATA >0006                          0008   0006     ..
DOOOA  DATA >0001                          OOOA   0001     ..
DOOOC  DATA >93E0                          OOOC   93E0     ..
*                                            LC   WORD(S)
LOOOE  EQU  $
       MOV  @D0008-LO(CODE),*SP+           OOOE   CEA8     0008
       MOV  @DOOOA-LO(CODE),*SP+           0012   CEA8     OOOA
       SETO *SP+                           0016   073A
       CLR  *SP+                           0018   04FA
       CLR  *SP+                           001A   04FA
       DATA CALL$,S$PRCS                   001C
       DATA CALL$,FILL_K                   0020
       DATA CALL$,PUT_KE                   0024
       DATA CALL$,PUT-TE                   0028
       MOV  @>0004(LF),*SP+                002C   CEA9     0004
       DATA CALL$,WAIT                     0030
       DATA CALL$,FILL_T                   0034
       MOV  @DOOOC-LO(CODE),*SP+           0038   CEA8     OOOC
       DATA CALL$,DELAY                    003C
       LI   R15,>0001                      0040   020F     0001
       MOV  @>0002(LF),R14                 0044   C3A9     0002
L0048  EQU  $
       C    R15,R14                        0048   838F
       JGT  L0054                          004A   1504
       DATA CALL$,POUR_C                   004C
       INC  R15                            0050   058F
```

```
       JMP   L0048                        0052   10FA
L0054  EQU   $
       MOV   @D000A-L0(CODE),*SP+         0054   CEA8   000A
       DATA  CALL$,E$PRCS                 0058
       B     @EXIT$P                      005C   0460   0000
       END
```

Figure 4-23 Reverse Assembled Object Code for the Tea Making Algorithm

4.13 BIBLIOGRAPHY

The following publications develop and extend the ideas presented in this chapter,

[1] Niklaus Wirth Algorithms + Data Structures = Programs

Prentice-Hall

[2] E W Dijkstra, O-H Dahl and C A R Hoare

Structured Programming

Academic Press

[3] Peter Freeman and Anthony I. Wasserman

Tutorial on Software Design Techniques

IEEE Computer Society

[4] Michael Jackson Principles of Program Design

Academic Press

[5] Carol A Ogdin Software Design for Microcomputers

Prentice-Hall

Texas Instruments Publications:

Component Software Handbook (MP918)

Microprocessor Pascal System User's Manual (MP351)

Microprocessor Pascal Executive User's Manual (MP385)

Realtime Executive User's Manual (MP373)

CHAPTER 5

COMPONENT SOFTWARE

5.1 WHAT IS COMPONENT SOFTWARE ?

Component Software is a means of packaging software to address what is perceived as the major problem of microsystems development for the next decade - the "software crisis".

Studies have shown that up to 90% of the development cost for a typical system using programmable hardware will be spent on software. Microprocessor hardware is cheap, but software development is expensive. With software forming the major investment for users, it is vital to manage software development effectively, and to make the most effective use of scarce software skills,

Where the product being developed is to be produced in large quantities (tens or hundreds of thousands), development cost is not significant - divided by a hundred thousand it does not add much to the selling price, But for an increasing number of microprocessor products that will be sold only in tens, hundreds or thousands, development cost is all important. For a 100-off product a single man-month of software development (at around $6000) will add $60 to the cost of each product - before any profit, A typical project will involve at least 4-6 months of software development.

Component Software is a way of providing packaged functions that are significantly more powerful than any currently available, either in software or in hardware, These functions consist of "encapsulated software" that can be purchased ready written and tested, and "plugged in" to a user's application, Unlike conventional applications software, the Component Software environment allows packaging of real time functions that can execute either concurrently or in sequence with other functions in an application system, This capability overcomes most of the restrictions of sequential software for writing real time control systems, and many other types of application, The framework .ensures complete security of function packages, so that functions cannot interfere with one another,

Because of the flexible packaging of Component Software, systems can be designed and constructed in terms of meaningful application-oriented functions, rather than

abstract software routines. Many of these functions can be purchased off the shelf, or reused from previous systems.

Component Software is the first step in a more radical approach to systems design using programmable components. Many functions first identified and packaged in this way will eventually be "canned" in silicon, as dedicated hardware functions.

Component Software is supplied as libraries of software modules stored on magnetic media (such as floppy discs), together with full documentation. The packages are designed to be configurable in many different ways, to suit individual application needs. Configuration involves selecting the software modules required from the library supplied, and linking them together with the user's application program. This semi-automatic process gives the system designer a higher level of programming capability(he can manipulate complete functional blocks in a real time environment), supplementing already available software development tools.

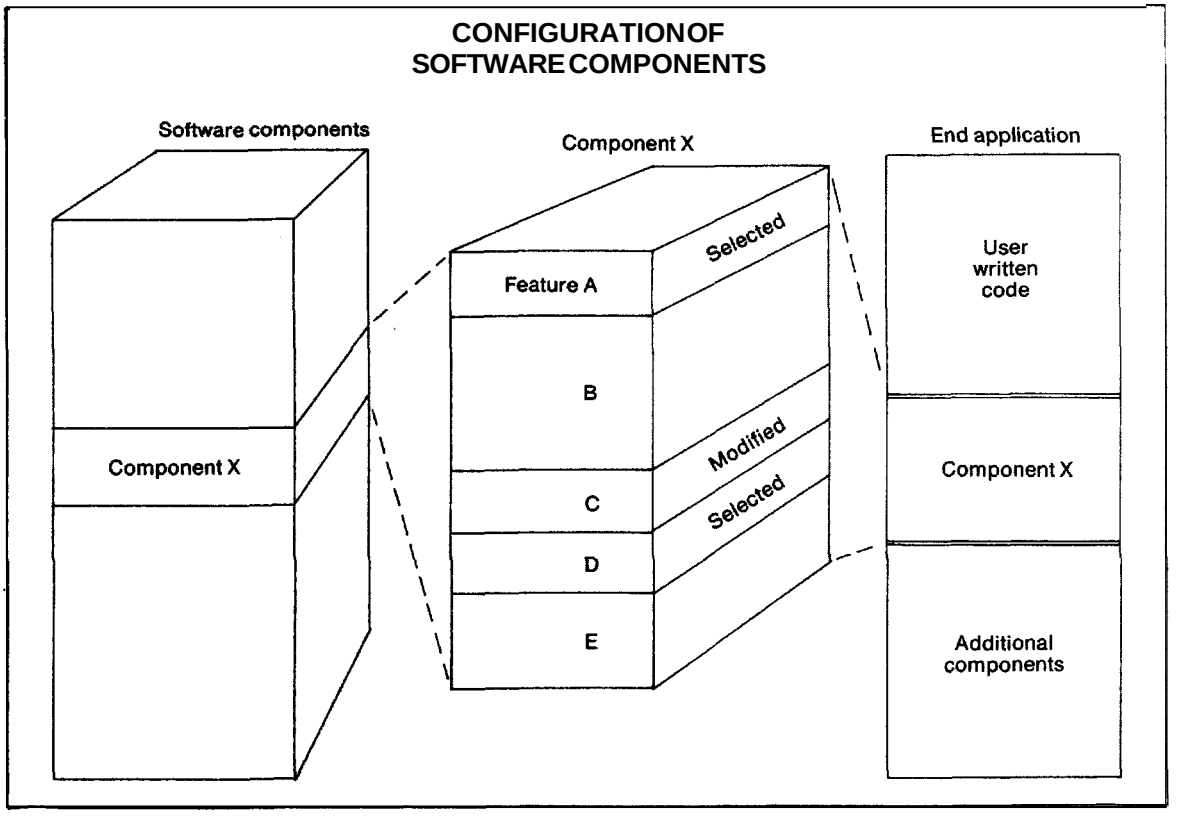


Figure 5-1 Configuration of Component Software Packages

Individual features of the package can be selected or left out, according to the needs of each application. Packages are designed to permit several levels of access - from a high level, trouble-free interface that requires minimum knowledge, to a low level interface that gives direct control over the workings of the package, but requires greater expertise to use effectively. System designers can choose whichever level is most appropriate for each particular application,

A typical Component Software package can be used in different ways in many different applications. A library of common application functions can be built up, which can supply component parts for new applications. Users can write their own Component Software packages - the Component Software Handbook, MP918, describes haw to do this. Texas Instruments (TI) encourages the production and sale of Component Software packages by other companies.

It is expected that configuration from pre-compiled object modules will supply most application needs, but TI also supplies source code as standard for all routines, For those applications which require it, functions can be customised at the most detailed level using standard Microprocessor Pascal and/or assembly language development tools.

5.1.1 The Functional Approach

Component Software makes possible a functional, application-oriented approach to system design. First, an application is analysed into the individual functions that are to be performed. This functional analysis can be done in whatever way is naturally appropriate for the application. Next, the requirements for each function, and the interaction between the separate functions, are unambiguously specified. A precise algorithmic description of the operation of each function will lead straightforwardly to a high level language software implementation (which can he optimised in assembly language if required). The structure of Component Software means that separately developed, concurrent functions can be connected together simply and with confidence. Testing can he carried out on each function individually, and on the system as a whole. Finally a choice of hardware can be made, from a range of options, to provide the required cost, performance and environmental suitability.

Traditional forms of system design rarely start with the application - they usually require choosing a hardware configuration, often with barely adequate information, at the start; and then building up software on top of this to adapt the hardware to the application requirements.

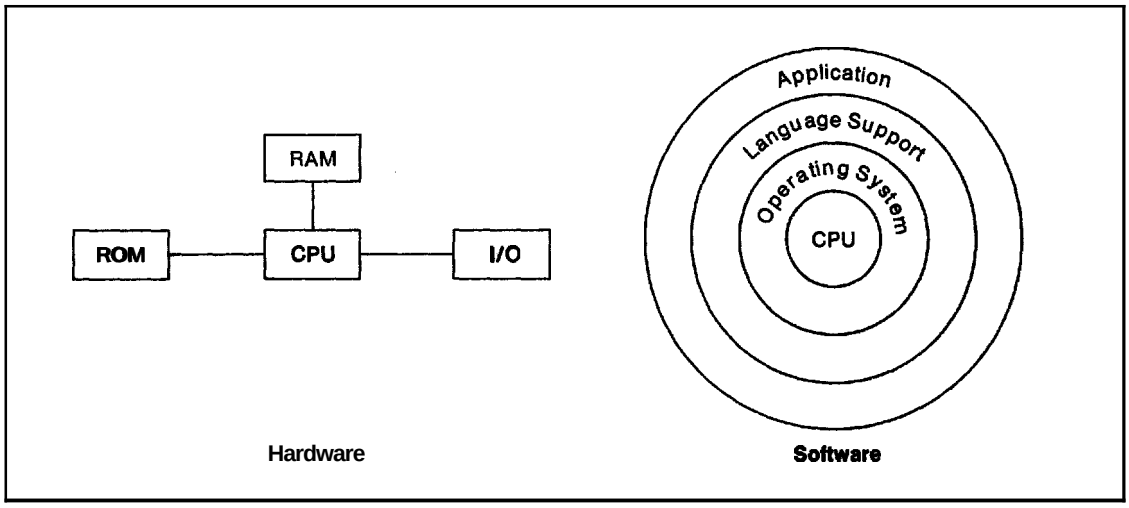


Figure 5-2 The Traditional Approach

Bridging the gap between the chosen microprocessor hardware and application requirements usually involves major design effort, with skills that are rare. In addition, the design produced is likely to be "brittle[1]' rather than flexible, because built into it are assumptions about a particular type of hardware and a particular set of application requirements. Incorporating new hardware or new requirements usually means major redesign of both hardware and software, and consequent problems of testing and reliability.

The functional approach places few arbitrary restrictions on the development process. Both the software algorithms(which determine how an application functions) and the hardware (which determines price and performance) can be varied independently, with minimal effect on the rest of the design. The constructs of Component Software are sufficiently flexible that systems can be structured according to the nature of the application, whatever it is, rather than being shaped by the necessities of the technology. Systems built like this are both more responsive to application requirements in the first place, and easier to change if the requirements alter.

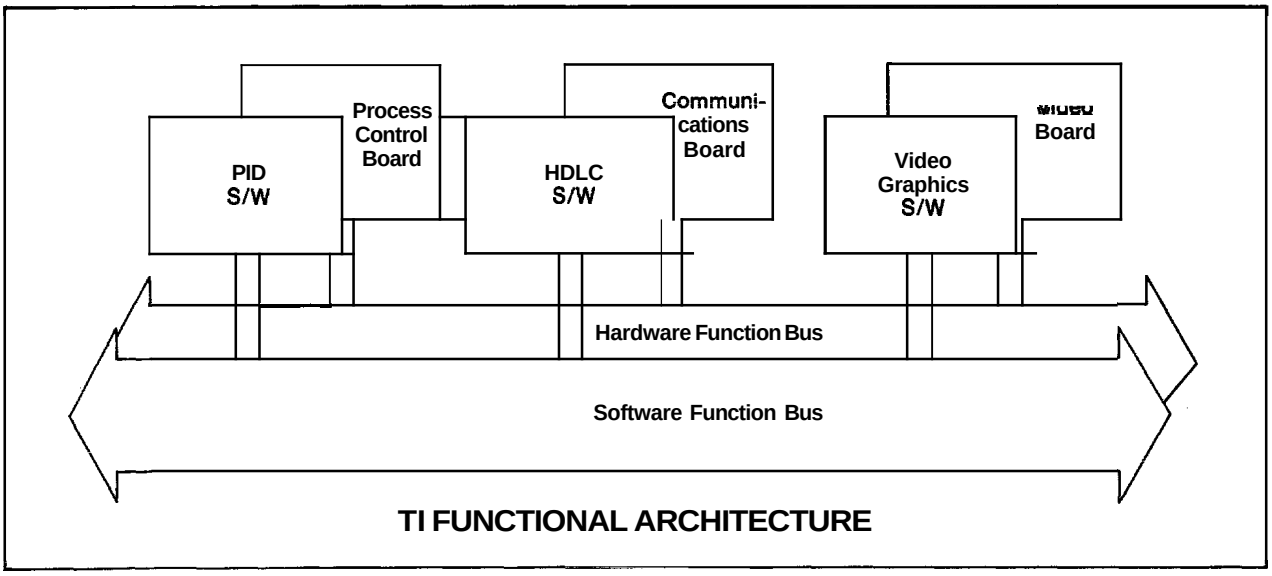


Figure 5-3 TI Functional Architecture

How to divide an application into functional parts for separate development may be immediately obvious from the nature of the application; or functional "packages" may be chosen according to the division of available engineering resource to implement them. Packages may also be chosen to encapsulate areas of a system which may be reused, or areas which are likely to change. In any case, the ability to encapsulate real time functions (which may have a concurrent structure - see below) can be used to advantage.

Systems can be upgraded incrementally by changing or replacing separately developed functions. The Component Software environment ensures that separate functions are enclosed, so that changes will have no effect on other parts of the system.

TI's microprocessor hardware provides a wide range of price, performance and environment options (available either as individual LSI and VLSI components, or in a range of prepackaged board modules), all with a common software interface. The 9900/99000 instruction set defines a low level standard interface; the Realtime Executive (Rx) defines a standard at a higher level of capability - the Software Function Bus - that incorporates concurrency, standard management of system resources, and all the features required to implement Component Software. Versions of Rx will be available to adapt the standard software interface to multiple processors and various types of memory configuration.

The functional approach can be seen as a generalisation of the "Top Down" and "Structured Programming" approaches which have been successful in achieving reliable software design. Here, the approach is applied to system design, in particular to the design of real time systems,

5.1.2 Function to Function Architecture

The functional approach of Component Software forms part of a broader architectural scheme called Function-to-Function Architecture, which integrates both hardware and software in the service of useful functions, Function-to-Function Architecture (FFA) defines a standard interconnect mechanism between complex functions, however they are implemented - in hardware, software, or a combination of both, It makes possible early definition and implementation of functions in the flexible medium of Component Software, Once the usefulness and reliability of a function has been proved, it can be migrated to progressively "harder" implementations. Those functions which justify it will eventually end up as custom VLSI silicon chips, The standard interconnect mechanism means that systems will be upgraded gradually by replacing individual functions to give improved cost, performance or features, without having to redesign the whole system,

5.2 THE COMPONENT SOFTWARE ENVIRONMENT

The Component Software Handbook, from which this chapter is extracted, gives further information on the construction and use of Component Software packages, and precise terminology. This section provides an overview of the Component Software environment. Terms such as "function", "program" etc are used here in a general rather than a specific technical sense, except where capitalised.

5.2.1 Concurrency

Component Software supports concurrency - i.e. simultaneous execution of a number of different software programs.

Conventional programming environments only allow the user to run one program at a time. However, a typical microprocessor system may be required to perform a number of different functions at once.

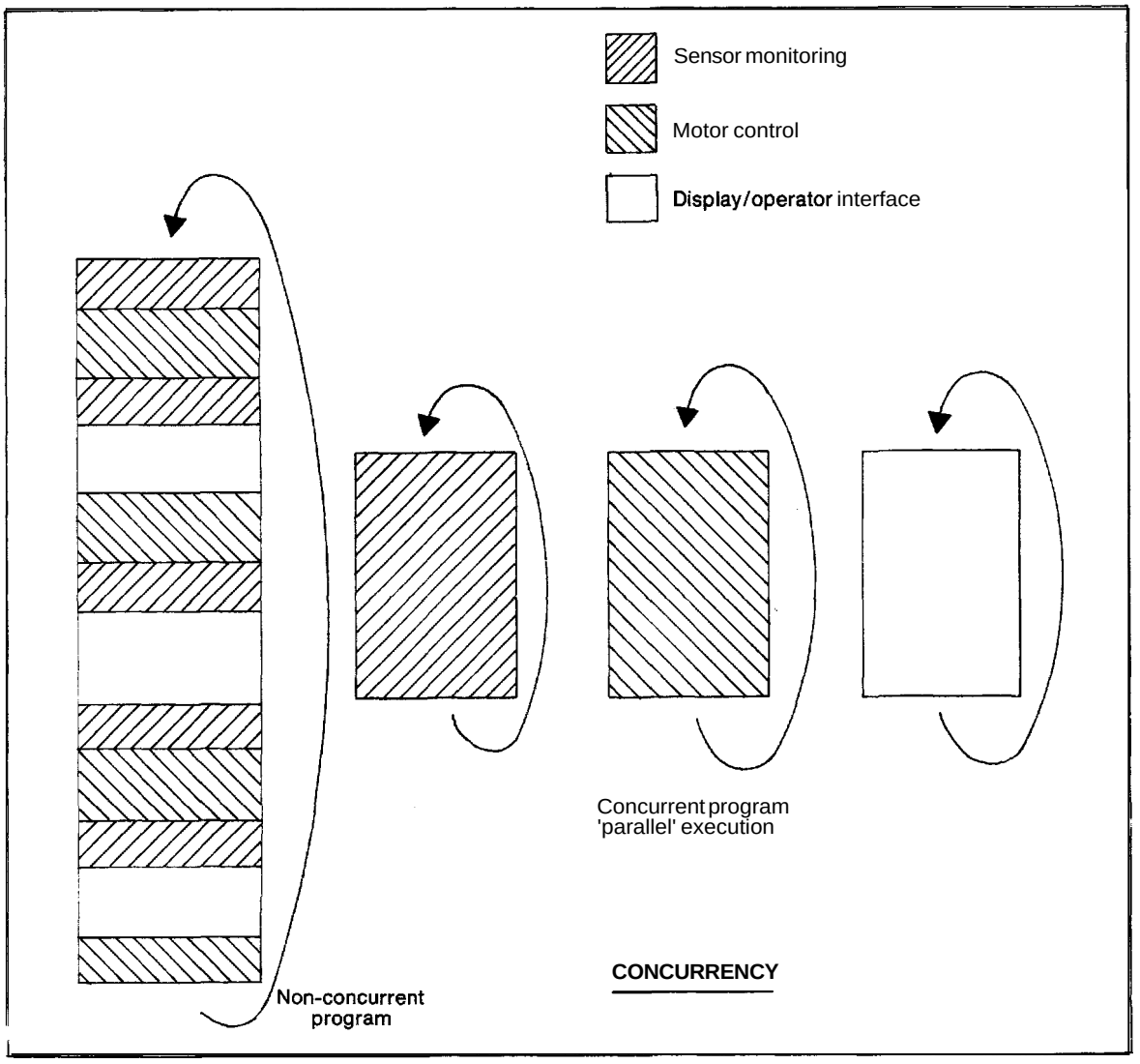


Figure 5-4 Concurrency

For example, a system controlling a group of manufacturing machines may be required to monitor and control each machine, continuously check safety conditions, select and record information for costing each job as it appears, and still to respond immediately to commands from its operator,

Reducing all of this to one sequential list of instructions (a conventional program) is a very difficult task. The result (if it turned out to be possible) would be a very convoluted program that breaks off in the middle of doing one thing to perform another, halts that to carry out a third, and so on. Such programs are difficult to understand and awkward to maintain. They are also nearly impossible to test,

Conventional software is built on the assumption that functions will be executed one at a time, in sequence, Each function must start, execute and terminate before another function can begin.

But the real world does not always (or even usually) behave like this. A typical real time application system will need to do several things "at once". Even though each individual task may only require periodic attention, the system must keep track of everything that is going on, carry out each task when it is required, and must also respond immediately and correctly if an unexpected event occurs, A control function, for example, may need to check the status of a machine or a chemical process continuously over a period of hours, However, the check may only require a small calculation every half second (say),

To dedicate a complete processor to this function would be wasteful; yet conventional application software provides no standard means of using the processor to perform another function in the meantime, while ensuring that the check gets made every half second, and that the two functions do not interfere.

Demands on the system may occur not only at fixed time intervals: from the system's point of view, it is completely impossible to predict when an operator is going to press a button, or when a temperature will exceed a safe margin - but it is important to respond quickly and reliably, and without disrupting the operation of the rest of the system.

For a specific application, it may be possible to solve these problems in a sequential program. However, to do so would require a great deal of effort, and would result in an ad hoc solution, very specific to one application. With software constructed in this way, it is not unknown for an apparently simple change in the specification (say, the need to check the status of a machine every quarter second rather than half second) to require a complete redesign of the system, Additional problems arise when trying to test such systems.

What is needed is a standard framework in which this class of problem is handled automatically. The system designer can then specify and write each individual function separately, and evaluate and test it independently, Applications can be built up by selecting the required functions and linking them together (semi-automatically) to construct a complete system - analogous to the process of connecting together ICs using a printed circuit board, This standard framework is provided by Component Software,

In the Component Software environment, functions are considered to be independent, and may have a sequential and/or a concurrent relationship with other functions, The designer may specify that one function must wait for another function to complete before it executes, but (unlike conventional software environments) he can also specify that the two functions should take place concurrently. For example, a user's program can initiate an 1/0 request (such as a read from floppy disc), but need not wait for it to complete before going on to do something else. The system will automatically complete the transfer, taking care of the hardware timings and delays of the floppy disc controller and the necessary format conversions, in a way that is completely transparent to the rest of the software,

Explicit support for concurrency is an important element in the framework, It makes possible the construction of systems which perform real tasks, easily, cheaply and reliably, and permits software to be structured in a natural way that reflects the real world. It allows a functional approach (as outlined above) to be applied to software - because the natural analysis of an application will rarely result in functions that have a simple sequential relationship.

5.2.1.1 Packaged Functions

Software libraries have existed before, but they have generally been libraries of routines that only execute sequentially, There is a limit to the type of function that can be placed in a purely sequential package,

Sequential software is well suited to a restricted class of operations - those operations that can be specified by a single list of instructions. Unfortunately, by no means all of the tasks to be performed in the real world can be specified as simply as this. Microprocessors, by virtue of their cheapness and effectiveness, are required to perform a wide variety of tasks which mainframe computers were never called upon to do. Consequently, a more powerful medium is needed to program them effectively - a framework which incorporates concurrency.

A "package" such as a process control function looks quite different from a sequential software routine. The package

may include a piece of code to be executed automatically every (say) half second, plus some routines callable by a user's program to set up and change the control parameters, obtain status information etc; and maybe some logging routines, again executed automatically at fixed intervals, to record selected data regularly on disc. The package contains a number of functions which must be executed at different times and in different ways - some automatically .at fixed time intervals, some on demand from the user's application program (perhaps halting the flow of the user's program while they execute, and perhaps not), and some on detecting a particular out-of-range condition (say).

Component Software is designed to accomodate such complex "packages" as this. Using the basic constructs provided by the Software Function Bus, algorithms written in a high level programming language (or in assembly language) can be combined in a variety of sequential and concurrent relationships to build a complete package implementing, say, a file manager or a machine controller. The simplicity of the basic constructs means that parts of any package can be isolated and tested independently, using interactive debugging tools.

The complete package (or such parts of it as are required) can be incorporated in a larger system easily and quickly, with the knowledge that it will not interfere with any other function in the system.

5.2.1.2 Implementation of Concurrency

Functions which execute concurrently can be regarded as taking place independently and simultaneously. Functional design, and the Component Software environment, makes no fundamental assumptions about how this concurrency is implemented. The "simultaneity" may involve two or more separate hardware processors, or may be simulated in software with a single processor.

In a single processor environment, concurrency is implemented by switching the processor between the different functions to be performed, according to the demands of the system and priorities set by the user. This switching is called scheduling. More generally, scheduling can be regarded as the allocation of available system resources to the different functions competing for them. The statement that "a function is separately scheduled" means that it competes independently for system resources, according to priorities set by the system designer. In a Component Software system, the designer chooses which functions are actively independent, and hence need to be separately scheduled. Generally, functions which have independent timing requirements, or which take place over long periods of time, should be separately scheduled.

Functions which are not separately scheduled can be regarded as "passive", and only execute when called on by an "active" function. The scheduling policy is designed to ensure that the task being performed by the processor is always the most urgent one, and in particular that external events (eg a signal from a device connected to the system) are responded to immediately. Scheduling is described in detail in the Microprocessor Pascal System User's Manual (MP351) and the Realtime Executive User's Manual (MP373).

With a single processor, concurrency provides the advantages of increased clarity of system design (which means easier maintenance, testing and upgrade), functional packaging, and improved throughput (because the processor need never be idle, waiting say for a slow output device to respond - it can switch to performing some other function). Concurrency means that the system has some degree of dynamic flexibility: it can respond to changes in the demand for any function by reallocating resources from less urgent functions.

With multiple processors, throughput will be further increased because there is more than one active processing element. Reliability may also be increased, because (with appropriate design) the whole system need not collapse if one processor fails. However, a multiple processor system is likely to be more expensive. It is intended that Component Software programs can be executed on the same processor or on a distributed network of processors, with minimal impact on the programs themselves or their interaction. The system designer will then choose the hardware to implement his functional design purely on the basis of cost and performance tradeoffs. Adding another processor, say, to increase throughput will no longer be a major design exercise. Currently, multiple processor systems can be built in which functions executing in different processors interact through file level messages across standard communication links (eg HDLC or EIA). Future versions of Rx will support more closely coupled multiple processor systems.

5.2.1.3 Levels of Concurrency

The Component Software environment permits concurrency not only between complete function packages, but within packages themselves, This means that a complex function, such as the HDLC Data Communications package, can be designed as a collection of subfunctions that may execute sequentially and/or concurrently.

Typically, a users program will pass a data record to the HDLC subsystem, for transmission over the HDLC communications network. The HDLC subsystem then performs all the work needed to transmit the record to its destination. Within the HDLC package are a number of concurrent functions which manage the different levels of HDLC protocol, interact with

the physical data link, and check that receipt of correct data is acknqwledged within a specified time interval. If acknowledgement is not received, or if an error is signalled, the HDLC subsystem will retransmit the data. Efficient and reliable implementation of this kind of "intelligent" operation requires concurrency. The Component Software environment permits such an intelligent function to be encapsulated in a single package which has a simple interface with the users program (for example, it can be accessed through straightforward sequential procedure calls).

The internal structure of such a function package is completely invisible to the user, unless he chooses to interact with the package at that level of detail. The package can be initialised automatically at power up, and will perform throughout as an'enclosed operation, complete in itself.

5.2.2 Code, Data and Re-entrancy

Component Software is designed to make efficient use of the memory space available in a microprocessor system, and to maintain strict separation between program code and data. Separation of code and data improves system integrity (making accidental modification of code less likely), makes possible re-entrancy (as described below), and permits easy partitioning into read only and read/write memory (ROM and RAM), which is often required in a microprocessor system.

The fundamental unit of instruction code in a Component Software system is the routine. A routine is a sequence of processor instructions that performs a particular operation.

Component Software provides a set of constructs that group routines together, define which routines will have access to which other routines, and determine how routines will interact (sequentially or concurrently). The Component Software Handbook describes the detailed structure of a Component Software package, and how to construct one. Within a separately compiled Component Software module (which will probably include several routines), the rules of scope define exactly which routines and which data structures are accessible at each point in the software. (See the Microprocessor Pascal System User's Manual for a complete discussion of scope,) Between modules, explicit EXTERNAL declarations in each module specify exactly what connections are to be permitted with other modules.

The structure of a Component Software system is shown in figure 5-5.

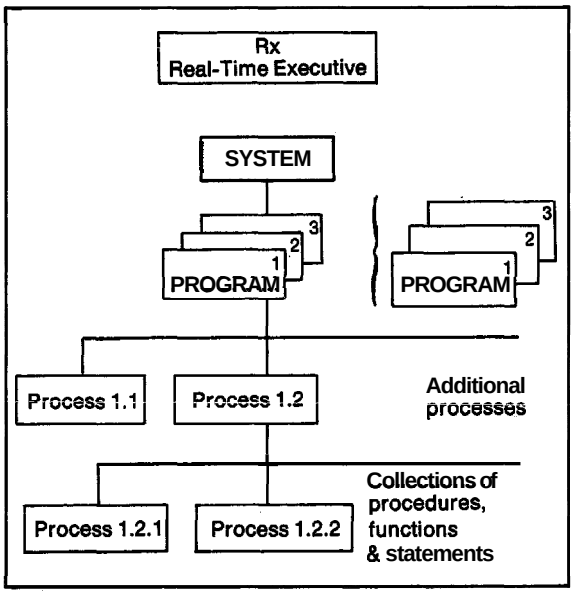


Figure 5-5 SYSTEMs, PROGRAMs and PROCESSes

For implementation as a Component Software package, application functions must be implemented as groups of PROGRAMs, PROCESSes, PROCEDUREs, and FUNCTIONs. A SYSTEM is likely to contain a number of independent, separately scheduled PROGRAMs. However, a PROGRAM may also have a hierarchy of dependent PROCESSes - separately scheduled, but related. Strictly, the term PROGRAM applies only to the single, "top level" routine in the group. The complete structure of a PROGRAM with all subordinate PROCESSes (and PROCEDUREs and FUNCTIONs - see below) is referred to as a PROGRAM family. Continuing the analogy, routines further up the hierarchical tree are referred to as "ancestors"; those lower down are "descendants". The PROGRAM family is a convenient package for a complete, independent function within a system.

PROGRAMs and PROCESSes are independent routines which are separately scheduled; however the hierarchical relationship makes it possible to isolate and develop separately not only single routines, but also complete groups of concurrent routines implementing a complex function.

PROGRAMs and PROCESSes are the "active" elements in a Component Software system. "Passive" routines can also be defined, which may be called on by an active PROGRAM or PROCESS to perform a specific function. These are PROCEDUREs and FUNCTIONs. (NB "FUNCTION" capitalised has a precise technical meaning, as distinct from the more general use of

"function").

A PROCEDURE or FUNCTION never competes directly for system resources; it always executes under the wing of a PROGRAM or PROCESS, and provides a particular "skill" that the PROGRAM or PROCESS may need at the time, PROCEDUREs and FUNCTIONs can be used to encapsulate functions which are simple enough not to require the power of the PROGRAM family construct to implement them,

Depending on where a PROCEDURE or FUNCTION is defined, it may be accessible to some or all of the routines in the system, PROCEDUREs and FUNCTIONs declared at the level of the SYSTEM are available to any routine. They may also be declared at some point in the hierarchy of a PROGRAM family, so that access to the PROCEDURE or FUNCTION is restricted to that PROGRAM family or part of that family,

The Microprocessor Pascal System User's Manual (MP351) and the Realtime Executive User's Manual (MP373) give more details about the structure of Component Software systems.

5.2.2.1 Memory Allocation

Before it is activated, a software system is simply a collection of dormant instruction code, grouped into routines, and probably stored in ROM. To perform any useful work, a routine must be activated and allocated data space with which to work, The stock of dormant routines can be regarded as the "repertoire" of the system, which is called upon as needed, The task of the system designer is, first, to ensure that there are adequate functions in the repertoire; second, to activate them as needed to perform the task required, When a Component Software SYSTEM is powered up, system data structures will be initialised, any 1/0 subsystems (see below) will be initialised, and any user defined initialisation will be performed, Typically, the PROGRAM(s) present in the SYSTEM will then be started, All action beyond this point is dependent on the system designer. He may

1. design a system that is a single sequential PROGRAM
2. use two or more concurrent PROGRAMs, each of which is sequential
3. within a PROGRAM, start more concurrent PROCESSes to create a PROGRAM family
4. incorporate Component Software packages, of which he he may or may not know the internal structure

Each call to start a PROGRAM or PROCESS is said to activate a new site of execution within the system, which executes independently of every other site of execution. In the following discussion, what is said about PROCESSes applies also to PROGRAMs: a PROGRAM is a special case of a "top level" PROCESS, Whenever a PROCESS is activated, it is allocated by the executive an appropriate amount of data memory from a pool (known as the heap). This allocated memory is returned to the heap when the PROCESS terminates, so that it can be allocated to other PROCESSes, Processor time is allocated to each PROCESS according to demand and the priority given to the PROCESS when it was started,

PROCEDUREs and FUNCTIONs that are called by a PROCESS borrow memory from that PROCESS's allocation, and use processor time scheduled to that PROCESS. The PROCESS gives its resource to execute that PROCEDURE or FUNCTION, and cannot do anything else until it is complete. Each PROGRAM or PROCESS can be thought of as an independent, single "thread" of logic within the system, with its own timing characteristics and separate existence. PROCEDUREs and FUNCTIONs provide a kind of "stored logic" that can be inserted in the thread of a PROGRAM or PROCESS at an appropriate time. PROCESSes may request additional memory from the heap while they are executing.

5.2.2.2 Multiple Activations

Because the instruction code for a PROCESS is completely separate from its data space, and is never changed, it can be activated more than once. For example, a factory may contain several identical machines, all controlled by one system, The control program for each machine is identical, and only one copy of the instruction code need exist, However, several activations of the control program may be present at the same time, using the same instruction code but different data spaces. There will be no conflict, The same applies to PROCEDUREs and FUNCTIONs: as the data space for executing any PROCEDURE or FUNCTION is allocated from the data space of the calling PROCESS, several PROCESSes may call a general purpose PROCEDURE (a matrix multiplication routine, for example) at the same time without problems. The routine code need only exist once within the system. This property of software is known as re-entrancy,

5.2.3 The Realtime Executive

The Realtime Executive (Rx) is the backbone and artery of a Component Software system; it supports the other functions and provides commonly needed services, Within Rx are the routines that allocate system resources (processor time, memory, I/O) between the different PROCESSes, according to

demand and priorities. Also within Rx are the standard procedures that allow one routine to call or start another. Finally, Rx contains the code that permits concurrent PROCESSes to synchronise their operation with other PROCESSes or external events, and allows PROCESSes to pass data to other PROCESSes.

The most basic synchronisation is achieved using a low level software mechanism called a semaphore. A semaphore allows one PROCESS to signal occurrence of an event (eg machine_operation_complete) to another,

It is Rx which sets up the Component Software environment, and maintains it. Rx establishes a "Software Function Bus" - a standard, concurrent interface into which Component Software functions can be "plugged".

5.2.3.1 Channels and Interprocess Files

Data communication between PROCESSes can take place over channels. A channel is simply a means of passing data from one PROCESS to another in a way which ensures that the integrity of the data is preserved (eg that one PROCESS does not try to read data until the other has finished writing it), and that the data is placed in an area of memory that will be accessible to both PROCESSes. Channels can also be used to provide a higher level of synchronisation.

A further method of communication is the interprocess file mechanism. This allows a PROCESS to write to another PROCESS exactly as if it were writing to an input/output device, using the standard file 1/0 primitives (see below).

The hierarchical system structure defines a clear relationship between the concurrent PROGRAMs and PROCESSes in a Component Software application, However, this may not be sufficient in all circumstances. The channel and interprocess file mechanisms allow any PROGRAM or PROCESS to connect to and exchange data with any other PROGRAM or PROCESS in the system (provided both "ends" prepare for and understand the exchange). These connections are made dynamically while the system is running. Connections of this kind can be "hard coded" into the routines when they are written, in which case they cannot be altered. However, it is also possible to write systems in which the connections can be modified at run time, either by an operator or by a piece of "intelligent" software, in response to changing requirements, or perhaps in response to failure of part of the system. With a system constructed using interprocess files, connections can be rerouted from a local PROCESS to an external device, or perhaps via a data link to a PROCESS in a completely different computer system. Requests for dynamic connections of this kind are made via executive routines which ensure that system integrity is preserved in making the

connection.

5.2.3.2 Rx vs Operating Systems

Many of the functions performed by the Realtime Executive (Rx) would be handled in a mainframe computer by an Operating System. Early computers suffered from the problems outlined above in the section on concurrency - namely, how to adapt a basically sequentially machine to a range of independent, probably simultaneous requirements. However, the scale of the problem for mainframe computers was different - requiring solutions to problems typically within hours or days rather than milliseconds. So human operators were introduced to share out the resources of "mainframe" computers between different users. Later, software Operating Systems (OSs) were designed to partially automate the process.

For mainframe computers, the tasks of programming and operating the computer remained very separate. Separate disciplines evolved, and people were trained to perform one job or the other.

A microsystem designer needs .to have direct control over both the programming of the functions to be performed, and the operation of the system. Typically, operation of the system (as regards controlling the execution of different functions) needs to be completely automatic in the final system, but the system designer should have a good measure of control over how this operation takes place - that is, just how the computer makes its millisecond-to-millisecond decisions on what to do next.

The requirements of an Operating System for a large general purpose computer, and an executive for a dedicated microcomputer system, are very different.

Traditional Operating Systems were designed to maximise the use of the computer's hardware resources - which at the time represented a huge capital investment. With cheap, distributed microcomputer power, the balance has shifted, and other factors - such as development, support and maintenance costs, and software correctness - are now more important than keeping the processor occupied 100 per cent of the time. In addition, a large, centralised general purpose computer has a complete set of resources, hardware and software, on hand at all times. There is no incentive for selecting the minimum set of resources required to implement a particular application. Where a product is to be produced in large quantities, the tradeoffs are quite different.

Operating Systems can afford to be large, monolithic structures that are always present for every application. An executive needs to be small, and tailored for each application (by configuring from a standard "kit of parts").

Thus, although Rx draws ow techniques iearnt from the design of operating systems, its structure is significantly different in many respects.

An Operating System is usually pictured as a set of concentric circles, centred on the (single) mainframe processor.

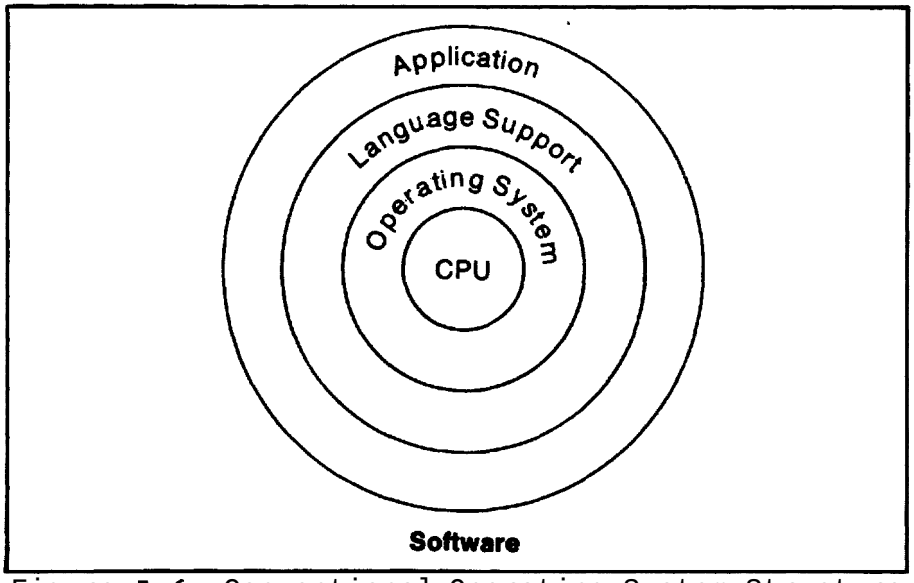


Figure 5-6 Conventional Operating System Structure

This structure is large, monolithic, and difficult to get inside (the shell is "hard"). An Operating System tends to be a union of all possible system requirements, and is difficult to split apart. Rx looks more like a "bus":

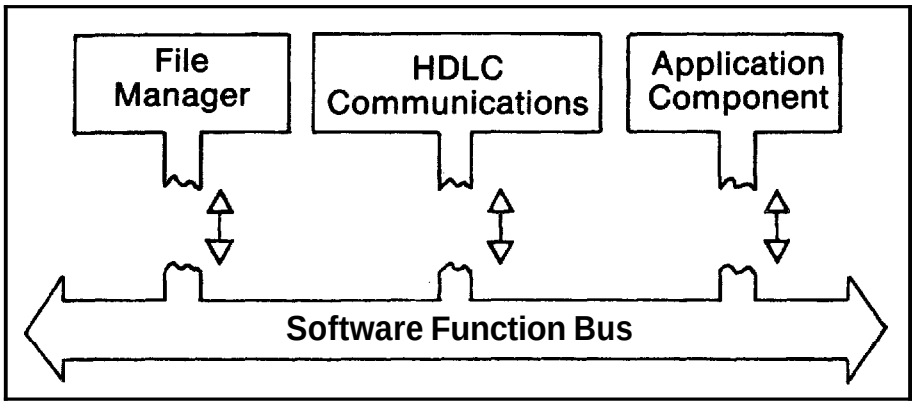


Figure 5-7 Software Function Bus

The Rx Software Function Bus establishes a set of conventions which are expected by the Component Software functions, This set of conventions can be implemented on virtually any hardware architecture, Versions of Rx will implement the standard Software Function Bus across a range of different single- and multiple-processor configurations, and memory schemes, Different Component Software functions can be "plugged into" the standard bus to expand the total capability of the system,

The requirements that led to the adoption of Component Software for application programs apply equally to systems software, Rx is itself a Component Software package - a "kit of parts" for constructing an executive customised to each application,

The Rx executive is "built" for each particular application by selecting (automatically) the functions actually used by the application, from a library of executive functions,

5.2.4 File I/O Standards

The Component Software environment standardises input and output so that systems can be built up using any combination of 1/0 devices without danger of conflict. Systems can incorporate a wide range of standard hardware and software, and can also include custom I/O.

The concurrent nature of the Component Software environment permits many asynchronous devices to be handled simultaneously. An independent process is assigned to each device, associated with an appropriate interrupt. The execution of this device process is synchronised with the device, and the process is activated according to the needs of the device, 1/0 routines called by the user's process will be synchronised with the user, and will respond to the user's needs, The two will interact via channels, The concurrent structure thus manages automatically the timing and synchronisation between user program requests and hardware I/O operations.

5.2.4.1 I/O Subsystems

1/0 software is grouped into subsystems, each subsystem handling a particular class of devices - rotating mass store (magnetic discs), for example, or HDLC data communication devices,

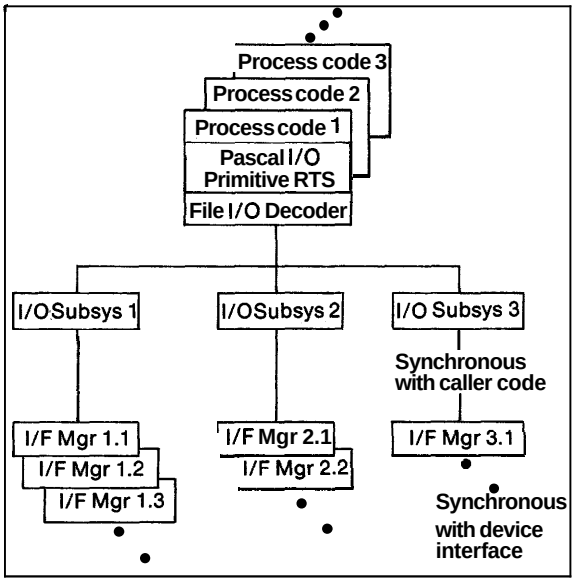


Figure 5-8 I/O Subsystems

Many Component Software packages will take the form of a complete 1/0 Subsystem. The 1/0 standards define a common set of high level operations on files (read, write, open, close etc), so that programs can be written without knowledge of the particular type of device they will be using, In this case, all device-dependent details will be hidden within the 1/0 subsystem.

The 1/0 standards also specify lower levels of interface, so that users can interface with 1/0 devices at a device dependent level, This will reduce the code size of the final application, but requires knowledge of the specific characteristics of the device, and of course means that application programs must be rewritten for use with a different device, In all, 5 levels of I/O interface are defined, Designers can choose to include as much or as little of the 1/0 structure as required,

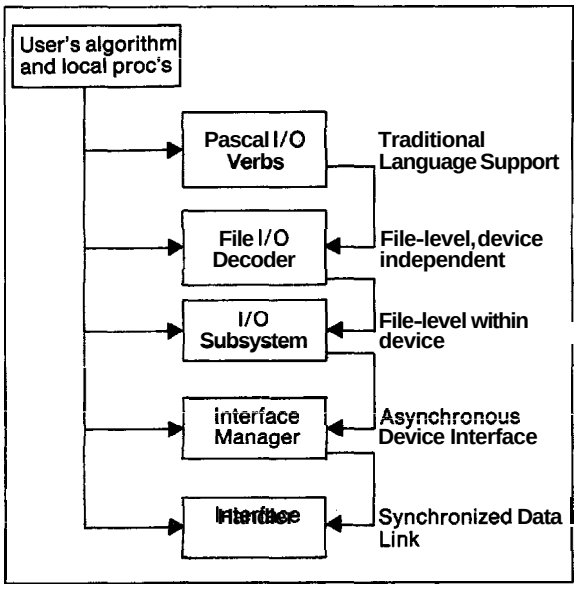


Figure 5-9 5 Levels of Interface to I/O Subsystems

The I/O standards provide for grouping of all hardware related details (I/O addresses, interrupt levels etc) in one system configuration module, for ease of system design. A standard method is provided for initialising I/O subsystems and for handling device interrupts. The I/O Standards and 1/0 Subsystems are discussed in more detail in the Component Software Handbook, MP918, and in the Device Indepedent File I/O User's Manual, MP355.

Texas Instruments supplies standard Component Software 1/0 subsystems for use with TM990 board modules and TMS99XX peripheral components. The I/O subsystems supplied by Texas Instruments are extensively documented and supplied with source code (as are all TI Component Software packages), and can be modified or used as templates to write I/O subsystems for custom hardware devices.

5.2.5 Configuration

Microcomputer systems typically differ in two respects from general purpose mainframe and mini computers. First, a microcomputer application is likely to be more cost sensitive. Second, a microcomputer system is likely to be dedicated to a specific application or range of applications, and will often be embedded in another piece of equipment.

These two requirements dictate the need for configuration. A microcomputer system cannot afford to include features (hardware or software) not actually used by the application.

A Component Software package is supplied as a library of software functions and subfunctions stored on a magnetic medium - such as a floppy disc. To build a system, the designer will write an application program that makes use of some of these functions, select the functions from the Component Software Library, and then link them together with his application program to build a target system. The process of selection and linking is largely automatic, and is called configuration.

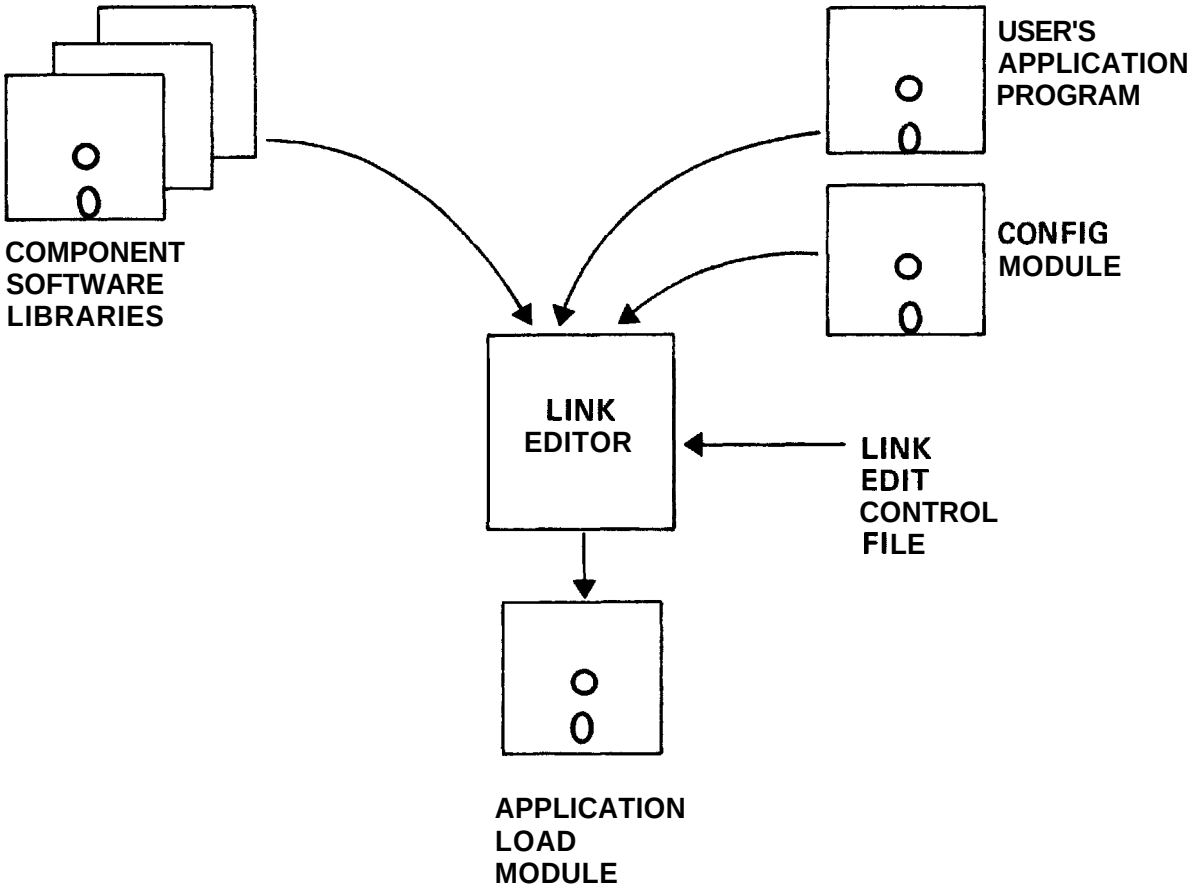


Figure 5-10 Configuration

Success of this approach depends on the division between functions being well chosen, so that a designer is not faced with having to include a software module only part of which he wants to use. This must be a prime consideration in the design of Component Software packages; the concurrent structure makes it easier.

5.2.6 Customisation

For the great majority of applications, configuration alone will he **sufficient to tailor** Component Software packages **to** particular needs. A range of different requirements have been foreseen in developing each package, and comprehended in the division of each package into functional modules,

However, for cases where configuration is insufficient, source code is included in Component Software packages, together with sufficient documentation to allow complete customisation. For example, the device service routines **(DSR's)** of an 1/0 subsystem package can be rewritten for non-standard devices, retaining the higher level routines. Component Software is **written in most cases in concurrent** Microprocessor Pascal, and supplied with documentation which fully describes the structure of the package, so that customisation is relatively easy.

5.2.7 Microprocessor Pascal

The Component Software environment supports **TI's** Microprocessor Pascal. Pascal was designed as a high level, application oriented language in which the sequence of steps required to perform a particular task (an algorithm) can be expressed easily and naturally. Writing a Pascal program requires little more than a precise specification of **what** the program is to do. This means that programs can be developed easily, quickly and reliably. Complex programs can be written much more quickly than in assembly language, and with fewer errors. It also means that the program developed is independent of any particular set of hardware.

TI's Microprocessor Pascal extends the original Pascal definition by incorporating within the language the constructs of Component Software. PROGRAMs, **PROCESSes,** PROCEDUREs and FUNCTIONs can be declared directly in the language. Synchronisation and communication mechanisms (eg semaphores) are also directly available. Microprocessor Pascal extends the scope of the Pascal language to the area of real time systems, retaining the original philosophy of the language and developing it for the real time environment.

Using Microprocessor Pascal, results can be achieved more quickly with less resource and less headaches. Management of projects becomes simpler and more rewarding, because Pascal programming is easier to schedule and control, These points have been proved by software projects undertaken within Texas Instruments (TI). TI has adopted Pascal as a corporate standard language, and trained thousands of programmers to use it, (Contact TI for details of courses on Microprocessor Pascal programming, and other subjects.)

Because Microprocessor Pascal can be "read" like English it is partly self documenting. Comments can be inserted to explain anything which is not made clear by the code itself. With a well written program, paper documentation can be reduced to a description of the program and data structures and of the routine functions, and, where appropriate, a Users Guide.

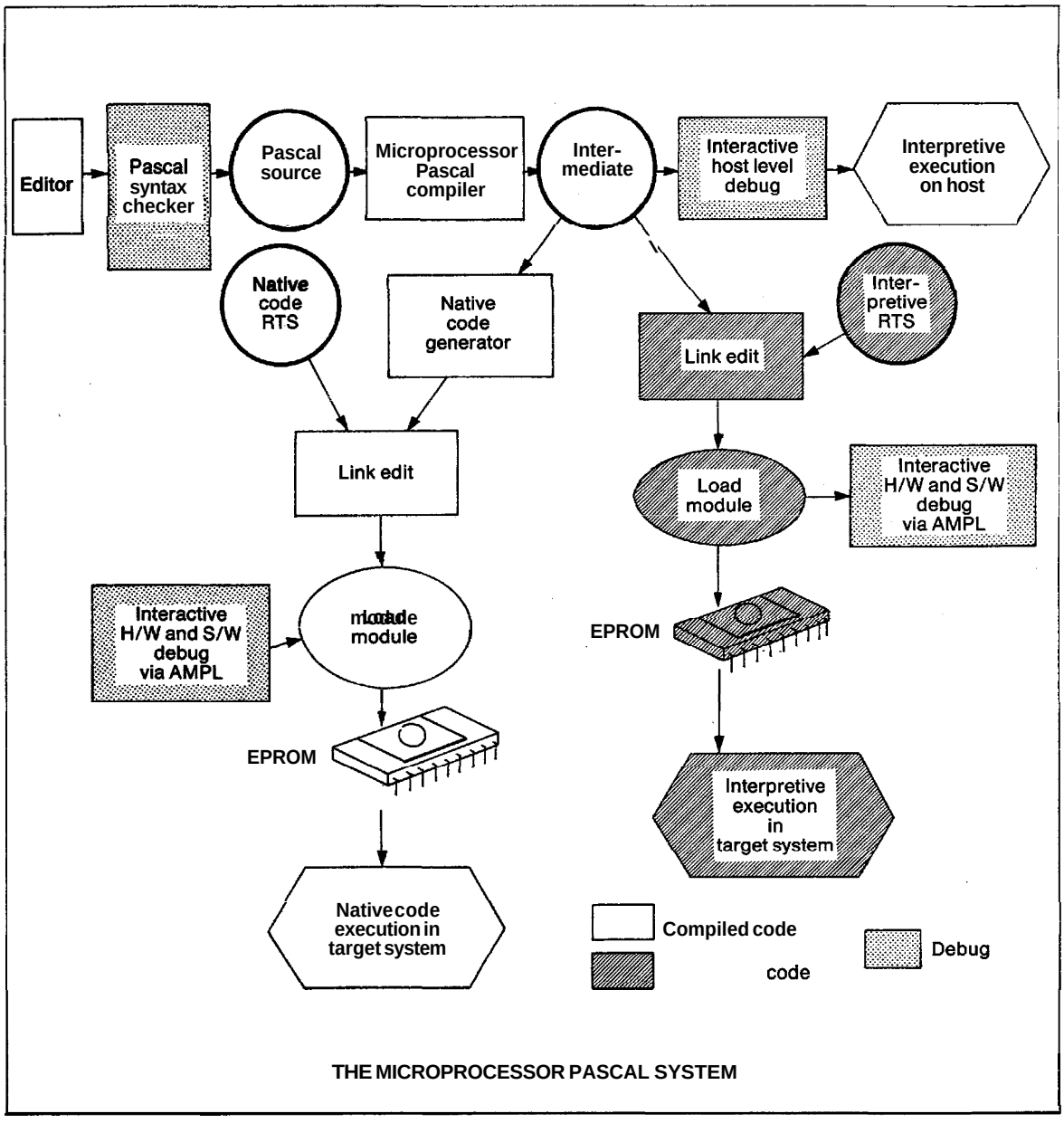
5.2.7.1 Code Efficiency

Use of a high level language inevitably produces code that is larger than a custom, hand crafted assembly language solution. However, the code produced by the Microprocessor Pascal code generator is efficient (a great deal of optimisation is performed automatically). Studies have shown that the code is, typically, slightly less than 1.5 times the size that would be expected from an experienced assembly language programmer. The compiler may well produce better (and certainly more reliable) code than an inexperienced assembly language programmer. Design tradeoffs are such that in most cases the extra memory cost, for all the systems that will be produced, works out less than the extra man months of software development time that would be needed in assembly language. When the further considerations of reliability, maintainability and development time are added, it is not difficult to justify the use of high level language.

The Microprocessor Pascal system includes a reverse assembler which turns the output of the code generator into assembly language source code. This code can be hand optimised in critical areas to squeeze the last ounce of performance from the system. Where code size is critical, Microprocessor Pascal programs can be executed interpretively instead of in native machine code. Interpretive execution is slower, but optimises use of memory.

5.2.7.2 Programming Support Environment

Microprocessor Pascal provides not only a language, but a complete design system for the development of microprocessor software. It provides a range of interlinked software tools, including a syntax checking text editor and extensive testing facilities within both host and target microcomputer systems. These tools make up a Programming Support Environment which guides software development from initial design through to final implementation and testing.



THE MICROPROCESSOR PASCAL SYSTEM

Figure 5-11 The Microprocessor Pascal System

The Microprocessor Pascal system is available on a wide range of single and multi-user, floppy and hard disc-based development computers, according to the needs of each user.

5.2.7.3 Microprocessor Pascal and Component Software

Pre-written Component Software functions (sequential or concurrent) can be accessed from within a user's Microprocessor Pascal program simply by declaring them EXTERNAL within the user's application program.

The Component Software packages themselves have been written in Microprocessor Pascal, for reliability, ease of understanding, and ease of customisation. A few have been recoded in assembly language to optimise performance in critical areas.

5.2.8 Other Languages

Although Component Software packages will generally be written in Microprocessor Pascal, the Software Function Bus (and hence the Component Software environment) is language independent. The low level "housekeeping" functions provided by Rx do not depend on any particular language. Application programs, and Component Software packages, written in assembly language interface directly with Rx. Microprocessor Pascal programs interface with Rx through an intermediate set of run time support functions. With the addition of suitable run time support, the Software Function Bus is capable of supporting any application language. Run time support functions and development tools for other languages will be added as the need becomes apparent.

Candidates for such addition may be not only the standard programming languages, but also special purpose languages and operator interfaces designed for specific application needs, such as process control. A range of programming languages is possible, permitting software development both "off line" in a separate development system and "on line" in the application microcomputer system itself.

5.2.9 Hardware

The Software Function Bus permits flexible selection of hardware implementations. Rx will adapt a standard software interface to a variety of hardware configurations, built from board modules or LSI components. TI's adoption of a standard instruction set for its 16-bit microprocessors (and minicomputers) has made this much easier.

For several years now, a range of compatible 16-bit microprocessors has been available from TI (the 9900 family). These processors have been designed to meet a wide range of price/performance goals. The recently announced 99000 family shares the same instruction set, with a number of advanced architectural features (such as storage of frequently used software functions in on-chip macrostore). The Software Function Bus provides a "cushion" against hardware changes, and protects software investment against potentially disastrous architectural changes.

The architecture of the 9900/99000 family is perfectly suited to the Component Software environment. The fast "context switch'' efficiently implements both concurrency, and the program modularity required' by all modern high level languages. Memory-to-memory architecture provides great flexibility in implementing independent, cooperating software functions.

At the board level, many special purpose Component Software packages correspond exactly to prepackaged microcomputer board modules. For example, the File Manager package corresponds with the TM990/303 Floppy Disc controller board. Matching software and hardware modules are designed to form complete Electronic Function Packages (EFPs) that can be incorporated directly in a system.

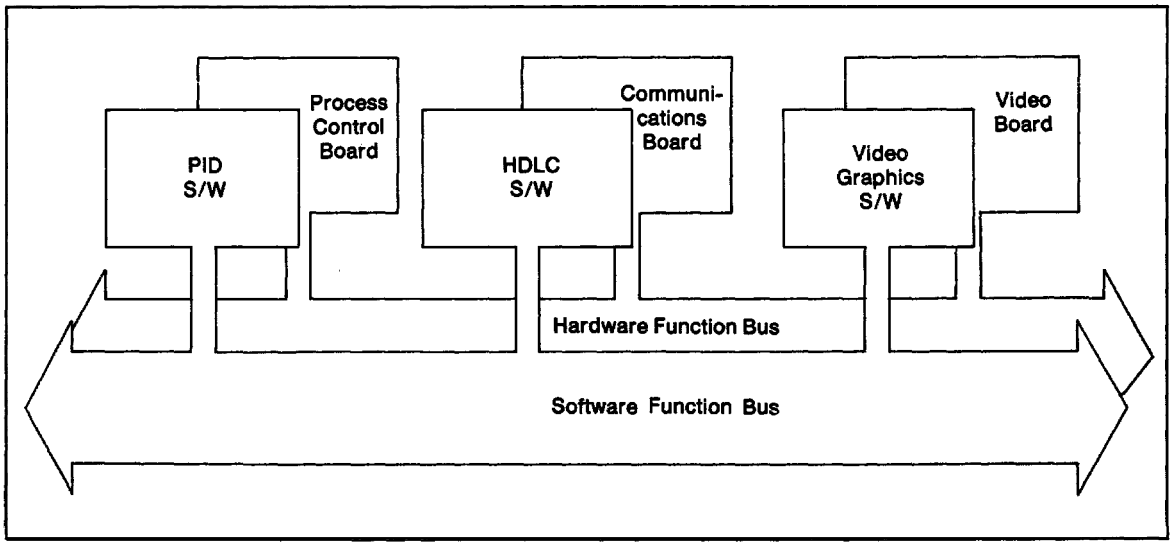


Figure 5-12 Software/Hardware Correspondence

5.2.10 Component Software Products

The first Component Software packages supplied by TI provide "system management" functions such as file storage and data communication between different systems. Later products will be designed for more specific application areas - process control and video graphics output, for example.

The Realtime Executive is available separately for assembly language users (it is supplied as a standard part of the Microprocessor Pascal package). The Microprocessor Pascal run time support functions will also be available as separate Component Software packages (Data Pack, Maths Pack, and Device Independent File 1/0 Pack), These functions can be called from assembly language programs to provide features such as floating point arithmetic, device independent files and structured data types.

Component Software packages will be available from other vendors as well as TI. The framework of Component Software is available to any manufacturer or software house that wishes to write and sell Component Software packages.

Contact Texas Instruments for a list of the Component Software packages currently available.

5.2.11 Silicon Functions

Taking a wider perspective, Component Software can be regarded as a development ground for functions which will eventually find their way into VLSI silicon, as dedicated hardware Microfunctions, VLSI integration will reduce the cost and increase the performance of Electronic Function Packages, so that future systems will be built from distributed networks of silicon Microfunctions, interconnected via a standard Function Bus.

This functional architecture is far more flexible than conventional microcomputer architectures, based on the mainframe model. Within a functional system, individual function packages can be incorporated that have a specialised architecture designed for particular needs,

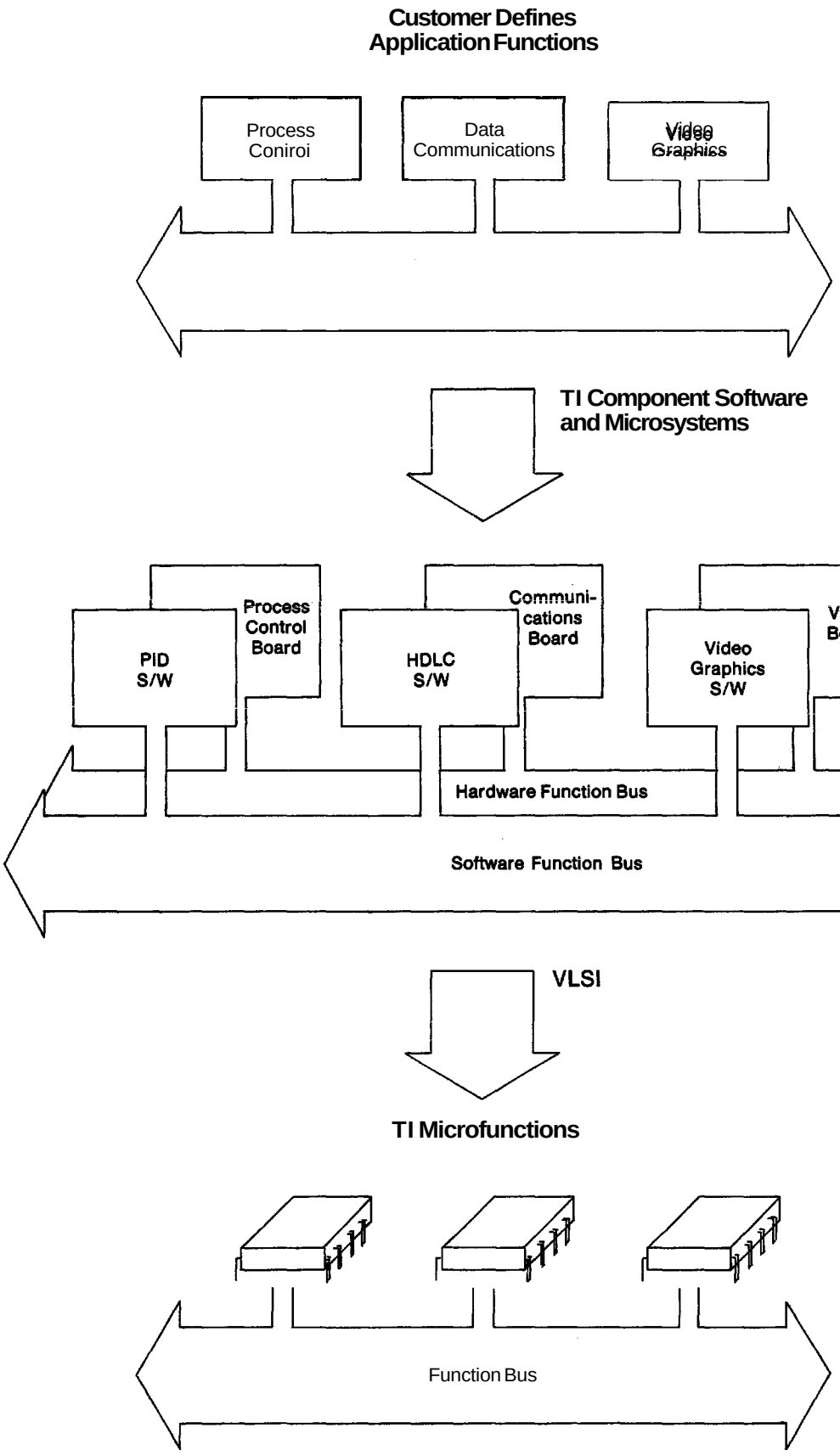


Figure 5–13 The Functional Approach

Function-to-Function Architecture defines a standard set of interconnection mechanisms for functions, hardware or software. This will permit replacement of software functions by their hardware equivalents, and vice versa. Software provides flexibility and fast development, hardware gives performance and cheapness (when it can be produced in quantity). In future, it will be possible to choose whether software or hardware (and what type of software or hardware) is appropriate at each point in a system, and to use the technology most exactly suited to the needs,

Component Software permits the development and tailoring of new functions in a flexible medium, quickly and cheaply, Such a development ground is needed if the potential of VLSI is to be exploited effectively.

New functions will be initially provided as Component Software libraries, permitting many different configurations from a standard "kit of parts". TI will eventually "can" particular configurations of these functions in silicon.

5.3 Bibliography

Texas Instruments Publications:

Component Software Handbook	(MP918)
Device Independent File 1/0 User's Manual	(MP386)
Microprocessor Pascal System User's Manual	(MP351)
Microprocessor Pascal Executive User's Manual	(MP385)
Realtime Executive User's Manual	(MP373)

CHAPTER 6

MICROPROCESSOR PASCAL

6.1 INTRODUCTION

Pascal was originated in the early 1970's by Professor Niklaus Wirth and Kathleen Jensen of ETH University, Zurich, Switzerland (see reference [1] in the Bibliography). 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 particuiariy 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 4.5, but it also has extremely flexible data structures which are very necessary for manipulating complex applications. In fact, the Pascal language is very close to the design language described in Section 4.4 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,

One of the greatest advances in Pascal is the data structuring facilities that are an integral part of the language. The concept of the data type has been greatly expanded to allow not only the usual types (eg INTEGER, REAL, CHAR, ARRAY, etc) but also more complex structures based on these types (eg SET and RECORD). Further, the user is able to define his own data types that totally satisfy his own requirements.

To ensure that these data structuring facilities are properly managed and controlled, the language encompasses a feature that is known as strong type-checking. This means that when a variable is defined it is declared to be of a particular type, As variables are used, the compiler checks that they are used correctly and consistently. This strong type-checking increases program reliability,

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 standardization on a particular low-level machine architecture,

6.2 TEXAS INSTRUMENTS' IMPLEMENTATIONS

Several years ago, Texas Instruments recognised that a single programming language was required as as a corporate standard for all software, whether for mainframes or minicomputers. The selected language would be used to cover the following areas:

- o Systems programming
- o Applications programming
- o Industrial real-time control

This led to in an in depth study of the 20 most prominent languages (including ALGOL68, BCPL, BLISS, C, CLU, Pascal, PL/I, etc) to determine which, if any, could satisfy these requirements.

After exhaustive tests, it was decided that a programming language based on Pascal (which was designed primarily as a teaching language) but having adequate extensions to operate in a teal-time environment most suited the requirements. This resulted in Texas Instruments Pascal (TIP) which was designed to compile and execute on large machines (the Texas Instruments DS 990/10 and the IBM 370). 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.

After the release of the TIP compiler, it soon became apparent that the language would be extremely useful for programming microprocessors for industrial and control applications. For this reason, a variant called Microprocessor Pascal was developed. This has fewer extensions than TIP and is therefore more easily implemented on small computers. In fact the compiler runs on a floppy disc based system that uses the TMS9900 microprocessor as its central processing unit.

The two languages are fundamentally the same, but provide slightly different features to support their different areas of application.

Because microcomputer systems usually have to operate in real-time, concurrency is an integral part of the Microprocessor Pascal language. 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 do the rest. This can greatly simplify a complex problem. Synchronization of processes is accomplished by signalling devices called semaphores. Higher level communication between processes can be handled by interprocess files. Further information on concurrency is presented in section 6.8 and also Section 5.2.1.

During the design of Microprocessor Pascal, it was recognised that a language on its own (no matter how good) is not enough. What is also required is what has become known as a 'programming support environment' - that is a collection of 'tools' that aid and simplify the design of complex application systems. The Microprocessor Pascal System (see section 6.4) was designed for this purpose.

6.3 MICROPROCESSOR PASCAL LANGUAGE OVERVIEW

6.3.1 Features

Microprocessor 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 more easily understood, and better, more reliable software. Microprocessor Pascal's structured statements include IF, CASE, FOR, WHILE and REPEAT: they are described in section 6.7.

Microprocessor Pascal provides extensive data structuring: RECORD and ARRAY data structures can be combined and nested to any level. The POINTER data type permits powerful structures such as linked lists and trees. It also permits dynamic storage allocation. These data structures are described in section 6.6.

In addition to the standard data types, Microprocessor Pascal allows the user to define his own data types, which can have values represented by meaningful names. The type concept was introduced in Section 4.6. Its implementation in Microprocessor Pascal is described in section 6.6.

Data typing allows data to he grouped according to use. It

can clarify the design of a program so that, for example, it is easier to change at a fate stage in devefopment. Compiler checks on type compatibility can reduce the risk of undetected errors in program code.

Microprocessor 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 can create a program that is largely self-documenting.

Microprocessor 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 section 6.3.6.

The concurrent structure of Microprocessor 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 overhead.) The mechanisms involved in concurrency are described later in more detail (see section 6.8) and also in Section 5.2.1. Additional information can be obtained from the Microprocessor Pascal System User's Manual.

6.3.2 Stack and Heap

Like the majority of modern high-level languages, Microprocessor Pascal has a stack architecture. The stack is an area of read/write memory from which sections (called stack frames) are allocated to a routine (procedure or function) at the time it is invoked. When the routine 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 8.4.4) forms a natural basis for implementing stack frames.

Data is completely separated from program code, so that Microprocessor Pascal adapts naturally to the ROM/RAM environment of a microcomputer. This means that Microprocessor 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.

When the target system is initially started, all available RAM is in a common pool called the heap. As programs and processes are activated they are allocated their stack space from the heap. This is returned to the heap (for re-use) when the program or process terminate.

In addition to the storage provided in the stack, Microprocessor Pascal is able to dynamically allocate areas of memory (known as heap packets), under program control, from the heap. This is accomplished using the standard procedures NEW and DISPOSE, and the pointer variable described in section 6.6.13. (NEW and DISPOSE are described in the Microprocessor Pascal System User's Manual.)

6.3.3 Systems and Programs

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

6.3.4 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 (and 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 concurrently with all the other processes in the system, including the one that calls, or STARTs it.

6.3.5 Declarations and Statements

For the programmer there are two principal parts to any Microprocessor Pascal system, program, process, procedure, or function: 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 specify the data that the program is to work with; the statements specify exactly what is to be done with this data.

The statement body is a collection of Microprocessor Pascal statements that is enclosed by a BEGIN...END compound statement.

```
PROGRAM factorial;     (* PROGRAM DECLARATION          *)

VAR i,j,n : INTEGER;   (* VARIABLE DECLARATIONS        *)
                       (* Declare variables named      *)
                       (* I, J, N of type integer      *)

BEGIN (* factorial *)  (* PROGRAM 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
    i := i + 1;        (* Use I and J to compute       *)
    j := i * j         (* factorial N                  *)
  END;
  Writeln(j)           (* Output value of factorial N *)
END,  (* factorial *)
```

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.

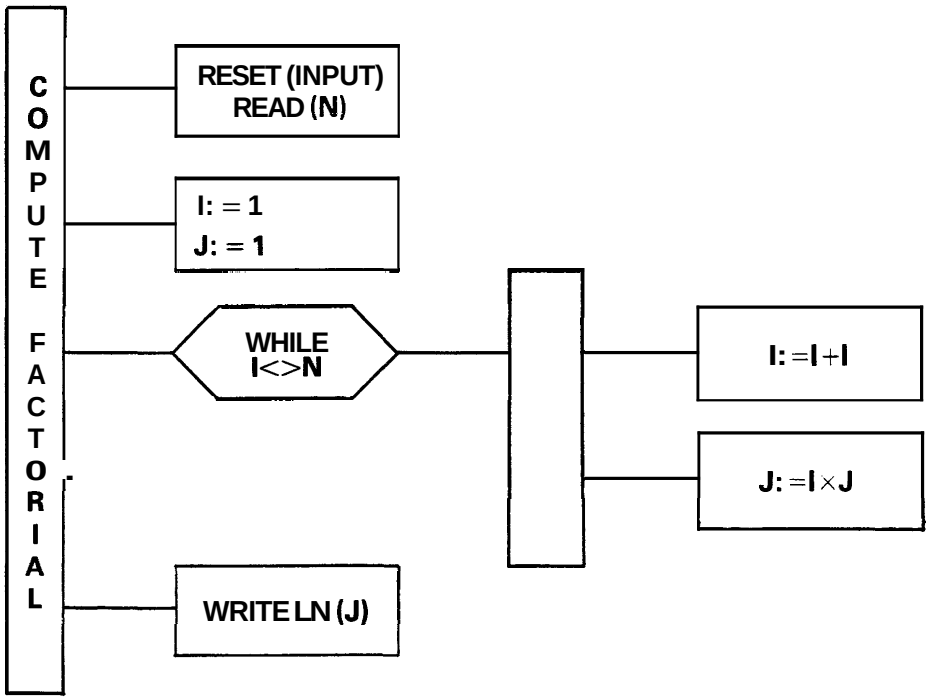


Figure 6-1 Program Structure Diagram

Microprocessor 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.

Microprocessor 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 statements within the BEGIN...END compound statement following the WHILE clause are indented to show that they are one level down in the program hierarchy. In fact, the indentation reflects the appearance of the structure diagram for the program (Figure 6-1). (See Section 4.5 for a description of structure diagrams,) Formatted in this way, the program is much more readable and the structure can be seen at a glance.

6.3.6 Block Structure

One of the key features of Microprocessor Pascal is its block structure, The basic ideas of block structuring are discussed in Section 4.9.

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 Microprocessor Pascal program consists of a hierarchy of blocks, nested one within another. A system block, which is a complete Microprocessor Pascal system, contains a number of program blocks, which in turn can contain process blocks, procedure and function blocks, etc. This hierarchy is displayed in Figure 6-2. (The lexical hierarchy is shown in Figure 6-3, and the corresponding concurrent structure in Figure 6-4.)

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, ie outside the block in which they are declared. For example, in the system of Figure 6-2, the declarations in PROGRAM A cannot be referenced in PROGRAM B or PROCESS 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.

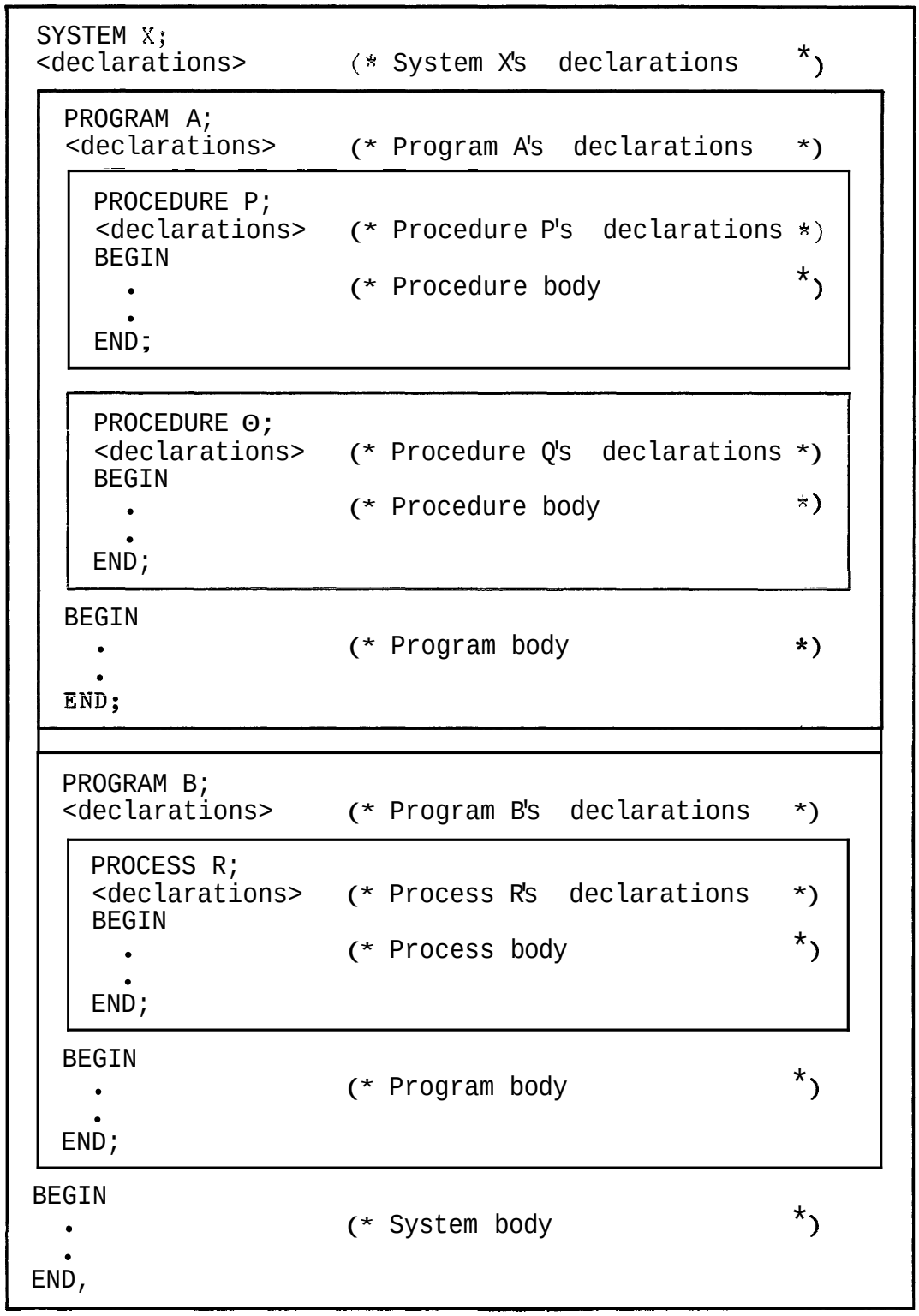


Figure 6-2 System Structure

Block structure and scope rules are 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.

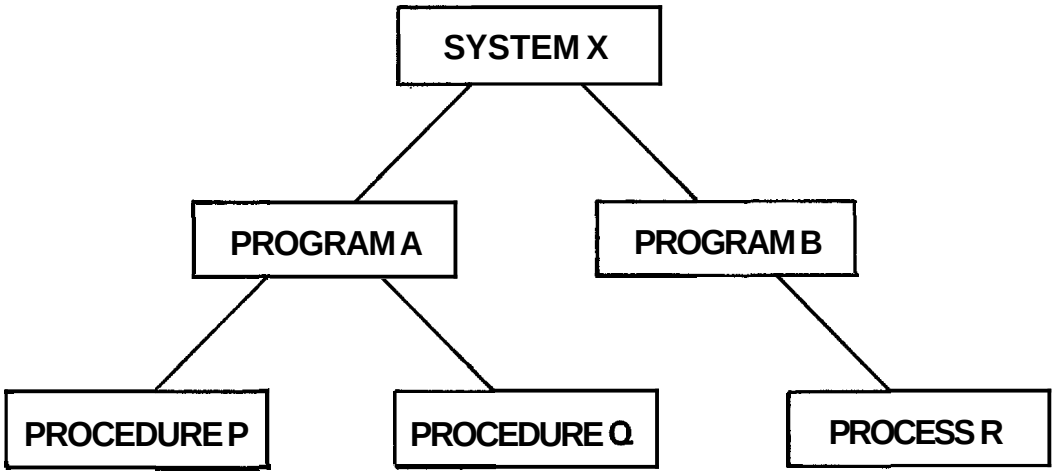


Figure 6–3 Lexical Hierarchy

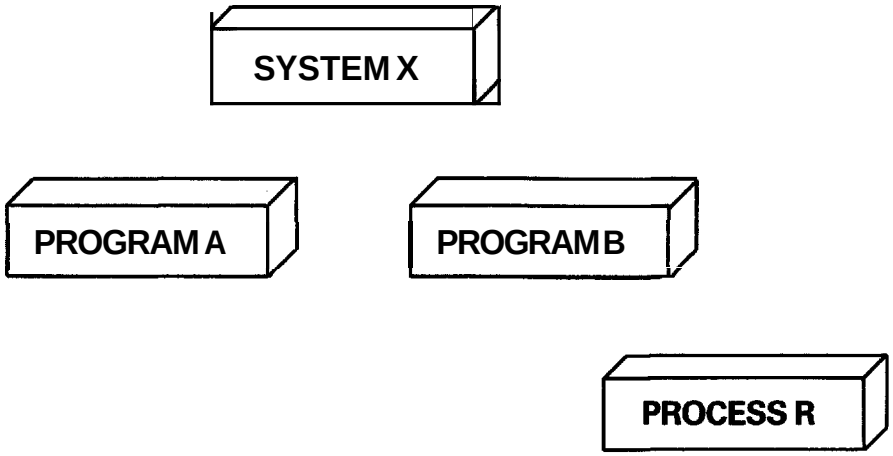


Figure 6–4 Concurrent Structure

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, biock structuring (combined with Microprocessor 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.

6.4 MICROPROCESSOR PASCAL SYSTEM - PROGRAMMING SUPPORT ENVIRONMENT

The Microprocessor Pascal System is a powerful integrated, software development tool set that provides a development environment for the design, coding, and debugging of Microprocessor Pascal applications for microcomputers,

This system was designed from the start to execute efficiently on the 'small' single-user floppy disc based FS 990/4 and TMAM 9000 minicomputers. The system is also supported on the much larger, hard disc multi-user DS 990/10 and /12 computers.

Currently there are four major components in the Microprocessor Pascal System to assist in software development:

- An 'intelligent', interactive, screen-based editor for source preparation, with syntax-checking capability,
- A compiler that produces interpretive code.
- An interactive host debug interpreter.
- A code generator that transforms interpretive code into TMS9900 native object code.

Two executives support the execution of the user's system on a target microcomputer. One supports the interpretive code

produced by the compiler (MPIX - Microprocessor Pascal Interpretive Executive); the other supports the object code produced by the code generator (MPX - Microprocessor Pascal Executive). These executives are functionally identical, so that the user has a choice of running either interpreted or compiled code on his target system.

6.4.1 Microprocessor Pascal Editor

The Microprocessor Pascal System features an interactive, screen-based editor that allows the user to create and modify Microprocessor Pascal source files. Some 'intelligence' has been built into this editor to allow it to recognise certain Microprocessor Pascal language keywords and to automatically indent the source text being entered into easily distinguishable blocks of code that show the program structure.

When editing, a page of text is displayed on a visual 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 he 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 explicit edit commands, including find string, replace string, etc.

When creating a source file, the editor *assists line-by-tine* 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.

Most editors (even screen-based ones) use a line numbering mechanism to access a particular source line within the source file. The first line in the file is "line 1" (or 10 or 100), the second line is "line 2" (or 20 or 200) and so on. Such mechanisms can be cumbersome to use, especially when inserting source lines and also when going back to perform modifications on an already partially modified source file. To overcome these problems, the Microprocessor Pascal system editor is completely cursor driven and does not use a line numbering mechanism.

A number of edit commands (MOVE, COPY, DELETE and PUT) operate on blocks of code. The required block is indicated by:

- Positioning the cursor to the first line in the block and press the function key F5.

- Positioning the cursor to the last line of the block and press the function key F6.

- If a destination position is required (MOVE and COPY) then reposition the cursor to the required source line. (The block of code will be inserted into the program immediately after this line.)

(The function keys are the grey keys, numbered F1 to F8, that are located above the normal 'QWERTY' keyboard on the 911 VDU.)

The HELP command (press the CMD key and type the word HELP followed by the return key) displays a full list of the available edit commands, along with the meaning of each function key,

After the program has been entered, the user can perform a Microprocessor 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 English language 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 has been done, the CHECK command can be reentered and checking will resume from the earliest point at which the text was changed. (The syntax 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 the need to exit the editor, execute the compiler, print the listing, and re-edit the source file for each mistake, The entire process becomes a single interactive step.

The CHECK facility is entirely optional. The Microprocessor Pascal System Editor can be used for text files other than Microprocessor Pascal source.

The available edit commands are:

ABORT	Exit the editor
INPUT	Change the edit file
QUIT	Save the edited file and ABORT
SAVE	Save the edited file and INPUT

BOTTOM	Position the cursor to the end-of-file
TOP	Position the cursor to the top-of-file
+/- int	Position the cursor up/down "int" lines
CHECK	Check syntax of edit file
HELP	Display edit commands available
INSERT	Insert the specified file
SHOW	Show the specified file
COPY	Copy the specified block to current cursor posn
DELETE	Delete the specified block
MOVE	Move the specified block to current cursor posn
PUT	Write the specified block to the specified file
FIND(tok,n)	Find the "n"th occurrence of "tok"
REPLACE(tok1,tok2,n)	Replace "tok1" by "tok2" "n" times
TAB(inc)	Set the tab increment to "inc"

The function key operations are:

F1	Roll down the file
F2	Roll up the file
F4	Duplicate this line
F5	Start block delimiter (<------- in cols 72 to 80)
F6	End block delimiter (-------> in cols 72 to 80)
F7	Compose/Edit
F8	Split line from the current cursor position
CMD	Go into command mode (+------+ in cols 72 to 80)

6.4.2 Microprocessor Pascal Compiler and Code Generator

The Microprocessor Pascal Compiler generates interpretive code from a Microprocessor Pascal source file, This code can be executed directly using the interpretive debugger or the Microprocessor Pascal Interpretive Executive (MPIX). Passing this interpretive code through the Microprocessor Pascal Code Generator produces native 9900 object code that will run under the Microprocessor Pascal Executive (MPX).

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 source file will be functionally identical, apart from considerations of speed and code size.

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

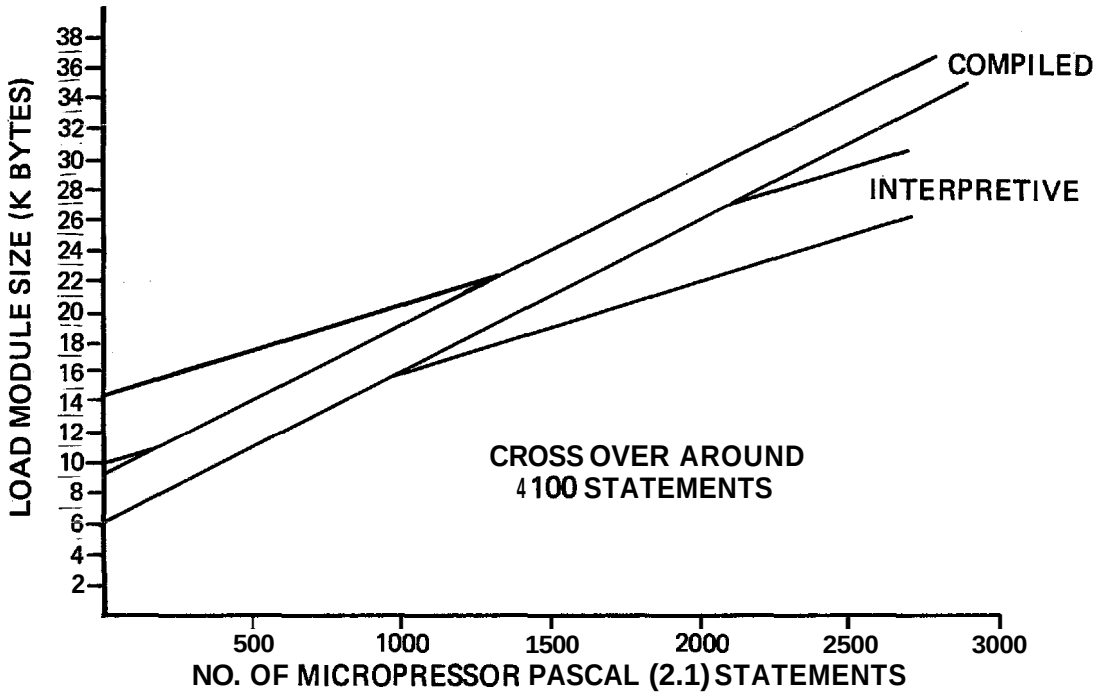


Figure 6-5 Interpretive vs Compiled Characteristics

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 FS 990/4 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, processing approximately 100 lines of source code per minute). On the DS 990/10, where there are no memory restrictions, it executes as native code to maximize the speed.

Various compiler options are available. These options include:

LIST	Produce source listing
MAP	Produce variable map
STATMAP	Produce statement displacement map
DEBUG	Include debug information in code
ASSERTS	Generate code for ASSERTS statement checks
CKINDEX	Generate code for array index checks
CKPTR	Generate code for NIL pointer checks
CKSET	Generate code for set expression checks
CKSUB	Generate code for subrange assignment bounds checks

The host debugger can be used to check the functionality of the application program. When satisfied that the program works correctly it can be transferred to the actual target hardware where any hardware dependent parts of the program

can be verified using the AMPL in-circuit emulator. The Microprocessor Pascal System is supplied with two sets of AMPL procedures (one for MPIX, the other for MPX) that present the same user interface as the host debugger. Any necessary 'fine tuning' or customisation can also be performed at this stage,

6.4.3 Microprocessor Pascal Host Debugger

The Microprocessor Pascal Host Debugger is an interactive interpreter that allows the user to control and monitor execution of a Microprocessor Pascal target application system, 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 Microprocessor 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 Microprocessor 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 connect interprocess files (discussed in section 6.8.5.4) using the Connect Input Pile 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 in the case of output.

Interrupts can he simulated using the SIMulate Interrupts command,

The system has three ways of dealing with CRU 1/0 (for a description of the CRU see section 8.9). 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 iikeiy to have a different CRU configuration from the development system.

The Microprocessor 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.

6.5 MICROPROCESSOR PASCAL LANGUAGE

Before describing the major features of the Microprocessor Pascal language (data types, control structures, concurrency, etc) it is first necessary to explain some of the basics of the language.

6.5.1 Basic Language Elements

A Microprocessor Pascal application program 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.

6.5.2 Character Set

The Microprocessor Pascal character set is:

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

6.5.3 Keywords

Keywords are reserved words with a fixed meaning; they may not be used as identifiers. Although they are written as a sequence of letters, they are interpreted as a single symbol. A full list of these keywords is given below.

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

In program text, it is convient to write keywords in upper case to distinguish then from user-defined identifiers in lower case, Microprocessor Pascal does not require this distinction, but it is helpful in making programs more readable.

6.5.4 Identifiers

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. he-convention followed in this chapter is that all identifiers are written in lower case when they appear in examples, but they will be in upper case whenever they appear in the text,

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 called.

6.5.5 Language Element Separators

At least one separator must occur between two constants, identifiers, keywords or special symbols. No separators can occur within these elements (except spaces within strings). Separators include spaces, end of lines, comments or remarks. For example, in the statement:

```
WHILE X<10
```

a space separates WHILE and X. This is not equivalent to:

```
WHILEX<10
```

as WHILEX could be a legal identifier. However, a space is not necessary between X and '<' because '<' is not permitted within an identifier and thus serves to delimit it.

6.5.6 Comments

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.

6.5.7 Constants

Part of the declaration section for a program, process, etc, consists of the(constant declaration part>. This allows an identifier to be used as a synonym for a constant and can make a program more readable. These constants are defined by:

CONST <constant declaration list>

where (constant declaration list> is one or more of the following:

```
<identifier> = <constant> ;
```

where <constant> may be a signed real constant, string constant, character constant, integer constant expression or a previouly defined constant identifier, An integer constant expression may consist of: integer constants and/or constant identifiers along with any of the integer arithmetic operators. For example:

```
CONST max      = 500;
      asterisk = '*';
      one_half = 0.5;
      half_max = max DIV 2;
```

"Application parameters[1]' that are liable to change between systems (eg the number of capstan lathes in an engineering shop) are best handled by defining them as constants. Doing this would mean changing only a few statements right at the begining of the application program instead of having to search the whole program for instances where the parameter values are used (and possibly even missing some of them).

6.5.8 Variables

Variables are used to reference areas of storage within a module, A variable declaration associates an identifier to 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) ;
```

<identifier list> is a list of identifiers separated by commas. <type definition) (described in section 6.6) can be a standard type (INTEGER, REAL, etc), the name of a type defined in a type declaration statement, or a new type definition which can take any of the forms allowed in a type declaration, In the last case, the new type will not have any name associated with it (the declaration of PROFIT below is an instance of this), For example:

```
VAR nyears            : INTEGER;
    amount,value,rate : REAL;
    ten_years         : vector;
    profit            : ARRAY [1..10] OF BOOLEAN;
```

(Type VECTOR is defined in section 6.6.1.)

A variable can either be a simple identifier which references the entire variable, or may be a qualified variable which is used to reference part of a structured variable - for example a record or an array.

6.5.9 Expressions

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

6.5.9.1 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)

6.5.9.2 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, which specifies 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

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.

6.5.9.3 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) ((parameter list>)

eg sqrt(max)

where <function identifer> is the name of the function to be called. <parameter list> is one or more <parameter>s, separated by commas, as specified by the function definition. <parameter> may be any variable, constant or expression so long as it matches the declared type.

6.5.10 Assignment Statement

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

<variable> := <expression>

eg 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.

6.5.11 Routine Declaration

A PROCEDURE declaration packages a self contained sequence of operations that performs some action, and also associates this action with a particular identifier. This action can then be performed from anywhere within the program (so long as it is in scope - see section 6.3.6) by simply invoking the appropriate procedure.

The general form for a PROCEDURE declaration is:

```
PROCEDURE <identifier> ( <parameter list> ) ;
<declarations>

BEGIN                                                .

END ;
```

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

VAR (identifier list> : <type definition) ;

(identifier list> is one or more identifiers separated by

commas. <type definition) is described in section 6.6. If no parameters are required then the "(" and ")" can be omitted. VAR is optional(see below),

<declarations> can be one or more:

LABEL declaration	refer to manual
CONST declaration	section 6.5.7
TYPE declaration	section 6.6
VARS declaration	section 6.5.8
COMMON declaration	refer to manual
ACCESS declaration	refer to manual
PROCEDURE declaration	
FUNCTION declaration	below

There are two methods of parameter passing. Call by value will cause a copy of the actual parameter's value to passed over to a new storage location in the procedure. This parameter can then be modified by the called procedure without affecting the value of the actual parameter variable in the caller's stack. Call by reference will cause the address of the caller's actual parameter variable to be passed over to the procedure. Modifying a call by reference parameter modifies the contents of the caller's actual parameter variable. (More detail on the parameter passing mechanisms is given in Section 4.10.1.)

If a parameter is to be passed by reference then the keyword VAR should be included before the appropriate <identifier list>:

```
PROCEDURE add_five_plus_inc ( VAR update : INTEGER;
                                  inc    : INTEGER);
CONST five = 5;

BEGIN
  ( Modify the caller's actual parameter by INC+5 }
  update := update + five + inc;
  { Modify local variable INC - does not affect
    the caller's actual parameter                 }
  inc := inc +3
  .
END;
```

<declarations> and the BEGIN ... END; can be replaced by the keyword EXTERNAL, which informs the compiler that that particular procedure is defined outside this program module.

A FUNCTION declaration is similiar to a PROCEDURE declaration. The only difference is that the first line is of the form:

```
FUNCTION <identifier> ( <parameter list> ) :
                        <type definition) ;
```

The function's result is returned by assigning the required value to the function identifier, ie:

```
FUNCTION return_6x ( value : INTEGER ) : INTEGER;

BEGIN
  return_6x := value * 6
END;
```

Microprocessor Pascal implements additional structures that can be used to package concurrent statement blocks (PROGRAMs and PROCESSes). These are defined in a similiar way to procedures and can have parameters in a simfliar way (but parameters must all he passed by value). However, programs and processes are STARTed rather than called and once started exist as separate concurrent "sites of execution" within the system.

A PROGRAM or PROCESS declaration is identical to a PROCEDURE declaration, except that the first line is:

PROGRAM <identifier> (<parameter list>) ;

or

PROCESS <identifier> (<parameter list>) ;

The <declarations> can include other PROCESS declarations. The (parameter list> cannot contain variable parameters (ie the keyword VAR is not allowed in(parameter list>).

See sections 6.3.3 to 6.3.6, 6.9 and Section 5.2.2 for the concurrent structures of Microprocessor Pascal.

6.6 DATA TYPES

A data type 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.

In Microprocessor Pascal, data types can he split into three distinct classes. These are:

Simple types	INTEGER, LONGINT, REAL, CHAR, BOOLEAN, SEMAPHORE, Subrange and Enumeration
Structured types	ARRAY, RECORD, SET, POINTER and FILE

User defined types Specified by the TYPE statement

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, if possible, are stored in one word. Packing may economize the storage requirements of a data structure, at the expense of efficiency of access of the components.

One example of a packed array is a string, which can be defined as:

PACKED ARRAY [<index type>] OF CHAR

In this structure, characters are stored one per byte instead of the usual one per word. <index type> is described in section 6.6.9.

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

6.6.1 User Defined Types

A type declaration introduces an identifier as the name of a new data type. The identifier 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 performed by the variable (VAR) declaration, as described above (section 6.5.8).

6.6.2 Integer and Longint Type

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 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 section 6.13.6.

6.6.3 Boolean Type

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 is less than TRUE. Thus the relational operators can be used with BOOLEAN operands to provide additional operations. For example:

=	Equivalence
<>	Exclusive OR

6.6.4 Char Type

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 #:

'A'	ASCII character A
'#0D'	ASCII character 'carriage return'

6.6.5 Enumeration Type

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:

- o There is a distinct set of values which a variable of that type can take.
- o The 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), These can he regarded as primitive values, just like '7' or '125': it is not necessary to translate them into hit patterns, or know how they are represented within the computer, any more than it is necessary for most purposes 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)	Successor of X
PRED(x)	Predecessor of X
ORD(x)	Integer ordinal value of X within the set of values(not defined for INTEGER or LONGINT)

eg SUCC(wed) = thu, PRED(wed) = tue, ORD(wed) = 3

Scalar types are useful for counting purposes. For example,

to index into an array or control the number of iterations of a FOR loop (see section 6.7.5):

```
FOR today := mon TO fri DO
  total_takings := total_takings + takings[today];
```

The variable TODAY is declared to be of type DAYS; the array TAKINGS is declared to be indexed by type DAYS*

6.6.6 Subrange Type

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 insert runtime checks to ensure variables do not exceed their specified subrange. This can be a great help in debugging. Subrange types can also be used in declaring array bounds, 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 makes it easy to change the size of an array at a late stage in development, simply by altering one or two TYPE statements. (Arrays are discussed in section 6.6.9.)

6.6.7 Real Type

The type REAL can be used to represent real values with 6-7 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.

6.6.8 Semaphore Type

The type "semaphore" is used for process synchronisation and communication (more about this later, see section 6.8). Operations on variables of type semaphore are performed by functions and procedures which must be declared EXTERNAL to the program. Arithmetic operations are not valid for semaphore variables.

6.6.9 Array Type

An array type consists of an ordered 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)

<component type> can be any type except FILE. This means that it is possible to have arrays of arrays, of records or of any other structured type. <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 type. For example:

```
VAR a,b : ARRAY [1..20, 25..50, 1..2];
.
.
a := b;
```

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

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

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

eg 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, eg A [5] [6]

which is exactly equivalent to A [5, 6]

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

6.6.10 Record Type

A record type consists of a group 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>

<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;
```

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. Here,

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.

Pascal also allows record variants, which means that part of a record can be interpreted in more than one way. This would allow, 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 (see Section 4.7.4). Record variants are described in detail in the Microprocessor Pascal System User's Manual.

6.6.11 Set Type

Pascal allows a set type, in which 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.

6.6.12 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 as 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 MPX or MPIX executive performs the transfer in internal memory without involving any external storage devices.

The form of a file type definition is:

RANDOM FILE OF <component type)

or

FILE OF <component type>

or

TEXT

The component type of a file can be any type except pointer or file. The number of components (ie 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 CHAR;
             id_num : INTEGER
           END;

VAR f        : FILE OF INTEGER;
    employee : RANDOM FILE OF rec;
    temp     : TEXT;
```

The following standard procedures and functions are available for file manipulation:

CLOSE	Close the file
EOF	Check for EOF (end-of-file)
EOLN	Check for EOL (end-of-line)
READ	Read components of the file
READLN	Read components from a text file until EOL
RESET	Open file for input
REWRITE	Open file for output
WRITE	Write components to the file
WRITELN	Write components and EOL to a text file

See the Microprocessor Pascal System User's Manual for further details.

6.6.13 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)

read as ''pointer to a(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 PL/I is allowed to roam freelv throughout memory, and can

cause chaos if the programmer makes a small error in manipulating it. (In Microprocessor Pascal it is always possible to do such things using the type transfer function, for instance, 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.

```
TYPE ptr  = @list;
     list = RECORD
              value : REAL;
              loc   : 0..FF
            END;
```

PTR is declared to ''point to the type LIST" and variables of type LIST can only be used to point to records of type LIST.

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 (see section 6.3.2). The constant NIL can be assigned to any pointer variable, which.means it points to nothing at all. A compiler option (CKPTR) is available to check if a reference is made to a NIL pointer.

```
( Declare NEXT and TEMP as pointers to records of
  type LIST }
VAR next,temp : ptr;
  .
{ Set TEMP to point to the NIL record of type LIST }
temp@ := NIL;
{ Allocate new record of type LIST from the heap, and
  set NEXT to point to it )
new(next);
( Set VALUE field of record pointed to by NEXT to 2.5 }
next@.value := 2.5;
```

The operators that can be applied to pointer variables with compatible types are:

:=	Assignment
=	Equal (TRUE if the, operands point to the same address)
<>	Not equal

Pointers allow storage to be dynamically allocated from a

storage area called the heap, using the standard procedure NEW, Pointers can also be used to construct "advanced" data structures (see reference [2] in the Bibliography) such as linked lists and binary trees. A linked list is easily created by defining a record type which contains one field that is a pointer to the next record in the list. Similarly, a binary tree of records can be constructed by defining a 'right link' and 'left link' pointer within the record,

6.6.14 Type Compatibility and Transfer

Microprocessor 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 (eg in specifying an array index type),

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

- o They are identical types,
- o Both are subranges of the same enumeration type,
- o Both are string types with the same length,
- o Both are pointer types which point to identical types,
- o Both are set types with compatible base types.
- o 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 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 order). 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 is:

<variable> :: <type identifier)

eg i := 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 Microprocessor Pascal System User's Manual).

6.7 CONTROL STRUCTURES

This section is primarily concerned with the Microprocessor Pascal statements that implement the control structures which were introduced in Chapter 4 of this book (Section 4.5).

6.7.1 Procedure Statement

The procedure declaration (see section 6.5.11) defines a subprogram which can be called up simply by writing its name in a procedure statement. A procedure statement corresponds to one of the terminal boxes on the right hand side of a structure 'diagram (see Figure 4-14), which is expanded as a separate algorithm in the procedure declaration (Figure 4-15).

The general form of a procedure statement is:

(procedure name> ((parameter list>)

eg calculate_mean(a, 5, 4*x)

Parameters must match in number and type with those declared with the procedure, If the procedure has no parameters then only(procedure name> is required.

6.7.2 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 in all higher level constructs.

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

(statement list> is a list of Microprocessor 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;
  xl := x2;
  x2 := exchange
END
```

The semicolon is used to separate Microprocessor 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 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;
ESLE 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 is no 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 Section 4.5.1.

6.7.3 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 Microprocessor Pascal statement, including compound statements and further 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 always to use explicit BEGIN...ENDs to make the logical structure perfectly clear.

6.7.4 CASE Statement

The CASE statement is an extension of the IF statement to

allow more than two choices. 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
```

<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 4.5.2.

6.7.5 FOR Statement

The FOR statement provides for the repeated execution of a given statement for a progression of values which 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 may 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 this 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;
```

6.7.6 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 <expression> is false initially, <statement> is not exe'cuted 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.

Example:

```
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 initially. 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 (see Section **4.5.5)** is intended to be a WHILE (or a FOR), and is best kept as

such. A REPEAT....UNTIL construct can then be written explicitly as:

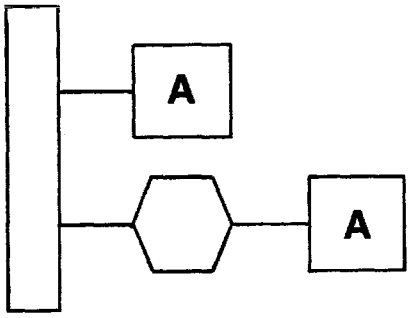
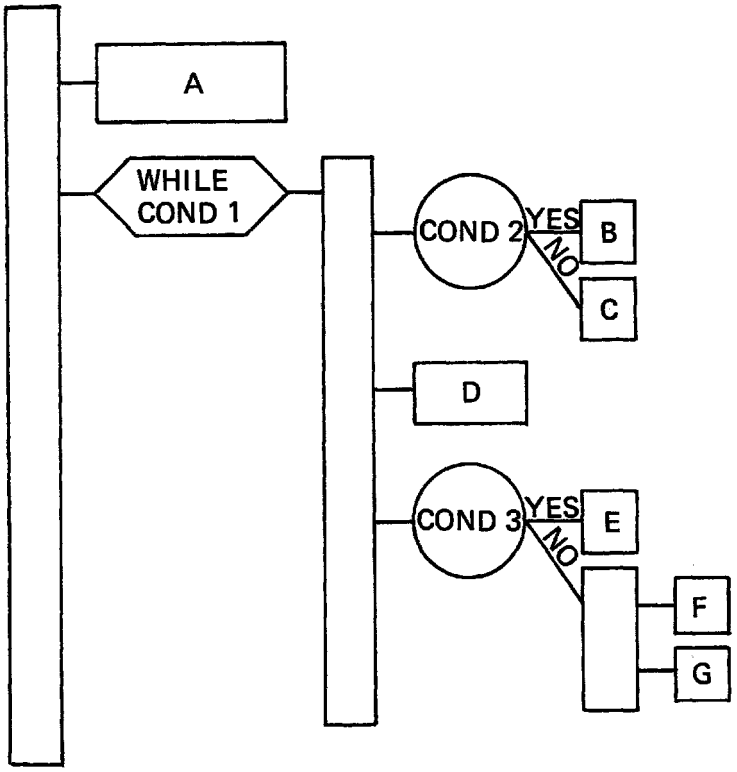


Figure 6-6 Repeat Until Construct

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

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




```
BEGIN
  A;

  WHILE COND 1 DO
    BEGIN
      IF COND 2
        THEN B
        ELSE C;

      D;


      IF COND 3
        THEN E
        ELSE
            BEGIN
              F;
              G
            END
    END
END
```

Figure 6-7 A Sample Program

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

When the control structures are used in conjunction with the data typing features it is possible to produce a program that is clear, uncomplicated (but never the less complex) and largely self-documenting. Although the following program lines are a little whimiscal, they do illustrate the point.

```
CONST number_of_people         = 50;
      expected_number_of-legs = number_of_people DIV 2;

VAR animal :(lion, tiger, cat, dog, rhino);

BEGIN
  CASE animal OF
    dog: pat_it_on_the_head;
    cat: stroke_its_back;
  OTHERWISE
    IF life_is_not worth_living THEN hang_around
    ELSE run_for_it
  END
END;
```

6.7.7 ESCAPE Statement

The ESCAPE statement is a 'structured jump'. It is used for premature termination of a structured statement, procedure, program or process, 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 or process name.

An escape label, followed by a colon, may prefix any structured statement, (The structured statements are: compound statement, IF, CASE, FOR, WHILE and REPEAT statements.) 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 (see below), 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;
```

6.7.8 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' which is difficult to follow and prone to error. In some languages (eg FORTRAN), GOTOs are necessary because the constructs necessary to implement control structures directly are not available. This is not the case in Microprocessor Pascal, which has a complete set of sequence, selection and iteration constructs that are sufficient to implement any program algorithm. In almost every case where a GOTO might 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.

Although the GOTO statement has been included in Microprocessor Pascal it has deliberately not been made easy to use. All labels used must be declared in advance.

6.8 CONCURRENCY

Concurrency is an integral part of the Microprocessor Pascal language and an understanding of this concept is built into the Microprocessor Pascal System tools (in particular, the compiler and the host and target debuggers), In a target environment, concurrent execution of a multiple process system is supported by the MPX and MPIX executives.

Concurrency is the simultaneous execution of a number of different software programs, or processes. Further information on concurrency is given in Section 5.2.1.

This section describes some of the functions performed by the executive, and also the mechanisms provided for synchronization and communication between processes,

6.8.1 Processes

The term "process" as used in this section applies to all concurrent units in Microprocessor Pascal (implemented using the keywords SYSTEM, PROGRAM or PROCESS - see section 6.3.3 and section 6.9).

When a SYSTEM is first executed, the <system body> is automatically started, However, all other processes, must be explicitly activated using the START statement, The <system body> should only contain the code to initialise the system, which will typically consist of a series of START statements,

On process activation, stack space is allocated to the process from the heap, The amount of stack space to be allocated to a process is set using the concurrent characteristic:

```
{# STACKSIZE = required_stack_size }
```

which is part of the process declaration.

A process can be in one of three states:

- Ready - the process is able to run (but there is a higher priority processes currently executing).
- Active - the process is being executed. Under Microprocessor Pascal, the active process (there can only be one) is always the ready process with the highest priority.

- Blocked - the process is suspended (waiting for an event from another process to occur) and unable to run until the event has occurred.

6.8.2 Process Record

Each process has a unique process record. This is used by the executive to access information particular to a given process (where its stack is located, its identity, its priority, etc). The process record is also used for storing a process's volatile environment: display, program counter (PC), workspace pointer (WP), and status register (ST). (For an explanation of PC, WP and ST see section 8.4.3.)

The display is a 16-word area containing addresses of the stack frames which can be accessed by the currently executing routine (ie data areas of other blocks which are in scope). The display is a 'short cut' means of access to remote stack frames that is quicker than tracing back through the stack frame linkage.

6.8.3 Process Scheduling

The executive Run-Time Support (RTS) determines which of several concurrent processes is to be executed next based on process readiness and process priority. The scheduling policy used is known as pre-emptive priority scheduling.

Every process in a SYSTEM has a priority in the range 0 (highest or most urgent) to 32766 (lowest or least urgent). This is specified by the concurrent characteristic:

```
{# PRIORITY = required_priority_level }
```

which is part of the process declaration. Priorities 0 to 15 are reserved for interrupt device handling processes.

Through the process records, the executive maintains two queues: one is a circular list of all the processes known in the system; the other, the ready queue, is a priority ordered queue of processes that are in the ready state. The scheduling algorithm takes the first process in the ready queue and makes that the active process. This process is allowed to continue its execution until either it terminates, it becomes blocked, or a higher priority process that was blocked becomes ready.

When a process becomes blocked, it is removed from the ready queue and the active process becomes the next process in the ready queue. If a process's state is changed form blocked

to ready, it is inserted into the ready queue according to its priority. (The process will be inserted into the ready queue after processes with the same priority. Interrupt device handling processes are inserted into the queue before processes with the same priority.) If the process which has just become ready is inserted into the ready queue in front of the active process, then the processor is pre-empted and the new process becomes the active process.

To ensure that there is always at least one process in the ready state, the executive RTS automatically creates the 'idle process' (with the lowest priority possible - 32767) on system initialisation.

6.8.4 Process Synchronization

Processes are independent but it is often necessary for them to synchronize their actions. The simplest way of doing this is via the semaphore and its primitive operations wait and signal. Although these operations are implemented as routines (ie a collection of instructions) they must be executed as though they are single machine instructions. Until the routines have completed, nothing must access the semaphore, the queues operated on, or the wait and signal 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 in Section 4.11.1.

6.8.4.1 Semaphores

The semaphore is considered to be so fundamental to process synchronization that it is a predefined Microprocessor Pascal type (like an INTEGER or REAL), Although the compiler recognises the type semaphore (and allocates one word for each semaphore variable), a semaphore variable is, in fact, a pointer to a structure that is allocated from the heap at run-time by the INITSEMAPHORE procedure.

The required Microprocessor Pascal statements to create a semaphore are:

```
PROCEDURE initsemaphore(VAR sema : SEMAPHORE;
                        value : INTEGER); EXTERNAL;
.
VAR semaphore_name : SEMAPHORE;
.
initsemaphore(semaphore_name,initial_value);
```

After executing the INITSEMAPHORE routine, the variable SEMAPHORE_NAME will reference the newly created semaphore,

which will have its counter component set to INITIAL_VALUE, For most applications INITIAL_VALUE will be set to zero.

A semaphore consists of three elements:

- o A non-negative counter of unserviced events.
- o A queue (possibly empty) of suspended processes, This queue uses First In First Out (FIFO) ordering.
- o A check word that allows the executive to ensure that semaphore operations are actually being performed on semaphores,

The Microprocessor Pascal RTS gives greater flexibility in handling semaphores by providing routines in addition to the basic WAIT and SIGNAL operations(a full list of these can be found in section 6.13.9.3).

6.8.4.2 Wait Operation

A WAIT operation decrements the semaphore's non-negative counter if it is non-zero, otherwise the issuing process (the active process) is put into the blocked state, (The process is removed from the scheduling ready queue and inserted into the semaphore queue,)

6.8.4.3 Signal Operation

A SIGNAL operation increments the semaphore's non-negative counter if the semaphore queue is empty, otherwise the first process in the queue is put into the ready state. (The process is removed from the semaphore queue and reinserted into the scheduling ready queue,)

The classic producer/consumer situation is an obvious example of process synchronisation, In this, one process produces an item (eg a buffer full of text) while another one consumes it, A simplified version of this is shown below,

```
PROCESS producer              PROCESS consumer
  .                             .
  . (make item)                 .
  .                             .
  signal(done)                  wait(done)
  .                             .
  .                             . ( use item }
  .                             .
```

CONSUMER must WAIT for ITEM to be made before it attempts to use it. If PRODUCER has already made ITEM, the semaphore DONE (initialised to zero) is SIGNALed and CONSUMER will be able to continue. Otherwise CONSUMER will be suspended which will allow PRODUCER to make ITEM. When ITEM has been made, the SIGNAL will cause CONSUMER to be removed from the semaphore queue and inserted back into the scheduling ready queue.

If the CONSUMER and PRODUCER processes are cyclic, then the above example cannot be relied upon as there is no guarantee that CONSUMER has finished with ITEM before PRODUCER replaces it with a new one. A more complete example is:

```
PROCESS producer                PROCESS consumer
BEGIN                           BEGIN
  WHILE TRUE DO                   WHILE TRUE DO
  BEGIN                           BEGIN
    wait(available);                wait(done);
      .                               .
      .  { make item }                .  { use item }
      .                               .
    signal(done)                    signal(available)
  END                             END
END;                            END;
```

The semaphore AVAILABLE is initialised to one so that on the first time around the loop, PRODUCER does not get suspended,

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 two or more processes are suspended, awaiting a condition that can not happen because there is no active process to cause the needed event to occur.

For example, if two simultaneously executing processes (A and B) both require exclusive access to resources (X and Y), the following sequence may 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 he waiting for X (which A has). One possible way to ensure that this does not happen is to force both processes to request the resources in the same order. However, in some situations this might not be practical or efficient. Here either (or both) processes must check the availability of succeeding resources and, if unavailable, release those already acquired.

6.8.5 Interprocess Communication

To implement a practical function it is usually necessary for a process to be able to communicate with other processes in the system. Microprocessor Pascal supports four mechanisms for interprocess communication. These are described below.

6.8.5.1 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 (the semaphore is initialised to one), 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;
```

The WITH statement above is used to simplify references to components of a record structure. This allows B.MUTEX and B.SHARED_VARIABLE to be referred to by the identifiers MUTEX and SHARED_VARIABLE respectively.

6.8.5.2 Message Buffers

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 (ie the messages are buffered). In this context a "message" is any structure which can be copied from one process to another.

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 and NEXT_OUT are set to zero.

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 max_messages +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 max_messages +1;
   signal(mutex);
   signal(not_full)
END;
```

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 MAX MESSAGES and then adding one allows the message buffer to be- used in a circular fashion (a

buffer managed in thfs way is known as a circular or ring buffer).

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

6.8.5.3 Channels

The channel mechanism permits communication between any two (or more) concurrent routines (PROGRAMs or PROCESSes) in a system, Channel data structures are not pre-defined in the program code, but are allocated dynamically from the system heap as required, Channels provide a standard, pre-written set of routines for exchanging messages,

Channels also provide more flexibility, The two previous mechanisms do not allow communication between PROGRAMs, or between PROCESSes defined within different PROGRAMs (as variables cannot be defined at the SYSTEM level),

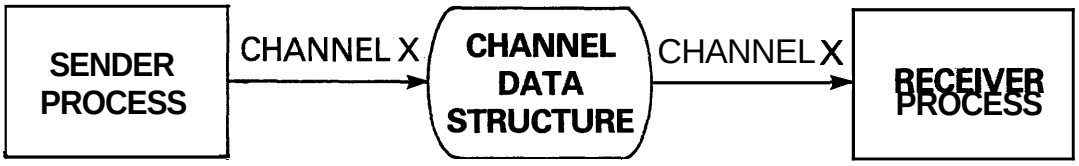


Figure 6-8 Channel Mechanism

Channels are referenced by channel names (in fact, channel names are 16 bit numbers). There is a system-wide directory of channel names, maintained by the executive, which is referenced whenever a PROCESS or PROGRAM wishes to "connect" to a channel. It is also possible to allocate channels which are specific to an individual software package (for example, the Interprocess File Subsystem makes use of a locally defined set of channels for internal operations),

In order to use the channel mechanism:

- o All participating concurrent routines must agree on the channel name to be used, This is hard-coded into the routines,
- o Each participating routine requests the executive to allocate and initialise the data structures for a particular channel name using

the C$INIT procedure,

- A routine that wants to send data along the channel allocates a message buffer using C$ALLOCATE. The required message is written into the appropriate fields of the message buffer which is then "transmitted" using C$SEND. A call to C$WAIT ensures that the transmitting routine does not access the message buffer until the receiving routine has finished processing it, When processing has completed, the message buffer can either be re-used or returned to the system heap using C$DISPOSE.

- A routine that wishes to receive data calls the procedure C$RECEIVE. This routine will wait until a message has been sent, if one is not already available, When the message has been processed, C$ACKNOWLEDGE is used to inform the sending routine that the message buffer is no longer being used,

A typical data declaration sequence is:

```
CONST channel_no      = any_user_required_number;
TYPE  msg_buffer_ptr  = @msg_buffer;
      msg_buffer      = RECORD
                          { Any required structure }
                        END;
      channel_id_ptr  = @INTEGER;

VAR buffer     : msg buffer_ptr;
    channel_id : channel_id_ptr;
```

The sending routine is:

```
                  { Allocate channel CHANNEL_NO from
                    the heap and reference it through
                    the variable CHANNEL_ID          }
C$INIT(channel_no,channel_id);
                  { Allocate a message buffer and refer-
                    ence it through the variable BUFFER }
C$ALLOCATE(size(buffer),buffer);
    .
    .  { Fill the message buffer }
    .
                  { Send the filled message buffer
                    referenced by BUFFER             }
C$SEND(channel_id,buffer);
                  { Wait for the receiver to finish
                    processing the message buffer    }
C$WAIT(buffer);
                  { Return the "used" message buffer
                    back to the system heap           }
C$DISPOSE(buffer);
```

The receiving routine is:

```
                { Allocate channel CHANNEL_NO from
                  the system heap and reference it
                  through the variable CHANNEL_ID     }
C$INIT(channel_no,channel_id);
                { Wait for the next message buffer
                  sent via the channel CHANNEL NO
                  and reference it through the-       )
                  variable BUFFER
C$RECEIVE(channel_id,buffer);
    .
    .   ( Process the message }
    .
                { Inform the sender that the message
                  buffer is no longer in use          }
C$ACKNOWLEDGE(buffer);
```

A concurrent routine can "disconnect" itself from a channel by calling C$TERM. When all routines have been disconnected from a channel then the channel data structures will be returned to the system heap.

Other channel procedures available include C$NOTIFY (signal the calling process whenever a message arrives on the specified channel), C$CRECEIVE (check to see if a message has arrived hut do not wait if none has), and C$CWAIT (check if' the message has been processed but do not wait if it has not),

6.8.5.4 Interprocess Files

The fourth communication mechanism is implemented using file variables (see section 6.6.12) that communicate through interprocess files. Interprocess files allow concurrent routines to write to other concurrent routines exactly as if they were writing to external devices, However, the communication mechanism is handled entirely in internal memory (by the Interprocess File Subsystem), The standard file I/O procedures (READ, WRITE, etc) are used in exactly the same way as for external files.

Each interprocess file has a character string name which is identical to the names of all file variables connected to it.

A fife 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.

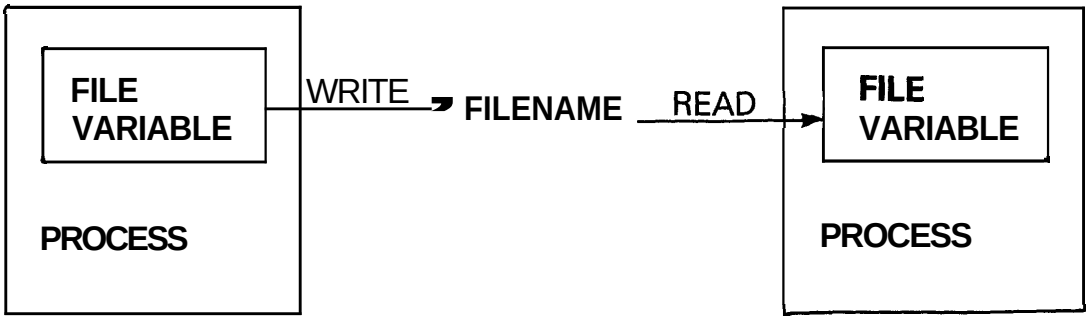


Figure 6-9 Interprocess File Mechanism

Files must be opened by calling the procedure REWRITE for write operatipns 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 file channel with the same name as the file variable. If no file 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 file channel. A file 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 interprocess file TRANSFER, Process A opens the interprocess file TRANSFER for 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(.....);
.                                .
```

where these two processes are activated by

```
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 the second example for process A as this is automatically performed on the default output text file OUTPUT,

6.9 MODULARITY

One of the most important features not addressed by Wirth's original definition of Pascal is that of modularity, Modularity allows a problem to be defined in terms of a number of separate, self-contained, sub-problems (each of which has a clearly defined interface). A sub-problem can, in turn, be broken down into further sub-problems, Typically, this decomposition continues until each sub-problem is of a manageable size,

In Microprocessor Pascal, the language constructs SYSTEM, PROGRAM and PROCESS enforce a modular approach to program development, This hierarchical concurrent structure permits the construction of complex concurrent functions which can be encapsulated in a single package,

The fundamental unit of modularity is the PROGRAM; this represents an independent function which has its own unique "site of execution', Although functions execute concurrently with each other (with no possibility that one will interfere with another), the code that the function consists of typically executes sequentially,

However, in a complex function, it may be necessary to create the function from a number of independent concurrent sub-functions, This situation is catered for by the PROCESS construct. Like PROGRAMs, PROCESSes are separate "sites of execution" which are activated by being STARTed; they are not simply "called' like PROCEDUREs and FUNCTIONs,

The complete structure of a PROGRAM with all subordinate PROCESSes (and PROCEDUREs and FUNCTIONs) is referred to as a PROGRAM family, The PROGRAM family is a convient package for a complete, independent function within a system. The concurrent structure is described in Section 5.2.2.

If, for example, a function was to be designed to control a lathe, sub-functions required might be 'monitor the chuck speed', 'control the cutting depth' and 'control the cutter position'.

```
PROGRAM Control_lathe;
declarations;

      PROCESS Monitor_chuck_speed;
      declarations;
      BEGIN ( Monitor_chuck_speed }

      END;  { Monitor_chuck_speed }

      PROCESS Control_cutter_depth;
      declarations;
      BEGIN { Control_cutter_depth }

      END;  ( Control_cutter_depth }

      PROCESS Control_cutter_position;
      declarations;
      BEGIN { Control_cutter_position }

      END;  ( Control_cutter_position }

BEGIN ( Control lathe }
  START  Oni it Or-chuck_speed;
  START Control_cutter_depth;
  START Control cutter_position
END;  { control_lathe }
```

As each function, and sub-function, are separate "sites of execution" and, once **STARTed,** execute totally independently of the system, the user is able to specify the **concurrent** characteristics(heapsize, stacksize and priority) to be used for each, These are defined by:

```
      BEGIN ( program or process body }
        {# STACKSIZE = amount_of_stack;
            HEAPSIZE = amount_of_heap;
            PRIORITY = program_or_process_priority }

      END;  ( program or process body }
```

Under Microprocessor Pascal, an application is put together from functions to form a system. A SYSTEM consists of a number of declarations(constants, types, commons, PROGRAMs, procedures and functions) and a <system **body>.** The (system body) contains the instructions that are first executed when the system is initialized; it also specifies the concurrent characteristics to be used while this initialization is being **performed.**

```
SYSTEM Look_after_shop_floor;
CONST  declarations;
TYPE   declarations;
COMMON declarations;

       PROGRAM Control_lathe;
       declarations;
       BEGIN { Control_lathe }

       END;  { Control_lathe }

       PROGRAM Control_miller;
       declarations;
       BEGIN { Control_miller }

       END;  { Control_miller }

BEGIN { Look_after_shop_floor }
  {# system concurrent characteristics }

  START Control lathe;              { system body }
  START control_miller;

END.  { Look_after_shop_floor }
```

Modularity is further enhanced by allowing the user to develop and compile modules in complete isolation from each other and to link them together into a consistent system at "configuration time". These modules may contain PROCEDURE, FUNCTION and/or PROGRAM definitions (along with any necessary data declarations). In this case, only one module must have a real system body. The others must have a "null system body", declared by:

```
SYSTEM System_dummy_name;
declarations;

     PROCEDURE definitions;
     FUNCTION  definitions;
     PROGRAM   definitions;

BEGIN { System_dummy-name }
  {$ nullbody }
END.  { System_dummy_name }
```

When the modules are linked together to form the system, there will be only one <system body). PROCEDUREs, FUNCTIONs, PROGRAMs or PROCESSes that are not defined in a module but are used within it are accessed by declaring them as EXTERNAL.

Further development of this modular approach, to encompass hardware as well as software, leads to a functional approach (see Section 5.1.1).

Note: "FUNCTION" capitalised has a precise technical meaning, as distinct from the more general use of "function",

6.10 INTERRUPTS

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 section 8.10.

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 a Microprocessor Pascal system are assigned a priority, in the range 0 to 32,766. The first 16 priorities, 0 to 15, are reserved for use by device processes,

A process with a priority of (eg) 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.

The procedure EXTERNALEVENT is used to associate a semaphore with a particular interrupt level, 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 (eg if it has not finished processing the last interrupt). 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 ( interrupt_7 }
      {# priority=7;..... };
      WHILE TRUE DO
      BEGIN { do forever }
        wait(level);
        ( process interrupt level 7 }
      END   ( forever loop }
    END;  { interrupt_7 }

    PROCESS spurious_7(level: SEMAPHORE);
    .
    BEGIN { spurious_7 }
      {# priority=7;..... };
      wait(level);
      ( process spurious interrupt }
    END;  ( spurious_7 }

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 }
```

If a fast device is incorporated into the system, the Microprocessor Pascal interrupt handling mechanism may be too slow and it may be necessary to write an assembly language interrupt handler. To cover this eventuality, the user can "hook" the assembly language routine into the system in two ways,

- o Using the ASSEMBLYEVENT procedure.
- o Setting the appropriate interrupt vector (in the "RXINIT" module) to reference the assembly language routine and its workspace. In this case the interrupt is handled totally outside the Microprocessor Pascal run-time environment.

The ASSEMBLYEVENT procedure is used as follows:

```
CONST level = required_interrupt_level_value;

TYPE workspace = ARRAY [1..16] OF INTEGER;

VAR asm_wp : workspace;

PROCEDURE assemblyevent(VAR wp : workspace;
                        entry point : INTEGER;
                              level : INTEGER);  EXTERNAL;
PROCEDURE asm_idt; EXTERNAL;


assemblyevent(asm_wp, location(asm_idt), level );
```

where ASM_IDT is the entry point label of the assembly language interrupt handler. LOCATION returns the address of ASM_IDT.

Note: The host debugger does not support assembly language routines.

6.11 INPUT/OUTPUT

611.1 CRU Operations

Microprocessor Pascal supports direct 9900 CRU operations (for those unfamiliar with the CRU concept see Section 8.9) via the following standard procedures:

```
CRUBASE(base)
LDCR(width, value)
SRO(disp)
SBZ(d'isp)
STCR(width, value)
```

and the BOOLEAN function:

```
TB(disp)
```

Although these are written as procedure calls, the Microprocessor Pascal compiler actually transforms the calls into in-line code.

6.11.2 Memory-Mapped 1/0

Communication to a memory-mapped device is performed by:

- Describing the structure of the device's dedicated memory space in a type declaration (if

the device has a control register it will be necessary to describe the individual control flags in a packed record structure). In the example below, this is the type identifier CNTL_REG,

- o Declaring a pointer variable that points to this type (CNTL_REG_PTR below),
- o Initialising this pointer variable to point to the actual address of the memory-mapped device via a "type transfer" (see section 6.6.14).

Having done this, assigning a value to the pointer variable (or the appropriate field of it, if it is a packed record) causes the value to be "written" to the device.

Referencing the variable on the right hand side of an assignment statement, or anywhere an expression is required, will cause the device to be "read",

For example: If an 8 bit digital to analogue converter is located at hex address >FC06, then the following Microprocessor Pascal statements will cause the value 127 to be written to the device,

```
CONST address_of_the_device = #FC06;
      value_to_be_output    = 127;

TYPE cntl_reg_ptr = @cntl_reg;
       cntl_reg = INTEGER;

VAR  dac : cntl_reg_ptr;

  dac::INTEGER := address_of_the_device;

  dac@ := value_to_be_output;
```

As the D/A only has an 8 bit resolution, CNTL_REG could be defined as:

```
TYPE cntl_reg =
       PACKED RECORD
         fill   : 0..255;   "8 unused bits
         output : 0..255    "8 bit output value
       END;
```

The output operation now becomes;

```
  dac@.output := value-to_be_output;
```

If a sequence of operations is to be performed on the memory-mapped device then the Microprocessor Pascal keyword WITH can be used to "shorten" the variable name (see section 6.8.5.1).

For a 12 bit analogue to digital converter, located at hex address C01A, the following Microprocessor Pascal statements will cause the device to be read.

```
TYPE bits12              = 0..#FFF;
     a_to_d_cntl_reg_ptr = @a_to_d_cntl_reg;
     a_to_d_cntl_reg =
        PACKED RECORD
          .
          start_conversion_flag  : BOOLEAN;
          end_of_conversion_flag : BOOLEAN;
          .
          input_bits : bits12;
          .
        END;

VAR  a_to_d       : a_to_d_cntl_reg_ptr;
    input_reading : bits12;

a_to_d::INTEGER := #C01A;  { Set a_to_d address }

WITH a_to_d DO
BEGIN
  ( If another reading is available then get it,
    then initialise the A/D for the next reading }
  IF end_of_conversion_flag THEN
  BEGIN
    input reading := input_bits;
    start-conversion := TRUE; { Set start conversion }
    start=conversion := FALSE { pulse                 }
  END;
  .
END;
```

6.11.3 Files

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, File types are described in section 6.6.12 above, and in the Microprocessor Pascal System User's Manual.

6.12 DIGITAL VOLTMETER (DVM) EXAMPLE

This example consists of four independent "do forever" processes that synchronise their actions via semaphores.

The system structure for this example is shown below:

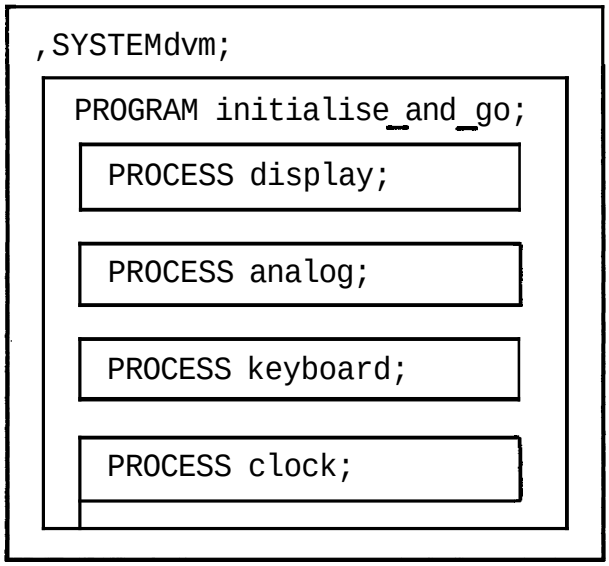


Figure 6-10' DVM Example - Lexical Hierarchy

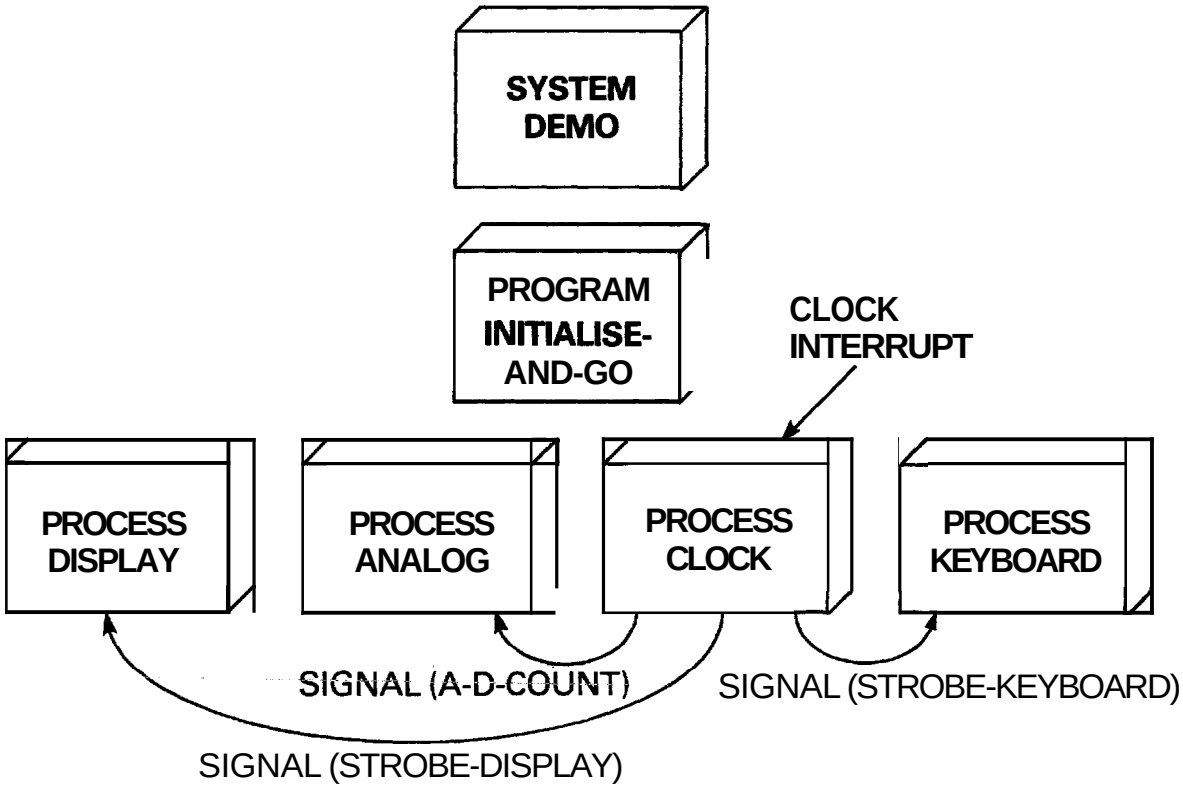


Figure 6-11 DVM Example - Concurrent Structure

```
{*************************************************************
*                                                            *
* Microprocessor Pascal Concurrency Demonstration Program    *
*                                                            *
*                 Dave Wollen, EMTC, Bedford                 *
*                                                            *
*                       15 Oct 1979                          *
*                                                            *
*  DESCRIPTION                                               *
*     The program implements a simple digital voltmeter      *
*     using a special demonstration box,   The main          *
*     purpose is to illustrate Microprocessor Pascal,        *
*     especially concurrent processing, and for this         *
*     reason the system has been implemented as a number     *
*     of separate processes synchronized by semaphores,      *
*                                                            *
*     The A/W used includes a strobed keyboard, strobed      *
*     LED display(with  decoders) and a Texas Instruments    *
*     TL505 A/D converter.  The system will run on a         *
*     Texas Instruments TM990 microprocessor module with     *
*     at least >2AF0 bytes of program memory,  The on-       *
*     board TMS9901 is used to provide clock interrupts,     *
*                                                            *
*     The H/W is set up in such a way that the keyboard      *
*     may not be used when the analogue input switch is      *
*     in the ON position.                                    *
*                                                            *
*  OPERATION                                                 *
*     When the analogue input switch is "OFF" a threshold    *
*     voltage can be keyed in(hundredths  of a volt),  with  *
*     the system accepting only the last four digits         *
*     keyed.   To start converting, key "GO" and turn on     *
*     the analogue input switch,  The input voltage will     *
*     be constantly monitored and displayed; if it rises     *
*     above the entered threshold the display will show      *
*     9999 until it falls below threshold once more,  To     *
*     alter the threshold, turn off analogue input, key      *
*     "STOP" and enter new value,                            *
*                                                            *
*************************************************************}
SYSTEM demo; {$debug)
TYPE   non_neg = 0..32767;
       interrupt = 0..15;

  PROCEDURE initsemaphore(VAR sema: SEMAPHORE;
                              count: non_neg); EXTERNAL;

  PROCEDURE externalevent(sema: SEMAPHORE;
                          level: interrupt);  EXTERNAL;

  PROCEDURE wait(sema: SEMAPHORE);  EXTERNAL;

  PROCEDURE signal(sema: SEMAPHORE);  EXTERNAL;

  PROGRAM initialise_and_go;
```

```
CONST   interrupt_level = 3;
VAR     threshold; analog_value: ARRAY [0..3] of 0..9;
        converting: BOOLEAN;
        time: SEMAPHORE;
        time_to_strobe_display: SEMAPHORE;
        time_for_a_d_count: SEMAPHORE;
        time_to_strobe_keyboard: SEMAPHORE;

  PROCESS clock;
  CONST   clock_mode = 0;      enable_clock_interrupt = 3;
          timer_on_9901 = #100;   period_for_58hz = #65D;

  {This process synchronises all others.  It initialises
   the 9901 clock register and waits for each level 3
   interrupt, after which it signals to other processes
   that they can resume.  If the period between
   interrupts is made too short, other processes will
   not run to completion; for the sake of brevity no
   attempt is made to cope with this.)

  BEGIN(clock)
    {# STACKSIZE=50; HEAPSIZE=0; PRIORITY=interrupt_level}
    crubase(timer_on_9901);
    ldcr(15, period_for_58hz);
    WHILE TRUE DO
    BEGIN
      sbz(clock_mode);
      sbo(enable_clock_interrupt);
      wait(time);
      signal(time_to_strobe display);
      signal(time_for_a_d_count);
      signal(time_to_strobe_keyboard)
    END
  END;  {clock)

  PROCESS display;
  CONST   num_of_bits = 9;      display_base = 288;
          high_byte = 8100;     low_byte = 0;
  VAR     dig_ptr: 0..3;
          byte_selector: 0..#100;
          display_output: 0..#199;

 (This  process strobes and updates the display when
   it has been signalled to do so.  It simply converts
   the appropriate two digits of threshold or analog_
   value(depending  on the current mode) to a bit
   pattern(including  the strobe bit) and outputs this
   pattern to the CRU.}

  BEGIN(display)
    {# STACKSIZE=50; HEAPSIZE=0; PRIORITY=16}
    dig_ptr := 2;
    crubase(display_base);
    WHILE TRUE DO
    BEGIN
```

```
    wait(time_to_strobe display);
    IF dig_ptr = 2 THEN-
      BEGIN
        dig_ptr := 0;
        byte-selector := low_byte
      END
    ELSE
      BEGIN
        dig_ptr := 2;
        byte-selector := high_byte
      END;
    IF converting THEN
       display_output := analog_value[dig_ptr]
                       + analog_value[dig_ptr + 1]*16
                       + byte_selector
    ELSE
       display-output := threshold[dig_ptr]
                       + threshold[dig_ptr + 1]*16
                       + byte_selector;
    ldcr(num_of_bits, display_output)
  END  {while}
END;  {display)

PROCESS analog_to_digital_converter;
CONST   a_d_base = 308;     comparator_on_505 = 4;
        A_input_to_505 = 0;  B input_to_505 = 1;
        t0 = 25;             ti = 25;
        Vref = 250;          ratio = Vref DIV tl;
        max_count = 32767 DIV ratio;
TYPE    conversion_period = (pre_con, in_t0, in_tl, in_t2);
VAR     count: 0..max_count;
        when: conversion_period;
        limit, millivolts: INTEGER;

{This process implements all the A/D conversion.  The
 TL505 requires a specific sequence of events to occur
 for conversion, and the final representation of the
 analog value is the value held in a S/W counter, which
 may then be scaled etc as required,  The symbols used
 in this process correspond to those used in the 505
 data sheet, to which further reference should be made.
 If the current mode is "not converting'' then the 505
 control lines are kept high,)

BEGIN(analog_to_digital_converter)
  {* STACKSIZE=50; HEAPSIZE=0; PRIORITY=16}
  crubase(a_d_base);
  WHILE TRUE DO
  BEGIN
    wait(time_for_a_d_count);
    IF converting THEN
      BEGIN
        count := count + 1;
        CASE when OF
          pre-con : BEGIN
```

```
                    sbz(A_input_to_505);
                    sbz(B_input_to_505);
                    when := in_t0;
                    count := 0
                  END;
          in_t0   : IF count = t0 THEN
                  BEGIN
                    sbo(A_input_to_505);
                    sbo(B_input_to_505);
                    when := in_t1;
                    count := 0
                  END;
          in_t1   : IF count = t1 THEN
                  BEGIN
                    sbz(A_input_to_505);
                    when := in_t2;
                    count := 0
                  END;
          in_t2   : IF tb(comparator_on_505) THEN
                  BEGIN
                    sbz(B_input_to_505);
                    when := in_t0;
                    millivolts := ratio * count;
                    count := 0;
                    limit := threshold[3]*1000
                           + threshold[2]*100
                           + threshold[1]*10
                           + threshold[0];
                    IF millivolts > limit THEN
                       millivolts := 9999;
                    FOR i := 0 TO 3 DO
                    BEGIN
                      analog_value[i] := millivolts MOD 10;
                      millivolts := millivolts DIV 10
                    END
                  END {if tb)
        END (case)
      END (if converting)
    ELSE
      BEGIN
        when := pre_con;
        sbo(A_input_to_505);
        sbo(B_input_to_505)
      END
  END (while)
END; (analog_to_digital_converter)

PROCESS keyboard_input;
CONST   width = 4;          strobe = 0;
        key_input = 312;    nothing = FF;
VAR     row: 306,.312;
        key_push, last_push: 0..15;
        updated: BOOLEAN;

  PROCEDURE update_inputs;
```

```
VAR       key: 0..12;

  PROCEDURE change_threshold;

  {This procedure shift threshold and accept most
   recent key into least significant position)

  BEGIN {change_threshold)
    FOR i := 3 DOWNTO 1 DO threshold[i] := threshold[i-1];
    threshold[0] := key
  END; (change_threshold)


{This procedure decode the keyboard and take appropriate
 action.  The keyboard is arranged as follows:

                LSB.--       ...MSB

          310     1    2    3     4
          308     5    6    7     8
          306     9    0   go    stop          }

BEGIN {update_inputs)
  CASE key_push OF
    #E      : key := 1;
    #C, #D  : key := 2;
    8..#B   : key := 3;
    0..7    : key := 4
  END;
  key := key + 4*abs((row - 310) DIV 2);
  CASE key OF
    10  : key := 0;
    11  : IF NOT converting THEN converting := TRUE;
    12  : converting := FALSE;
  OTHERWISE
  END;
  IF NOT converting AND key < 11 THEN change_threshold;
END; (update_inputs)


(This process strobes the keyboard and debounces
 and decodes any input when signalled to do so.  If
 the mode is "converting", the only key of interest
 is "stop".  Keys are active when low.)

BEGIN(keyboard_input)
  {# STACKSIZE=50; HEAPSIZE=0; PRIORITY=16}
  row := 306;
  key_push := nothing;
  last_push := nothing;
  updated := FALSE;
  WHILE TRUE DO
  BEGIN
    wait(time_to-strobe_keyboard);
    crubase(row);
```

```
        sbz(strobe);
        crubase(key_input);
        stcr(width, keg_push);
        crubase(row);
        sbo(strobe);
        IF key_push = nothing THEN
          BEGIN
            updated := false;
            row := row + 2;
            IF converting OR row = 312 THEN row := 306;
          END
        ELSE
          IF key_push = last_push AND NOT updated THEN
          BEGIN
            update_inputs;
            updated := TRUE
          END;
        last_push := key_push
      END
    END; (keyboard _input)


  {This program is used to initialise all the semaphores,
   zero the threshold and analog_value arrays and start
   all the other processes)

  BEGIN(initialise _and_go)
    {# STACKSIZE=300; HEAPSIZE=800; PRIORITY=16}
    initsemaphore(time_to_strobe display, 0);
    initsemaphore(time_for_a_d_count, 0);
    initsemaphore(time_to_strobe_keyboard, 0);
    initsemaphore(time, 0);
    externalevent(time, interrupt_level);
    converting := FALSE;
    FOR i := 0 TO 3 DO
    BEGIN
      threshold[i] := 0;
      analog_value[i] := 0
    END;
    START display;
    START analog_to_digital-converter;
    START keyboard_input;
    START clock
  END; (initialise _and_go)

BEGIN {demo)
  {# STACKSIZE=300; HEAPSIZE=0; PRIORITY=16}
  START initialise_and_go
END. (demo)
```

6.13 REFERENCE SECTION

6.13.1 System Commands

Compile a Microprocessor Pascal program in background	* BATCH
Generate native code	CODEGEN
Collect MPIX run-time support	COLLECT
Compile a Microprocessor Pascal program	COMPILE
Copy text files	# COPY
Copy text files	* COPYSRC
Debug a compiled Microprocessor Pascal program	DEBUG
Delete temporary files	* DELETE
Create/edit a file	EDIT
Execute a compiled Microprocessor Pascal program	EXECUTE
Generate routine map	GENMAP
Print a stored file	* PRINT
Delete synonyms used	* PURGE
Reverse assemble object code	RASS
Save an edited file	SAVE
Execute SCI command	* SCI
Display a stored file	SHOW
Separate object modules	SPLIT
Terminate a Mircroprocessor Pascal session	QUIT
File utility program	# UTILITY
Wait for background task to finish	* WAIT

* Only for DX990 users
\# Only for FS990 and TMAM9000 users

6.13.2 Utility Commands (990/4 and TMAM9000 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 program execution	TE

6.13.3 Edit Commands

Help	CMD HELP
Edit/compose mode	F7 key
Syntax check	CMB CHECK
Terminate and save edit	CMD QUIT
Terminate without saving	CMD ABORT
Change editing files	CMD INPUT
Save the edited file	CMD SAVE
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

NOTES

CMD HELP
Strike the CMD key and then type in the word HELP.

[exp]
Indicates that item EXP is optional. Optional items may be omitted (they default to 1) along with any preceding comma.

pattern
Is either an identifier or a string of characters enclosed within double quotes.

6.13.4 Debug Commands

Getting Started/Finished	
Resume execution	GO
Terminate DEBUG session	QUIT
Help	HELP(command)
Load saved program	LOAD ("pathname")
Copy commands from file	COPY ("pathname")
Show unresolved externals	SE
Status Displays	
Display process	DP([process])
Display all processes	DAP
Breakpoints/Single Step	
Assign breakpoint	AB(routine,[statement number])
Delete breakpoint	DB(routine,[statement number])
Delete all breakpoints	DAB(process)
List breakpoints	LB([process])
Select single step mode	SS([process],[flag])
Showing/Modifying Data	
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 memory 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)
Tracing Execution	
Trace process execution	TP([process],[flag])
Trace routine entry/exit	TR([process],[flag])
Trace statement flow	TS([process],[flag])
Monitor Process Scheduling	
Select default process	SDP(process)
DEBUG the process	DEBUG(process,[flag])
Assign breakpoint to process	ABP(process)
Delete breakpoint from process	DBP(process)
Hold process	HP(process)
Release process	RP(process)
Interprocess File Simulation	
Connect input file	CIF(file1,[file2])
Connect output file	COF(file1,[file2])
Interrupt Simulation	
Simulate interrupt	SIMI(level)
Selection of CRU Mode	
Select CRU mode	CRU([process],cru mode)

NOTES

[x]
Indicates that the item X is optional. Parenthesis may be omitted if all the parameters are optional or defaulted.

process
If omitted it defaults to that set by SDP. It may be either a name (youngest instance of the PROCESS) or an integer constant (older instance of a particular PROCESS), found using DAP.

routine
May he 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 with PROCESS (this is followed by '.').

flag
Is an identifier that is either TRUE or FALSE: if TRUE the command is enabled; if FALSE the 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
An 8 character Microprocessor Pascal file name identifier enclosed in double quotes.

file2
A file pathname enclosed in double quotes. If omitted it defaults to the user's terminal.

cru mode
One of the following:

EXECUTE	Execute all CRU instructions
OFF	Ignore all CRU instructions
DEBUG	Default - All CRU 1/0 is user-simulated

6.13.5 File Manipulation Routines

CLOSE(f)
Place file F in closed state.

DECODE(s,n,stat,q)
Convert string S, starting at the Nth component of S, into a form compatible with the read variable Q (see NOTE 2) and store in Q. Status of the operation is returned in STAT,

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 Nth component. The status of the operations is returned in STAT.

EOF(f) : BOOLEAN FUNCTION
Returns a value of TRUE if the file F is not open for input or is in the end-of-file state.

EOLN(f) : BOOLEAN FUNCTION
Returns a value of TRUE if the last character of the current line in the file F has been read.

FILENAMED(S) : **ANYFILE** FUNCTION
Connects the file variable of type **ANYFILE** to the file with the name S (S is a string constant).

MESSAGE(x)
Write the string X to the system message **file.**

READ(f,v1,..,vn) Sequential
READ(v1,..,vn) ---> **READ(INPUT,v1,..,vn)** Text
READ(f,recnum,v1,..,vn) Random
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,v1,..,vn)
READLN(v1,..,vn) ---> **READLN(INPUT,v1,..vn)**
READLN(INPUT)
Read the components of a text file into the specified variables 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)** is 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,

WRITE(f,v1,..,vn) Sequential
WRITE(v1,..,vn) ---> WRITE(INPUT,v1,..,vn) Text
WRITE(f,recnum,v1,..,vn) Random
Write the components to a Sequential, Text or Random file from the specified variables V1..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,v1,..,vn)
WRITELN(v1,..,vn) ---> WRITELN(OUTPUT,v1,..vn)
WRITELN(OUTPUT)
Write the components to a text file F from the specified variables V1..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 or E:M or E:M:N

E is an expression of type CHAR, INTEGER, LONGINT, REAL, BOOLEAN, or a string.

M (INTEGER expression) is the minimum field width. If omitted and E is REAL, floating point format is used.

N (INTEGER expression) specifies that 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 8) in the form:

E hex number or E:M hex number

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 are:

INTEGER	10	LOWGINT	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.

V is a CHAR - next character is read.

V is a string (length L) - next L characters are read.

V is BOOLEAN - either the character T or F is read or the identifier TRUE or FALSE.

V is INTEGER, LONGINT or REAL - 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.

6.13.6 Arithmetic Routines

All 'routines' preceded by '*' must be declared EXTERNAL.

ABS(x: INTEGER or LONGINT or REAL **) :** as arg FUNCTION
Returns the absolute value of **X.**

* **ARCTAN(** x: REAL) : REAL FUNCTION
Returns the arc tangent of the value X.

* **COS(** x: REAL **) :** REAL FUNCTION
Returns the cosine of the value X.

* **EXP(** x: REAL **) :** REAL FUNCTION
Returns the exponential value of the value X.

FLOAT(x: INTEGER or LONGINT **) :** REAL FUNCTION
Converts the value X into a real number.

* LN(x: REAL **) :** REAL FUNCTION
Returns the natural logarithm of the value X.

LINT(x: INTEGER or LONGINT or REAL **) :** LONGINT FUNCTION
Converts the value X into a long integer number.

LROUND(x: REAL **) :** LONGINT FUNCTION
Converts and rounds the value X into a long integer number.

LTRUNC(X: LONGINT or REAL) : LONGINT FUNCTION
Truncate the value X into a long integer number,

ODD(X: INTEGER or LONGINT) : BOOLEAN FUNCTION
Returns TRUE if the value of X is odd; FALSE otherwise.

ROUND(X: REAL) : INTEGER FUNCTION
Converts and rounds the value X into an integer number.

* SIN(X: REAL) : REAL FUNCTION
Returns the sin of the value X.

SQR(X: INTEGER or LONGINT or REAL) : as arg FUNCTION
Returns the squared value of X.

* SQRT(X: REAL) : REAL FUNCTION
Returns the square root of the value X.

TRUNC(X: LONGINT or REAL) : INTEGER FUNCTION
Truncate the value X into an integer number.

6.13.7 CRU Routines

The CRU 'routines'are expanded in-line by the comiler.

```
TYPE base-range        = 0..#1FFE;
TYPE width-range       = 1..16;
TYPE displacement-range = -128..127;
```

CRUBASE(base: base_range)
Set the CRU base address for subsequent CRU operations.

LDCR(width: width range; out_value: INTEGER)
Output WIDTH number of bits from the value OUT_VALUE to the CRU lines, starting from the CRU base address.

SBO(disp: displacement_range)
Set the specified bit (DISP + CRU base address) to a '1'.

SBZ(disp: displacement_range)
Set the specified bit (DISP + CRU base address) to a '0'.

STCR(width: width_range; VAR in_value: INTEGER)
Input WIDTH number of bits from the CRU, starting from the CRU base address, to the variable IN_VALUE.

TB(disp: displacement_range) : BOOLEAN FUNCTION
Returns TRUE if the specified bit (DISP + CRU base address) is a '1' and FALSE if it is '0'.

6.13.8 Miscellaneous Routines

CHR(x: BOOLEAN or INTEGER or scalar) : INTEGER FUNCTION
Returns the character with the ordinal value X.

LOCATION(x: module or unpacked type) : INTEGER FUNCTION
Returns the address of X.

ORD(x: BOOLEAN or CHAR or scalar) : INTEGER FUNCTION
Returns the ordinal value of X.

PACK(a: packed array; i: INTEGER; z: unpacked array)
Pack the components of array A into the packed array Z, starting at the Ith element of A.

PRED(x: enumeration) : enumeration FUNCTION
Returns the predecessor of X in the enumeration list.

SIZE(x: type or variable) : INTEGER FUNCTION
Returns the size (in bytes) of X.

SUCC(x: enumeration) : enumeration FUNCTION
Returns the successor of X in the enumeration list.

UNPACK(z: packed array; a: unpacked array; i: INTEGER)
Unpack the components of the packed array Z into the array A starting at the Ith element of A.

6.13.9 Rx Routines

All Rx procedures/functions called directly must be declared EXTERNAL.

6.13.9.1 Processor Management (Scheduling) Routines

```
TYPE non_device_priority = 16..32766;
```

```
SETPRIORITY( VAR oldvalue: non_device_priority;
             newvalue: non_device_priority )
```

Changes the priority of the first non-device process in the scheduling queue.

SWAP
Removes the first non-device process from the scheduling queue and inserts it behind the last process with the same priority.

6.13.9.2 Semaphore Routines

```
TYPE nonneg = 0..32766;
TYPE semaphorestate = (awaited, zero, signaled);
```

CKSEMAPHORE(sema: semaphore) : BOOLEAN FUNCTION
Returns TRUE if SEMA is a valid semaphore,

CSIGNAL(sema: semaphore; VAR waiter: BOOLEAN)
Performs a conditional signal operation on SEMA. If a waiter exists on this semaphore, a SIGNAL operation is performed on it and WAITER is set to true,

CWAIT(sema: semaphore; VAR received: BOOLEAN)
Performs a conditional wait operation on SEMA. If it has been SIGNALed, a WAIT operation is performed on it and RECEIVED is set to true,

INITSEMAPHORE (VAR sema: semaphore; count: nonneg)
Allocates and initializes the semaphore SEMA to COUNT and sets the queue management to FIFO.

SEMASTATE(sema: semaphore) : semaphorestate FUNCTION
Returns the state of the semaphore SEMA.

SEMAVALUE(sema: semaphore) : INTEGER FUNCTION
Returns the count of SEMA's initial value plus the total number SIGNALs performed on it minus the total number of WAITs performed on it.

SIGNAL(sema: semaphore)
Performs a SIGNAL operation on SEMA,

TERMSEMAPHORE(VAR sema: semaphore)
Returns the space occupied by the semaphore SEMA to Rx.

WAIT(sema: semaphore)
Performs a WAIT operation on SEMA,

WAITSIGNAL(wait_for, signal_the: semaphore)
Performs a WAIT operation on WAIT_FOR and a SIGNAL operation on SIGNAL_THE in an indivisible manner.

6.13.9.3 Semaphore Attribute Routines

```
TYPE interrupt_level = 0..15;
```

ALTEXTERNALEVENT(sema: semaphore; level: interrupt level)
Attaches the semaphore SEMA to the interrupt LEVEL - as the alternative receiver of an interrupt.

EXTERNALEVENT(sema: semaphore; level: interrupt_level)
Attaches the semaphore SEMA to the interrupt LEVEL as the primary receiver of an interrupt.

NOALTEXTERNALEVENT(level: interrupt_level)
Detaches any semaphore which has been designated the alternative receiver of the interrupt LEVEL,

NOEXTERNALEVENT(level: interrupt_level)
Detaches any semaphore which has been designated the primary receiver of the interrupt LEVEL.

6.13.9.4 Interrupt Routines

```
TYPE interrupt_result = -1..15;
TYPE word16           = ARRAY [ 0..15 ] OF INTEGERS;
TYPE wp               = @word16;
```

ASSEMBLYEVENT(VAR interrupt_wp: wp; interrupt_pc: INTEGER;
level: interrupt_level)
Assign the assembly language routine whose entry point is INTERRUPT PC to the interrupt LEVEL, INTERRUPT_WP is the workspace - to be used by this routine.

INTLEVEL : interrupt_result FUNCTION
Returns the interrupt level of the interrupt currently being serviced (0 to 15) or -1 if no interrupt is being serviced,

MASK
Disables all interrupts except for interrupt level 0.

NOASSEMBLYEVENT(level: interrupt_level)
De-assign the assembly language routine for interrupt LEVEL.

SETMASK(new_mask: interrupt_level;
VAR old_mask: interrupt_level)
Sets the interrupt mask to NEW_MASK (all interrupts less urgent than this value are disabled). The original value of the interrupt mask is saved in OLD_MASK.

UNMASK
Enables all interrupts which have a higher priority than the calling process.

6.13.9.5 Process Management Routines

```
TYPE processid = @processid;
```

MY$PROCESS : processid FUNCTION
Returns the process identification of the calling process.

P$ABORT(p: processid)
Causes process P to be marked for termination. P is aborted when it is next active; after it has returned from all Rx routines and is out of all user-defined critical regions.

P$LASTPROCESS(p: processid) : processid FUNCTION
Returns the identification of the last process started by P, or NIL if the last attempted start was unsuccessful.

P$SUCCESSFUL(p: processid) : BOOLEAN FUNCTION
Returns the status of the last process management operation performed by process P.

START$TERM(VAR oldvalue: BOOLEAN; newvalue: BOOLEAN)
Specifies the exception handling mode when processes can not be successfully started. If NEWVALUE is TRUE (default), an unsuccessful START causes the calling process to fail; else an unsuccessful START is ignored. The original value of the exception handling flag is preserved in OLDVALUE.

6.13.9.6 Heap Management Routines

```
TYPE pointer     = @INTEGER; { @any_structure }
TYPE byte_length = 0..32767;
```

DISPOSE(VAR p: pointer) Translated to FREE$ by compiler
Deallocate the heap packet specified by P and set P to NIL.

FREE$(VAR ptr: pointer)
Returns the area referenced by PTR to the heap, PTR is set to NIL.

HEAP$TERM(VAR oldvalue: BOOLEAN; newvalue: BOOLEAN)
Allows the user to specify what action heap overflow causes: error termination of the process calling NEW, or NEW$; or to ignored the condition. If NEWVALUE is TRUE (default) then error termination. The original value of the heap overflow flag is saved in OLDVALUE.

NEW(VAR p: pointer) Translated to NEW$ by compiler
Allocate a heap packet of, at least, the required size and return the address of this packet in P.

NEW$(VAR ptr: pointer; length: byte length)
Allocates, at least, LENGTH bytes of contiguous memory from the heap (if available). PTR is set to the address of this memory area.

6.13.9.7 Channel I/O Routines

```
TYPE cid        = @INTEGER;
TYPE msg_record = RECORD
                    { application defined record }
                  END;
TYPE msg_ptr    = @msg_record;
```

C$ACKNOWLEDGE(msg: msg_ptr)
The reciever acknowledges the receipt of the message referenced by MSG.

C$ALLOCATE(msg_size: INTEGER; VAR msg: msg-ptr j
Allocates a heap packet which will contain the message to be sent. The heap packet will he in two parts: a fixed length header (used by the channel routines to synchronise inter-process communication) and a message of length MSG_SIZE. The address of this heap packet is returned in MSG.

C$CRECEIVE(c: cid; VAR msg: msg-ptr)
Checks to see if a message has been sent to channel C. If a message is present, its address is returned in MSG. Otherwise MSG is set to NIL. (No waiting is performed.)

C$CWAIT(msg: msg-ptr; VAR received: BOOLEAN)
Conditionally waits for a message to be processed. If the message referenced by MSG has been processed, RECEIVED is set to TRUE. Otherwise it is set to FALSE.

C$DISPOSE(VAR msg: msg-ptr)
Return the heap packet specified by MSG to the heap and set MSG to NIL.

C$INIT(name: integer; VAR c: cid)
Allows the calling process to gain access to channel NAME, and returns the "address" of this channel in C. All subsequent calls to channel routines should reference this channel by C.

C$NOTIFY(c: cid; sema: semaphore)
Associate the semaphore SEMA to the channel C. Whenever a message is sent to this channel, the semaphore is signalled.

C$RECEIVE(c: cid; VAR msg: msg-ptr)
Waits for a message to be sent to channel C. The address of this message is returned in MSG.

C$SEND(c: cid; msg: msg-ptr)
Sends the message referenced by MSG to channel C.

C$TERM(VAR c: cid)
Disconnects the calling process from channel C. When all processes are disconnected from the channel, the structures associated with the channel are returned to the heap.

C$WAIT(msg: msg-ptr)
Waits for the message referenced by MSG to be processed.

6.13.9.8 Interprocess File Transfer Routines

F$CHABORT(VAR f: ANYFILE)
Aborts all file channels with the same name as F. All connected files are disconnected. Any subsequent 1/0 transfers to the file causes an exception to be raised. Any files suspended on the file channel are activated with an exception

F$CHBUFFERS(VAR f: ANYFILE; n: INTEGER)
Ensures that any file channels associated with file F have the capability of buffering at least N components before any producers are suspended,

F$CLENGTH(VAR f: ANYFILE) : INTEGER FUNCTION
Returns the component length of the file F.

F$CONDITIONAL(VAR f: ANYFILE; flag: BOOLEAN)
Causes the conditional attribute for file F to be reset to FLAG, This attribute defaults to FALSE (READs and WRITEs wait for buffers).

F$EOC(VAR f: ANYFILE) : BOOLEAN FUNCTION
Indicates whether 'end-of-consumption' has been set on the file channel associated with the file F.

F$LASTSUCCESSFUL(VAR f: ANYFILE) : BOOLEAN FUNCTION
Indicates whether the last file channel transfer made by file F was successful or not.

F$STEOC(VAR f: ANYFILE)
Sets 'end-of-consumption' on the file channel associated with file F. When all reading files disconnect, no files are allowed to connect to the file channel until all connected writing files close.

F$STLENGTH(VAR f: ANYFILE; length: INTEGER)
Allows the first text file to connect to a file channel to set the file channel component length (defaults to 80 characters),

6.13.9.9 Exception Handling Routines

ERR$CLASS : INTEGER FUNCTION
Returns the exception condition's class code.

ERR$REASON : INTEGER FUNCTION
Returns the exception condition's reason code.

ERR$RSET
Clears the current process' exception codes.

EXCEPTION(class_code, reason code: INTEGER)
Forces a routine to fail with the specified exception codes.

ONEXCEPTION(exception_hndlr: INTEGER)
Specifies the address of the routine (EXCEPTION_HNDLR) to be invoked when an exception condition occurs. The address of the routine can be found using the LOCATION function.

RE$START
Causes the entire system to be restarted.

6.13.9.10 Critical Transaction Routines

CT$ENTER
Indicates entry into a critical transaction.

CT$EXIT
Indicates exit from a critical transaction.

6.13.9.11 Rx Error and Exception Codes

System Crash Codes

Unable to boot system	= 1
No exception handler	= 2
No interrupt handler	= 3
Illegal interrupt or XOP	= 4
Scheduling queue in error	= 5
ROM/RAM partition error	= 6
Process list is in error	= 7
Invalid heap pointer	= 8

Class Codes

Run-time support error	= 0
User error	= 1
Scheduling error	= 2
Semaphore error	= 3
Interrupt error	= 4
Process management error	= 5
Exception error	= 6
Memory management error	= 7
File error	= 8
Text file error	= 9
Channel error	= 10
I/O decoder error	= 11
Interprocess communication error	= 12

Reason Codes (Run-Time Error)

Stack overflow	= 2
Division by zero	= 4
Floating point error	= 5
Set element out of bounds	= 6
Assert error	= 7
Missing OTHERWISE in CASE	= 8
Array index error	= 9
Pointer equals NIL	= 10
Subrange assignment error	= 11
LONGINT array index error	= 12
LONGINT subrange error	= 13
Halt called	= 20

Reason Codes (Scheduling Error)

Scheduling queue invalid	= 1
Scheduling queue priority error	= 2

Reason Codes (Semaphore Error)

Semaphore invalid	= 1
Semaphore count error	= 2
Semaphore operation error	= 3
Semaphore count overflow	= 4
Semaphore in handler priority error	= 5

Reason Codes (Interrupt Error)

Interrupt invalid	= 1
Interrupt level invalid	= 2
Interrupt semaphore invalid	= 3
Interrupt not handled	= 4
Interrupt incorrect trap vector	= 5
Interrupt handler priority error	= 6

Reason Codes (Exception Error)

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

Reason Codes (Process Management Error)

Not a process	= 1
Aborted	= 2
Not started - invalid priority	= 3
Not started - negative 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 for process frame	= 10

Reason Codes (Memory Management Error)

Heap invalid	= 1
Heap overflow error	= 2
Heap packet error	= 3
Invalid packet error	= 4

Reason Codes (File Error)

File is not open for reading	= 1
File is not open for writing	= 2
Sequential read past end-of-file	= 3
Open error	= 4
Read error	= 5
Write error	= 6
No memory for file descriptor	= 7
No memory for pathname	= 8
File not closed	= 9
Invalid parameter passed to F$STLENGTH	= 10
Not a text file	= 11

Reason Codes (Text File Error)

Text conversion - parameter out of range	= 1
Text conversion - field width too large	= 2
Text conversion - incomplete data	= 3
Text conversion - invalid character in text field	= 4
Text conversion - value too large	= 5
Text read past end of file	= 6
Text field exceeds record size	= 7

Reason Codes (Channel Error)

No memory for buffers	= 1
No memory for semaphores	= 2
No memory for channels	= 3

Reason Codes (I/O Decoder Error)

Empty file identifier list	= 1
File identifier not found	= 2
File identifier not released	= 3

Reason Codes (Interprocess Communication Error)

No heap for pathname record	= 1
No heap for name field	= 2
No heap for file variable record	= 3
No heap for port variables	= 4

6.13.10 Backus-Naur Form (BNF) Syntax Definitions

::=	"Is defined to be'
< >	For enclosing non-terminal symbols (ie entities defined by a production rule)
[ ]	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

6.13.10.1 Compiler Options

<option control comment>::= "{" $(option list> "}"

<option list>::= <option> { , <option> }

<option>::= [NO] <option identifier) |
[RESUME] <option identifier)

where <option identifier) is one of the following:

COL72 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
Enables run-time checks for array bounds,

CKPTR Default=FALSE
Enables run-time checks for pointers equal to NIL,

CKSET Default=FALSE
Enables run-time checks for set element expressions.

CKSTJB Default=FALSE
Enables run-time checks for subrange assignments in bounds.

DEBUG Default=FALSE
Statement numbers are incorporated into the code for use by

LIST Default=TRUE
Enables printing of source listing, error lines are always listed,

MAP Default=FALSE
Prints a map of the routine's variables and common areas after listing the routine.

NULLRODY Default=FALSE
No code is to be generated for the empty system body.

PAGE **Default=FALSE**
Continues printing at the top of a new page.

STATMAP Default=FALSE
A map of displacements for each statement in the object module is to be generated by the code generator.

6.13.10.2 Concurrent Characteristics

These may only appear immediately following the initial BEGIN of a system, program or process declaration.

<concurrent characteristics>::=
"{" # <concurrent characteristic list> "}"

<concurrent characteristic list>::=
<concurrent character) { ; <concurrent character) }

<concurrent character>::=
<concurrent character keyword) =(parameter identifier) |
<concurrent character keyword) = <integer constant)

<concurrent character keyword>::= **HEAPSIZE** | PRIORITY | STACKSIZE

6.13.10.3 System Declaration

For a single program with no processes the syntax is:

<system>::= PROGRAM <identifier> ;(program block> .

The general syntax for a system is:

<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>::= <empty> |
LABEL(statement label> { , <statement label> } ;

<empty>::=

(statement label>::= <digit> { <digit> }

<constant declaration part>::= <empty> |
CONST(constant declaration) { ;(constant declaration) } ;

```
<constant declaration>::= <identifier> = <constant> ; |
     <identifier> = <integer constant expression) ;

<type declaration part>::= <empty> |
     TYPE <type declaration) { <type declaration) }

<type declaration>::= <identifier> = <type> ;

<variable declaration part>::= <empty> |
     VAR <variable declaration) {(variable declaration) }

<variable declaration>::= <identifier list> : <type> ;

(identifier list>::= <identifier> ( , <identifier> }

<common declaration part>::= <empty> |
     COMMON(variable declaration) ((variable declaration) }

<access declaration part>::= ACCESS <identifier list) ; | <empty>

<system routines>::= ( <system routine) }

(system routine>::= <program declaration) |
                    <procedure declaration2 |
                    <function declaration)

<program declaration>::= <program header> <program block> ; |
                         <program header> FORWARD ; |
                         <program header) EXTERNAL [ PASCAL ] ;

<program header>::=
         PROGRAM <identififier> [ <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)
```

6.13.10.4 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 type> |
                     <pointer type> | <file type> | <set type>

<unpacked structure type>::= <array type> | <record type>

<array type>::=
     ARRAY "[" <index type> { , <index type> } "]" OF <type>

<index type>::= BOOLEAN | CHAR | <scalar type> | <identifier> |
                <subrange type>

<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>
```

6.13.10.5 Statement Syntax

```
<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) ]

<procedure identifier>::= <identifier>

<actual parameter list>::=
     ( <actual parameter) { , <actual parameter) } )

<actual parameter): := <expression> | <variable>

<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)

<compound statement>::= BEGIN <statement> { ; <statement> } END

<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)
```

6.13.10.6 Expression Syntax

```
 <expression>::= <boolean term> | <expression> OR <boolean term>

 <boolean term>::= <boolean factor) |
                   <boolean term> AND <boolean factor)

 <boolean factor>::= [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>
```

6.13.10.7 Variable Syntax

```
<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)
```

6.13.10.8 Constant Expression Syntax

```
<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>
```

6.13.10.9 Lanugauge Element Syntax

```
<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 | EXTERNAL |
                    FALSE | FILE | FOR | FORWARD | FUNCTION | GOTO |
                    IF | IN | INPUT | INTEGER | LABEL | LONGINT |
                    MOD | NIL | NOT | OF | OR | OTHERWISE | OUTPUT |
                    PACKED | PASCAL | PROCEDURE | PROCESS | PROGRAM |
                    RANDOM | REAL | RECORD | 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 ]

<digits>::= <digit> { <digit> }

<real constant>::= <digits> . <digits> |
                   <digits> . <digits> E <scale factor) |
                   <digits> E <scale factor)

<scale factor>::= [ <sign> ] <digits>

<sign>::= + | -
```

6.14 BIBLIOGRAPHY

[1] Kathleen Jensen and Niklaus Wirth
Pascal User Manual and Report
Springer-Verlag

[2] Niklaus Wirth Algorithms + Data Structures = Programs
Prentice-Hall

TI Publications

Microprocessor Pascal System User's Manual (MP351)
Microprocessor Pascal Executive User's Manual (MP385)
Realtime Executive User's Manual (MP373)
Component Software Handbook (MP918)
Microprocessor Pascai Euroboard Application Report (MP814)

CHAPTER 7

POWER BASIC

7.1 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 sf the Algol based languages (Algol 68, Pascal, etc), the BASIC language is easy to learn and supports a variety of useful features.

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. This is not directly executable. 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. At this time, 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) simply 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 are:

- 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 and link edit steps 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 whole 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 the 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,

7.2 POWER BASIC

Power BASIC is a family of software products designed for the industrial user. It provides all of the facilities of RASIC plus specially designed features to support real-time industrial control applications. At the time of writing, three members of the Power BASIC family are available: Evaluation Power BASIC, Development Power BASIC, and Configurable Power BASIC. New members may be added to satisfy particular requirements.

Power BASIC is designed to run on the TM990 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 743 KSR 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.

7.2.1 Evaluation Power RASIC

Evaluation Power BASIC is a four-EPROM package that resides on either a TM990/100M or a /101M CPU module. Additional RAM in the form of TM990/201 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 (RASE statement, CRB and CRF functions), It also allows the user to load a program from (LOAD command) and save a program to (SAVE command) digital cassettes.

Evaluation Power BASIC is intended for users to try out the features of Power BASIC, It was not designed for serious development work, apart from experimental applications.

Used with the /101M CPU board, Evaluation Power BASIC supports the following execution environments:

- o Single-user, single-partition
- o Single-user, two-partition
- o Two-user, two-partition

The appropriate environment is selected via the 5-pole DIP on the /101M CPU board. Section 2.9 of the TM990 Power RASIC 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). This enables Evaluation Power BASIC to be used to control two separate tasks, the execution of each being synchronised 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, say).

In the following code, partition #1 gathers input from the terminal and passes it across to partition #2 via the COM array. COM(0) is used to synchronise 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.

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 'initialise'  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
```

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.

7.2.2 Development Power BASIC

Development Power BASIC is a six-EPROM package that resides on either a TM990/100M or a /101M CPU board plus either a TM990/302 Software Development Board or a TM990/201 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 EPROM, Development Power BASIC also allows the user to write interrupt service routines in the Power BASIC language and to associate each of these routines to a particular interrupt level (using the TRAP, IMASR, 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, when the TM990/302 Software Development Board

is configured into the system, there is a two-EPROM Enhancement Software Package that can be used to extend the capabilities provided by Development Power BASIC. This package allows the user to LOAD and SAVE a Power BASIC program on low cost audio cassettes. The PROgram command gives the ability to 'burn' a Power BASIC application into TMS2716 EPROMs. The enhancement package also provides decimal print formatting and complete error message reporting.

7.2.3 Configurable Power BASIC

Configurable Power RASIC 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). It allows the user to generate an application target system of minimum size by deleting the Power RASIC editor along with any parts of the interpreter that are not used.

Configurable Power BASIC consists of 3 parts: a host interpreter, a configurator and an object library. This library is a collection of routines, each of which implements a specific Power BASIC statement or function.

The configurator determines what Power BASIC features are required by the user's application program and creates the following files:

- A link editor control file containing an INCLUDE statement for each object routine (from the object library) that is required by the application program. If the application program contains any CALL statements, the user supplied assembly language routines are also INCLUDEd.

- A "root" module containing the Power BASIC application program in its encoded internal form.

- A "map" file containing a summary of all Power RASIC statements and functions used by the application. Any errors encountered are immediately reported to the user and are also recorded in this file.

The TX990 Link Editor (TXSLNK) takes the link editor control file and uses the object library and the "root" module to produce a customised Power BASIC run-time module. This run-time module is then programmed into TMS2716 EPROMs. Inserting these EPROMs into a CPU board (like the TMS990/101M board), starting at address 0, and toggling the reset switch causes the Power RASIC application program to be activated.

The internal code used in the "root" module is compatible with the internal code used by Development Power BASIC. This means that the "root" module can be programmed into TMS2716 EPROMs on its own and these can then be inserted into a board system containing Development Power BASIC. When the EPROMs are inserted at address >3000 the application program is automatically executed whenever the reset switch is toggled, However, if the EPROMs are inserted elsewhere then the following command must be issued to execute the program

LOAD <address>

where <address> is the start address of the first pair of the "root" module's EPROMs,

Note: Due to features that have been added (eg the memory word, MWD, function) to the Configurable Power BASIC host interpreter and to Development Power BASIC there are differences between releases, A "root" module generated with Configurable Power BASIC C.1.4 should use Development Power BASIC D.1.6; Configurable Power BASIC C.1.6 should use Development Power BASIC D.1.10.

The host interpreter provides all the features of Development Power BASIC. and the 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 1/0 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 ROPEN 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 RDEFS (define sequential file) or the RDEFR (define random file) statements, and deleted by the BDEL statement, The COPY statement allows the user to copy a file to another file or to a device: this can be used to backup a file, to concatenate several files together, or to print a file. Reading from and writing to files or devices can be performed by the "BINARY" statement:

BINARY <exp>

where <exp> specifies the required 1/0 operation. BINARY 1 lets the user specify how many bytes are to be involved in subsequent I/O operations to a particular file or device (the default is 6 bytes), BINARY 2 is a write operation, BINARY 3 is a read operation. BINARY 4 allows the user to

access a particular byte within a specified record (this is for relative record, random, files only).

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 VDU) by either supplying the 'x' and 'y' co-ordinates or using the following 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' on the VDU 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";@(4,11);"INPUT NAME"
```

Note: The column values range from 0 to 79 (80 characters). The row values range from 0 to 23 (24 lines) for the 911. and from 0 to 11 (12 lines) for the 913.

Other features of Configurable Power BASIC include:

BYE	Terminate a Configurable Power BASIC session.
DIGITS	Specify the number of digits to be printed in free format.
EQUATE	Specify an alternate name for a variable or an array element.
NUMBER	Set the initial and increment values for the automatic line numbering facility.
PURGE	Delete the specified lines.
SOURCE	Show how much memory the program will occupy when saved.
SPOOL	Specify the secondary output device controlled by the UNIT statement.
STACK	Interrogate the GOSUB stack.

The following diagram (Figure 7-1) illustrates how Configurable Power BASIC minimises an application program's memory requirements.

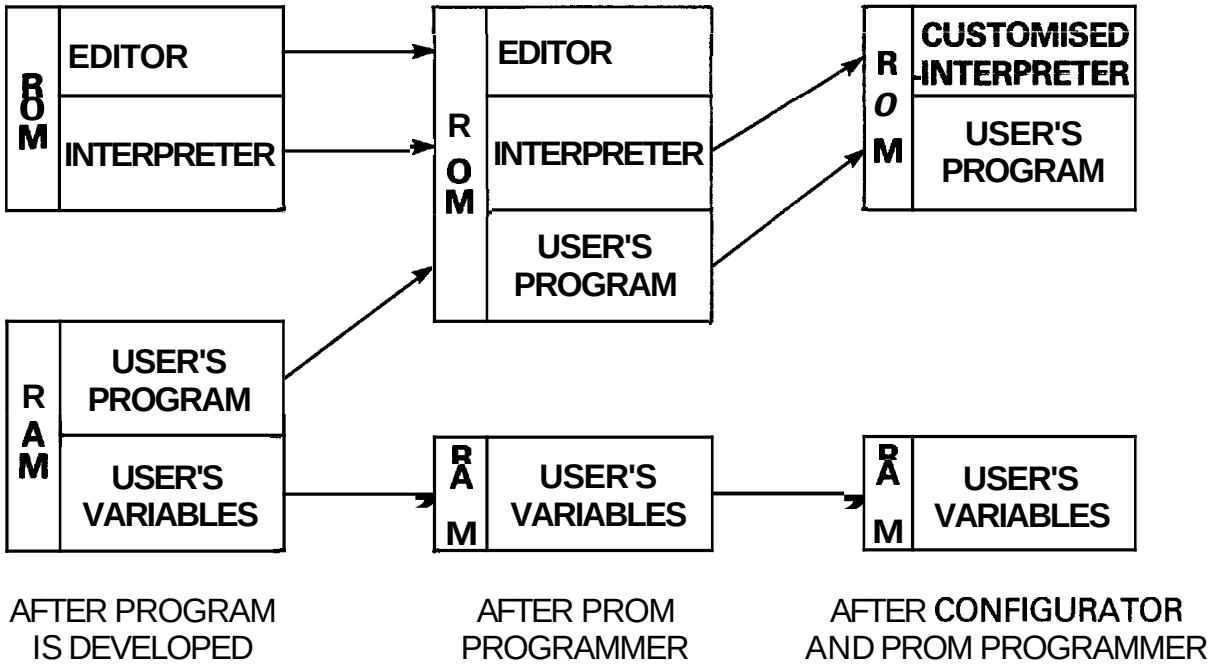


Figure 7-1 Code Minimisation

7.3 BASIC LANGUAGE OVERVIEW

Power BASIC is an uncomplicated, easy to learn language that is based upon a few simple concepts. A Power BASIC program consists of a series of numbered statement lings that are executed in ascending numerical order. A line normally contains one Power BASIC statement, although the statement separator 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:

```
A2 = 5 + 7
```

When the above line is executed, the variable 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 in one of the following:

- o on the left-hand side of an assignment statement
- o in an INPUT statement
- o in a READ statement

Variable names are restricted to one to three letters or a combination of a letter and a number in the range 0 to 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.

Each statement in a Power BASIC program has a line number:

```
10     A = 5 * B
20     PRINT A
etc
```

The line numbers specify the order in which the program statements are to be executed (ie its sequence).

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 (see section 7.6.1.2); it must be combined with the GOTO statement if the alternatives will not fit on one line, The FOR..NEXT statement implements iteration (see section 7.6.1.4). In general, programming constructs (see Section 4.5) have to be built by the programmer using IFs, FORs and GOTOs.

Subroutines or procedures (see section 7.6.2) 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 Power BASIC is its simplicity. Programs can be entered and executed easily even by users who are not skilled programmers, Power 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 Power BASIC is particularly simple and easy to use; even novices can learn to develop a Power BASIC program in a matter of hours. Power 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, Power 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.

7.4 POWER BASIC OPERATION

7.4.1 Operating Modes

Power BASIC has two operating modes: Keyboard mode and Execution mode.

Keyboard Mode is automatically entered when Power BASIC is initialised. In this mode, 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.

Execution Mode is entered by issuing either a RUN, a CONT or a GOTO statement. This causes 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 (these are referred to as commands). A full list of these commands is given in section 7.8.5.

7.4.2 Editing Source Statements

The simplest way to modify (or edit) a line is to re-type the whole line. However, Power BASIC also supports a simple editor that allows the user to easily modify previously entered source statements. The available edit commands are:

ESC	Cancel input line
RUBOUT	Backspace and remove character
CR or LF	Enter the edited line
ctrl H	Backspace the cursor one character
ctrl F	Forward space the cursor one character
<ln> ctrl E	Display the line <ln> for editing

An attempt to forward space past the last character entered, or to backspace beyond the first character in the line will only cause the bell on the terminal to be rung,

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 (CR) or linefeed (LF) key is pressed, all characters displayed are entered, regardless of the position of the cursor,

Entering just 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 terminal. However, if the program is being loaded from cassette or floppy diskette (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 (it can not be stored correctly) and the load operation will continue,

7.4.3 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 initialises the line number to 10, Terminating a line with a carriage return disables this facility.

7.4.4 System Initialisation

Toggling the reset switch on the /100M or /101M CPU board 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. If a mismatch occurs between addresses >FBFE and >F000 then Power BASIC assumes that a /100M CPU board is being used; any memory that was found between these addresses is ignored and autosizing continues from address >EFFE. (A fully populated /100M microcomputer board only holds 1K of RAM. This is addressed from >FC00 to >FFFF.)

The Power BASIC interpreter then performs the auto-baud sequence. This initialises the serial 1/0 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). All output is then directed to Port A on the microcomputer board.

When all Power BASIC pointers have been initialised, 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.

7.5 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 (4 bytes for Evaluation

Power BASIC, 6 bytes in Development Power BASIC),

7.5.1 Variable Names

A variable name is either an alphabetic character followed by a number in the range 0 to 127 (eg Z100) or an alphabetic string up to three characters long (eg A, ST, and LST). The variable name can not be identical to a Power BASIC keyword, nor can it form the beginning of a keyword. The following variable names are not valid:

LIS	Begining of LIST (a Power BASIC command)
MEM	A Power BASIC function
TOT	First 2 letters are the Power BASIC keyword TO
12B	First character is not alphabetic
ABCD	More than 3 characters
I130	Number greater than 127
A.B	' ' not allowed in variable names

Note: There is a maximum of 140 different variable names in any one Power BASIC program.

7.5.2 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 following statement can be used:

```
TST=100
```

A value can be assigned to a variable by either 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)
```

The above statement assumes that the variables PI and NUM have already been declared (assigned a value). An attempt to use a variable that has not been declared will result in error 40 (UNDEFINED VARIABLE).

7.5.3 Numeric Representation

If a number can be represented in a 16-bit twos complement form, it is stored in integer format, otherwise it will be stored in floating point format.

7.5.3.1 Integer Variables

An integer variable can store a value in the range -32768 to +32767.

7.5.3.2 Floating Point Variables

Floating point format allows a real number in the range 10E-75 to 10E+74 to be stored. ('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.

7.5.4 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.

A variable is specified as containing a character string by preceeding the variable name with a dollar sign ($). In this form, a variable should be used to store up to 3 characters for Evaluation Power BASIC, or 5 characters for Development Power BASIC. The last byte is used to terminate the string and contains the null character (zero).

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 only interpreted when the string is being input from a terminal, read from a DATA statement, or when the string is being printed. Note: Attempting to use the character sequence '<>' in a string via an INPUT, READ or PRINT statement will cause problems. If these characters are required then the sequence '<3C><3E>' should be used.

7.5.5 Array Variables

An array is a number of variables (stored consecutively in memory) that is referenced by a single variable name, Individual variables (or array elements) are accessed by following the variable name with a number that identifies the position of the variable within the array, The number (this is known as an array subscript) is enclosed in parentheses or square brackets (internally the parentheses are converted into and stored as square brackets),

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 Power 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,

Note: The variable A and the array variable A(0) refer to two completely different variables,

7.6 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 (GOSUB 1000), that change the execution flow of the system, A full list of Power BASIC statements is given in section 7.8.6.

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:

{ iine number } statement [:: statement] { ! comment }

where { } indicates optional items
[ ] indicates that the item is repeated as many times as required - 0,1,....

Exceptions:

- o A NEXT statement should be the first statement on a line, otherwise it may not be located to terminate its corresponding FOR loop,
- o A DATA statement should be the only statement on a line,
- o A REM statement takes the remainder of a line as comment.

7.6.1 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,

7.6.1.1 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 control 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 Power BASIC (the WHILE, DO FOREVER and REPEAT UNTIL loops; more about these later).

7.6.1.2 IF THEN Statement

Often 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>
```

The Power BASIC statements in <sequence> are only executed if <condition> proves to be true. 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.

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 <stmt1>::IF <cond2> THEN <stmt2>
```

Is not the same as:

```
100 IF <cond1> THEN <stmt1>
101 IF <cond2> THEN <stmt2>
```

In the first case, <stmt2> is only executed if both <cond1> and <cond2> are true. In the second case, <stmt2> is executed if <cond2> is true, regardless of <cond1>.

The number of statements in <sequence> 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
```

If <cond1> is false, NOT(<cond1>) is true and program control is passed to the REM statement following the sequence. The REM statement is a remark (comment), and is ignored by the interpreter.

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 ioop 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 recognised Development Power BASIC boolean

primitive that returns a value of TRUE if its argument evaluates to FALSE; otherwise it returns a value of FALSE, Although it is not supported by Evaluation Power BASIC it is simple to effect the NOT function. All conditions can be written in the form:

<exp1><relop><exp2>

using this, the NOT function is implemented by taking the complement of the relational operator (<relop>):

<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, The statement:

IF <expression> THEN <statement(s)>

is shorthand for

IF <expression> <>0 THEN <statement(s)>

7.6.1.3 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 statement separator (::).

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  <cond1>  THEN  <seq1>
110  ELSE  <seq2>
120  REM  end IF THEN ELSE
```

In the above, <seq1> is only executed if <cond1> is true; if <cond1> 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.

7.6.1.4 FOR NEXT Statement

A simple loop construct (perform a sequence of statements a known number of times) can be implemented as follows.

```
 90 num=int
100 IF num>lst THEN  GOTO 350  ! IF NOT(num<=lst)
    .
    , Sequence to be performed
    . while num<=lst
    .
    Num=num+1                  ! increment loop count
    GOTO  100
350 REM end iterative loop
```

where INT is the initial value, LST is the final value and NUM is the loop counter. ! is another form of comment; anything after the ! is ignored.

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<1st THEN GOTO 350  ! IF NOT(num>=1st)
    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 is the required increment/decrement.

As this type of loop is frequently used, Power 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 <increment>
.
, Sequence to be performed
.
NEXT <var>
```

The <start>, <final> and <increment> values can be any valid numeric expression. If the value of <increment> is one, it and 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:

```
(increment)*(start value) > (increment)*(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 value of <start> and the sequence of statements between the FOR and NEXT statements are executed. When the NEXT statement is encountered the FOR variable is updated by the value of <increment>. Control is passed back to the FOR statement and while the condition:

```
(increment)*(FOR variable) <= (increment)*(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 and 10 for Development Power BASIC,

```
100 FOR K=1 TO 100
    .
    .
200 FOR J=9 TO 0 STEP -1
    .
    .
275 NEXT J
    .
    .
490 NEXT K
```

Correct nesting

No overlapping is allowed; inner loops must be closed before closing outer loops. Nested FOR NEXT loops must have different FOR variables; they cannot share control variables, Otherwise, loop boundaries will not be clearly defined,

```
100 FOR K=1 TO 100
    .
    .
200 FOR K=90 TO 160
    .
    .
387 NEXT K
    .
480 NEXT K
```

Incorrect nesting

Control variable shared; unclear loop boundaries

```
100 FOR K=1 TO 100 STEP 3
    .
    .
200 FOR J=9 TO 0
    .
    .
300 NEXT K
    .
400 NEXT J
```

Incorrect nesting

Overlapping loop boundaries

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 1=0 TO 10 :: sequence :: NEXT I
```

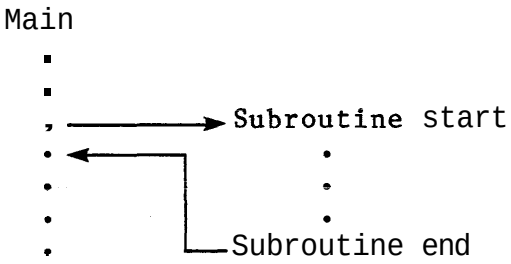
This effectively disables the ESCape key on the terminal while the loop is being executed (until the loop has completed it is not possible to interrupt program execution and return Power BASIC to keyboard mode). This is because Power BASIC only scans the keyboard looking for an 'escape'

character when a statement line has been completed. Also, if the initial check indicates that the loop is not to be executed, error 31 (FOR W/O NEXT) will result as the NEXT statement will not be found.

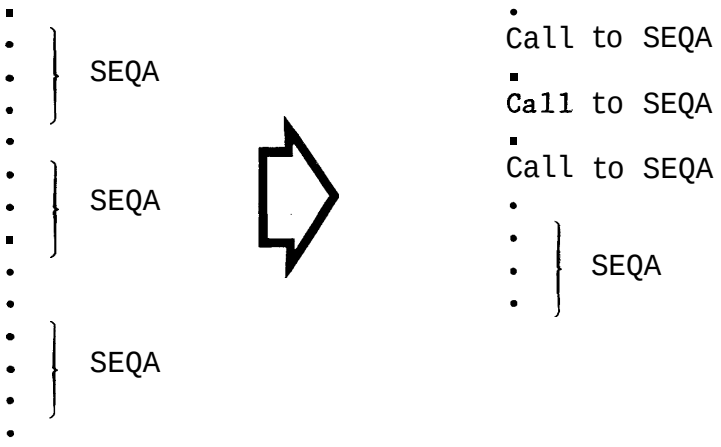
7.6.2 Subroutines

As previously stated, statements are normally executed in a straightforward sequential manner, A subroutine represents a method of executing a number of statements outside the normal sequence.

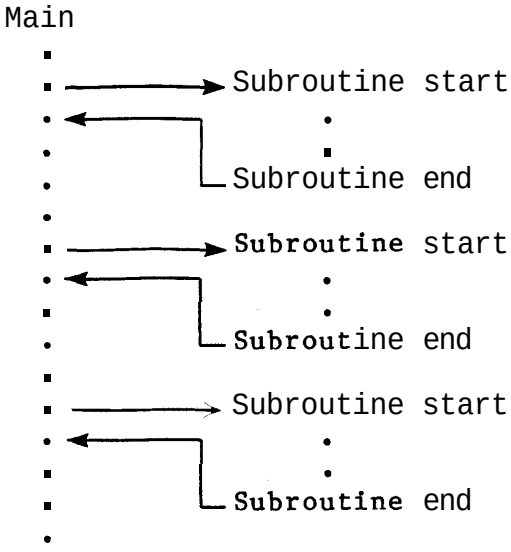
Pictorally, subroutine execution is:



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. For example:



Program execution would become:

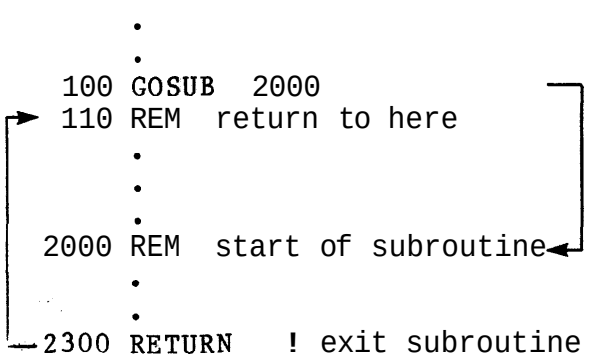


If the subroutine is large, or it is called from a number of different places, there can be a considerable saving realised in program storage against a small overhead in calling and in returning from the subroutine.

A Power BASIC subroutine is simply a sequence of statements that is entered via 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 these statements are:

```
GOSUB <ln>
RETURN
```

A subroutine is identified by its starting line number (<ln>), rather than by a name. For example:



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. If the

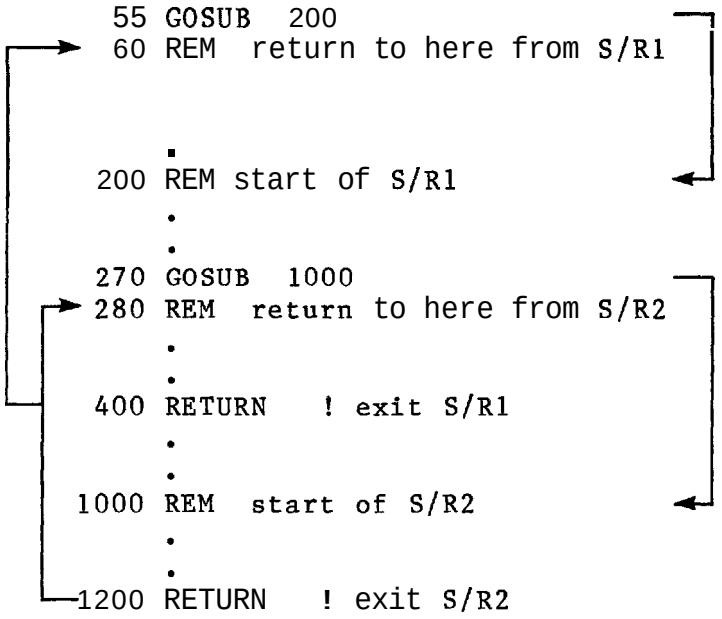
GOSUB 2000 had been followed by (eg) '::FLG=9' then the address of this statement would have been pushed onto the GOSUB stack.

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, allowing control to be passed back to line 110.

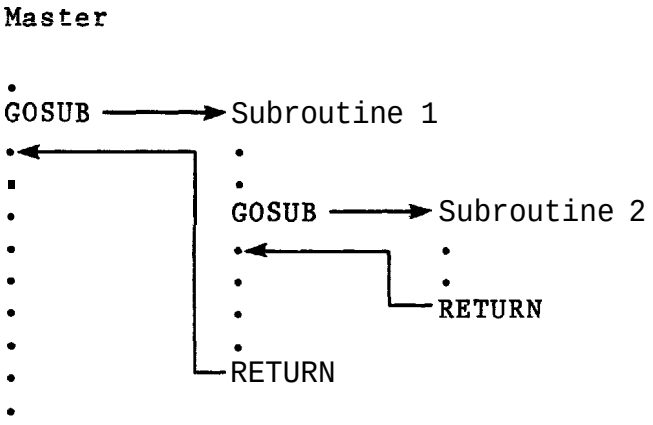
If a subroutine is exited by any way other than a RETURN statement, program flow can become unpredictable. Power BASIC performs no check that a subroutine has been exited (via a RETURN statement). Executing a RETURN statement when a subroutine has not been invoked will result in error 12 (STACK UNDERFLOW).

Subroutine calls may be nested (a subroutine may call another subroutine) up to a maximum of 10 levels for Evaluation Power BASIC and 20 levels for Development Power BASIC (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).

A program with nested subroutine calls is shown below:



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, and so the programmer must make his own checks that variables are not accessed incorrectly (inadvertently modified by a subroutine). If a subroutine can overwrite critical data, it is necessary to use temporary variables for storage of this data and the programmer must ensure that the subroutine only accesses this data through the temporary variables.

7.6.3 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 statement 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.

7.6.4 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. The address of the statement line following the one in which the error occurred is preserved on the GOSUB stack.

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.

```
  PO ERROR 1000
     .
     .
1000 REM  error handling routine
1010 IF 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), the message "ERROR=" and the error code are output 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 and a return is made to the line immediately following the read statement that caused the error. Obviously this "error routine" is not particularly useful (as the contents of the "read variables'' can not be relied upon), however it does serve to illustrate the use of the ERROR statement,

If the sequence of read operations is of the form:

```
 100 READ ....
     .
 200 READ ....
     .
 300 READ ...,
     .
```

Then replacing line 1030 by:

```
1030 POP:: ON SYS(2)/100 THEN GOTO 100,200,300,...
```

allows the "error routine" to be more useful. The POP

statement simply removes the top address from the GOSUB stack (in this case, the address of the line following the READ statement that caused the error).

Once an error has been trapped using this statement, no future errors will be trapped until another ERROR statement is executed.

Note: Use of the ERROR statement suppresses the automatic printing of error code/message.

7.6.5 CRU Operations

The 9900 supplies a bit-oriented method of 1/0 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 8.9.

7.6.5.1 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> (known as the software base address) must be twice that of the hardware CRU base address desired. For example; to access a device that has a CRU base address of 32, <exp> must evaluate to 64.

7.6.5.2 CRB Function

Single-bit 1/0 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 one if the specified bit is set, and zero if it is not set.

Example: Execute the sequence <seq1> if the 15th bit from the base address is a '1'.

```
IF CRB(15) THEN <seq1>
```

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.

Example: Set the 100th bit from the base address to '1'.

```
CRB(100)=200
```

7.6.5.3 CRF Function

The specified number of bits are written to or read from the CRU starting at the address set by the BASE statement. The number of bits to be transferred must be in the range 0 to 15. If the number is zero, all 16 bits are transferred.

Example: Transfer the 16 bit value minus one (hex >FFFF) to the CRU address specified by the RASE statement.

```
CRF(0)=-1
```

Example: Read an 8 bit value from the CRU base address and store the result in VAL.

```
VAL=CRF(8)
```

VAL will be in integer format with the value occupying the least significant byte of the integer word.

7.6.6 Memory Operations

The Power BASIC functions MWD and MEM allow the user to read or write to an individual word or byte in memory. However, care must be exercised when using these functions to ensure that no Power BASIC system variables are inadvertently corrupted.

These functions can also be used to directly interface to memory mapped 1/0 devices.

7.6.6.1 MEM Function

This function allows the user to read from or write to the specified memory byte location.

Example: Output the character 'A' to the device data register located at memory address >AE00.

```
MEM(0AE00H)=65          !DEC 65=ASCII 'A'
or  MEM(0AE00H)=ASC('A')
```

ASC returns the decimal ASCII code of the character

argument.

Example: Pick up the character in the device data register located at memory address >B000.

```
$CIN=%MEM(0B000H)%0
```

The single character string is terminated by the '%0'.

7.6.6.2 MWD Function

This function allows the user to read from or write to the specified memory word location. This function is particularly useful for loading small assembly language routines into memory. (The area of memory used must be outside the Power BASIC environment.)

Example: Load the assembly language program into memory starting from address >7000.

```
MWD(07000H)=.......     !Load 1st instruction
.
MWD(07XXXH)=045BH       !Load RT instruction
```

For large routines the above approach is not really suitable, An easier method is:

```
 100  DATA start address, .....
       .
 600  DATA ....., 045BH, term
      .
1000  READ str            !Get start address
1010  READ opc            !Get next instruction
1020  IF opc = trm THEN STOP
1030  MWD(str)=opc :: str=str+2 :: GOTO 1010
```

The first item to be read from the DATA statement is the actual address in memory where the program is to be loaded. The only other addition to the routine is some way of indicating when the end of the routine has been reached. In the above code, this is indicated by TRM (this is a unique value that does not appear anywhere within the routine to be loaded). It could, just as easily, have been indicated by including the length of the routine as the second item in the DATA statment at line 100. If this had been the case then a simple FOR NEXT loop could have been used.

Example: Check memory address >6000 to see if a particular EPROM set has been installed and if so, execute the assembly language routine located there. (This EPROM set is identified by the contents of its first word, it should be >1234.)

```
IF MWD(06000H)=01234H THEN CALL "routine",06002H,....
```

7.6.7 Assembly Language Routines

Although Power BASIC is one of the fastest BASIC interpreters commercially available, there are some situations where it may be advantageous, or even necessary, to write a routine in assembly language, Perhaps a complex operation has already been written in assembly language and it would certainly be easier, and simpler, to use this without having to recode it in the Power BASIC language. Or perhaps, to look after a high-speed device where timing is critical and a response is required in a matter of a few tens of microseconds. (At 3MHz and no memory wait states, the TMS9900 microprocessor executes an interrupt context switch in 7.3us; a MOV instruction takes between 4.7us and 10us depending upon the addressing mode used.)

With Development, and Configurable, Power BASIC, this sort of situation is provided for by the CALL statement. It allows the programmer to invoke an assembly language routine from within a Power BASIC program, The syntax for this statement is:

CALL <name>,<address>,<var1>,<var2>,<var3>,<var4>

where the string <name> is the assembly language routine's IDT, <address> is the address of the routine in memory. <var1>, <var2>, <var3> and <var4> are the routine's parameters (these parameters are optional and can be omitted, along with their preceeding commas, if they are not required).

When running under either Development Power BASIC or the Configurable Power BASIC host interpreter, the <name> operand is not checked (but it must be present) and the <address> operand is used as the routine's entry point. However, a customised Power BASIC target interpreter (derived from Configurable Power BASIC) uses the <name> operand to generate the routine's entry point and the <address> operand is not checked (but it must be present).

The assembly language routine is entered by a BL instruction, which stores the return address in register 11, A return to the Power BASIC interpreter is made by an RT pseudo-instruction (this is equivalent to a B *R11 instruction),

The parameters are passed across to the assembly language routine in registers 4, 5, 6 and 7 of the Power BASIC workspace. When a Power BASIC variable is a parameter, its contents are converted into a 16 bit twos complement integer value before being loaded into the appropriate register. Enclosing the variable name in parentheses causes the

address of the variable to be passed over. (The formats employed by the Power BASIC interpreter are given in section 7.7.1.) The routine can modify these four registers as necessary. If, however, more than four registers are required, the assembly language routine should be provided with its own workspace as modifying any of the other registers could cause the interpreter's execution to become unpredictable.

Example: Invoke the assembly language routine (IDT of TEST) located at memory address >8446, with parameters 10 and the address of the Power BASIC variable INC.

```
CALL "TEST",08446H,10,(INC)
```

On entry to the routine, R4 will contain 10 and R5 will contain the address of INC.

With the Configurable Power BASIC host interpreter, the user must first load the object program from either cassette or a floppy disk file. Details on how to do this are given in the Assembly Language Support for Power BASIC Application Report (MP719), available from TI. (A small assembly language routine can be 'loaded' using the mechanism described in section 7.6.6.2.)

7.6.8 Interrupts

Development Power BASIC allows the user to perform interrupt handling via a Power BASIC subroutine. This is achieved using the Power BASIC interrrupt statements IMASK, TRAP and IRTN.

With the TM990/100M and /101M microcomputer modules, all interrupt lines are connected to the onboard TMS9901 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 (ie >100) when used in the BASE statement to set the base address of the 9901. For an interrupt to be recognised 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 TMS9901 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 a '1') to the mask bit then the 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.

An 'open/close window' mechanism is used to recognise interrupts. This mechanism was chosen because it guarantees the integrity of the Power BASIC environment. Interrupts are only recognised after a Power BASIC statement has been executed, As the Power BASIC interpreter is not re-entrant (see Sections 8.13.7 to 8.13.9 inclusive), this is necessary to ensure that temporary/partial results and even Power BASIC system variables are not corrupted by executing a Power BASIC interrupt handler while the interpreter is in the middle of a statement.

After a statement has been executed, the interpreter sets the status register's interrupt mask to the 'open' value (this allows the processor to take the highest priority pending interrupt). If there is a pending interrupt, its priority level is stored in an internal 'flag register'. The interrupt mask is then reset to the 'close' value. If the 'flag register' is unchanged, the next Power BASIC statement is executed. Otherwise the 'open' value and the address of the next instruction to be executed are stacked. The 'open' value is reset to the incoming interrupt level minus one (this disables interrupts of an equal or lower priority) and the appropriate Power BASIC interrupt routine is then invoked. (On completion of the interrupt routine, both the 'open' value and the address of the next instruction to be executed are restored and the above sequence is then repeated.)

The 'open' and 'close' values are determined during system initialisation. This is performed by scanning the interrupt vectors (starting from interrupt level 15 and working down towards level 3) to find the lowest priority interrupt that is not handled by Power BASIC. Both 'open' and 'close' are set to the value of this interrupt level (if all interrupts are handled by Power BASIC, these two values are set to 3). This allows all enabled interrupts that are handled by

assembly language routines to be taken immediately they are recognised by the processor, no matter what the Power BASIC interpreter is doing. However, this means that all interrupt levels below the 'open' value must be handled by assembly language routines. If, for example, interrupt level 7 is handled by an assembly language routine, the Power BASIC interrupt statements can only be used in conjunction with levels 8 to 15.

Additional information on interrupts is contained in Section 8.10.

7.6.8.1 IMASK Statement

The IMASK statement is used to control the TMS9900 microprocessors's interrupt mask (bits 12 to 15 of the status register).

The 9900 recognises 16 distinct interrupt levels, level 0 is the highest priority interrupt and level 15, the lowest.

With the /100M and the /101M microcomputer modules, interrupt level 0 is reserved for the RESET function and interrupt level 3 for the real-time clock. Apart from these two, all other interrupt levels may be used by external devices. Several devices may even share the same interrupt level (if system considerations require it). If this is the case, the programmer must determine which device caused the interrupt by polling the devices' status registers.

An interrupt can only be recognised 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. If, for example, the interrupt mask is set to 5, then only interrupt levels 0 to 5 will be recognised 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 expression in the range 0 to 15.

Note: Care must be taken when using the IMASK statement as this causes the 'open' and 'close' values to be changed. ('Close' is set to the IMASK value. 'Open' is also set to this value if it is numerically lower than the current 'open' value.)

7.6.8.2 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 sf the first statement sf the interrupt routine.

7.6.8.3 IRTN Statement

The last statement of an interrupt subroutine must be an IRTN. When this statement is executed, the interpreter recognises 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

Before this statement is executed, the device that generates the interrupt signal must be reset. If this is not done then as soon as the IRTN statement has been executed the interrupt subroutine will be immediately re-entered (as the interrupt signal will still be present).

7.7 POWER BASIC STORAGE ALLOCATION

The paragraphs that follow discuss variable storage and the system memory map. This information is not necessary in order to write Power BASIC programs, but may be of interest to users.

7.7.1 Variable Storage

As a variable is allocated the same amount of memory no matter what it contains (4 bytes in Evaluation Power BASIC and 6 bytes in Development Power BASIC), swapping a variable's contents between integer, floating point or character string formats as the context requires presents no problem.

The memory space for variable storage starts in high memory and builds down towards low memory as each new variable is declared. Suppose variable storage starts at memory address >FE00. The first variable used will be allocated space as follows:

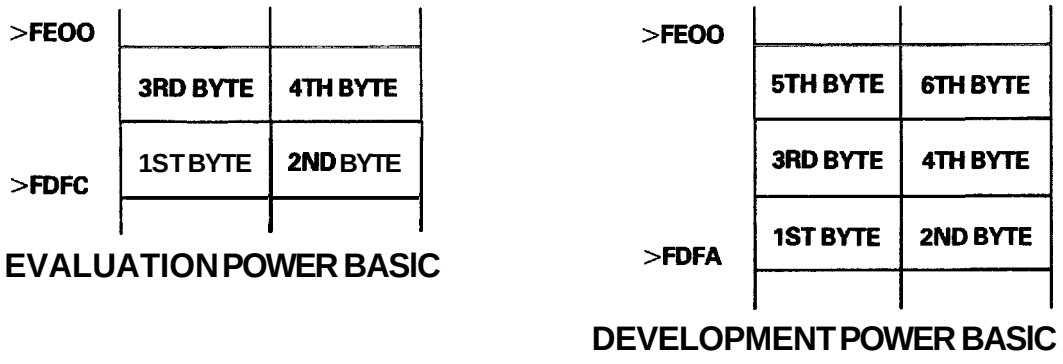


Figure 7–2 First Variable Allocation

The next variable will he allocated space as follows:

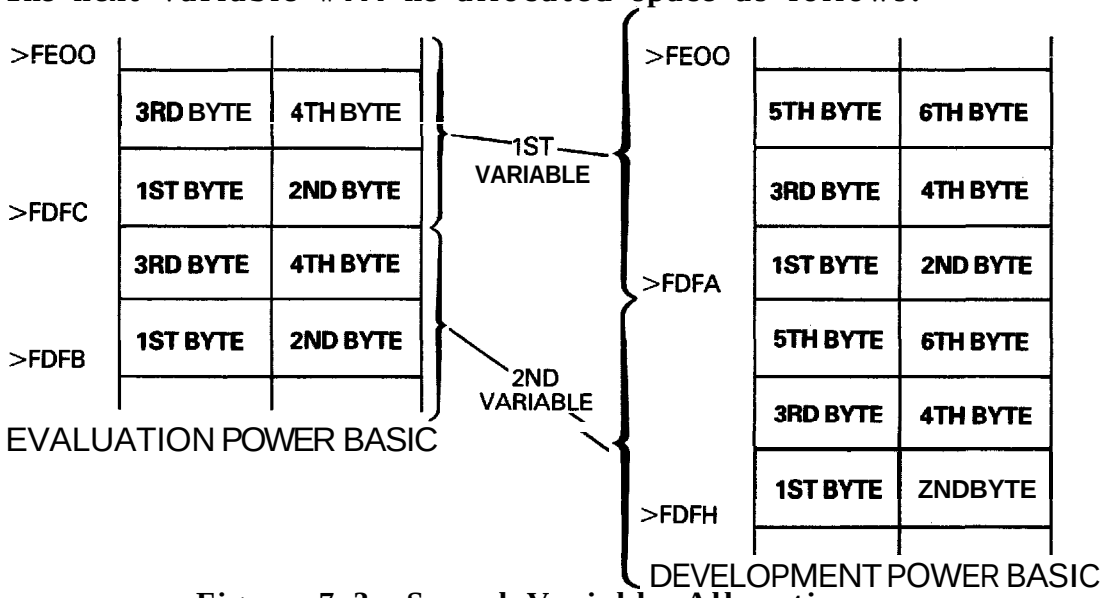


Figure 7–3 Second Variable Allocation

7.7.1.1 Integer Format

Integer numbers are stored as:

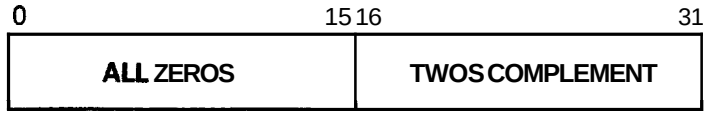


Figure 7–4 Integer Format

The first word (bits 0 to 15) is set to zero indicating an integer number. The second word (bits 16 to 31) contains the twos complement integer value, For Development Power BASIC the third word (bits 32 to 47) also contain zero.

7.7.1.2 Floating Point Format

A floating point number is represented internally as a fraction multiplied by a power of 16 (this power is known as the characteristic) and is stored as:

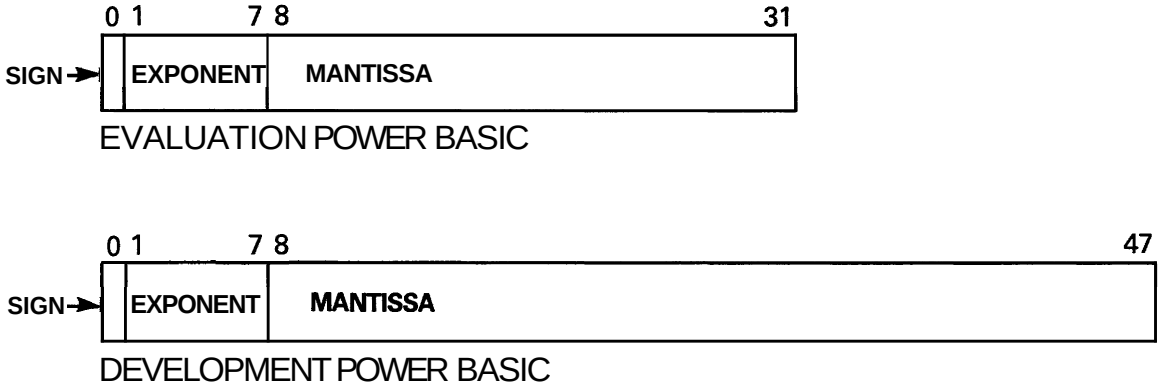


Figure 7-5 Floating Point Format

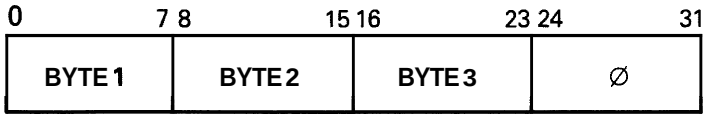
Bit 0 is the sign bit and represents the sign of the floating point number: 0 for positive, 1 for negative. Bits 1 to 7 hold the characteristic coded in Excess 64 notation (the true characteristic plus 64; this gives the characteristic a range of 0 to 127 representing a true exponent range of -64 to +63). The remaining hits (24 for Evaluation Power BASIC and 40 for Development Power BASIC) contain the normalised mantissa (the mantissa is normalised 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 twos complement notation.

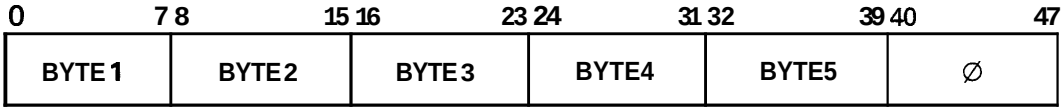
The conversion of a decimal real number into its approximate binary equivalent is described in Sections 8.13.2.3 and 8.13.2.4.

7.7.1.3 Character String Format

A character string is stored as follows:



EVALUATION POWER BASIC



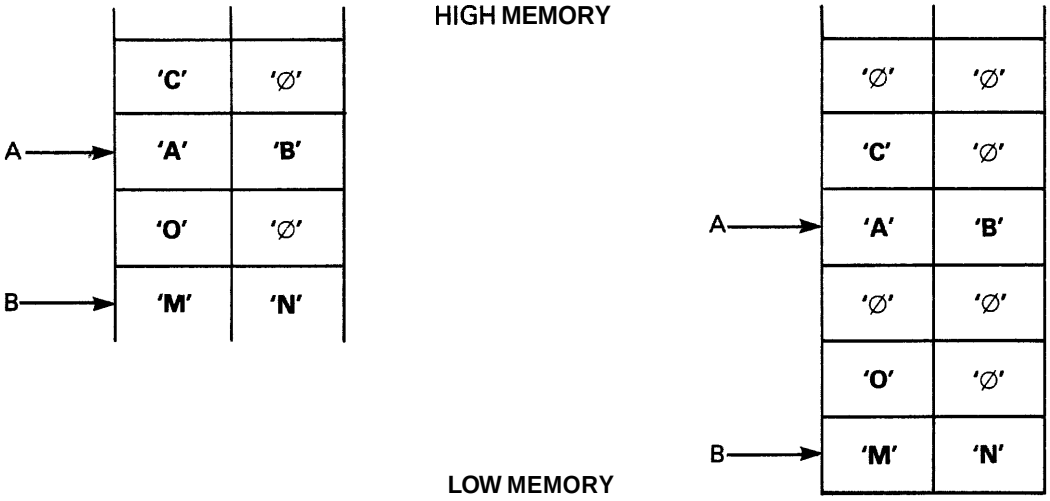
DEVELOPMENT POWER BASIC

Figure 7-6 Character String Format

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

Figure 7-7 Character String Storage Example

When a character string is too long to be held in a variable, an array should be used,

7.7.1.4 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 a dimension). For a one dimensional array the

stride is 4 for **Evaluation Power** BASIC and 6 for Development 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 of the array, 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:

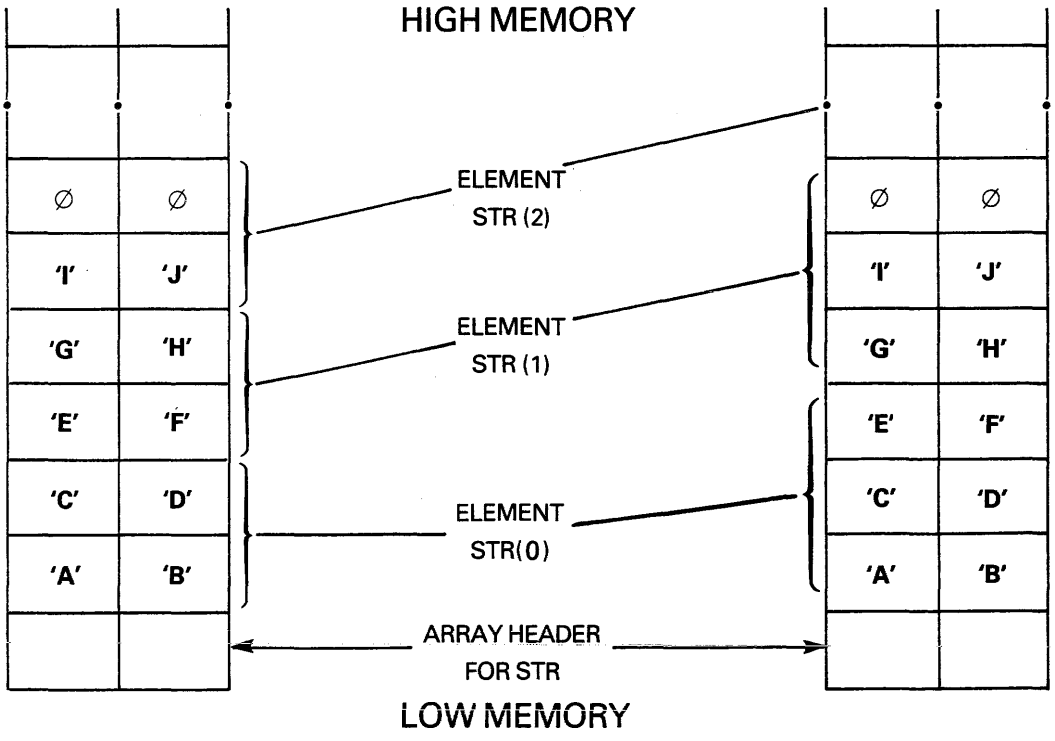


Figure 7-8 Array Storage

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 following the array subscript with a semicolon (;) and the number of the required byte. 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 a two dimensional array, which can be thought of as 26 one dimensional arrays each containing 10 elements. The stride for the first index will be 40 for Evaluation Power BASIC and 60 for Development Power BASIC; the stride for the second will be 4 for Evaluation Power BASIC and 6 for Development 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

7.7.2 System Memory Map

Any additional RAM to that supplied with the TM990/101M and /100M CPU boards must be configured to be contiguous and to

end at address >EFFF. For full details on how to do this, refer to Section 3 of the TM990/201 and TM990/206 Memory Expansion Boards Data Manual,

The lower limit of RAM is determined at system initialisation time by autosizing. This can be altered by:

NEW <exp>

where <exp> is the address of the first byte of RAM to be used by the system. (The first few bytes of RAM are reserved for system use.)

Once the system has been initialised, the memory map will look like this:

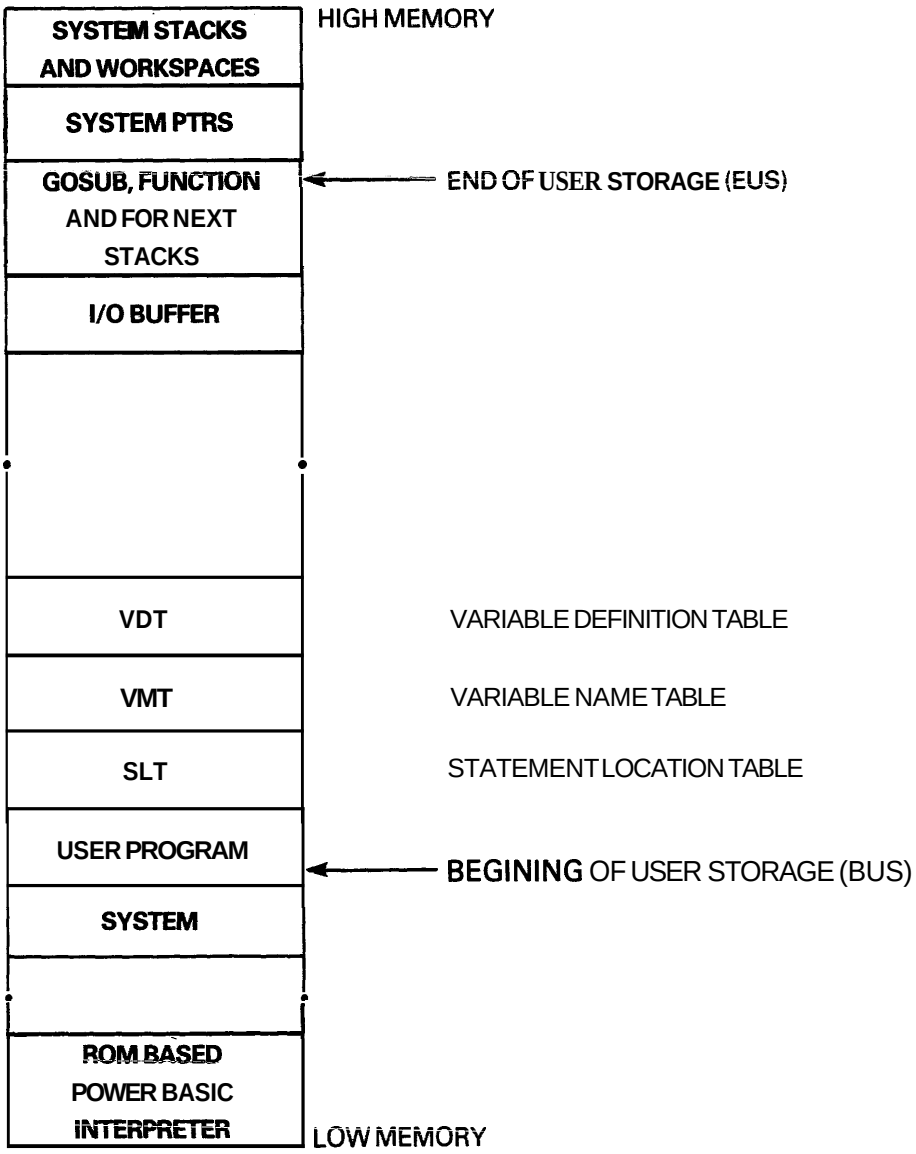


Figure 7-9 System Memory Map

When a Power BASIC statement is entered, it is checked for syntax errors, Syntactically correct statements are encoded to minimise 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 stored in their correct position in the program space.)

As the program grows the system tables (VNT, VDT and SLT) are moved up in memory in order to increase the size of each table and to expand the program space.

When a variable is first encountered, its name is encoded and entered into the Variable Name Table (VNT). As a statement is being encoded, all variable names present are replaced by their position within the VNT. This position number is then incremented by >74 to signify that an entry in the VNT is being referenced, For example, the statement:

```
LET   AJ=SIN (PI*RAD)
```

will initially be converted into something like:

```
LET   <77>=SIN(<76>,<75>)
```

The angle brackets are used to indicate a two digit hex number, <77> signifies the fourth entry in the VNT, <76> the third entry and <75> the second entry.

At run time, space is allocated to each variable as they are declared in the program; the address of this space is recorded in the Variable Definition Table (VDT). Variable storage is allocated from below the 1/0 buffer down towards BUS. If insufficient space exists, the run will terminate with error 10 (STORAGE OVERFLOW),

7.8 REFERENCE SECTION

An item preceded by an asterisk (*) denotes a feature that is not supported by Evaluation Power BASIC.

7.8.1 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 or THEN keyword
!	Tail remark indicator
;	Equivalent to PRINT

7.8.2 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.

7.8.3 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.

7.8.4 Edit Commands

	CR	Enter line into program source
	LF	Enter line into program source and enable the auto-numbering facility
	ESC	Cancel input line, return to keyboard mode
	DEL/RUBOUT	Backspace and delete character
*	Ctrl D <n>	Delete <n> characters
*	Ctrl I <n>	Insert <n> blanks
	Ctrl H	Backspace 1 chaaracter
	Ctrl F	Forwardspace 1 character
<1n>	Ctrl E	Display line <1n> for editing

7.8.5 Bower BASIC Commands

Power BASIC commands may not appear within a program,

Command		Function
CONtinue	*	Continue execution from last break
<ln> LISt		List currebt program from specified line **<ln>=Null, Line=First** line number **<ln>≠**Null, **Line=<ln>**
LOAd <exp>		Load 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 first line number
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

7.8.6 Power BASIC Statements

Power BASIC program lines are of the form:

{ line number } statement [:: statement] { ! comment }

where { } indicates optional items
[ ] indicates that the item is repeated as many times as required - 0,1,.,,.

Exceptions:

DATA should be the only statement on a line
NEXT should not be preceded by '::statement(s)'
REM should not be followed by '::statement(s)'

* BAUD <exp1> , <exp2>
Sets the baud rate of the serial 1/0 port(s) of the TMS9902 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> { , <parm> }
Transfers control to the assembly language subroutine <name> located at <add>, Up to 4 parameters, <parm>, are allowed in the statement (each separated by commas); these are passed to the subroutine in R4, RS, R6 and R7. (If a variable is contained in parenthesis, the address of the variable is passed,) The return address is contained in R11.

DATA <item> [, <item>]
Defines an internal data block for access by READ, <item> is either an expression or a string,

* DEF FN<i> { (<arg>) } = statement
Defines a single line arithmetic statement containing a maximum of 3, single letter, dummy variables <arg> (each separated by commas). <i> is the single alphabetic character function identifier, When caiiing FN<i> the dummy arguments are replaced by the actual parameters, which may be any Power BASIC variable, array element or expression,

DIM <var> (<num> [, <num>])
Allocates user space for the dimensioned array <var>. <num> is the number of elements in a dimension; each dimension starts at element 0.

* 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 returns to keyboard mode.

* ERROR <ln>
Specifies a Power BASIC 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> = <exp1> TO <exp2> { STEP <exp3> }
The FOR statement is used with the NEXT statement to open and close a program loop, Both identify the same FOR variable <var>. <exp1> is the start value, <exp2> is the end value and <exp3> is the stepsize. If STEP is omitted, a stepsize of 1 is assumed,

GOSUB <ln>
Transfers control to a Power BASIC subroutine starting at line <ln>. The address of the statement following the GOSUB statement is stored on the GOSUB stack.

GOTO <ln>
Transfers control to line <ln>.

IF <cond> THEN statement [:: statement]
The statement(s) following the THEN keyword are executed if the condition <cond> is true,

* IMASK <exp>
Sets the interrupt mask of the TMS9900 microprocessor to allow interrupts of higher or equal priority to <exp> (in the range 0 to 15).

INPUT <var> [; , <var>]
Take input (numeric or string) from the terminal and store it into next variable <var> 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. See section 7.8.14 for more details,

* IRTN
Is used to return from an interrupt routine; it restores the program environment existing prior to taking the interrupt,

{ 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 (when <exp>=i use the ith <ln> in the list). If <exp> is outside the specified range (less than 1 or greater than the number of <ln>s in the list) then drop through to the newt statement line.

* POP
Removes the top item from the GOSUB stack,

PRINT <exp> [, <exp>]
Prints (without formatting) the value of <exp>. See section 7.8.15 for more details,

* RANDOM <exp>
Sets the seed for the random number generator to the value of <exp>.

READ <var> [, <var>]
Takes input from the internal DATA block and stores it in the next <var> in the READ list,

REM <text>
Inserts comment lines (REMarks) into a user program. The whole line is taken as a comment.

RESTOR { <ln> }
Resets the DATA pointer to the specified DATA line <ln>. If <ln> is not present, the pointer is set to the first DATA statement in the program.

RETURN
Return from a Power BASIC subroutine, the return address is the last entry in the GOSUB stack,

STOP
Terminates program execution and returns to keyboard mode.

TIME { <item> }
Interrogate/set the 24 hour time of day **clock.**
<item>=Null - Output time in **HR:MN:SD** format
<item>=$<var> - Store time in string variable **<var>**
<item>=<exp1>,<exp2>,<exp3> **-** Set clock to specified time **(<exp1>=hours; <exp2>=mins; <exp3>=secs)**

* TRAP **<exp>** TO **<ln>**
Defines the entry point, **<ln>,** of a Power BASIC interrupt subroutine for interrupt level **<exp>.** Level 0 (RESET) and level 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, 1/0 port-A
<exp>=2, 1/0 **port=B**
<exp>=3, I/O ports A and B

7.8.7 Operators

7.8.7.1 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

7.8.7.2 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

7.8.7.3 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

7.8.7.4 Logical Operators

Perform bitwise operations on the operand(s). Operand(s) are converted into 16 bit integers before the operation.

LNOT A	* 1s complement
A LAND B	* Bitwise AND
A LOR B	* Bitwise OR
A LXOR B	* Bitwise exclusive OR

7.8.7.5 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 (=)

7.8.8 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>

7.8.9 CRU Operations

To use the following CRU functions it is first necessary to set the CRU base address via the BASE statement, (The value supplied to the BASE statement is twice the actual hardware base address,)

7.8.9.1 CRB Function

CRB (<exp>)
Read the CRU bit specified by the CRU hardware base address plus <exp>. <exp> is valid over the range -128 to +127.

CRB (<exp1>) = <exp2>
Set/reset the CRU bit specified by the CRU base address plus <exp1>. If <exp2>=0 then reset (0 the selected bit, otherwise set ('1') the bit. <exp1> is valid over the range -128 to +127.

7.8.9.2 CRF Functions

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 (<exp1>) = <exp2>
Output <exp1> bits of the value <exp2> to the CRU lines starting at the CRU hardware base address, <exp1> is valid over the range 0 to 15. If <exp1>=0 then 16 bits will be output.

7.8.10 Memory Functions

7.8.10.1 BIT Function

* **BIT (<var> , <exp>)**

Read the **<exp>th** bit of the variable **<var>.**

* BIT **(<var> , <exp1>) = <exp2>**

Modify the **<exp1>th** bit of the variable **<var>.** The selected bit is set to '1' if **<exp2>** is non-zero, otherwise it is set to '0'.

7.8.10.2 MEM Functions

MEM **(<exp>)**

Read the memory byte specified by **<exp>.**

MEM **(<exp1>) = <exp2>**

Set the memory byte specified by **<exp1>** to the value **<exp2>.**

7.8.10.3 MWD Functions

* MWD **(<exp>)**

Read the memory word specified by **<exp>.**

* MWD **(<exp1>) = <exp2>**

Set the memory word specified by **<exp1>** to the value **<exp2>.**

7.8.11 Miscellaneous Functions

78.11.1 NKY Function

NKY (<exp>)

Samples the keyboard in run-time mode. If <exp>=0 then return the decimal value of the last key struck. (Zero is returned if no key was struck.) If <exp>≠0 then compare the last key struck with the decimal value of <exp> and return a value of 1 (they are the same) or 0 (they are not the same).

7.8.11.2 SYS Function

* 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

7.8.11.3 TIC Function

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,

7.8.12 String Operations

<$var>	denotes either a literal string, enclosed in quotes, or a string variable
$<var>	denotes a string variable

A variable is specified as being a string variable by preceeding the variable name by a dollar sign ($).

An individual byte within a dimensioned string variable can be accessed by following the last array subscript with a semicolon (;) and the byte position,

$<var> = <$var>
Character Assignment: Copy characters into the string variable until a null (zero) byte is found.

$<var> = <$var> , <exp>
Character Pick: Copy <exp> characters into the string variable and then terminate the string with a null byte,

$<var> = <$var> + <$var> [+ <$var>]
Character Concatenation: Concatenate the strings into the string variable (in the specified order) and terminate the completed string with a null byte.

$<var> = <$var> ; <exp>
Character Replacement: Copy <exp> characters into the string variable (do not add the null byte).

* $<var> = / <$var>
Character Insertion: Insert the characters into the string variable.

* $<var> = / <exp>
Character Deletion: Delete <exp> characters from the string variable.

$<var>= % <exp> [% <exp>]
Byte Replacement: Replace the specified byte by the character equivalent of <exp>.

IF <$var><relop><$var> { , <exp> } THEN <sequence>
String Comparison: Where <relop> is a relational operator, If the second string is followed by a comma, <exp> indicates the number of characters to be compared.

* <var1> = <$var> , <var2>
Convert from ASCII to Binary: Convert the character string into its binary equivalent, The number delimiting character is stored in the first byte of <var2>.

* $<var> = <exp>

Convert from Binary to ASCII: Convert the number <exp> into an ASCII character string. The string is automatically terminated with a null character.

$<var>= # <$var> , <exp>

Formatted conversions can be made by preceding <exp> with the formatting operator (#) and a string.

7.8.13 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.

7.8.14 INPUT Options

INPUT <feature> <item> [**<del>** <feature> <item>]

<item>	Either a variable, a string variable, or an array element

<del>	Explanation
,	Delimit **<item>s** in the INPUT list
;	Delimit **<item>s** in the INPUT list. Suppress <CR> **<LF>** if at the end of the statement line

<feature>		Explanation
<string>		Prompt with <string> then get input
? **<ln>**	*	Upon an invalid input or control charcater, 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 and then get input

7.8.15 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 <item>s in the PRINT list and TAB to the next print field
;	Delimit <item>s in the PRINT list. Suppress <CR> <LF> if at the end of the statement line

<feature>		Explanation
<string>	*	Output <string>
TAB (<exp>)	*	TAB to column specified by <exp>
# <exp>	*	Print <exp> in hex free format
# , <exp>	*	Print <exp> in hex (word)
# ; <exp>	*	Print <exp> in hex (byte)
# <string>	*	Decimal formatting - (In Enhancement Software Package and Configurable Power BASIC). <string> can be

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

7.8.16 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	MORMALISE 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 FPAC's 2nd word - 16 bit twos complement number to floating point. Operand not used

Converting Integer. to Floating Point

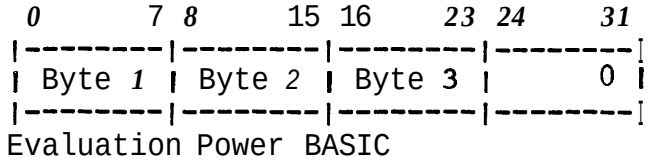
1) Set words 1 and 3 of 6-byte reserved area to zero.
2) Store integer number in 2nd word of area.
3) LOAD this 6-byte number into FPAC.
4) FLOAT FPAC.
5) STORE FPAC in 6 byte area.

```
DECNO  BSS  6
FLPT   BSS  6
       .
       CLR  @DECNO
       CLR  @DECNO+4
       LI   RO,NUM
       MOV  RO,@DECNO+2
       XOP  @DECNO,0
       XOP  0,10
       XOP  @FLPT,1
```

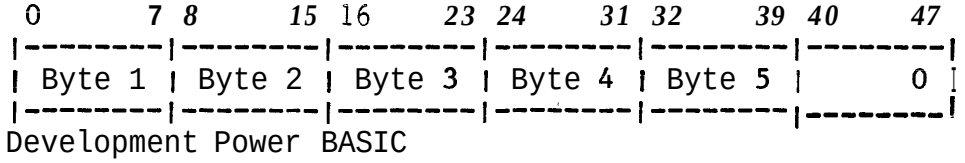
7.8.17 Variable Storage

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

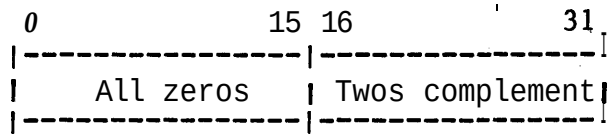


Evaluation Power BASIC

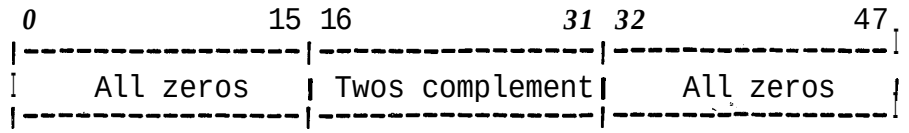


Development Power BASIC

Integer Format

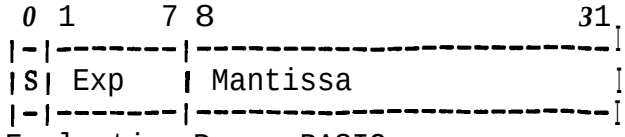


Evaluation Power BASIC

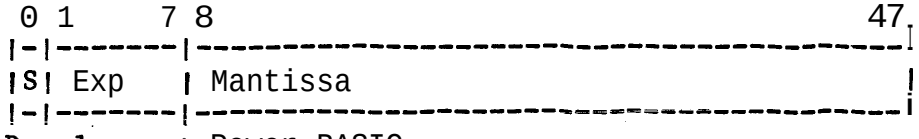


Development Power BASIC

Floating Point Format



Evaluation Power BASIC



Development Power BASIC

7.8.18 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
SO	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		

7.8.19 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

7.8.20 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

7.9 BIBLIOGRAPHY

TI Publications

Power BASIC Reference Manual (MP308)

Configurable Power BASIC Reference Manual (MP318)

TMS9901 Programmable Systems Interface (MP003)

TM990/100M Microcomputer User's Manual (MP321)

TM990/101M Microcomputer User's Manual (MP337)

TM990/201 and TM990/206 Memory Expansion Boards (MP334)

TM990/302 Software Development Board User's Guide (MP343)

Assembly Language Support For Power BASIC Application Report (MP719)

CHAPTER 8

ASSEMBLY LANGUAGE

8.1 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. Assembly language is therefore machine dependent and thus it has the capability to access all low-level features of the machine (memory, hardware registers, etc).

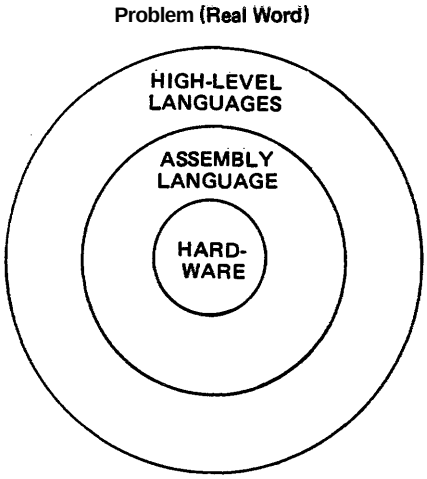


Figure 8-1 Assembly Language and the Computer

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), The disadvantage is that skill and a lot of time is needed to realize this compactness and efficiency, Using high-level

ianguages can speed up program production considerably and the program will he 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 1/0 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 difficult and requires very 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 section 8.13.2.1).

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.

8.2 INSTRUCTION FORMAT

An instruction consists of four fields, each separated from the other by at least one space. Several examples follow. The asterisk (*) in the first column indicates a comment line.

```
            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
```

The instruction fields are:

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 one. 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 (eg MOV). Assembler directives, assembly language instructions and pseudo-instructions are

covered by this term.

3) Operand field - This field specifies the opcode's argument(s); eg, where the data is to be taken from (source) and/or where 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 extremely 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:

- o LABEL field Starts in column 1
- o OPCODE field Starts in column 8
- o OPERAND field Starts in column 13
- o COMMENT field Starts in column 31

8.3 INSTRUCTION FORMAT RESTRICTIONS

Restrictions to instruction formats are listed below.

1) If a label is present it must start in column one; otherwise column one must be 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.

8.4 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.

or 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).

8.4.1 Byte

A byte is a group of eight binary digits (bits). The most significant bit (MSB) is designated hit 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).

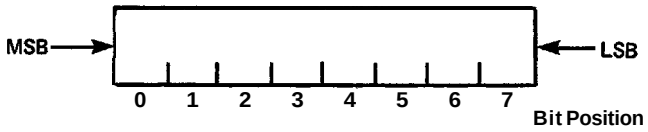


Figure 8-2 A Byte

8.4.2 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).

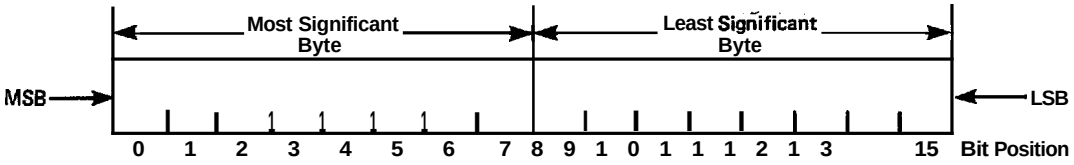


Figure 8-3 A Word

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.

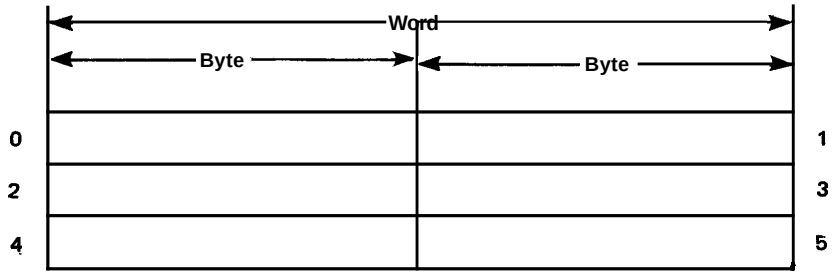


Figure 8-4 Memory Organisation

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 (eg MOVB, CB, AB, SB, etc). Using these instructions, it is possible to individually access/manipulate each of the bytes within a word.

8.4.3 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:

- Load the contents of one of the locations into a register.
- Add the contents of the other location into the register.
- Store the contents of the 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) - contains 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,

8.4.4 Workspace Registers

The TMS9900 does not provide a unique 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 (RO 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, when' a BLWP instruction is used to call a subroutine, or when an XOP instruction is executed. The workspace can also he 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 register 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 this instruction has been executed, RO 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,

8.4.5 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 a shift instruction is zero, the shift count is taken from bits 12 to 15 of R0. If these 4 bits are all zeros, the shift count is set to 16.

R11 Branch and Link instruction uses R11 to store its return address. Also the XOP instruction uses R11 to store the effective address of the source operand,

R12 Bits 3 to 14 of R12 contain the hardware base for CRU instructions.

R13 When a context switch occurs, R13 is used to store the old WP.

R14 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 (eg 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.

8.4.6 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 8.4.7.4) on the new register 13.

Hardware interrupts, XOP instructions and the BLWP instruction cause a context **switch** to take place. For an interrupt and an XOP instruction, the WP and PC are taken from the interrupt's or **XOP's** vector. The BLWP instruction requires the address of a two word area, containing the new WP and PC, as its operand. This two word area is known as a RLWP vector.

Executing a RLWP instruction does not affect the ST register. An XOP instruction causes the ST register's bit 6 to be set to a one. The hardware interrupt only changes the ST register's interrupt mask (bits 12 to 15); this is set to one less than the incoming interrupt level (a level six interrupt resets this mask to five).

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 WP
   0220  SUBWP   BSS     32        Define SUB's WP
   0240  SURPTR  DATA    SUBWP     Ref SUB's workspace
   0242          DATA    SUB       Ref SUB's entry point
                 .
                 .
                 .
         MAIN    EQU     $         Entry point for MAIN
                 LWPI    MAINWP    Load WP with >200
                 .
                 .
                 .
   1000          BLWP    @SUBPTR   Execute subroutine SUB
                 .
                 .
                 .
   1200  SUB     EQU     $         Entry point for SUB
                 .
                 .
                 .
   1300          RTWP              Exit from SUB
```

The context switch is shown diagrammaticaliy **in** Figures 8-5, 8-6 and 8-7.

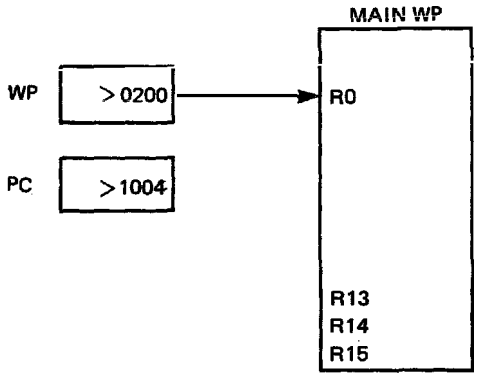


Figure **8-5** Before Executing the BLWP Instruction

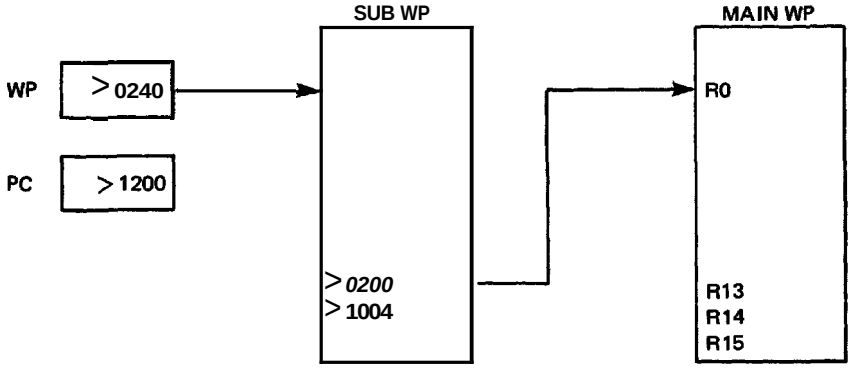


Figure **8-6** After Executing the BLWP Instruction

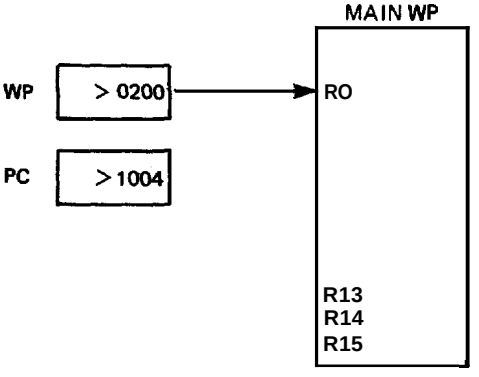


Figure 8-7 After Executing the RTWP Instruction

8.4.7 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.

8.4.7.1 Register Addressing

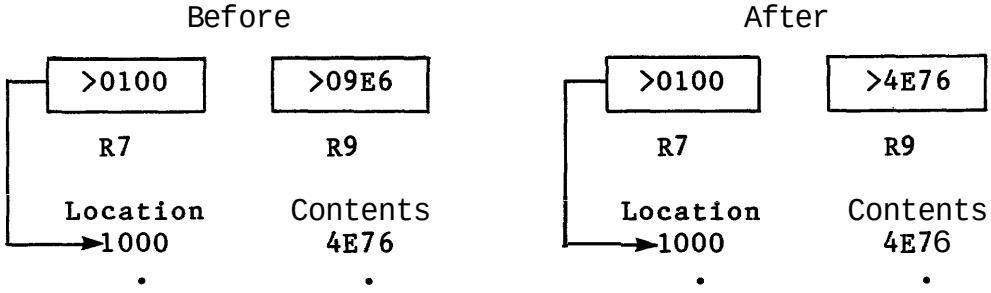
A workspace register contains the operand.

```
* Copy the contents of R4 into R10

      MOV   R4,R10
```

8.4.7.2 Register Indirect Addressing

A workspace register contains the address of the operand. To identify this mode the workspace register is preceded by an asterisk (*).

```
* Copy the contents of the address in R7 to R9

      MOV   *R7,R9
```



8.4.7.3 Symbolic Memory Addressing

A memory address contains the operand. To identify this mode, the memory address is preceded by an at sign (@). (If a symbolic name such as TABLE is used, the name must be defined somewhere in the program.)

```
* Copy the contents of the word at symbolic address TABLE
* into address >7C

      MOV   @TABLE,@>7C
```

Before		After	
Location	Contents	Location	Contents
007C	0471	007C	6483
.	.	.	.
.	.	.	.
TABLE	6483	TABLE	6483
.	.	.	.

8.4.7.4 Indexed Memory Addressing

A memory address contains the operand. The address is the sum of the contents of a workspace register and a symbolic address. This mode is written as an address preceded by an at sign (@) and followed by a workspace register enclosed in parentheses (the index **register).** Register **0** can not be used as an index **register.**

```
* Copy the contents of word at location (2 + contents of R7)
* into location (address of TABLE + contents of R10)

      MOV   @2(R7),@TABLE(R10)
```

Before

>1000	>0006
R7	R10

Location	Contents
1000	4849
1002	2041
.	.
TABLE	454D
.	5443
.	2052
.	5546

After

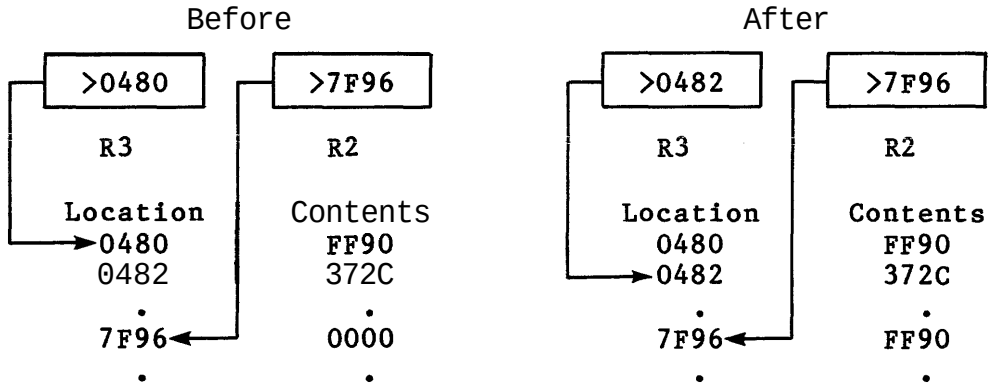
>1000	>0006
R7	R10

Location	Contents
1000	4849
1002	2041
.	.
TABLE	454D
.	5443
.	2052
.	2041

8.4.7.5 Register Indirect Autoincrement Addressing

This is similar to the 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 preceded by an asterisk (*) and followed by a plus sign (+).

```
*
* Copy the contents of the word at the address in R3 into
* the word at the address in R2. Increment R3 by 2

       MOV   *R3+,*R2
```



This mode is very useful for working with structures such as tables, where a succession of memory locations must be accessed in sequence.

8.4.8 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.

8.4.8.1 Immediate Addressing

This 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     Set WP= >FE70
       LI    R5,1000   Set R5= 1000
```

8.4.8.2 CRU Bit Addressing

This 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. For full details on the operation of the CRU, refer to section 8.9.

SRO	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

8.4.8.3 Program Counter Relative Addressing

This 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:

- Subtracts the value of the incremented PC (address of the next instruction) from the symbolic address.
- Halves 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 he stored in memory).

8.5 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  ....      lst instruction
```

Another way of defining this subroutine is:

```
CLOSE  EQU   $
       ....      lst instruction
```

Although both approaches produce the same machine code, the second clearly indicates a subroutine's entry point and thus aids program documentation.

Care must be exercised when using the second approach to ensure that the assembler's location counter is on an even address (ie a word boundary) when the subroutine name (CLOSE above) is defined. The only time this location counter might have an odd address is when the assembler has just allocated some space via the BYTE or TEXT directive. If this is the case then it is necessary to follow the directive by an EVEN directive, EVEN tells the assembler to increment its location counter by one if it contains an odd address (ie a byte boundary), otherwise it is ignored.

```
BOD    BYTE  >OD   or   MSG    TEXT 'ENTER COMMAND'
       EVEN
CLOSE  EQU   $
```

Note that this is not strictly necessary with the first approach as the assembler automatically forces its location counter to a word boundary when assembling instructions,

The Branch and Link instruction (BL) causes the address of the instruction following the BL 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 acce'ss the calling routine'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 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 address >1000 then 'BLWP @>1000' can 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 on register 13 of the new workspace. When the context switch is performed, register 13 of the new workspace automatically contains the address of the old workspace, Register 5, for example, of the old workspace can be referenced by using '@10(R13)' as the operand of an instruction, The indexed address is obtained by adding ten bytes to the contents of register 13. As register 13 contains the address of the old workspace, adding ten bytes (or five words) to this address means that the sixth word of the old workspace (or the old register 5) is accessed. (The first word, or old register 0, is accessed by adding zero to register 13; the second, or old register 1, by adding two; etc.)

The BLWP instruction is a very useful instruction for implementing modular software in assembly language (see Section 4.3).

Control is returned from a subroutine by either an 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),

An 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 a branch to the address contained in R11 (the register used by the BL instruction to store the return address).

8.6 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, if 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 three main methods of parameter passing and their implementation in 9900 assembly language are given below.

1) The parameter is stored in a register.

a) Subroutine invoked by BL instruction:

```
*
* Called routine has direct access to all the
* calling routine's registers
```

b) Subroutine invoked by BLWP instruction:

```
*
* Copy the contents of calling routine's workspace
* space register N into TEMP
*
        MOV   @2*n(R13),temp
```

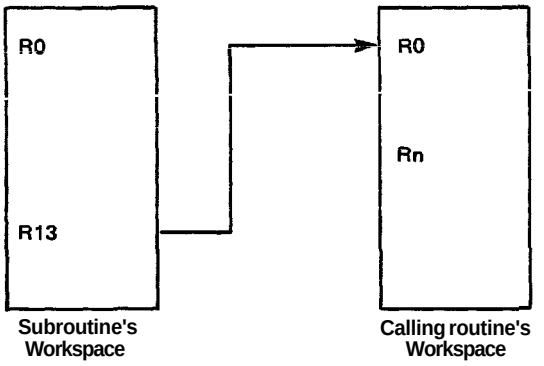


Figure 8-8 Parameter Passing 1

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. (Parameter numbering starts from zero.)

a) Subroutine invoked by BL instruction:

```
*
* Copy contents of the Mth word (Mth parameter) of
* the parameter block into TEMP

        MOV   @2*m(Rn),temp
```

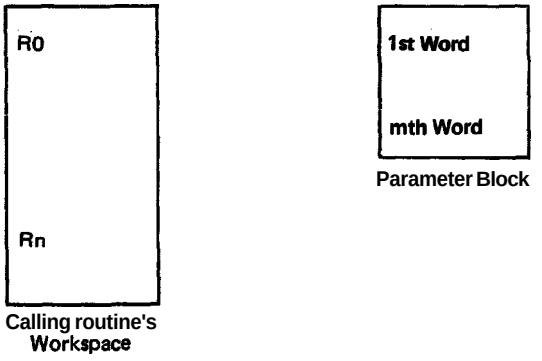


Figure 8-9 Parameter Passing 2

b) Subroutine invoked by BLWP instruction:

```
*
* Copy address in the calling routine's workspace
* register N into register S
*
*       MOV   @2*n(R13),Rs
*
* Now copy contents of Mth word of parameter block
* into TEMP

        MOV   @2*m(Rs),temp
```

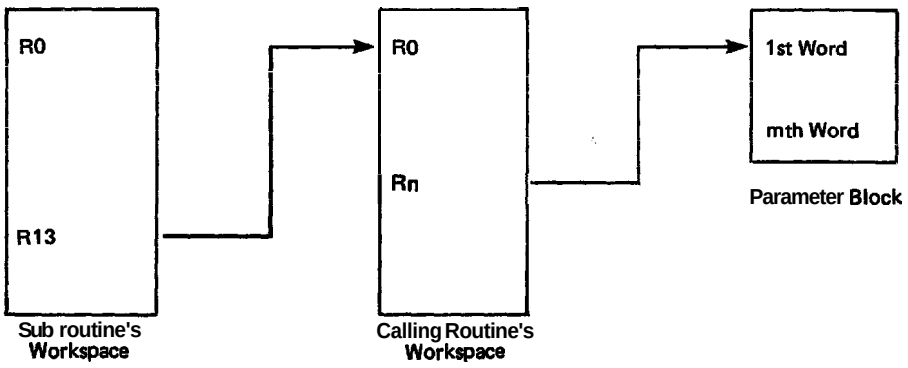


Figure 8-10 Parameter Passing 3

3) This is a variation on the previous method in that 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 to 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.

a) Subroutine invoked by BL instruction:

```
        BL    @SUBR       Call SUBR
        DATA  ....        Parameter block
        .

        .
SUBR    MOV   *R11+,temp  Get 1st parameter in TEMP,
                          update return address in R11

        .
        RT                Return
```

b) Subroutine invoked by RLWP instruction:

```
SUBADD  DATA  SUBWP       SUB's workspace
        DATA  SURR        SUB's entry point

        .
        BLWP  @SUBADD     Call SUB
        DATA  .....       Parameter block

        .
        .
SUBR    MOV   *R14+,temp  Get 1st parameter in TEMP,
                          update return address in R14

        .
        RTWP              Return
```

This in-line approach should only be used when the

data to be passed to the subroutine is constant (its value is known when the program is assembled), since program code is likely to be placed in ROM.

Note: Invoking a subroutine is faster using the BL instruction as no context switch takes place, but there is a risk that data might be inadvertently lost when any of the calling routine's registers are used for temporary storage purposes.

8.7 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 sequence, selection and iteration constructs used in software design. The sequence, selection and iteration constructs (and the notation used here) are described in Section 4.5.

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.

8.7.1 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.

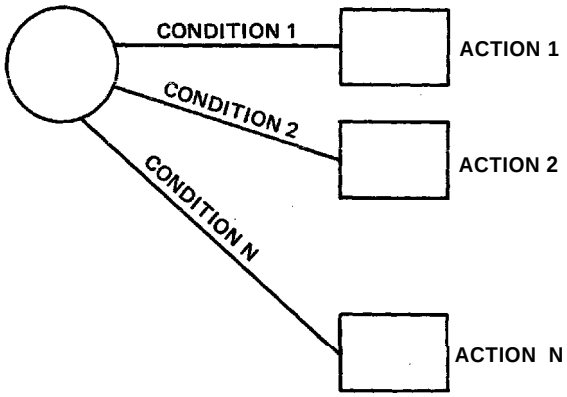


Figure 8–11 General Selection Construct

8.7.1.1 Condition Codes

Implementing the selection 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.

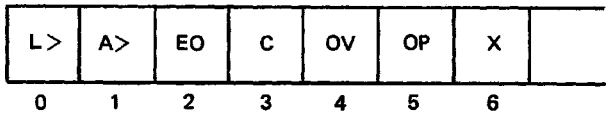


Figure 8–12 Condition Codes for the **TMS9900** Status Register

LOGICAL GREATER THAN **(L>)** contains the result **of** a comparison of **words/bytes** as unsigned binary numbers; as the sign bit is interpreted as part of the number, a negative number is logically greater than a positive one.

ARITHMETIC GREATER THAN **(A>)** holds the result of a comparison of **words/bytes** as signed binary numbers.

EQUAL (EQ) is set when the **words/bytes** being compared are equal. Also contains the TB CRU bit.

CARRY (C) is set by a carry out of the most significant bit of a **word/byte** during arithmetic operations. This bit is also used by the shift instructions to hold 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.

ODD PARITY (OP) is set when the result of a **byte operation** has odd parity (when the number of bits in a byte having a value of '1' is odd).

EXTENDED OPERATION (X) is set when an extended operation instruction is performed by software.

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. Full details on which flags are affected by a given instruction are given in the reference section of this chapter.

8.7.1.2 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.

Example: Depending on the contents of R2 (>10, =10, or <10) execute the sequence ACT1, ACT2 or ACT3 respectively. Then execute the sequence ACT4.

The structure diagram for this is:

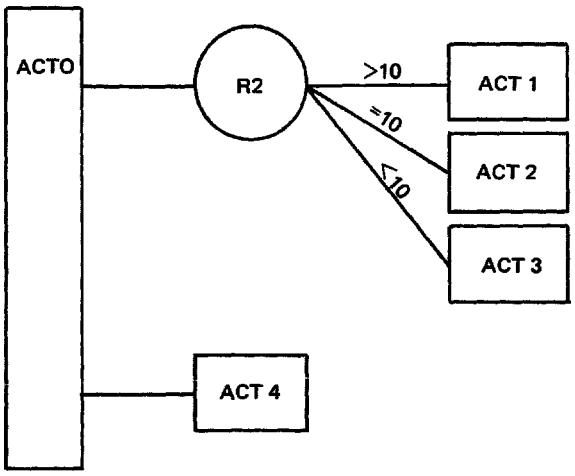


Figure 8-13 A Three Way Selection Example

This can be coded as:

```
ACT0    EQU   $
        CI    R2,10     Compare X2 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:

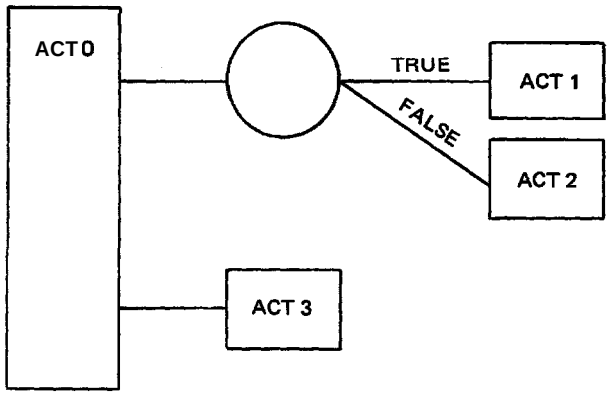


Figure 8-14 A Two Way Selection Example

This can be coded 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
```

8.7.2 Iteration

Quite often it is necessary for a sequence of instructions to be executed a number of times. One way of implementing this repetition 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** (SEQl) 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:

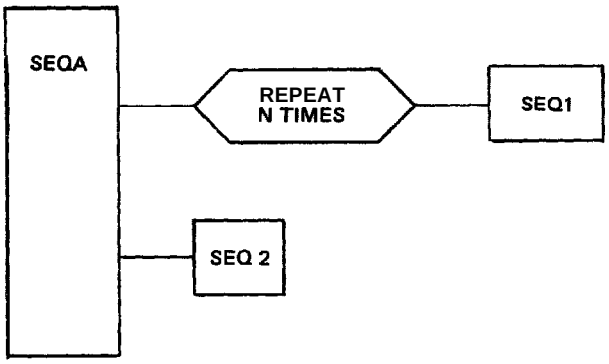


Figure 8-15 An Iteration Example (REPEAT)

This can be coded as:

```
SEQA   EQU   $
       MOV   @n,R0     Copy count into R0,sets flags
SEQAST JEQ   SEQ2      To SEQ2 if R0 = 0
SEQ1   EQU   $
       .
       Code for SEQl
       .
       DEC   R0        Decrement repetition count
       JMP   SEQAST    To SEQAST
```

```
SEQ2    EQU   $
        .
        Code for SEQ2
        .
```

If N is a constant (eg 20) then:

```
        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 is:

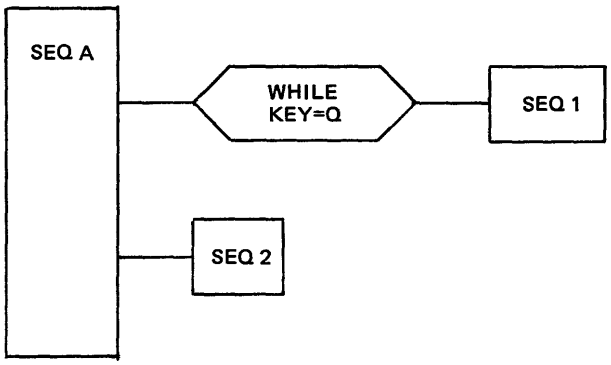


Figure 8-16 An Iteration Example (WILE)

This can be coded as:

```
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
        .
```

8.7.3 sequence

On the surface, the sequence is the simplest construct to implement, as it merely involves executing one instruction after another. Unfortunately, with assembly languages there is a great temptation to write programs in an unsequenced fashion with program flow jumping backwards and forwards in an irregular manner. This usually leads to 'spaghetti code'; code so convoluted and complex (often much more complicated than is actually necessary) that it is difficult to follow or understand and almost impossible to maintain.

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 (see Section 4.5.1). Further, it is important that a single block on the structure diagram be implemented as a single block of code.

This is, in fact, the simplest and the most natural way to write programs; it is certainly the easiest to follow.

Consider this structure diagram:

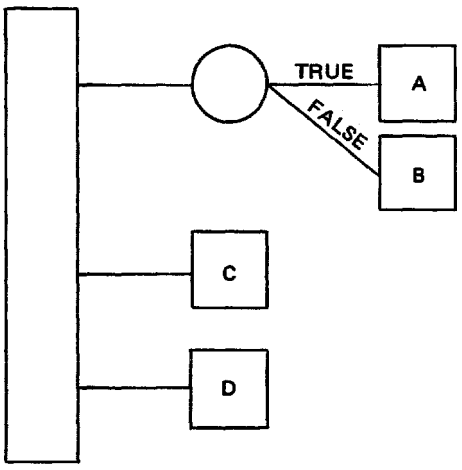


Figure 8-17 A Sequence Example

This can be coded in (at least) three ways:

```
        '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.

When a 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.

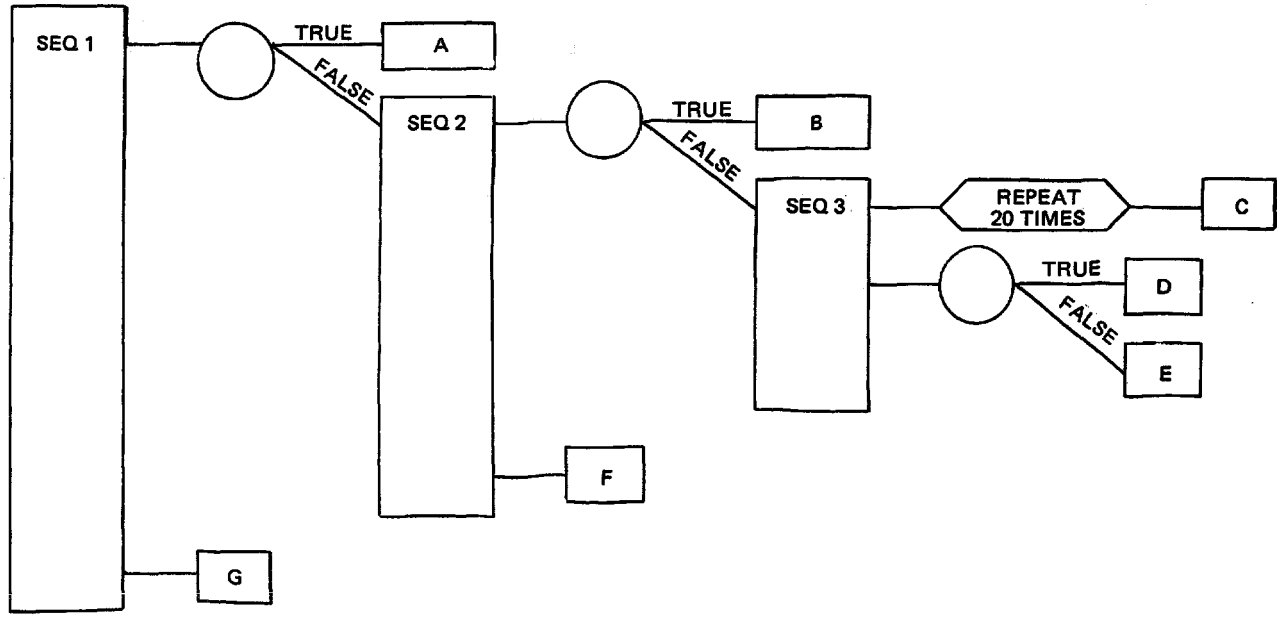


Figure 8-18 A Complex Structure

The 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   $         To here if true
        .
        Code for D
        .
        JMP   F         To F
E       EQU   $
        .
        Code for E
        .
F       EQU   $
        .
        Code for F
        .
G       EQU   $
        .
        Code for G
        .
```

8.8 PROGRAMMING FOR RX AND COMPONENT SOFTWARE

When writing a software system as a single unit, any method can be adopted for the use of memory, way of calling subroutines, etc, provided the system is internally consistent.

However, there is often a requirement for writing software that can:

a) Make use of existing pieces of software.

or b) Be used by other pieces of software.

or c) Be reliably updated at a later date, perhaps by someone other than the person who wrote it.

All these requirements dictate the use of standard conventions: a set of rules which are known to be complete and consistent, and can be written down.

Pieces of software developed according to such conventions will work together. (Of course, if one piece of software wishes to make use of another piece, it must know what functions are available in the second piece of software and how to access them.)' Conventions make it possible both to write pieces of software that will not conflict, and to 'package' them in standard ways. Software packages can be stored in libraries, then selected and connected together to form a new system.

TI's Component Software provides a framework of standard conventions within which pieces of software can be written separately to perform independent tasks. The pieces can then be 'plugged together' to build a system. The parts plugged together may have been written by the user, or they may have been bought 'off the shelf' from TI or other vendors.

TI's Realtime Executive (Rx) is the means of welding these separate parts together to make a complete, coherent system. Component programs call Rx routines to perform commonly needed operations (such as calling other routines, requesting additional memory space, etc). Rx manages all the resources of the system so that conflicts do not occur.

This is an extension of the program modularity described above (in relation to sequence, iteration, etc). Rx provides 'time modularity' too: it allows independent application functions to he written as separate programs with different demands on the time of the processor (some functions may need to be executed every 5ms, say; others only when an operator presses a key, or a particular device interrupts).

When building an application system, these functions are linked together, in a semi-automatic process know as configuration.

Rx provides a standard mechanism for handling interrupts, standard ways of dealing with file I/O, and standard methods for calling other routines (whether written in assembly

language, Pascal or other languages),

The benefits of this 'component' approach are:

- Systems can make use of existing Component Software packages.
- Software modules written according to Component Software standards can be used again, in other systems,
- Reliability is improved, because each task in a real time system can be programmed and tested separately, and then linked with the other parts to form the system.
- Systems can be upgraded easily, because the component parts can be separated out and replaced, changed or added to as necessary,
- Because of the above, systems can be developed more quickly and for less cost.

The conventions that must be followed mainly relate to calls between routines and the access to registers and memory,

In a high-level language, many of these requirements are taken care of automatically by the compiler. The assembly language programmer must himself ensure that the conventions are followed when writing the program,

The standards are set out in the Component Software Handbook and the Realtime Executive User's Manual, Adherence to these standards (which are not too restrictive) means that programs written can be used with other Component Software routines, whether written in Microprocessor Pascal or assembly language. See Chapter 5.

Routines to be used with Component Software should be written according to the Rx standards from the start. This is much easier than converting routines already written,

8.9 COMMUNICATIONS REGISTER UNIT

The 9900 supplies a bit-oriented method of 1/0 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' 1/0 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 1/0 used 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). The CRU bits required to control a particular device should be grouped together, If a system has several identical devices the same piece of code (structured as a subroutine) can be used for each. It is only necessary to set the CRU base address for the appropriate machine and call the subroutine.

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.)

8.9.1 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.

Diagrammatically this can be shown as:

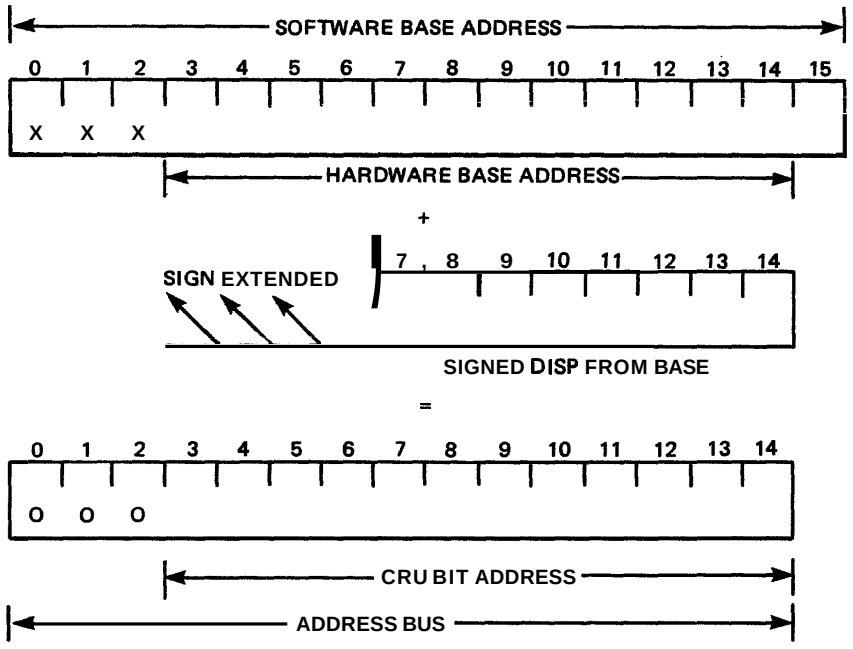


Figure **8-19** CRU Bit Addressing

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 CRU bit >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 CRU bit >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 will 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   $
        .
```

8.9.2 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 he 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:

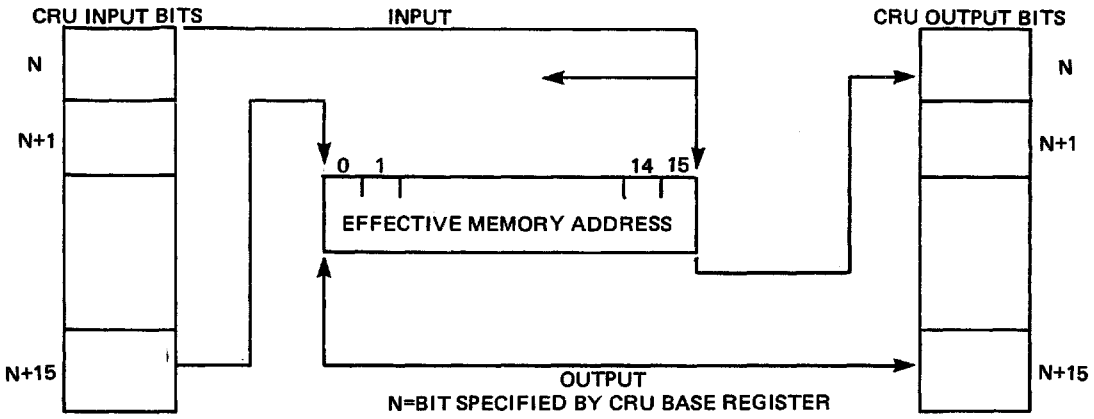


Figure 8-20 CRU Transfer Of More Than 8 Bits

For example, in a transfer involvfng 10 bits, the data Is taken from, or stored in, bits 15 to 6.

For a transfer of 8 bits or less:

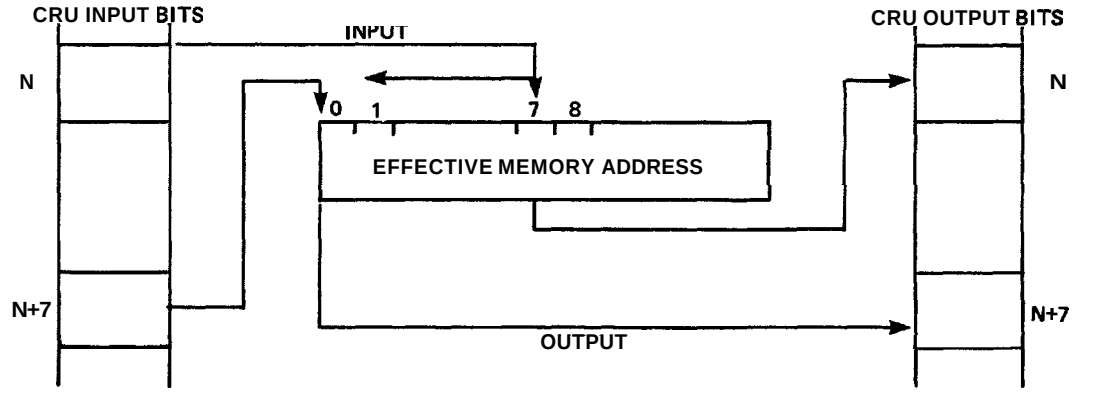


Figure 8-21 CRU Transfer Of 8 Bits Or Less

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 data 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
```

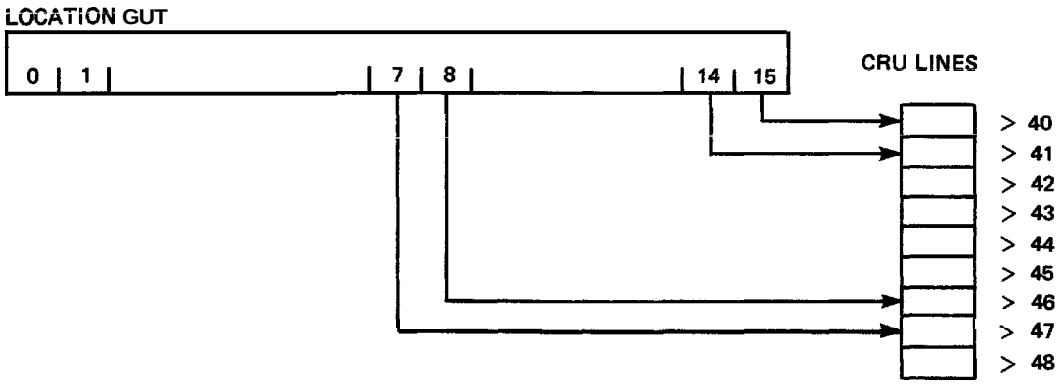


Figure 8–22 CRU Output Example

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
```

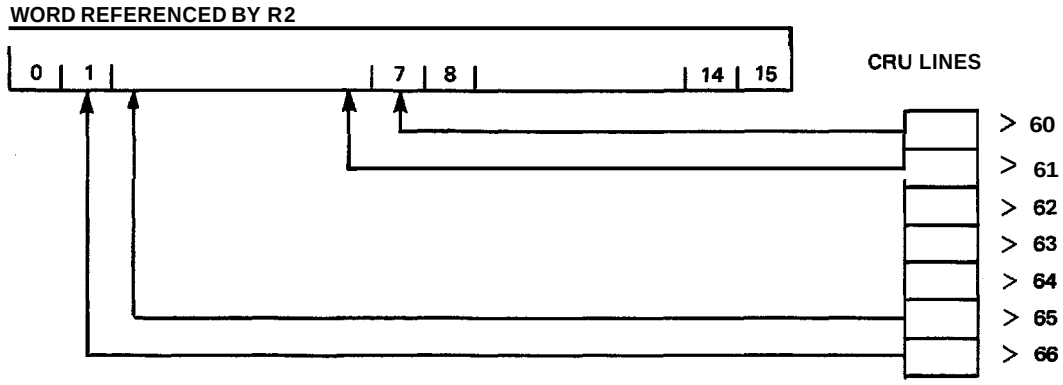


Figure 8–23 CRU Input Example

Note: If workspace register 2 had contained an odd address (ie if it referenced a word's least significant byte) then the input would have been stored in bits 15 to 9.

8.10 INTERRUPTS

In a real–time system, there are two mechanisms for determining when an external event has occured (for example, when a device connected to the computer needs to be

serviced): Polling and Interrupts.

In the polling mechanism, 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 considerable delay between the time when the device indicates that it is ready and the time when the program actually discovers that it is ready, Because of this, polling is only practical on a simple system, or when response time is not critical.

With the interrupt mechanism, 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 some other operation 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.

8.10.1 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 of the status register, determine which interrupts are to be recognised by the processor.

A device with a lower priority (higher 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 than or equal to the device's interrupt level.

Often, instead of being coupled directly to the 9900 microprocessor, interrupt lines are connected to a TMS9901 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 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 TMS9901 Programmable Systems Interface Data Manual,

Interrupt Mask Bits				Levels Allowed	Level setting Mask	
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	--	

Table 8-1 Interrupt Mask Table

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,

8.10.2 Interrupt Vectors

Every interrupt level has a two word dedicated area (known as the interrupt vector) containing:

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 (see Table 8-2).

A particular interrupt vector (for interrupt level 8, say) can be assigned the appropriate values by:

```
AORG    >20       Interrupt level 8 vector at >20
DATA    INT8WP    Workspace for interrupt level 8
DATA    INT8PC    Entry point for level 8 handler
```

Address	Level	**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**

Table **8-2** Interrupt Vector Table

8.10.3 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.** (Note: The level 0 interrupt, the RESET interrupt, will always be taken and can not be masked out,)

For example, if the processor is servicing a level 4 interrupt, only interrupts of level **3** and higher (ie levels **0** to **3)** will be recognized,

To process an interrupt, a context switch takes place, The contents of the interrupt vector's first word is stored in

the WP register and 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.

After storing the contents of the ST register, the processor decrements the incoming interrupt level by one and stores the result in the interrupt mask. This disables the current interrupt level, leaving only higher levels enabled. (This does not happen with level 0 interrupts.)

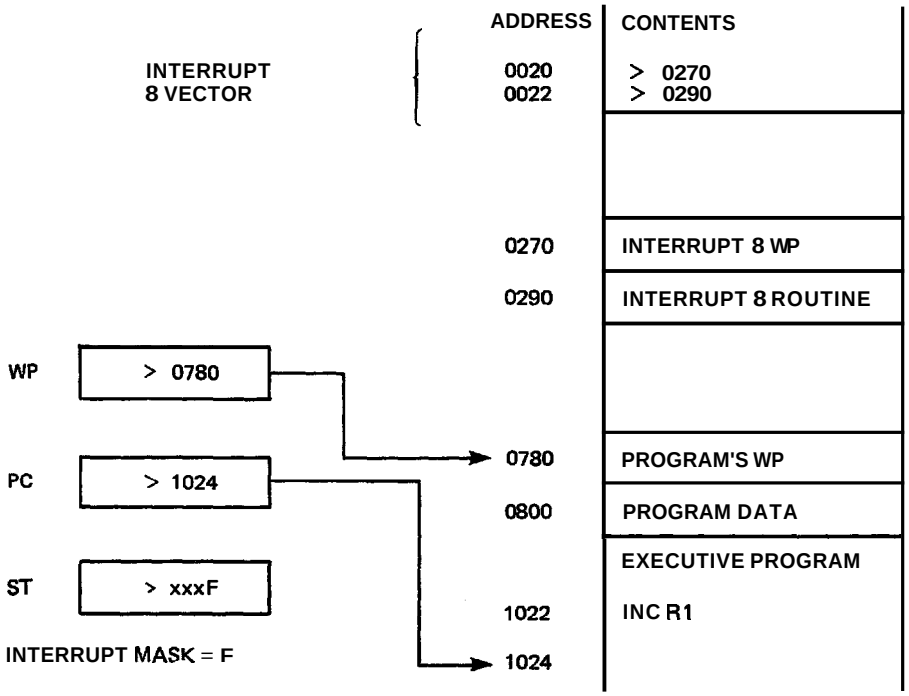


Figure 8-24 State Prior to a Level 8 Interrupt

No additional interrupt is taken until the first instruction of the service routine has been executed. If the first instruction is a 'LIMI 0' (Load Interrupt Mask Immediate with zero) then further interrupts will be inhibited.

The last instruction in the service routine must be an **RTWP.** This causes the processor to restore the contents of the WP, PC and ST registers from workspace registers 13, **14** and **15** respectively (ie 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 then becomes the programmer's responsibility to determine which device generated the interrupt by polling the devices and then executing the appropriate service routine.

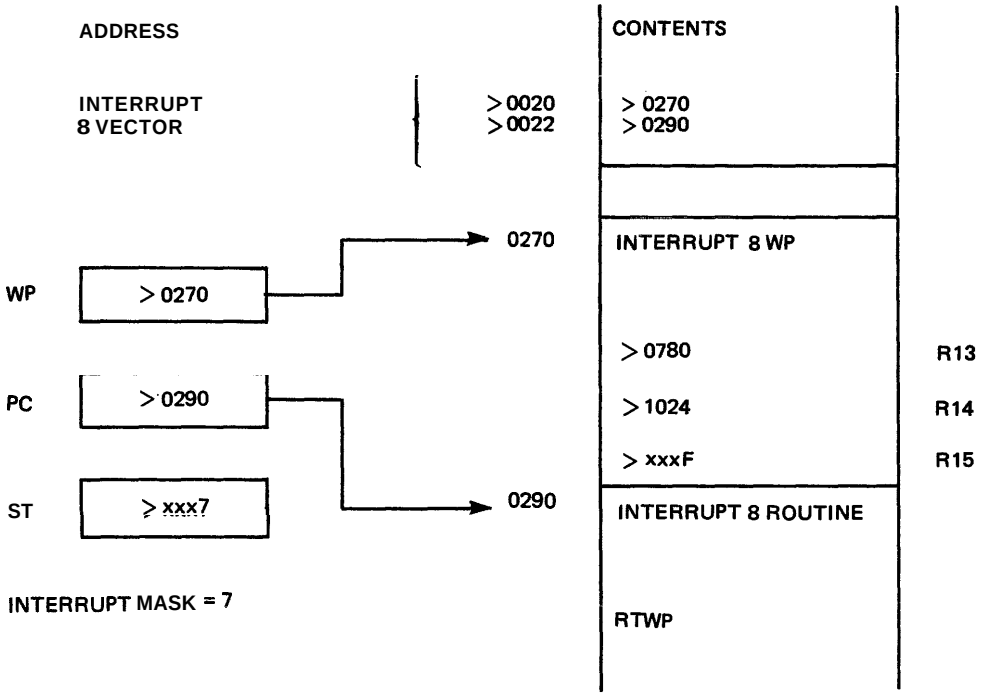


Figure 8–25 State After a Level 8 Interrupt

Any interrupt request must remain active until it is reset by the interrupt service routine. Interrupts that just disappear (without being reset) can cause program execution to become unpredictable; the interrupt level presented to the processor could become corrupted and subsequently the wrong interrupt service routine would be invoked. Failure to reset an interrupt will cause the processor to re–take the interrupt as soon as the service routine has completed.

8.10.4 Fault Tolerant Interrupt Systems

In an interrupt–driven control environment it is almost impossible to guarantee that only valid interrupt signals are going to be generated. This is especially true in electrically noisy environments (for example when switching on a motor). The system designer must be aware of the possiblity of receiving false interrupt signals and should be able to recognise the situations where these may occur. Further, part of the system design goal(s) should be concerned with overcoming this problem.

It is also a good idea to build a certain amount of fault tolerance into the system. Obviously the more that is built into the system the more reliable the system is going to be. However, this does increase the complexity and hence the cost of the system. Some systems may not require much (if any) fault tolerance; it may be sufficient to simply power down all the equipment in some ordered sequence. In others, a large amount may be needed, especially if the system is expected to recover from the fault. The actual

amount of fault tolerance built into a system depends on the design criteria (speed, simplicity, recoverability, reliability, cost, etc).

A classic example of including fault tolerance in a system is the overflow pipe in a domestic water supply, in particular, in the cistern. In normal operation, no overflow pipe is required; the ball-cock floats on top of the water and determines how much more water is needed, opening or closing the water inlet value as necessary. However, what happens if the ball-cock loses its buoyancy or the inlet value sticks open? It would mean water running down the walls, damaging carpets, furniture, etc. Typically this doesn't happen as the overflow pipe is included to cater for this problem. The system tolerates this type of fault: water overflows, but not on the carpet.

In an interrupt-driven environment, a simple piece of fault tolerance is to "tie" all unused interrupt levels to a common interrupt service routine (this is often referred to as a 'spurious interrupt handler'). What this handler actually does is entirely up to the user; it may be nothing more than an RTWP instruction or it may, for example, provide the user with some form of statistics on false interrupts. If the handler does anything other than the RTWP it will be necessary to either perform the 'LIMI 0' instruction or to allocate some memory to be used as a workspace (not necessarily a whole workspace, but at least three words for R13, R14 and R15) for each unused interrupt level.

Although this doesn't stop any false interrupt signals from being generated, it does ensure that a false interrupt on an unused interrupt level will not have disastrous side effects. How to cope with false interrupt signals on a used interrupt level is another problem. It may be possible to investigate the "interrupting" device and to determine whether it actually interrupted or not. Or it may be possible to state that a particular device can only interrupt when some specific set of conditions prevail; if all the coditions are met then assume that it was a true interrupt, otherwise it could be treated in a similar fashion to an unused interrupt level.

8.11 EXTENDED OPERATION INSTRUCTIONS

Extended operation instructions (XOPs) enable the user to extend the existing instruction set by defining additional "instructions" that are implemented by software routines. XOPs provide a kind of "fast subroutine call" for often performed operations. The 9900 supports 16 extended operation instructions, numbered 0 to 15.

If the program is running under an operating system, XOP instructions are often predefined by the system. They are used as a method of calling operating system routines that perform specific functions. These functions, in particular input/output operations, are provided by the system as it is not safe to allow a user to implement them (they could, too easily, affect other users). The XOP mechanism isolates the user from the internal workings of the operating system. Extended operation instructions, used in this manner are also known as extracodes or supervisor calls (SVCs).

This type of instruction is often 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 XOP instruction requires an operand; this allows a parameter to be passed over to the service routine.

One potential problem with XOPs is that there is only one set of XOPs in each system. Where a system can execute multiple programs, there is a potential conflict over use of XOPs, as different programs may wish to use the same XOP number for different operations.

8.11.1 Defining Extended Operation Instructions

XOP is a valid assembly language mnemonic; unfortunately, it does not convey any information 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. DXOP has 2 operands:

1) The mnemonic by which the XOP is to be known.

2) The number of the XOP involved.

This directive associates the mnemonic with a particular XOP (it does not generate any code). When the mnemonic appears as an instruction opcode, the assembler generates the machine code to execute the appropriate XOP routine. (It translates the mnemonic into the correct XOP instruction and then assembles that.) For example:

```
DXOP   CALL,4
.
.
.
CALL   @FRED
```

The first instruction associates the mnemonic CALL to XOP 4. The second is an example of an XOP instruction (although it doesn't look like it). The effect of these two instructions is to execute the XOP 4 instruction with the symbolic address FRED as its parameter.

8.11.2 Extended Operation Instruction Vectors

Like a hardware interrupt, an extended operation instruction has a two word dedicated vector containing:

1) The address of the workspace to be used by the XOP.

2) The address of the XOP routine's entry point.

These vectors are located at memory addresses >40 to >7F (see Table 8-3).

Before an extended operation instruction is executed, its vector must contain the appropriate values. For the CALL extended operation above:

```
AORG    >50       CALL's vector at >50
DATA    CALLWP    Workspace for CALL
DATA    CALLPC    Entry point for CALL
```

8.11.3 Extended Operation Instruction Execution

When an extended operation instruction is executed, the processor performs the following sequence:

1) Locates the XOP's 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 instruction'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,

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 XQP 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

Table 8-3 XOP Vector Table

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 shown earlier

CALL @FRED can be written as XOP @FRED,4

The latter does not require the DXOP directive to be used. However, it is recommended that the first approach be adopted as the mnemonic can indicate what the routine actually does and thus aids program readability.

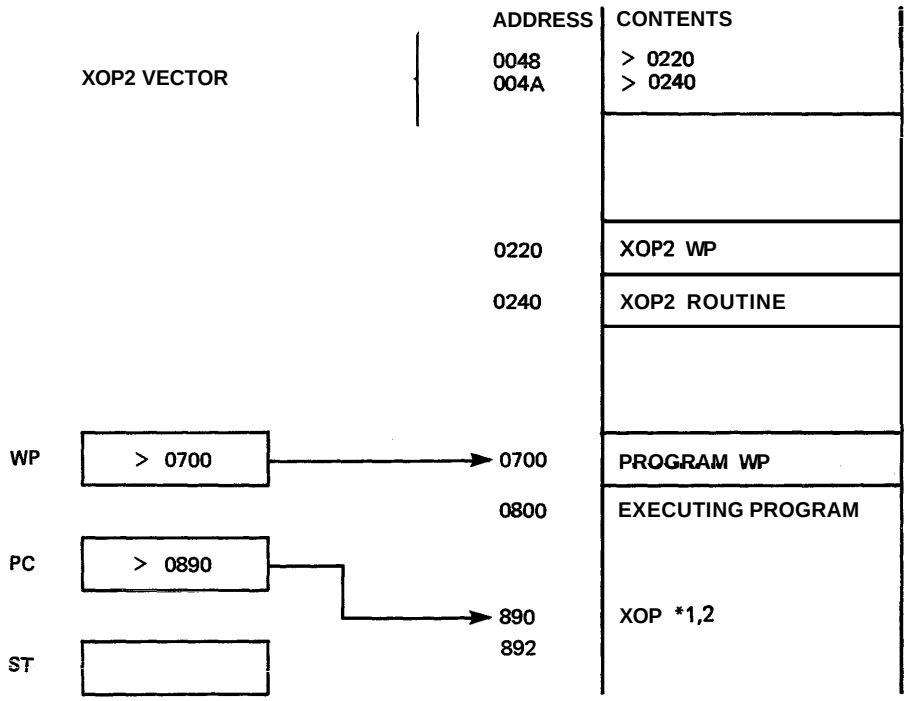


Figure 8–26 State Before Executing the XOP 2 Instruction

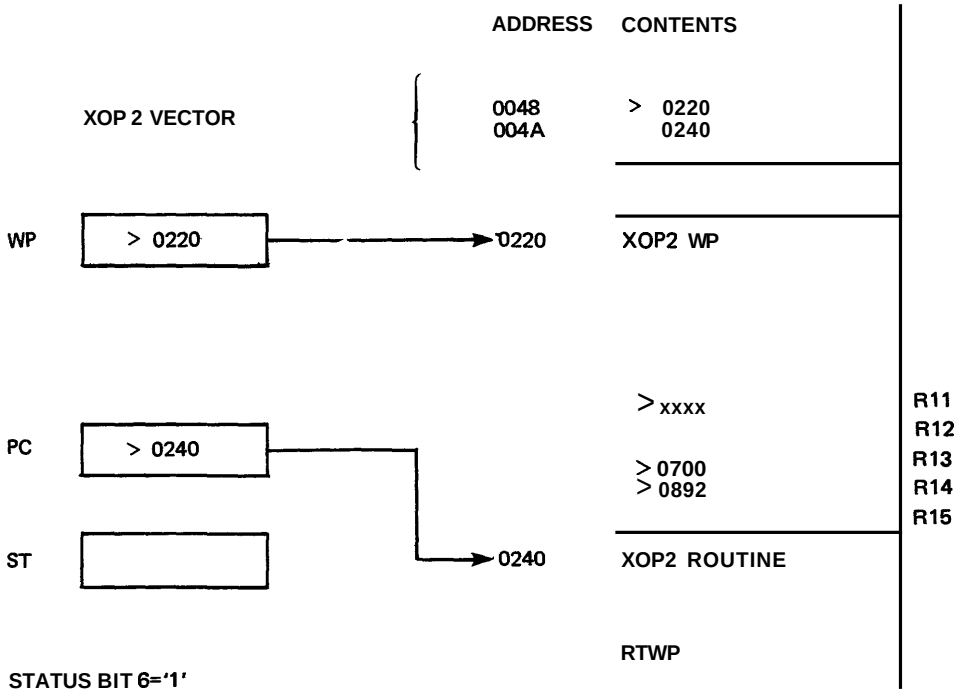


Figure 8–27 State After Executing the XOP 2 Instruction

8.12 9900199000 FAMILY

The 9900/99000 family of microprocessors gives a choice of different cost/performance/environment options using the same software. Because of the nature of some of the options (eg the 9995 is designed for use as a microcontroller) there are small differences in architecture which are outlined below.

Modifications to assembly language software to run on a different processor in the family are usually quite straightforward. For high level language (eg Pascal) programs the differences will be taken care of within the Rx executive.

8.12.1 TMS9900

- NMOS technology
- 16 bit data bus
- 3MHz
- 3 power rails (+5V, -5V and +12V)
- 4 phase clock
- 64 pin package
- Up to 64K byte address space
- 16 prioritized interrupts
- Memory-to-memory architecture
- 3 dedicated registers - PC, WP and ST
- 16 general registers - R0 to R15
- Workspace register set - any 32 byte block of RAM
- 5 workspace register addressing modes
- 16 extended operation instructions (XOPs)
- Serial 1/0 via CRU - up to 4K bits
- 3 single bit and 2 multiple bit CRU instructions
- Automatic context switch for interrupts, XOPs and subroutines
- 69 instructions, includes hardware multiply (MPY) and divide (DIV)
- DMA capability
- 5 external instructions

8.12.2 SBP9900A

- Integrated injection logic (I2L) technology
- Fully static operation
- Single phase clock
- Up to 3MHz at 500mA injector current
- Approved to MIL standard 883B and BS9000

- Single power rail
- *64* pin package

8.12.3 TMS9980A

- 8 bit data bus
- Up to 16K byte address space
- 4 prioritized interrupts
- On chip 4 phase clock generator
- 40 pin package

8.12.4 TMS9981

As for the 9980A except:

- No -5v rail required
- On chip crystal oscillator

Note: The TMS9981 has a different pin out to the TMS9980A.

8.12.5 TMS9995

- *8* bit data bus
- On chip oscillator and clock generator
- Single +5V power rail
- *40* pin package
- Optional automatic first wait state generation
- 12MHz (internally divided by *4)*
- On chip RAM (256 bytes) organised as 16 bit words
- On chip decrementer/event counter
- 5 prioritized interrupts
- Macro Instruction Detect feature
- Arithmetic overflow interrupt
- Up to 32K bits of serial I/O via CRU
- Minimum memory cycle time of 333ns
- Instruction pre-fetch
- CRU flag register (16 bits)
- Signed multiply (MPYS) and divide (DIVS)
- Load WP and ST from register (LWP and LST)

8.12.5.1 Macro Instruction Detect

The Macro Instruction Detect (MID) feature enables the user to extend the instruction set in a similar way to the XOP instructions,

An XOP instruction, which is a valid 9900 assembly language instruction, occupies a range of opcodes: for example, the

'XOP 0' instruction uses opcodes >2C00 to >2C3F; the 'XOP 1' instruction uses >2C40 to >2C7F; etc, When the processor encounters an XOP instruction it evaluates the address of the XOP instruction's vector, uses the least significant 7 bits of the instruction to determine the address of the source operand, stores this address in the XOP's workspace register 11, and performs a context switch to the appropriate routine. (Full details on XOPs is given in section 8.11.)

With the MID feature, the user can implement some, one, or all, of the undefined instruction opcodes (such as the opcodes >0000 to >007F) in software. When an undefined opcode (a MID opcode) is encountered by the 9995 processor, a non-maskable level 2 interrupt is generated. This causes the processor to perform a context switch using the interrupt level 2 vector. The level 2 interrupt handler must identify which software routine actually implements the particular opcode and then pass control to that routine. A routine may implement a single opcode, or a range of opcodes (like the XOP instruction). This is totally up to the user to decide when designing the level 2 interrupt handler and its callable routines.' The MID opcode instruction can he accessed by:

```
MOV  @-2(R14),temp  Copy opcode into TEMP
```

As the processor stores the incremented program counter when the context switch takes place, a simple RTWP instruction returns control to the interrupted program at the instruction following the MID opcode.

If any MID opcode instructions are executed in the level 2 interrupt handler itself then care must he taken to ensure that the original program context is not lost, and also that the handler does not cycle endlessly.

8.12.5.2 Arithmetic Overflow

The user can cause the processor to generate an arithmetic overflow interrupt (a level 2 interrupt) whenever an instruction sets the arithmetic overflow status bit (status bit 4). This is done by setting the arithmetic overflow interrupt enable status bit (status bit 10) to a '1 and enabling level 2 interrupts via the processor's interrupt mask. Both of these operations can be performed using the 'LST register' instruction.

8.12.5.3 Test for MID or Arithmetic Overflow

The MID interrupt and the arithmetic overflow interrupt both generate level 2 interrupts (they share the same interrupt vector). Thus, when a level 2 interrupt is taken by the

processor, the level 2 interrupt handler must determine what actually caused the interrupt. Was it a MID? Or was it an arithmetic overflow? When this has been decided the appropriate routine can be invoked. (Note: Before control is returned to the interrupted program, the interrupt must be reset, otherwise the level 2 interrupt handler will be immediately re-taken.)

If the MID flag (at on chip CRU software base address >1FDA) is a '1' then a MID caused the interrupt (this is reset by writing a '0' to the MID flag) otherwise it was an arithmetic overflow (this is reset by masking the arithmetic overflow status bit to a '0').

8.12.5.4 On Chip CRU Flag Register

The CRU flag register consists of 16 read/write CRU bits (named FLAGO, FLAGl, ..., FLAGF) starting at a CRU software base address of >1EE0. The first 5 of these flags (FLAGO to FLAG4) are used internally, but the remaining 11 are user definable.

8.12.5.5 On Chip Decrementer/Event Counter

The decrementer can be configured as either a timer or an event counter using FLAGO, and enabled/disabled using FLAGl. When FLAGO is set to '0', the decrementer functions as a timer, and when it is set to '1' it is an event counter (the level 4 interrupt line is used as the input for the event counter). If FLAGl is set to '0', the decrementer is disabled, but if it is a '1', the decrementer is enabled to generate a level 3 interrupt.

The decrementer is configured by:

- Set FLAGO to the required mode.
- Load the required 16 bit start count into the decrementer register (this is located at memory address >FFFA). In timer mode, the count is decremented every fourth CLKOUT cycle (every 1.333us). (A count of >3A98 gives a 'delay' of 20ms, while a count of zero disables the decrementer.) When the count reaches zero, a level 3 interrupt is generated, the original count is reloaded and decrementing continues,
- Enable the decrementer by setting FLAGl to '1'.
- Enable level 3 interrupts by setting the interrupt mask to 3 or higher,

Note: The 256 bytes of internal RAM is distributed as 252 bytes from address >F000 to >F0FB, and 4 bytes from >FFFC to >FFFF. (These last 4 bytes are the two-word LOAD vector.) This RAM can not be switched out of the address map. The internal RAM is automatically selected when any of the above addresses are referenced, regardless of what is located at these addresses off chip.

8.12.6 SBP9989

- Integrated injection logic (I2L) technology
- Fully static operation
- Single phase clock
- Up to 4.4MHz at 500mA injector current
- Conforms to MIL standard 883B
- Single power rail
- 64 pin package and chip-carrier 68 pin
- Multiprocessor interlock signal (MPILCK)
- Extended instr. processor present signal (XIPP)
- Interrupt acknowledge signal (INTACK)
- Arithmetic overflow interrupt
- Memory map enable signal (MPEN)
  - to drive TIM99610 memory mapper chip
  - as an extra address bit for 2 * 64K byte pages
- Signed multiply (MPYS) and divide (DIVS)
- Load WP and ST from register (LWP and LST)

8.12.6.1 MPILCK

In an environment consisting of a number of microprocessors, where some sharing of the system memory is necessary (if only for the microprocessors to communicate with each other) there is a possible software memory contention problem: one, or more, processors are attempting to read the contents of a piece of memory while another processor is attempting to modify it. While the piece of memory is being read from, no processor should be allowed to modify it. Similiarly, while the memory is being written to, no processor should be allowed to read it.

This problem is more acute if the memory location in question is used to allow or inhibit access to another piece of memory (in software, such a memory location is known as a semaphore).

What is required is some mechanism that implements a 'test and set' operation in an indivisible manner while also inhibiting access to the semaphore. This is performed via the MPILCK (multiprocessor interlock) signal, which is generated whenever the ABS instruction is executed. If the semaphore is initially set to >FFFF to indicate that it is not in use, exclusive access to the piece of memory can be

guaranteed by:

```
test  ABS  semaphore     Is the semaphore in use?
      JGT  test          +ve - semaphore in use
```

The ABS instruction 'converts' a negative value into a positive value and sets the status bits according to the original value. If the semaphore is not in use (contains the negative value -1), the ABS instruction resets the semaphore value to 1 and resets the arithmetic greater than status bit to '0'; program control will 'drop through' the JGT instruction. When the semaphore is in use (contains the positive value 1), the ABS instruction simply sets the arithmetic greater than status bit to '1'; program control will be sent hack to the 'test instruction',

When a processor has finished with the piece of memory, the semaphore is reset to >FFFF (the semaphore is not in use),

8.12.6.2 XIPP

The extended instruction processor present (XIPP) signal is the same as the attached processor present signal used in the 99000 family processors, It works in a similar manner to an attached processor using the MID feature (except that the 9989 does not have a macrostore). This is defined below in sections 8.12.7.1 and 8.12.7.2.

8.12.6.3 INTACK

The interrupt acknowledge (INTACK) signal allows the 9989 to acknowledge the presence of an interrupt during times when it has handed over control of the system bus to an extended instruction~processor,

8.12.7 TMS99000 Family

- Scaled NMOS (SMOS) technology
- Multiplexed 16 bit address and data bus
- Single +5V power rail
- Up to 24MHz (internally divided by 4)
- 40 pin package
- On chip oscillator and clock generator
- Minimum memory cycle time of 167ns
- Instruction pre-fetch
- Privileged mode
- Bus status codes to identify processor activity
- Multiprocessor interlock signal (MPILCK) via bus status codes
- Multiprocessor support instructions - test memory bit (TMB), test and clear memory bit (TCMB), and

test and set memory bit (TSMB)
- Macrostore emulation of user defined instructions
- Attached processor present signal (APP) - with access to PC, WP and ST registers
- Macro Instruction Detect feature
- Arithmetic overflow interrupt
- Interrupt acknowledge signal (INTA)
- Up to 16K bits of serial 1/0 via CRU
- Up to 16R bits of parallel I/O via CRU
- Optional automatic first wait state generation
- Memory map enable signal (ST8) to drive TIM99610 memory mapper chip
- Memory expansion instructions via macrostore - load map file (LMF), long distance source (LDS), and long distance destination (LDD)
- Signed multiply (MPYS) and divide (DIVS)
- Load WP and ST from register (LWP and LST)
- Stack support instructions - branch and push link to stack (RLSK), and branch indirect (BIND)
- Double precision 32 bit instructions - add double (AM), subtract double (SM) , shift left arithmetic double (SLAM) and shift right arithmetic double (SRAM)

8.12.7.1 Macrostore

In the 99000 family, the concept behind the MID (the ability to define 'new instructions' that are implemented in software) has been extended to allow these routines to be stored in a high-speed memory that is addressed independently of main memory. This high-speed memory (minimum cycle time of 167ns) is known as **macrostore**.

When a MID opcode is detected by the processor, program control is transferred to the macrostore.

The first few words of the macrostore contain a specially ordered table. Each entry in this table defines the macrostore address of the routine that implements a particular group of MID opcodes, This address table is used to determine whether the MID opcode is, in fact, implemented by a macrostore routine. If so, program control is passed to the appropriate routine. If not, a level 2 interrupt is generated. Although a special internal, 16 word, workspace (this is known as macrostore RAM, or MRAM) is used when the processor is executing out of macrostore, it is a simple matter to access data in the user's main memory. When the macrostore routine has completed, an exit is made from the macrostore (program control is returned to the user's program) via an RTWP instruction.

If the user defined instruction allows the standard addressing modes (register, register indirect, symbolic, etc) for the source and/or the destination operand then the

appropriate MID routine must calculate the operand's actual address, (This is automatfcaily performed by the microcode for the standard instructions,) To save the overhead of having to do this calculation in software, the 99000 family sf processors provide the EVAD (evaluate address) macrostore instruction.

Internal to the 99000 family processors is a 1K byte macrostore ROM (MROM) which can be expanded to 61K bytes using off chip high-speed ROM, PROM, or even RAM,

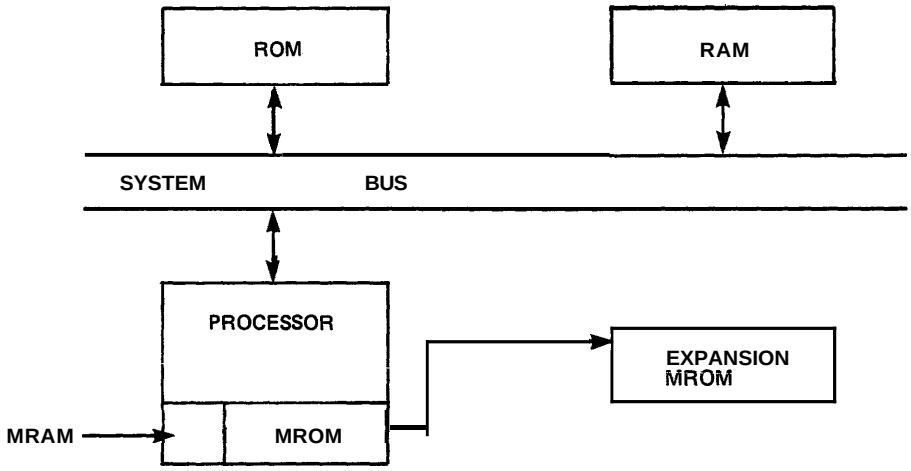


Figure 8-28 Macrostore

The macrostore can be addressed in three different modes:

- o Standard mode - The on chip MROM and MRAM are both enabled, This allows the software routines in MROM to be used.
- o Prototyping mode - The MROM is disabled but the MRAM is enabled. This allows the user to re-configure the system so that a lk byte block of the off chip macrostore is used as though it was the MROM. This enables the user to try out and test the macrostore routines before committing them to mask.
- o Baseline mode - All macrostore is disabled. Only the baseline 99000 instruction set can be executed; with the exception of the parallel CRU instructions this is identical to that of the 9995.

8.12.7.2 Attached Processors

To increase system throughput, some of the macrostore routines can be taken out of the macrostore and implemented in an attached processor (a specially designed unit to

handle a particular function) which is attached to the system via a special interface. If these routines are frequently used, or relatively slow and complicated (such as floating point arithmetic routines), then a considerable speed improvement will he noticed. (Floating point routines, for example, could be replaced by a high-speed floating point processor,)

When the system processor encounters a MID opcode it outputs a MID bus status code. Any attached processor that recognises the MID opcode can then inform the system processor that it is prepared to execute the opcode (using the attached processor present signal), If this happens, the system processor relinquishes the bus to the attached processor and waits until the attached processor signals that it has finished.

Before giving up the bus, the system processor copies its internal WP, PC and ST registers into RAM, When it regains control of the bus these hardware registers are reloaded from RAM, This allows the attached processor to access the user's workspace, to access any multiple word operands (updating the PC to skip over these operands as necessary) and to return status information.

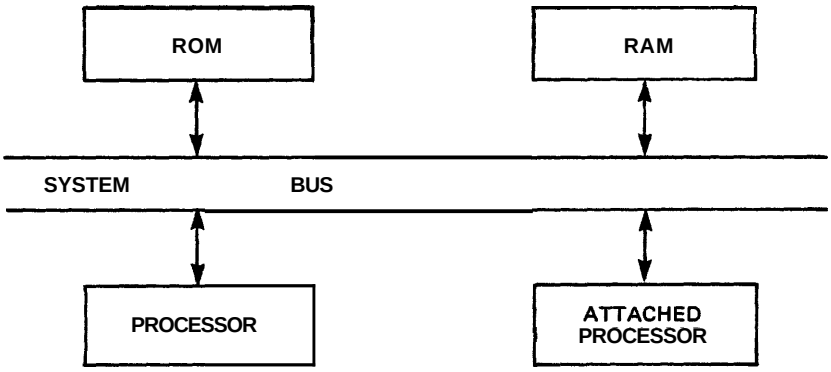


Figure 8-29 Attached Processor

Unfortunately, attached processors can not simply be attached to a high-speed bus without limit. They are not completely self-contained computing systems as they require the services of the system bus (to access memory, for example), and they operate by suspending (or blocking) execution of the main program until they have completed their operation, Even so, an attached processor can increase the system throughput for specific operations by 10 to 100 times.

8.12.7.3 Attached Computers

Attacked computers, on the other hand, only require the services of the system bus infrequently (when the macrostore instruction invokes them with the required parameters, for some hand-shaking signals and for completion signaling).

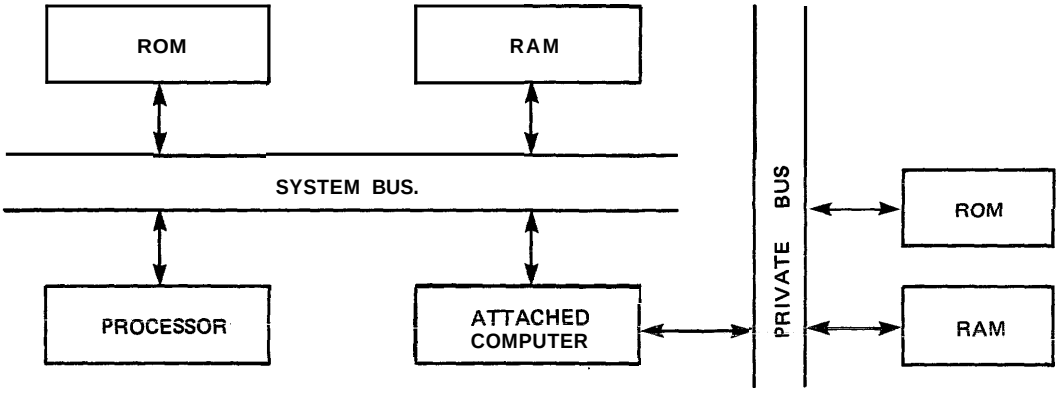


Figure 8-30 Attached Computer.

As attached computers are totally self-contained systems, no blocking action is necessary, which means that they can execute in a true parallel fashion. An attached computer can increase the system throughput for particular operations up to 1000 times.

The complete procedure when a MID opcode is encountered by the processor is shown in Figure 8-31.

8.12.7.4 Interrupts

All interrupts (except RESET) are inhibited while executing from macrostore. However, there are two instructions that allow the user to test for any pending interrupts while executing a routine in macrostore. Using these, MID opcodes requiring long execution times can be written so that they can be interrupted and resumed after the interrupt has been serviced. If the MID opcode is being handled by an attached processor when a pending interrupt is detected, the attached processor can temporarily return control to the system processor to handle the interrupt. Upon completion of the interrupt servicing, the system processor returns control back to the attached processor. (When the interrupt is

taken by the system processor it automatically outputs the interrupt acknowledge bus status code, INTA, which can be used to reset the interrupting device.)

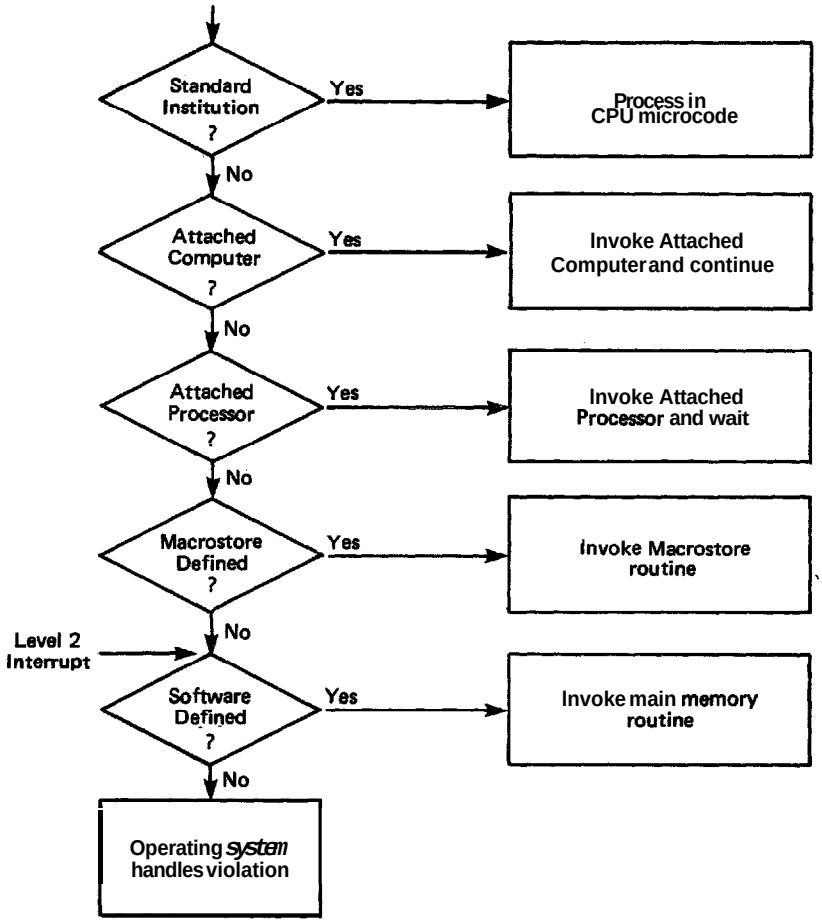


Figure 8-31 Full TMS99000 Instruction Sequence

8.12.7.5 MPILCK

In a multiprocessor environment where communication is performed via shared memory it is necessary to have a mechanism that allows a portion of memory to be exclusively 'owned', so that while one processor is accessing that portion every other processor in the system is physically inhibited from accessing it. This is guaranteed via the multiprocessor interlock (MPILCK) bus status code and the multiprocessor support instructions (TMB, TCMB and TSMB); the ABS instruction can also be used.

8.12.7.6 CRU Operations

On the 99000 family of processors, CRU operations use bits 0 to 14 of register 12 (instead of just bits 3 to 14 with the TMS9900). This expands possible CRU I/O operations from the

previous maximum of 4K bits (with the TMS9900) to a new maximum of 32K bits. The 32K bfts is split into two 16K bit blocks; the first block (0 to 16K) is used for serial I/O transfers, and the second block (16K to 32K) is used for parallel I/O transfers. (If the most significant hit of register 12 is set to a '1' then a parallel transfer is indicated otherwise it is a serial transfer.)

For parallel CRU operations, the count supplied to the LDCR and the STCR instructions is used to select either an 8 or a 16 bit transfer and also to specify whether or not the CRU base address is to be incremented by 2 after the transfer has been performed. (With serial CRU operations, the count is used to specify how many bits are to be transferred.) The possible valid values for the count, using parallel CRU, are shown below:

	Binary Count	
Byte transfer	0010	R12 not altered
	0011	R12 post incremented by 2
Word transfer	1010	R12 not altered
	1011	R12 post incremented by 2

All other values for count are reserved for future expansion of the parallel CRU capability and should not be used.

When operating in user mode (status bit 7 is set to '1'), an attempt to execute an LDCR or an STCR instruction using a CRU base address in the range >1C00 to >7FFE or >9C00 to >FFFE is flagged as a privileged opcode violation. (This condition generates a level 2 interrupt request and also inhibits transfer of the remaining bits.)

Note: The SBO, SBZ and TB instructions should be used with caution when an access is made within the parallel CRU address space. SBO and SBZ will set/reset the CRUOUT line (the same line as data bit D15), while the other 15 bits (DO to D14) will be undefined. TB takes its value from the CRUIN line (the same line as data bit DO).

There will be different versions of the 99000, each supporting an extended instruction set, implemented in the macrostore. These instruction sets will be tailored to particular requirements, eg:

99105 Baseline version, instruction set as 9995, no macrostore
99110 High performance floating point package
99120 Realtime executive (Rx) kernel

8.13 ALGORITHMS AND TECHNIQUES

The paragraphs that follow provide information about algorithms and techniques that are applicable to 9900 assembly language programming.

8.13.1 Invoking the 9900 Family of Assemblers

The 9900 family of assemblers are upward compatible. However, there are restrictions on the use of certain instructions. The first three instructions below are only valid on the 990/10 or /12 minicomputers with map option. The remaining five instructions (external instructions) perform specific functions on the /10, /12 and the /4 minicomputers. Although they are not illegal for the TMS9900 microprocessor, the functions they actually perform are dependent upon the external hardware.

Long distance destination	LDD
Long distance source	LDS
Load memory map file	LMF
Clock off	CKOF
Clock on	CKON
Idle	IDLE
Load ROM and execute	LREX
Reset 1/0	RSET

8 1 3 1 1 LRLA

The Line-By-Line Assembler is a two-EPROM package that is used in conjunction with the TIBUG monitor supplied with the TM990/101 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 (9E6 in some versions)
?	E

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 TM990/402 Line-By-Line Assembler User's Guide.

8.13.1.2 SYMBOLIC

SYMBOLIC is a ROM resident two-pass assembler (see footnote) that is supplied with the TM990/302 Software Development Board. It takes source statements stored on audio cassette (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 Relocating Loader to load the absolute program into the board's user memory.

A two-pass assembler reads the source program twice. 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 instruction opcodes and the completed symbol table.

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 (the DP command), setting the program counter, workspace pointer and status register to the appropriate values using the **IR** command, and then issuing the EX command.

See the **TM990/302** Software Develpoment Board User's Guide for further details.

8.13.1.3 TXMIRA

TXMIRA is a two-pass assembler that runs on a **990/4** minicomputer under the floppy disc based TXDS Control Program. The assembler is invoked by replying to the Control Program prompts as follows:

TXDS Control Program Prompts	User Replies	
PROGRAM:	**DSCX:TXMIRA/SYS**	return
INPUT:	**DSCX:NAME/ASM**	return
OUTPUT:	DSCX:NAME/OBJ,DSCX:NAME/LST	return
OPTIONS:		return

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. The second output 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 or TXSLNK) must be used to resolve any external references **(REFs)** contained in the program.

If the program has been written to run on a TM board based

system then it may be possible to test and debug it using TIBUG or the Software Development Board. However, the AMPL in-circuit emulator (if one is available) could make the testing a lot easier, simpler, quicker and less painful.

If the program has been written to run on a /4 then there are two options available. If it doesn't use any operating system facilities then the EX or RU commands of the TXDS Standalone Debug Monitor (TXDBUG) can be used. If it does use operating system facilities and if the Operator Communications Package (OCP) has been included in the generation of the /4 operating system (using GENTX) then OCP may be used.

For a program to run on the /4 the first three words of the program must contain (in the following order):

1) The address of the initial workspace.

2) The address of the program's entry point.

3) The address of the error handling routine to be invoked when the operating system detects a non-fatal error. If this address is less than 15 then it is assumed that an error handler is not included in the program.

As the 990/4 minicomputer is based around the TMS9900 microprocessor it is possible to use the AMPL in-ciruit emulator to debug a /4 based program. Note: there can be timing problems with the host cpu.

8.13.1.4 SDSMAC

SDSMAC (Software Development System Macro Assembler) is a multipass macro assembler that runs on a 990/10 or /12 minicomputer under the hard disc based DX10 operating system. This assembler is invoked by issuing an 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** or **/12** it is a good idea 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) or even the TX Link Editor (TXXLE command) to resolve all external references in the assembled program.

For a TM board based or for a **990/4** based program refer to the relevant comments under TXMIRA above.

For a **990/10** or **/12** minicomputer the fully linked (if necessary) program must be installed as either a procedure, task or overlay (using the IP, IT or **I0** commands). (For most applications the program is usually installed as a task.) This can then be executed using the XT (execute task) command, or debugged using the XD (execute debug) command and the SCI debugger commands.

The first three words of the **990/10** or **/12** based program must contain task information; this is the same as for a **990/4** based program and is described under TXMIRA.

8.13.2 Number Representations

The information in this subsection discusses how numbers are formed and how they are stored internally. Note: The **TMS9900** performs all arithmetic using twos complement notation; it does not contain any instructions that directly manipulate fractional, floating point or binary coded decimal numbers. If a program needs to use these types of number systems, then the user must supply the routines to actually perform the required arithmetic operations. It will also be necessary to provide the routines to convert between the required number system and the twos complement

form, The TMS99110 provides floating point instructions in macrostore,

8.13.2.1 Number Systems

A number in the decimal, base 10, system is composed of the digits 0 to 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

$$2*10^3 + 9*10^2 + 7*10^1 + 6*10$$

Note: $10^0 = 1$

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: $2^0 = 1$

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.**

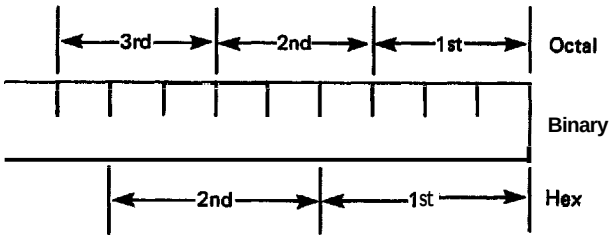


Figure 8-32 Bit Grouping

1001111111011010 => 1001 1111 1101 1010 => 9FDA

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.

8.13.2.2 Representation of Negative Numbers

Negative numbers are stored in twos 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', it is negative. The other 15 bits (bits 1 - 15) hold the twos complement value of the number. For a positive number this is simply the binary representatdon 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. (16 bits for the 9900.)

1096 ⟶ 0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 0

2) Take the ones 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 hit.

```
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.

8.13.2.3 Representation of Fractions

The general equation to convert a binary fraction into its decimal equivalent is:

$$0.d1\ d2\ \ldots\ldots\ dn = d1*2^{-1} + d2*2^{-2} + \ldots\ldots + dn*2^{-n}$$

where dl ..... 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.

8.13.2.4 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: a sign bit, a 7 bit exponent and a 24 bit mantissa:

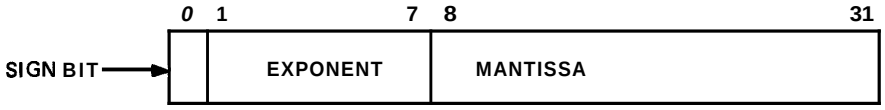


Figure 8-33 Floating Point Format

The sign bit (bit 0 of the first word) is used to show whether the number is positive or negative (a '1' means that it is negative). A real number is converted into the form 'fraction*exponent'. The fractional part is stored in the 24-bit mantissa field in true form and not twos complement. The exponent part is stored in the exponent field in "Excess 64 notation".

The most significant hex digit of the mantissa must be normalized (ie it must contain a value other than zero). This is performed by shifting the number four bits to the left (one hex digit) and decrementing the exponent value by one until the mantissa is normalized.

Excess 64 notation means that the number stored in the exponent field is 64 greater than the actual value of the exponent part. Thus, the true exponent values 0 to 63 will be stored as 64 to 127. The exponent field values 0 to 63 are used to represent the true exponent range of -64 to -1.

Consider the number -107.5

Binary Form	Frac*Exp Form	Normalised
01101011.1000	$0.0110101110000 * 16^2$	No change

In floating point form 1 1000010 0110101110000....0

The number -107.5 would be stored as >C26B8000 (sign = -ve, exponent= +2).

Consider the number 0.03125

Binary Form	Frac*Exp Form	Normalised
0.000010000	$0.000010 * 16^{0}$	$0.10000 * 16^{-1}$

In floating point form 0 0111111 1000000000000....0

The number 0.03125 would be stored as >3F800000 (sign= +ve, exponents -1).

8.13.2.5 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).

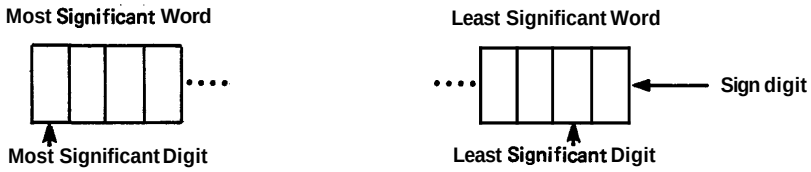


Figure 8-34 A Possible BCD Format

8.13.3 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. Typically, if the program is loaded at any other address than the program will not execute correctly.

However, it is possible to write a program such that without any modifications at all it will execute at any position in memory. A program that exhibits this form is said to be written in Position Independent Code. (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.)

The real value of position independent code may not be immediately obvious so consider the following: You have an EPROM based monitor (like TIBUG) and want to add new capabilities to it (say an assembler, a disassembler and a

floating point package) and these are also going to be EPROM based. Where are these extra EPROMs going to be placed in memory? At the same address? Possibly, but this would require you to power down your system, remove the unwanted EPROM(s) and insert the required ones. And then you would have access to only one of these extra facilities at any one time, At different addresses? This would be better as it allows access to all of the new features at any time.

After a while, you could have built up a healthy selection of extra monitor facilities and a number of useful application packages. The only problem is that all of them are specific to some particular address, What happens when you want to use a combination of these packages and extra facilities? It is quite likely that you will have an address clash (two packages requiring the same memory address) and it will become necessary to go hack and re-assemble one of them (taking great care that another address clash doesn't happen), Now you've got two versions of a piece of software that only differ in their load addresses. Nothing wrong with this but it does mean that any updates (a bug corrected or new facilities added) must be applied to both pieces of software, This leads to a proliferation of near identical parts and that is a real headache from a maintenance point of view,

If the packages are written in position independent code then only one copy of a package is ever required, When one of the packages is wanted its EPROM(s) are simply inserted in any unused memory space, A package is then invoked with the address of the package's EPROM(s) as the start address.

The calling sequence for position independent code is shown below, along with the relocatable code equivalent,

```
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, This is obtained by:

```
START EQU   $
      LI    R10,>045      Load R10 with RT instruction
      BL    R10           Execute instruction in R10
ENTRY EQU   $             R11 contains address of ENTRY
      MOV   R11,R4        R4 contains address of ENTRY
```

Note: START is the real entry point for the position

independent code program.

8.13.4 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 (see Figure 8-35).

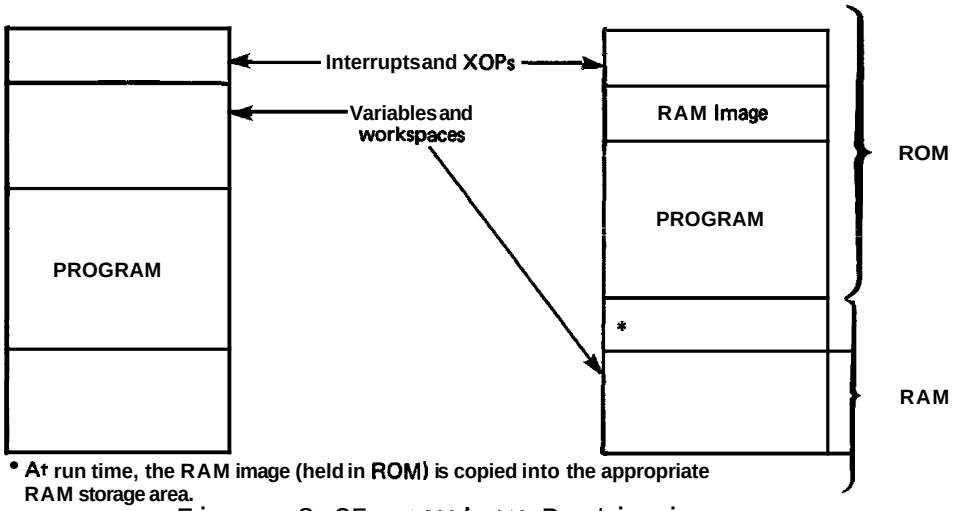


Figure 8-35 ROM/RAM Partioning

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 RSS   0
*
*
*  Initial variable values in ROM

VALUES DATA  100
       DATA  25
         .
         .
       TEXT  'READY '
         .

*
*
*  Initialisation loop

ENTRY  EQU   $
       LI    R1,TEMP1       R1 points to TEMP1
       LI    R2,VALUES      R2 points to 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
       MOV   R1,@TEMP1     Set TEMP1=100
       LI    R1,25
       MOV   R1,@TEMP2     Set TEMP2=25
         .
```

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,

Note: A complete ROM/RAM system must satisfy the following three conditionds.

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 PC to access this routine. It may be necessary to allocate some RAM to each unused interrupt level's WP, but this depends on exactly what the spurious interrupt handler does.

If any XOPs are used then the appropriate XOP trap vectors must be included.

If the LOAD vector is not used then it should be treated as though it was an unused interrupt level. Typically this vector is used to perform a'warmstart' operation; it allows the user to halt the application program (usually when an error has been detected) and for it to be restarted from a known state (eg immediately before the code that copies the RAM image into memory).

8.13.5 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 is almost, but not exactly, the same. 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 but to the macro processor it is simply text. The macro processor is only concerned with macro related operations, and source lines containing none of these are output unchanged. 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 their replacement text and all other macro operations removed.

Diagrammatically, this can be expressed as:

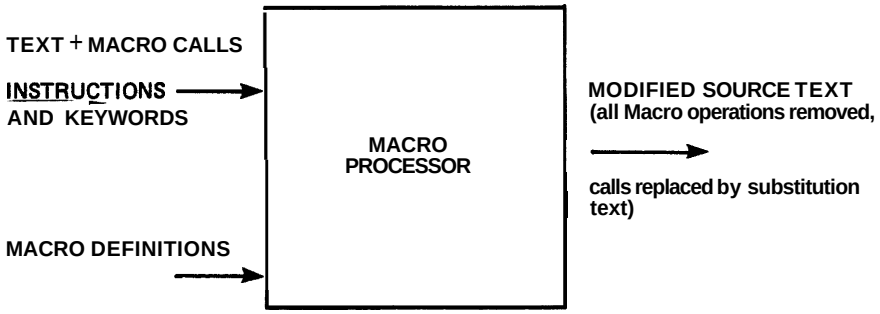


Figure 8-36 Macro Processor Operation

A macro processor has two phases: Macro Definition and Macro Expansion.

Macro Definition - A macro is defined and subsequently included into its macro library,

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 he placed in the object module at the point of the call.

The SDSMAC assembler supports a macro language (ie it is 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.

8.13.5.1 Macro Definition

Macro definition is performed by the $MACRO instruction. All source lines following this instruction up to but excluding the definition terminator ($END instruction) constitute a macro.

```
Mname   $MACRO   parm
        .        }
        .        } Macro
        .        }
        $END
```

MNAME is the name of the macro. PARM is the list of parameters (separated by commas) used by the macro.

$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.

8.13.5.2 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, *R6 the following is required:

```
        AGAIN    R4,*R6
```

SDSMAC supports conditional assembly through the $IF, $ELSE and $ENDIF 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 is:

```
$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

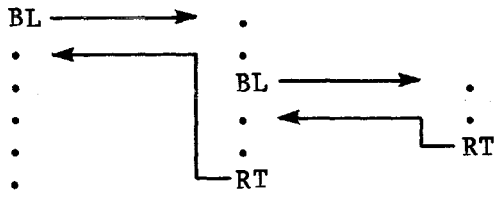
SDSMAC also supplies a number of keywords such as $PCALL (parameter appears as a macro instruction operand) and $PIND (parameter is an indirect workspace register address) that enable the programmer to test a variable's attribute component. These keywords are used with the logical operators AND ('&'), OR ('++'), Exclusive OR ('&&') and NOT ('#'). For example:

P2.A & $PCALL	This expression has a noii zero value when the variable P2 is a parameter supplied in a macro instruction; esle the value is zero.

8.13.6 Nested Subroutines

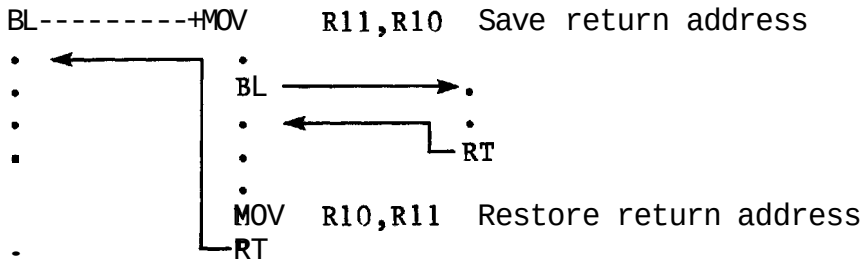
A subroutine is nested when it is invoked by another subroutine. The only problem with nested subroutine calls is that of ensuring that a subroutine's return address is not lost or overwritten. This is particularly troublesome if the subroutines are called via a BL instruction (the return address is stored in workspace register 11).

Conceptually the flow of control is as follows:



Executing the second RL 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:



In the above piece of code, the instructions:

```
MOV    R10,R11
RT
```

can he replaced by:

```
BL     *R10
```

8.13.7 Stacks

Another way of performing this saving and restoring of return addresses is by implementing a stack mechanism, An area of memory is set aside to be used 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 (ie it points to the last item added to the stack). This register is usually referred to as the stack pointer, A stack can be represented graphically by:

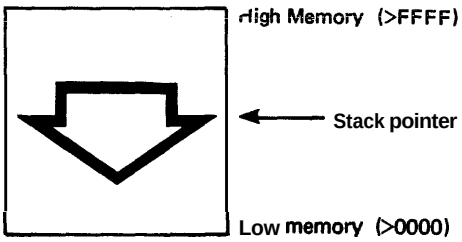


Figure 8-37 Stack Representation

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 he 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 plus two; otherwise the first word in the stack will not be used,

8.13.8 Recursion

A nested subroutine has already been defined as a subroutine that is called by another subroutine. In this definition 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 not perform recursion endlessly,

Recursion presents problems, For example, how is a subroutine's return address to be saved? 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.

8.13.9 Re-entrancy

For code to be re-entrant the following two conditions must be satisfied,

The subroutine code must not modify itself. Modifying code is an extremely dangerous practice; it is very difficult to debug and is actively discouraged. Storing the code in ROM ensures that this can not be done, If self modifying code is included then the program will not work as expected,

On entry to the subroutine, the data local to the subroutine must be correctly initialized. This also implies that the data local to previous invocations must be preserved, and restoredon exiting the routine. The simplest way of performing this is using a stack:

```
ENTRY  EQU  $
       PUSH R11          Save return address
       PUSH @ARG1        Save ARGl
       PUSH @ARG2        Save ARG2
       •
       •
       PUSH RO           Save RO
       LI   RO,...
       MOV  RO,@ARG1     Reset ARGl
       LI   RO,...
       MOV  RO,@ARG2     Reset ARG2
       •
       •
       POP  RO           Restore RO
       •
       POP  @ARG2        Restore ARG2
       POP  @ARG1        Restore ARG1
       POP  R11          Restore return address
       RT
```

Note: The stacked items are popped in reverse order. PUSH and POP are macros as defined in section 8.13.7.

8.13.10 Automatic Workspace Allocation

Transparent stacking of workspaces can he achieved by calling all subroutines through a special purpose XOP named CALL, defined below. 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.

```
*
* CALL XOP
* This routine automatically stacks workspaces down
* through memory.  An 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 wksp's R13
        A     R13,R1         Pt to new wksp's R13
        MOV   R13,*R1+       Move return WP
        MOV   R14,*R1+       Move return PC
        MOV   R15,*R1+       Move return ST
        MOV   R11,R14        Get Subroutine's entry pt
        AI    R13,-32        Hit next wksp
        RTWP                 Call subroutine
```

An example of using this routine follows:

```
XOPWP   EQU   >FF00          Assign wksp
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
        .
        .
SURR    EQU   $              SUB's entry point
        .
        .
        RTWP                 Return to caller
```

Another way of implementing this stacking mechanism is shown below, This method assumes that register 7 contains the address of a BLWP vector (this vector is built in RAM at run time as the workspace address field of the vector must be updated after each call). A routine is invoked by issuing a BLWP *R7 instruction (in the code this the CALL$ DATA word).

```
CALL$   EQU   >417           BLWP *R7 Instruction
        RORG
STACK   BSS   stacksize*2    Allocate space for stack
WP1     BSS   32             Initial workspace
        .
CALLVEC EQU   $              Call handler vector
NXTWP   BSS   2              Next WP to be allocated
HNDLR   BSS   2              Entry pt for call handler
```

```
*
* Routine entry - set up call handler vector

        LI   R1,CALLVEC       Ref vector
        MOV  R1,R7            Save address of vector
        LI   R2,ENTRY         Ref handler
        LI   R3,WP1-32        Ref 1st stack WP
        MOV  R3,*R1+          Set NXTWP
        MOV  R2,*R1           Set HNDLR
        .
        DATA CALL$,SUBR       Call SUBR(shown above)
        .
ENTRY   EQU  $                Call handler entry point
        MOV  @7*2(R13),R7     Get address of CALLVEC
        AI   *R7,-32          Set address of next WP
        MOV  *R14+,R11        Get routine's entry
        RT                    Invoke routine
```

Only minor modifications are required to either implementation to allow a user stack to be incorporated; this would also allow a simple check to be made to determine if stack overflow has occurred (stack overflow checking is not performed in either mechanism above). For the CALL$ version this is shown below.

In the initialization loop:

```
        LI   R8,STACK         Set user stack start addr
```

ENTRY now becomes:

```
ENTRY   EQU  $                Call handler entry point
        MOV  @7*2(R13),R7     Get address of CALLVEC
        MOV  @8*2(R13),R8     Get address of user stack
        AI   *R7,-32          Set address of next WP
        C    R8,*R7           Overflow?
        JH   error            Y - error
        MOV  *R14+,R11        Get routine's entry
        RT                    Invoke routine
```

Pictorially this can be shown:

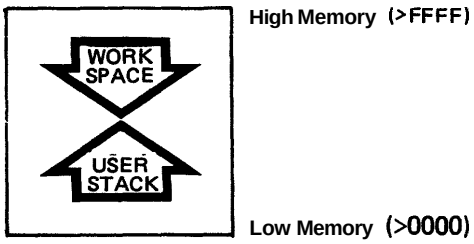


Figure 8-38 A Stack/Workspace Allocation Implementation

Workspace allocation starts from high memory and builds down towards low memory; NXTWP contains the address of the next workspace to be allocated. The user stack starts at low memory and builds up towards high memory; R8 contains the address of the next word to be used in the stack. In the allocation routine stack overflow is detected when the content of R8 is logically greater than the content of NXTWP. However, stack overflow can still occur and so the code that performs the 'push' operation must also check for stack overflow (if no check is made then all workspace register sets could become corrupted).

A final improvement on the allocation routine (shown below) removes the necessity for this additional checking. With this the first word of the routine to be 'called' contains a count of the number of words that are stacked in the routine. ENTRY now becomes:

```
ENTRY   EQU   $              Call handler entry point
        MOV   @7*2(R13),R7   Get address of CALLVEC
        MOV   @8*2(R13),R8   Get address of user stack
        AI    *R7,-32        Set address of next WP
        MOV   *R14+,R11      Get routine's entry
        MOV   *R11+,R6       Get 'stack count'
        A     R8,R6          Get final stack address
        C     R6,*R7         Overflow?
        JHE   error          Y - error
        RT                   N - Invoke routine
```

The 'called" routine SUBR becomes:

```
SUBR    EQU   $              SUB's entry point
        WORD  stack count    Words to be stacked
        .
        .
        .
        RTWP                 Return to caller
```

'PUSH routine' becomes:

```
        MOV   item,*R8+      Stack <item>
```

'POP routine' is:

```
        DECT  R8             Back up stack ptr
        MOV   *R8,item       Stacked object to <item>
```

This final version allows the call handler (CALL$) to be used with a recursive subroutine. On entry to the recursive subroutine it is not necessary to save the return address or any of the registers as these have already been saved in the previous workspace; it is only necessary to load the relevant local data (named ARG1 to ARGn in the re-entrancy

section). Note: Any items that have not been explicitly popped from the stack will automatically be lost when the RTWP instruction is executed,

8.13.11 Jump Table

Suppose it is necessary to branch to a label (Li) depending on the value of a key (i); if i=1, then 11, if i=2 then L2, etc. Assume that RO contains the key, This can be written as:

```
            CI    RO,1
            JEQ   L1
            CI    RO,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 RO,i with a DEC RO

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 and then using the index mode of addressing on the key as follows:

```
TABLE   DATA  L1,L2,....,LN     Table of addresses
            •
            •
            A     RO,RO          KEY->word offset
            JLE   UNDER          KEY<=0?
            CI    RO,2*N
            JGT   OVER           KEY>N?
            B     @TABLE-2(RO)   Keys start from 1 not 0
```

This assumes that all the keys within the range 1 to N are used. If, for example, the key range is 1 to 40 and keys 2, 14 and 29 are not used, the address table (TABLE above) must still contain entries for these three keys; it is necessary to supply an 'unused key label'.

If there are large gaps of unused keys then a large amount of extra memory could be used unnecessarily, Suppose you are only interested in determining if a 'key' is; a space, a comma, a double quote, a single quote, a semi-colon, a full stop or a question mark. These characters have the following ASCII codes; >20, >2C, >22, >27, >3B, >2E and >3F. With the above method this would require a table of 32 entries and the key would have to be modified to bring it within the range 1 to 20).

In this situation the following jump routine can provide considerable memory savings, especially if this type of checking has to be performed in a number of different places. Note: This time the table is organised by frequency with the most frequently used key as the first entry in the table. (Assume that the high byte of Rx contains the key.)

```
        BL    @JUMPRx
TABLE   BYTE  TABLE-L1/2,<key1>
        BYTE  TABLE-L2/2,<key2>
        .
        BYTE  TABLE-Ln/2,<keyn>
        DATA  0
NOTFND  EQU   $    Return here if specified key not found
        .
        .
L1      EQU   $
        .
        .
Ln      EQU   $
        .
```

Here the Li are arranged so that they lie within a range of +127 and -128 words from TABLE, Each entry in TABLE consists of a signed word displacement (from TABLE to the corresponding label - Li) and a <keyi> byte opcode. The DATA 0 word indicates that there are no more entries in TABLE,

After executing the BL instruction the return address (ie the address of TABLE) is stored in R11.

JUMPRx compares the key to the next <keyi> entry in TABLE. If they are the same then the displacement field is 'added' to the address of TABLE and a branch is then made to this address, Otherwise the pointer into TABLE is incremented to the next <keyi>. If the value of this entry is zero then the specified key is not in the table and a return is made to the instruction immediately following the DATA 0 word.

The actual working of the JUMPRx routine is shown below. in the brief desciption above the displacement field is not simply added to TABLE address (hence the 'added'). The displacement field is in words and needs to be expressed in

Bytes; simply doubling it is not sufficient as it is a signed quantity (it is necessary to preserve the sign). Further, a MOVB instruction is used to copy the displacement from TABLE into a register; this automatically causes the displacement to be stored in the register's high byte and it needs to be in the low byte for the add instruction to work correctly. In the code below, this is performed by the SRA R4,7 instruction (an arithmetic shift is used so that the sign bit is propogated).

```
JUMPRx  EQU   $
        MOV   R11,R3      Save return address
*       CLR   R4          Needed for 80 and 81 processors
JUMP    MOVB  *R11+,R4    Get the current displacement
        JEQ   JUMPNO      If 0 then not found
        CB    Rx,*R11+    KEY = <keyi>?
        JNE   JUMP        No - back for next <keyi>
        SRA   R4,7        Yes - Disp to low byte and *2
        A     R3,R4       Add TABLE address to offset
        B     *R4         Goto Li
JUMPNO  INC   R11         Not found - skip over 2nd byte
        RT                'Error return'
```

Although the TMS9980 and the TMS9981 microprocessors force all instruction executions to be from a word boundary it is possible for the contents of the program counter (PC) to be odd. Normally this presents no problems. However, if the PC is used to index into a table then the wrong byte in this table could be accessed.

This can, in fact, happen with the JUMPRx routine above as executing the BL instruction causes the incremented PC (the address of TABLE) to be stored in R11. The problem revolves around the contents of R4 before the SRA instruction is performed. If bit 8 of this register is a '1' then R11 will contain an odd address when this routine is called the next time (assuming this bit is not cleared in the meantime). To guarantee that JUMPRx will work correctly the CLR R4 instruction is needed. (Note: This is not really necessary for the TMS9900 microprocessor as bit 15 of the PC is never used nor saved,)

8.13.12 Miscellaneous Techniques

A number of miscellaneous 'tricks' and techniques that may prove useful to the assembly language programmer are listed below.

8.13.12.1 Swapping Register Values

Often when writing a program consisting of a number of routines the required value is already stored in a register,

but not in the right register for the routine. Usually, this problem is overcome by using a spare register to swap the contents of the two registers:

```
MOV  Rx,temp     Save Rx contents into TEMP
MOV  Ry,Rx       Required contents to Rx
MOV  temp,Ry     Original contents of Rx to Ry
'call routine'
```

However, this is not always possible (all the registers are in use and there is no 'free RAM' available). Here, the following piece of code can be used:

```
XOR  Rx,Ry       Ry contains bit-wise difference
XOR  Ry,Rx       Set Rx to original contents of Ry
XOR  Rx,Ry       Set Ry to original contents of Rx
'call routine'
```

8.13.12.2 Error Return

Occasionally it is necessary to return some information from a called routine to inform the calling routine that something 'unexpected' happened and that some specific action is necessary (ie an error occuried). This sort of information can be returned in a number of different ways: by setting a particular register to a specific value; by setting (or resetting) a certain bit in the status register (ST); by branching directly to an error routine; etc.

Register setting. The most common error indicators used are:

```
CLR  Ry      or    SETO Ry       Set error flag
.                  .
MOV  Ry,Ry         INC  Ry       Error flag set?
JEQ  error         JEQ  error    Y - error routine
```

Status bit setting. With XOP and BLWP instructions this can be performed by anding workspace register 15 (the old ST) with >F (this clears all the status bits except the interrupt mask). The required status bit can then be set to '1' using an ORI mask instruction (the AI mask instruction can also be used); 'mask' is >2000 (for EQ bit), >1000 (for C bit), etc. On return to the calling routine these status bits are interrogated using the appropriate jump instructions; JEQ or JNE for the EQ bit; JOC or JNC for the C bit; etc.

```
ANDI R15,>F        XOP routine - clear status bits
.
ORI  R15,>1000     Error - set Carry bit
.
RTWP               Return to calling routine
```

The ANDI instruction could be replaced by:

```
        SB    R15,R15         Clear R15's high Byte
```

The calling sequence is:

```
        XOP   ....            Calling routine - issue XOP
        JOC   error           C bit set? Y - error
```

For BL instructions:

```
        SETO  temp            Error flag = no
        .
        CLR   temp            Error - error flag = yes
        .
        MOV   temp,temp       Set status bits in ST
        RT                    Return to calling routine
```

A variation on these is for the word immediately following the call to contain a jump to an error return, If an error occurs in the called routine then a return is made to the JMP instruction. A normal return to the calling routine causes the return address to be incremented past the JMP instruction.

```
        'error test'          Called routine - Error?
        JEQ   errrtn          Y - to error return
        .
        INCT  R14             Skip over error return
errrtn  RTWP                  Return to calling routine
```

The calling sequence is:

```
        BLWP  ....            Calling routine - issue BLWP
        JMP   error           Error return
        .                     Normal return
```

Suppose the routine to be called converted data input from a terminal (ie from ASCII) to binary, Then any of these mechanisms could be used to inform the calling program that the input data was not a decimal number but a hexadecimal number. Further, these mechanisms can be combined to allow multiple returns, for example:

```
        BLWP  ....            Convert ASCII to binary
        JMP   hexno           Hex number return
        JEQ   zero            Zero 'return'
        .                     Normal return
```

8.13.12.3 Buffered I/O

In a microprocessor application it is often necessary to output information to a terminal. The most efficient way of doing this is not a byte at a time but as a string of

bytes. An area of memory is set aside as an output buffer, and bytes are written into this buffer until a line is complete. A terminating character is then added to the end of the line. The output routine is invoked and printing continues until the termination character is encountered,

Note: Typically, it is not possible to mix byte and word operations on the buffer; it is all right starting off writing words to the buffer and occasionally writing two bytes together to it. The problem ocurrs when you start off with bytes and want to write a word to it. If the buffer pointer contains an odd address then performing a word operation will cause the last byte entered to be overwritten. It is often very difficult to guarantee that when you want to write a word to the buffer that the buffer pointer contains an even address.

```
        RORG
OUTBUF  BSS  80          Allocate output buffer
        .
        LI   Rx,OUTBUF   Ref the output buffer
```

A byte is written to the buffer:

```
        MOVB @char,*Rx+ Output 'char'
or      MOVB Ry,*Rx+    Output high byte of Ry
```

A word can be written to the buffer:

```
        MOVB Ry,*Rx+    Output high byte of Ry
        SWPR Ry         Swap bytes over in Ry
        MOVB Ry,*Rx+    Output new high byte of Ry
```

When the line is complete the terminator is added:

```
        SB   *Rx,*Rx    Add termination char (null)
```

In the code above the termination character is a null byte (a byte containing 0). This is used to simplify the actual terminal output routine, instead of comparing each character with the terminator all that has to be done is to take the next byte from buffer and move it into a register. Doing this causes the processor to set/reset its status bits according to the value of the byte moved; if it is zero then the EQ status bit is automatically set.

```
OUTPUT  EQU  $           Output routine entry point
        LI   Rx,OUTBUF  Ref the output buffer
OUTP1   MOVB *Rx+,Ry    Get next char to be output
        JEQ  OUTND      Null? Y - finished
                        N - output this character
        .
        JMP  OUTP1      Back for the next character
OUTND   return
```

With most terminals it is also necessary to add the carriage return and linefeed characters to the buffer before storing the termination character. The actual code required to output a character to a terminal (a KSR743) is included later.

8.13.12.4 Increment Register by 4

The TMS9900 contains a number of instructfons that allow registers to be incremented. INC increments a register by one and INCT increments a register by 2. For increments greater than these the A (add) and AI (add immediate value) instructions have to be used. However, the C (compare) instruction can be used to increment a register by four, and it only takes up one word. The AI requires 2 words. The A only takes one word, but the source register must have already been loaded with the value four. The compare instruction is used as follows:

```
C     *Rx+,*Rx+  Rx=Rx+4
```

8.13.12.5 Non Destructive Memory Sizing

In this example a simple memory check is also performed; it only checks to see if each bit in the word can be set to a '1' and a '0'. (A full memory checking algorithm would be extremely complex and could literally take days to run. For a practical system, some compromise is obviously necessary.)

```
            LI    R2,start    Ref start address (high memory)
            LI    R5,end      Ref end address (low memory)
NEXTWD      C     R2,R5       Finished?
            JL    done        Y
            MOV   *R2,R3      N - save original contents
            INV   R3          Invert all the bits in copy
            MOV   R3,*R2      Write back to test address
            C     *R2,R3      Same?
            JNE   nomatch     N - end of RAM found
            INV   *R2         Y - restore original contents
            DECT  R2          Ref next word to be tested
            JMP   NEXTWD      Back for the next word
done        EQU   $
nomatch     INCT  R2          Back up to last 'good' word
```

Note: Memory autosizing operations should not be performed on an area of memory that contains memory mapped devices as this could cause the devices to become corrupted.

8.13.12.6 Simple Clock using the 9901

The 9901 Programmable Systems Interface is a CRU-driven

device that is used to regulate (enable or disable) incoming interrupt signals without interfering with the 9900 microprocessor, It is also contains an interval timer that can be programmed to generate level 3 interrupts when the interval period has elapsed. This device car; be in one of two modes (clock mode or interrupt mode), The mode is selected by writing either a '0' (interrupt mode) or a '1' (clock mode) to the 9901's control bit (bit 0 in the 9901's CRU address space),

Clock mode allows the user to program the interval timer with a 14 bit value; a copy of this value is decremented every 64 system clock cycles (for a system clock frequency of 3MHz this means a decrement every 21.3us). The value 1875 (in the code below) corresponds to an interval of 40ms.

Interrupt mode allows the user to enable or disable a particular interrupt level. An interrupt level is enabled by writing a '1' to the appropriate mask bit (mask bit 5 corresponds to interrupt level 5) and disabled by writing a '0' to the mask bit.

The initialization code below sets the 9901's CRU base address to BASE, selects clock mode and then loads the interval timer for a 40ms delay. (The LDCR instruction writes 15 bits to the 9901; the first bit causes clock mode to be selected as it is a '1' , the other 14 bits contain the required delay,) It is now necessary to enable interrupt level 3, otherwise no interrupt will be allowed through to the 9900 when the specified interval delay has expired, Level 3 interrupts are enabled by selecting interrupt mode (SBZ 0) and writing a '1' to the mask bit 3 (SBO 3). Now the 9901 will pass any level 3 interrupts through to the 9900, however the 9900 will not recognise any interrupts until the status register's interrupt mask is set to a sufficiently low value. This is performed by the LIMI 3 instruction, (Note: A DORG directive is used to allocate memory for the workspaces, starting at address FREE, DORG is similar to the AORG directive except that no code is actually produced for the DORG section, however, all references to a DORG'd label are resolved,)

```
          DORG free
WP1       BSS  32          Define RESET interrupt's  WP
CLKWP     BSS  32          Define clock interrupt's  WP
SPURWP    BSS  32          Define spurious interrupt WP
          AORG 0
          DATA WP1         Define RESET(level  0) vector
          DATA START
          DATA SPURWP,SPUR Level 1 not used
          DATA SPURWP,SPUR Level 2 not used
          DATA CLKWP       Define level 3 vector
          DATA CLOCK
```

```
SPUR    EQU  $          Spurious interrupt handler
        •
        •
START   EQU  $          Initial entry point
        •

*
* Set current time - zeroise all clock values

        LI   R2,CLKWP   Ref clock's WP
        CLR  *R2+       Clear clock handler's R0
        CLR  *R2+       Clear clock handler's R1
        CLR  *R2+       Clear clock handler's R2
*       CLR  *R2        Clear clock handler's R3
*
* Initialise the 9901
*
        LI   R12,base   Set 9901 CRU s/w base addr
        LI   R1,1875*2+1  40ms delay + clock mode
        LDCR R1,15      Set 9901 interval timer
        SBZ  0          Select interrupt mode
        SBO  3          Enable level 3 interrupt
        LIMI 3          Set interrupt mask to 3
```

The clock interrupt handler is:

```
CLOCK   EQU  $
        LI   R12,base   Set 9901 CRU s/w base addr
        SBZ  0          Select interrupt mode
        SBO  3          Reset level 3 interrupt
        CI   R0,>24     24th tic?
        JHE  CLK1       Y - 1 second elapsed
        INC  R0         N - increment tic count
        RTWP            Return
CLK1    CLR  R0         Reset tic count
        INC  R1         Increment second count
        CI   R1,60      60 secs elapsed?
        JLT  CLK2       N - return
        CLR  R1         Y - reset second count
        INC  R2         Increment minute count
        CI   R2,60      60 mins elapsed?
        JLT  CLK2       N - return
        CLR  R2         Y - reset minute count
        INC  R3         Increment hour count
        CI   R3,24      24 hours elapsed?
        JLT  CLK2       N - return
        CLR  R3         Y - reset hour count
CLK2    RTWP            Return
```

In the clock interrupt routine above the interrupt signal is reset by selecting interrupt mode and re-enabling the level 3 mask bit.

The above routine can be modified, very simply, to drive a clock display (a circuit for this is described in the Time

of Day Clock Application Sheet).

8.13.12.7 Simple 1/0 Routines using the 9902

The 9902 is a CRU-driven asynchronous communications controller. It allows the user to receive and transmit asynchronous serial data over a wide range of baud rates.

The receive routine reads a character from the 9902 receive buffer register (CRU bits 0 to 7 in the 9902's CRU read address space) into the high byte of register 0. Data is present when the read CRU bit 21 (RBRL - Receive Buffer Register Loaded) is set to '1'. If data is there then the character is read into the register (only 7 bits are actually read), the RBRL bit is reset by a write to CRU bit 18 (RIENB) and the return address is increment to skip over the 'no character return'.

```
GETCH   LI    R12,base   Set the CRU base address
        TB    21         Character ready - RBRL set?
        JNE   GETC1      N - return
        CLR   RO         Clear receiving register
        STCR  R0,7       Read character (only 7 bits)
        SBZ   18         Reset RBRL
        INCT  R11        Skip over 'no char return'
GETC1   RT
```

The calling sequence is:

```
        BL    @GETCH     Get next character input
        JMP   no char    No character return address
        .                Character return address
```

The transmit routine assumes that the character to be transmitted is stored in RO (this character is masked down to 7 bits). When the terminal is ready (bit 27, Data Set Ready - DSR - is set) a Request To Send is issued (sets bit 16 - RTS). Before the character can be sent the Transmit Buffer Register Empty flag (bit 22 - XBRE) must be set. When this occurs the character is passed to the 9902. (Note: Although the character has been masked to 7 bits, 8 bits are actually passed across. In the 9902, the character is initially loaded into the Transmit Buffer Register and is not sent until the most significant bit of this register is written to. If only 7 bits are passed across it is necessary to include either a SBZ 7 or a SBO 7 instruction). The RTS flag is then reset.

```
OUTCH   LI    R12,base   Set the CRU base address
        ANDI  R0,>7F00   Mask to 7 bits
OUTC1   TB    27         DSR ready?
        JNE   OUTC1      N - wait until it is
        SBO   16         Set RTS
OUTC2   TB    22         XBRE empty?
        JNE   OUTC2      N - wait until it is
        LDCR  R0,8       Y - send character
        SBZ   16         Reset RTS
        RT
```

Note: If the terminal is a slow printer (below 1200 baud) then whenever a carriage return character is sent a delay of around 200ms is needed to allow the print head to return to the left hand margin.

Before the 9902 can be used it must first be initialized, For this the following sequence must be used:

- o Write to bit 31 (RESET), This initializes the transmitter and receiver, and sets all the load control flags,
- o After a reset the first 8 data bits written to the 9902 are used to set up the Control Register, This selects character length, parity, the number of stop bits to be generated, and the clock predivider.
- o If the interval timer is not required then it is necessary to reset the Load Interval Register flag (bit 13 - LDIR). Otherwise the next 8 data bits written to the 9902 are used to specify the interval delay,
- o The next 12 data bits sent to the 9902 are used to select the receive data rate. If the Load Transmit Data Rate Register flag (bit 11 - LXDR) has not been explicitly reset then these 12 bits will also be used to select the transmit data rate.

In the code below the first LDCR instruction loads the Control Register with >62; this means that 2 stop bits are generated and that each character is 7 bits with even parity. (As a multiple bit CRU instruction of less than 9 bits is involved it is necessary to store the >62 byte in R1's high byte.) The second LDCR instruction causes the receive and transmit data rate registers to be set to RATE, The actual value of RATE depends on the system clock frequency; for a 3MHz system clock a value of >638 corresponds to 110 baud, >4D0 to 300 baud, and >1A0 to 1200 baud. (Full details are in sections 2.1.2.3 and 2.1.2.4 of the TMS9902 Asynchronous Communications Controller Data Manual.)

```
LI    R12,base   Set the 9902 CRU base address
SBO   31         Reset the 9902 (RESET)
LI    R1,>6200
LDCR  R1,8       Initialise the control reg
SBZ   13         No interval reg (LDIR)
LDCR  rate,12    Init REC/XMIT data rate
```

8.13.12.8 Automatic Baud Rate Determination

The receive line (RIN, bit 15 on the 9902) of a terminal to EIA port communication cable is usually in the SPACE condition (it is held at logic level '1') when nothing is being received, When a key is pressed on the terminal, the terminal puts the RIN line into the MARK condition (pulls the line down to logic level '0') by generating a start bit, This start bit is followed by 7 data bits (the least significant bit first) and a parity bit. At least 1 stop bit is then generated to put the line back into the SPACE condition,

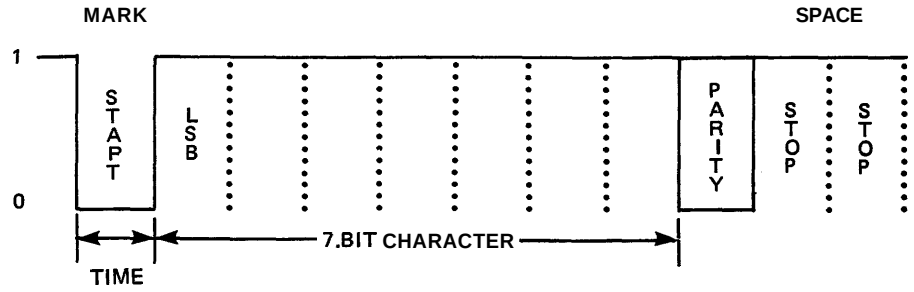


Figure 8-39 TMS9902 Character Timing

The 9902's RIN pin can he interrogated to determine when the line goes into the mark condition (when a start bit is received), If the least significant bit of the character being received is a '1' (eg the character 'A'), then the length of time taken for the RIN pin to go from the mark condition back to the space condition can be calculated. From this, the rate at which bits are being received (the receive baud rate) can be determined, This baud rate is then used to initialize the receive and transmit data rate registers.

The code below operates by counting the number of times the RIN pin is interrogated while waiting for it to be pulled up from the mark condition to the space condition, This count (stored in R3) is then compared against a table of 'maximum times around the interrogation loop for a given baud rate'. The corresponding baud rate is then loaded into the receive and transmit data rate registers,

```
*
* Initialize the 9902
*
        LI   R12,base  Set the 9902 CRU base address
        SBO  31        Reset the 9902 (RESET)
        LI   R1,>6200
        LDCR R1,8      Initialise the control reg
        SBZ  13        No interval reg (LDIR)
        CLR  R3        Clear loop count
*
* Wait for the start bit
*
SBAUD   TB   15        Space condition?
        JEQ  SBAUD     Y - test RIN pin again
*
* In the mark condition - wait until RIN goes back
* to the space condition
*
SBAUD1  INC  R3        Update loop count
        TB   15        Space condition?
        JNE  SBAUD1    N - retry the RIN pin
*
* Back in the space condition - find baud rate
*
        LI   R4,BAUDTB-2  Ref max value table
SBAUD2  INCT R4        Try next entry
        C    R3,*R4+   Loop count <= table entry?
        JH   SBAUD2    N - higher, back for next
*
* Baud rate found - set receive and transmit data
* registers, wait until character received, and
* throw the character away
*
        LDCR *R4,12    Y - set rec/xmit baud rate regs
SBAUD3  TB   21        RBRL set?
        JNE  SBAUD3    N - character not complete
        SBZ  18        Reset RBRL
```

The 'baud rate' table (BAUDTB) below works for a 3Mhz system clock (eg for a TM990 /100 or /101 CPU board). Each entry in the table consists of two fields; a loop count (in the description above this field was referred to as the 'maximum times around the interrogation loop for a given baud rate') and the baud rate corresponding to this value.

```
BAUDTB  DATA >0007,>001A  19200 baud
        DATA >000E,>0034   9600 baud
        DATA >001D,>0068   4800 baud
        DATA >003B,>00D0   2400 baud
        DATA >0075,>01A0   1200 baud
        DATA >00EA,>0340    600 baud
        DATA >0246,>04D0    300 baud
        DATA >7FFF,>0638    110 baud
```

Note: For processors other than the TMS9900 it may be

necessary to adjust the loop count entries (eg for a TMS9995 microprocessor using internal RAM),

8.13.12.9 Packed Data

In a number of instances a binary variable is required; such a variable only has two possible values (eg the state of a switch, either on or off) and can be stored in a single hit. Unfortunately the assembler does not support a bit structure (it only recognises the word, byte and text structures) and storing one bit's worth of information in a word (or even a byte) can be rather wasteful, especially if a number of these binary variables are required,

Packing a number of these binary variables into a word solves the memory wastage problem, however, it does make it a little more complicated to access the individual variables; you can not do a straight value comparison nor a 'MOV var,var' instruction to set the status register's status bits.

An individual binary variable can be set using the SOC instruction (Set Ones Corresponding), reset using the SZC instruction (Set Zeros Corresponding), toggled (change it's state from '1' to '0' or from '0' to ' 1 using the XOR (Exclusive OR) instruction, and tested using the COC (Compare Ones Corresponding) and/or the CZC (compare Zeros Corresponding) instructions, (Note: The ANDI logical instruction can be used to isolate a particular binary variable, which can then be tested using the compare or move instruction,)

The SOC instruction sets the bits in the destination operand to a ' 1 that correspond to a '1' in the source operand, All other bits in the destination operand are unchanged, Example: Set the binary variable in bit position 10 of a packed word:

```
            LI   Rx,>0020     Bit 10 = '1' (rest = '0')
            SOC  Rx,@PACKED   Set bit in PACKED

 or  MASK   DATA >0020        Bit 10 = '1' (rest = '0')
            .
            SOC  @MASK,@PACKED Set bit in PACKED
```

Note: This can also be performed by:

```
            MOV  @PACKED,Rx  Copy PACKED into register
            ORI  Rx,>0020    Set the bit
            MOV  Rx,@PACKED  Copy updated word to PACKED
```

The SZC instruction resets the bits in the destination operand to a '0' that correspond to a '1' in the source operand. All other bits in the destination operand are

unchanged, Example: Reset the binary variable in bit position 10 of a packed word:

```
        LI   Rx,>0020     Bit 10 = '1' (rest = '0')
        SZC  Rx,@PACKED   Reset bit in PACKED

or MASK DATA >0020        Bit 10 = '1' (rest = '0')
        .
        SZC  @MASK,@PACKED Reset bit in PACKED
```

Note: This can also be performed by:

```
        MOV  @PACKED,Rx   Copy PACKED into register
        ANDI Rx,>FFDF     Reset the bit
        MOV  Rx,@PACKED   Copy updated word to PACKED
```

The XOR instruction performs a bit by bit exclusive or of the two operands, and stores the result in the destination (second) operand, A bit-wise exclusive or operation sets the result bit to a '1' if the source and destination bits are different, otherwise the result bit is reset to '0'.

```
MASK    DATA >0020        Bit 10 = '1' (rest = '0')
        .
        MOV  @PACKED,Rx   Copy packed data into Rx
        XOR  @MASK,Rx     Toggle bit in PACKED
        MOV  Rx,@PACKED   Restore updated data
```

The COC instruction sets the EQ status bit to '1' if all the bits in the destination operand that correspond to a '1' in the source operand are '1's.

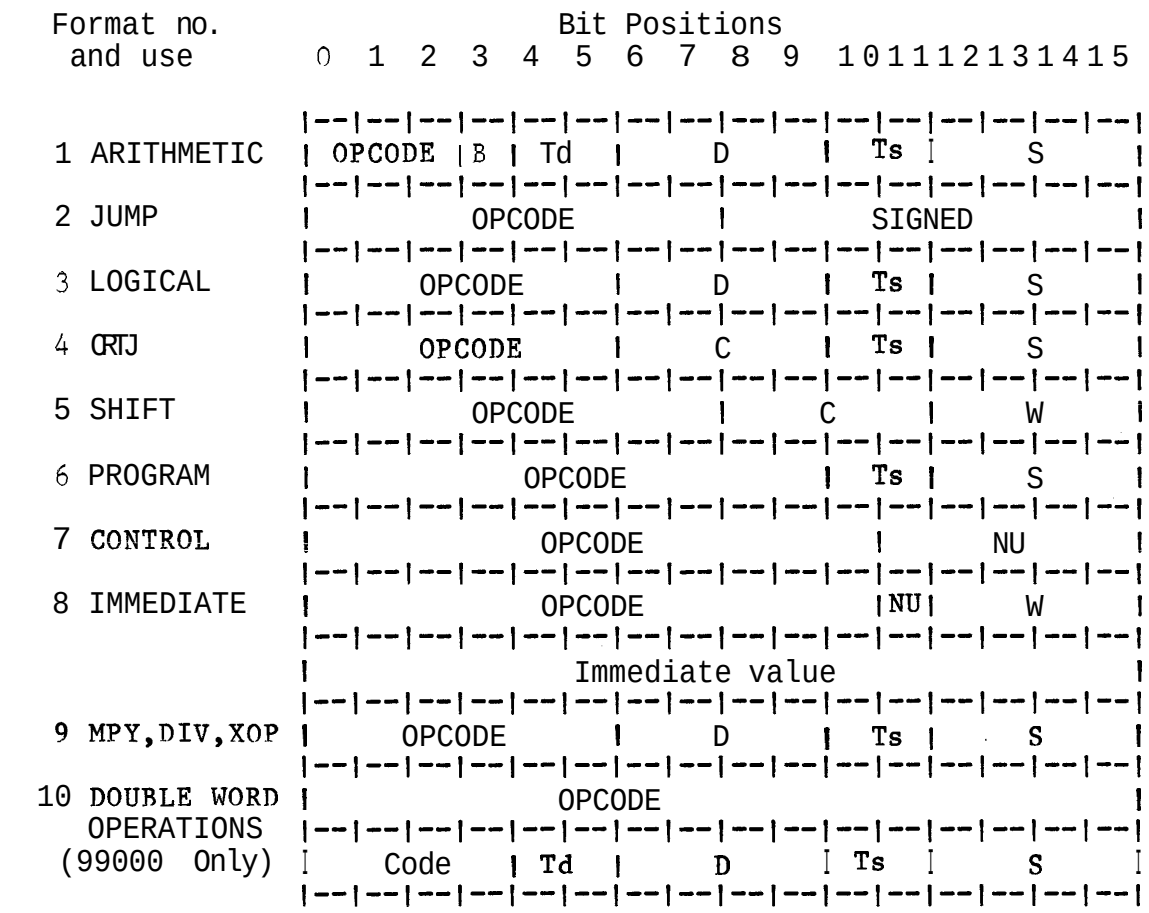
```
MASK    DATA >0020        Bit 10 = '1' (rest = '0')
        .
        MOV  @PACKED,Rx   Copy packed data into Rx
        COC  @MASK,Rx     Rit 10 set to '1'?
        JNE  NOT1         N - goto NOT1
        .                 Y - drop through to here
        .
NOT1    EQU  $            Rit 10 was not set to '1'
```

The CZC instruction sets the EQ status bit to '1' if all the bits in the destination operand that correspond to a '1' in the source operand are '0's.

```
MASK    DATA >FFDF        Bit 10 = '0' (rest = '1')
        .
        MOV  @PACKED,Rx   Copy packed data into Rx
        CZC  @MASK,Rx     Bit 10 reset to '0'?
        JNE  NOT0         N - goto NOT0
        .                 Y - drop through to here
        .
NOT0    EQU  $            Bit 10 was not reset to '0'
```

8.14 REFERENCE SECTION

8.14.1 Instruction Formats



Note: For AM/SM Code='0100'
For SLAM/SRAM Code='0100';Td='00';D is shift count
For TMB/TCMB/TSMB Code='0000';Td='00';D is bit number

OPCODE - Assembly language mnemonic
B - Byte indicator (1 = byte, 0 = word)
Td/Ts - Destination/Source address mode
D/S - Destination/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 by 1 - byte; 2 - word; 4 - double word

An extra word is required for each operand code of 2.

8.14.2 Status Register

0	1	2	3	4	5	6	7	8	9	10	11	12	15
L>	A>	=	C	O	P	X	PR	M			OE	EM	Int. mask

0 - L> Logical greater than
1 - A> Arithmetic greater than
2 - = Equal/TB indicator
3 - C Carry from most significan hit
4 - O Overflow
5 - P Parity
6 - X Software implemented XOP in progress
7 - PR Privilege mode (99000)
8 - M Map select (9989 and 99000)
10 - OE Overflow enable (9995, 9989 and 99000)
11 - EM Emulate XOP enable (99000)

Interrupt mask: F - All interrupts enabled
0 - Only interrupt level 0 enabled

8.14.3 Interrupts

Vector address	Workspace Pointer (WP)
Vector address+2	Entry point (PC)

Note: 1) Interrupt vectors 0–15 from 0 TO >3C
(only levels 0 - 4 for 9980A, 9981 and 9995)
2) XOP vectors from >40 to >7C
3) LOAD vector at >FFFC
4) Interrupt 0 is the RESET interrupt

8.14.4 CRU

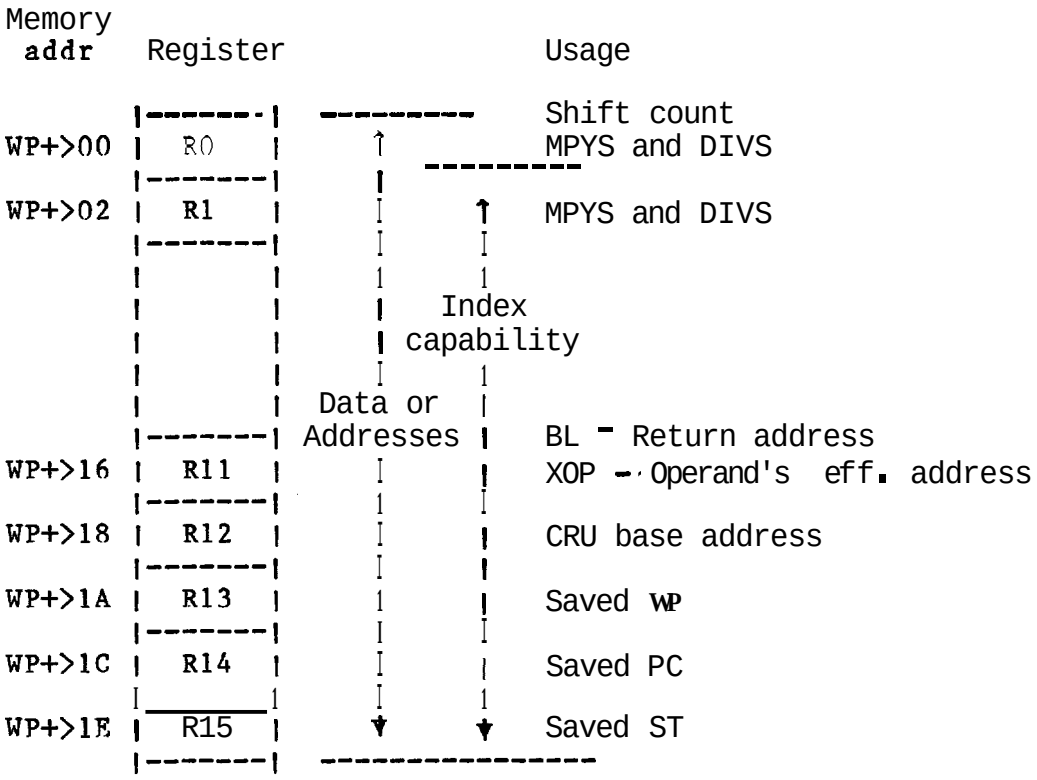
R12 - Base address for CRU operations
bits 3 - 14 used (all but 9995 and 99000)
bits 0 - 14 used (9995 and 99000)

Transfers < 9 bits - high byte used
Transfers > 9 bits - low byte used

Parallel CRU (99000 only) - CRU base address -ve

Transfer	Count	Effect on R12
Byte	0010	Not altered
	0011	Post incremented by 2
Word	1010	Not altered
	1011	Post incremented by 2

8.14.5 Register Restrictions



MPY and DIV use two consecutive registers, the first is supplied as the source operand. If R15 used then the word following R15 is used as the second register.

8.14.6 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 - 15)
* - Result is compared to zero

Additional symbols for 9989, 9995 and 99000

R* - Register pair R1 and R2
G1,G1+2 - General memory address double word

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	AR	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		External
CLOCK ON	03A0	7		CKON		External
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 CORRES.	2000	3	2	COC	G,R	ST2=AND of RBITS corres. to GBITS=1
COMPARE ZEROS CORRES.	2400	3	2	CZC	G,R	ST2=NAND of RRITS corres. to GBITS=1
DECREMENT BY ONE	0600	6	*0 -- 4	DEC	G	(G)-1->(G)
DECREMENT BY TWO	0640	6	*0 -- 4	DECT	G	(G)-2->(G)

Instruction	Opcode	Format Type	Status Bits Affected	Format		Effect
DIVIDE	3C00	9	4	DIV	G,R	INT (R)/(G)->(R) REM (R)/(G)->(R+1)
EXECUTE INSTRTJCTION	0480	6		X	G	Execute instr at G
EXTENDED OPERATION	2C00	9	6	XOP	G,T	(>40+4*T)->(WP) (>42+4*T)->(PC) Eff add of G->(R11) (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		IDLE; External
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	Is 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		3EQ	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 CRU	3000	4	*0-2,5	LDCR	G,T	T bits (G) -> 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		External
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		External
RETURN WORKSPACE POINTER	0380	7	0 -- 6 12-15	RTWP		(R13)->(WP) (R14)->(PC) (R15)->(ST)

Instruction	Opcode	Format Type	Status Bits Affected	Format		Effect
SET BIT TO ONE	1D00	2		SBO	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 CORRES, BYTE	F000	1	*0-2,5	SOCB	G1,G2	(G1) OR (G2) ->(G2)
SET ONES CORRES. WORD	E000	1	*0 -- 2	SOC	G1,G2	(G1) OR(G2) ->(G2)
SHIFT LEFT ARITH, ※	0A00	5	0 -- 4	SLA	R,C	Shift left C bits and '0' fill
SHIFT RIGHT ARITH. ※	0800	5	0 -- 3	SRA	R,C	Shift right C bits and MSR fill
SHIFT RIGHT CIRCULAR ※	0B00	5	0 -- 3	SRC	R,C	Shift right C bits and LSR into MSR
SHIFT RIGHT LOGICAL ※	0900	5	0 -- 3	SRL	R,C	Shift right C bits and '0' fill
STORE CRU	3400	4	*0-2,5	STCR	G,T	T CRU bits ->(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	Gl,G2	(G2)-(G1)->(G2)
SWAP BYTES	06C0	6		SWPB	G	Interchange bits 0-7 with bits 8-15 of G
SET ZEROES CORRESPONDING BYTE	5000	1	*0-2,5	SZCB	61,G2	(INV(G1)) AND (G2) ->(G2)
SET ZEROES CORRESPONDING WORD	4000	1	*0 -- 2	SZC	G1,G2	(INV(G1)) AND(G2) ->(G2)
TEST BIT	1F00	2	2	TB	E	(R12)+E->ST2

※ If C=0 then count taken from bits 12 - 15 of R0.
If this is zero then C=16.

Additional Instructions for 9995 and 9989

Instruction	Opcode	Format Type	Status Bits Affected	Format		Effect
LOAD ST FROM REGISTER	0080	8	0 - 15	LST	R	(R)->ST
LOAD WP FROM REGISTER	0090	8		LWP	R	(R)->WP
SIGNED DIVIDE	0180	6	*0-2,4	DIVS	G	INT(R*)/(G)->(R0) REM(R*)/(G)->(R1)
SIGNED MULTIPLY	01C0	6	*0 -- 2	MPYS	G	MSW((R*)*(G))->(R0) LSW((R*)*(G))->(R1)

Additional Instructions for 99000 Family

Instruction	Opcode	Format Type	Status Bits Affected	Format		Effect
ADD DOUBLE	002A	10	0 -- 4	AM	G1,G2	(G1,G1+2)+(G2,G2+2) --> (G2, G2+2)
BRANCH INDIRECT	0140	6		BIND	G	(G) -> (PC)
BRANCH AND PUSH STACK POINTER	00B0	8		RLSK	R,I	(W)-2 -> (W) (PC)+4 -> ((W)) I -> (PC)
LOAD ST FROM REGISTER	0080	8	0 - 15	LST	R	(R)->ST
LOAD WP FROM REGISTER	0090	8		LWP	R	(R)->WP
SHIFT LEFT ARITHMETIC DOUBLE ※	001D	10	0 -- 4	SLAM	Gl,C	Shift (G1,G1+2) left C bits; '0' fill
SHIFT RIGHT ARITHMETIC DOUBLE ※	001C	10	0 -- 3	SRAM	G1,C	Shift (G1,G1+2) right C bits; MSB fill
SIGNED DIVIBE	0180	6	*0-2,4	DIVS	G	INT(R*)/(G)->(R0) REM(R*)/(G)->(R1)
SIGNED MULTIPLY	01C0	6	*0 -- 2	MPYS	G	MSW((R*)*(G))->(R0)
SUBTRACT DOUBLE	0029	10	0 -- 4	SM	G1,G2	(G2,G2+2)-(G1,G1+2) ---> (G2, G2+2)
TEST MEMORY BIT	0C09	10	2	TMB	G1,T	(G1+Tbit) -> ST2
TEST AND CLEAR MEMORY BIT	0C0A	10	2	TCMB	G1,T	(G1+Tbit) -> ST2 0 --> (G1+DISP)
TEST AND SET MEMORY BIT	0C0B	10	2	TSMB	G1,T	(G1+Tbit) -> ST2 1 --> (G1+DISP)

※ If C=0 then count is taken from bits 4 - 7 of R0.

8.12.7 Pseudo-Instructions

Instruction	Format	Effect
NO OPERATION	NOP	JMP $+2
RETURN	RT	B *R11

TRANSFER VECTOR for a 'BLWP @label1' (SDSMAC only)

```
label XVEC  wpadd,pcadd      label  DATA  wpadd
                                    DATA  pcadd
                                    WPNT  wpadd
```

8.14.8 Assembler Directives

{ } - The item in parenthesis is optional
(,x) - Any number of 'x's (each preceded by a comma)

All directives (except OPTION) may be preceded by a label and followed by a comment. Strings are enclosed in single quotes,

ABSOLUTE ORIGIN - AORG exp - absolute value
Defines an absolute code block and loads the location counter with EXP.

RELOCATABLE ORIGIN - RORG {exp)
Defines a relocatable code block and loads the location counter with EXP; if EXP not present then uses:
- o Current length of program segment for absolute code
- o Length of data segment for data relocatable code
- o Length of common segment for common relocatable code

DUMMY ORIGIN - DORG exp
Defines a dummy code block (no code is generated but it allows a module to access symbols defined in another module) and loads the location counter with EXP,

DATA SEGMENT - DSEG
Defines a data relocatable block and loads the location counter with:
- o Max location counter from data relocatable code
- o Zero

DATA SEGMENT END - DEND
Terminates a DSEG and defines a program relocatable block. Loads the location counter with:
- o Max location counter from program relocatable code
- o Zero

COMMON SEGMENT - CSEG (string)
Defines begining (or continuation) of named common relocatable code block and loads the location counter with;
- o Zero if named common block previouly unused
- o Max location counter from already used named common relocatable code

If STRING (6 characters) not present then refers to blank common segment,

COMMON SEGMENT END - CEND
Terminates a CSEG and defines a program relocatable code block. The location counter is loaded as for DEND,

PROGRAM SEGMENT - PSEG
Defines a program relocatable code block and loads the location counter with:
- o Max location counter for program relocatable code
- o Zero

PROGRAM SEGMENT END - PEND
Terminates a PSEG and defines a program relocatable code block. The location counter is loaded as for DEND,

BLOCK STARTING WITH SYMBOL - BSS exp
Reserves EXP consecutive bytes. If a label present it is assigned the address of the first byte of the block,

BLOCK ENDING WITH SYMBOL - BES exp
Reserves EXP consecutive bytes. If a label present it is assigned the address of the first byte immediately following the block.

INITIALIZE BYTE - BYTE exp (,exp)
Reserves successive bytes of memory and initializes them to their respective values sf EXP.

INITIALIZE WORD - WORD exp (,exp)
Reserves successive words of memory and initializes them to their respective values of EXP.

INITIALIZE TEXT - TEXT (-) string
Reserves successive bytes of memory and initializes them to the appropriate character in STRING (max 52 characters) if minus sign present then the last character in STRING is negated.

WORD BOUNDARY ALIGN - 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
Assigns the value of EXP to LABEL,

EXTERNAL DEFINITION - DEF symbol (,symbol)
Allows other programs to access a program's SYMBOLs.

EXTERNAL REFERENCE - REF symbol (,symbol)
Provides access to SYMBOLs defined in other programs.

SECONDARY EXTERNAL REFERENCE - SREF symbol (,symbol)
Provides access to SYMBOLs defined in other programs.

FORCE LOAD - LOAD symbol (,symbol)
Causes a special object tag to be generated for the Link Editor (effect INCLUDE SYMBOL), Used with SREF.

DEFINE EXTENDED OPERATION - DXOP sym,num
Defines SYM to be an XOP; NUM is the XOP number.

PROGRAM END - END (symbol)
Terminates the assembly (everything following is ignored). If SYMBOL present it is the program's entry point.

OUTPUT OPTIONS - OPTION key(,key)
Specifies the output and listing options to the assembler. KEY can he:

XREF - Print cross reference table,
OBJ - Print listing of the object code.
SYMT - Print symbol table,
NOLIST - Suppress listing (SDSMAC)
TUNLIST - Text statement unlist (SDSMAC)
DUNLIST - Data statement unlist (SDSMAC)
BUNLIST - Byte statement unlist (SDSMAC)
MUNLIST - Macro expansion unlist (SDSMAC)

PROGRAM IDENTIFIER - IDT string
Assigns a name (first 8 characters of STRING - enclosed in single quotes) to the program. Must precede everything that produces object code,

PAGE TITLE - TITL string
STRING (max 50 characters) supplies heading for the assembler listing. (If TITL not first source statement then no heading on first page of listing).

LIST SOURCE - LIST
Restores printing of the source listing after an UNL. The directive is not printed in the listing,

NO SOURCE LISTING - UNL
Inhibits the printing of the source listing. The directive is not printed in the listing.

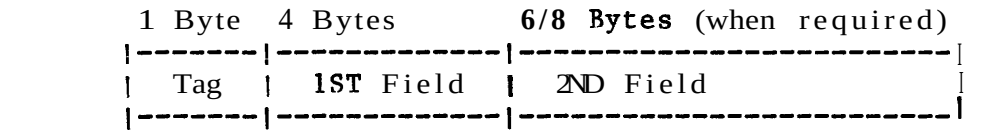
PAGE EJECT - PAGE
Causes the assembler to continue the source listing on a new page. The directive is not printed in the listing.

WORKSPACE POINTER - WPNT label SDSMAC only
Defines the current workspace (referenced by LABEL) to the assembler but produces no object code.

COPY SOURCE FILES - COPY file SDSMAC only
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
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.

8.14.9 Object Record Format and Code



TAG	1st FIELD	2nd FIELD	MEANING
0	Length of all relocatable code	8 char Program ID	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

8.14.10 Instruction Execution Times

8.14.10.1 TMS9900

Instruction	Clock Cycles	Memory Access	Add. Mod Source	Table Dest
A	14	4	A	A
AB	14	4	B	B
ARS 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 Set	16	3	A	-
ST4 Reset a	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	-	-
LWPI	10	2	-	-
MOV	14	4	A	A
MOVB	14	4	B	B
MPY	52	5	A	-
NEG	12	3	A	-
~RESET function	26	5	-	-
~LOAD function	22	5	-	-
Interrupt context switch	22	5	-	-

Instruction	Clock Cycles	Memory Access	Add. Mod Table Source	Dest
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
SOCR	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 b	8	2	A	-
XOP	36	8	A	-
XOR	14	4	A	-
Undefined opcodes	6	1	-	-

a Execution time is dependent upon the partial quotient after each clock cycle during execution

b Execution time is added to that of the instruction at the source address minus 4 clock cycles and 1 memory access

Address Modification Tables (A and B)

Addressing Mode	Clock Cycles A	Clock Cycles B	Memory Access A	Memory Access B
Register	0	0	0	0
Indirect	4	4	1	1
Indexed	8	8	2	2
Symbolic	8	8	1	1
Indirect with autoincrement	8	6	2	2

T=tc[C+(W*M)]

T - Total instruction execution time
tc - 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

8.14.10.3 SBP9900A

As for the TMS9900 except:

Instruction	Clock Cycles	Memory Access	Add. Mod Source	Table Dest
LIMI	14	2	-	-
X a	4	1	A	-

a Execution time is added to that of the instruction at the source address minus 4 clock cycles and 1 memory access

8.14.10.2 TMS9980A/TMS9981

Instruction	Clock Cycles	Memory Access	Add* Mod Source	Table Dest
A	22	8	A	A
AB	22	8	B	B
ABS Msb=0	16	4	A	-
Msb=1	20	6	A	-
AI	22	8	-	-
ANDI	22	8	-	-
B	12	4	A	-
BL	18	6	A	-
RLWP	38	12	A	-
C	20	6	A	A
CB	20	6	B	B
CI	20	6	-	-
CKOF	14	2	-	-
CKON	14	2	-	-
CLR	16	6	A	-
COC	20	6	A	-
CZC	20	6	A	-
DEC	16	6	A	-
DECT	16	6	A	-
DIV ST4 Set	22	6	A	-
ST4 Reset a	1104-136	12	A	-
IDLE	14	2	-	-
INC	16	6	A	-
INCT	16	6	A	-
INV	16	6	A	-
JUMP PC Changed	12	2	-	-
PC Unchanged	10	2	-	-
LDCR C=0	58	6	A	-
1<=C<=8	26+2C	6	B	-
9<=C<=15	26+2C	6	A	-
LI	18	6	-	-
LIMI	22	6	-	-
LREX	14	2	-	-
LWPI	14	4	-	-
MOV	22	8	A	A
MOVB	22	8	B	B
MPY	62	10	A	-
NEG	18	6	A	-
ORI	22	8	-	-
RSET	14	2	-	-
RTWP	22	8	-	-
S	22	8	A	A
~RESET function	36	10	-	-
"LOAD function	32	10	-	-
Interrupt context switch	32	10	-	-

Instruction	Clock Cycles	Memory Access	Add. Mod Source	Table Dest
SB	22	8	B	B
SBO	16	4	-	-
SBZ	16	4	-	-
SETO	16	6	A	-
SHIFT C/O	18+2C	6	-	-
C=0,R0=0	60	8	-	-
C=0,R0=N≠0	28+2N	8	-	-
SOC	22	8	A	A
SOCB	22	8	B	B
STCR C=0	68	8	A	-
1<=C<=7	50	8	B	-
C=8	52	8	B	-
9<=C<=15	66	8	A	-
STST	12	4	-	-
STWP	12	4	-	-
SWPB	16	6	A	-
SZC	22	8	A	A
SZCB	22	8	B	B
TB	16	4	-	-
X [b]	12	4	A	-
XOP	52	16	A	-
XOR	22	8	A	-
Undefined opcodes	8	2	-	-

a Execution time is dependent upon the partial quotient after each clock cycle during execution
b Execution time is added to that of the instruction at the source address minus 4 clock cycles and 1 memory access

Address Modification Tables (A and B)

Addressing Mode	Clock Cycles A	Clock Cycles B	Memory Access A	Memory Access B
Register	0	0	0	0
Indirect	6	6	2	2
Indexed	12	12	4	4
Symbolic	10	10	2	2
Indirect with autoincrement	12	10	4	4

Use the TMS9900 formula for calculating the TMS9980A and the TMS9981 instruction execution times

8.14.10.4 TMS9995

Instruction	Everything on chip		Everything but Src and Dst operands off chip		Everything but Dst operand off chip a		Everything off chip		Operand address derivation1	
	C1	XM1	C1	XM1	C1	XM1	C1	XM1	Src	Dst
A	4	0	5	2	6	4	8	8	A	A
AB	4	0	5	2	5	3	5	5	A	A
ARS	3	0	4	2	6	6	6	6	A	-
AI	4	0	6	4	6	4	8	8	-	-
ANDI	4	0	6	4	6	4	8	8	-	-
B	3	0	4	2	4	2	4	2	A	-
BL	5	0	6	2	7	4	7	4	A	-
BLWP	11	0	12	2	14 b	6 b	17	(12	A	-
C	4	0	5	2	6	4	7	6	A	A
CB	4	0	5	2	5	3	5	4	A	A
CI	4	0	6	4	6	4	7	6	-	-
CKOF	7	0	8	2	8	2	8	2	-	-
CKON	7	0	8	2	8	2	8	2	-	-
CLR	3	0	4	2	5	4	5	4	A	-
COC	4	0	5	2	6	4	7	6	A	-
CZC	4	0	5	2	6	4	7	6	A	-
DEC	3	0	4	2	6	6	6	6	A	-
DECT	3	0	4	2	6	6	6	6	A	-
DIV ST4 Set c	6	0	7	2	8	4	10	8	A	-
ST4 Reset	28	0	29	2	30	4	34	12	A	-
DIVS ST4 Set c	10	0	11	2	12	4	36	8	A	-
ST4 Reset	33	0	34	2	35	4	39	12	A	-
IDLE d	7+2I	0	8+2I	2	8+2I	2	8+2I	2	-	-
INC	3	0	4	2	6	6	6	6	A	-
INCT	3	0	4	2	6	6	6	6	A	-
INV	3	0	4	2	6	6	6	6	A	-
JUMP - All	3	0	4	2	4	2	4	2	-	-
LDCR C=0	41	0	42	2	43	4	44	6	A	-
1<=C<=15	9+2C	0	10+2C	2	11+2C	4	12+2C	6	A	-
LI	3	0	5	4	5	4	6	6	-	-
LIMI	5	0	7	4	7	4	7	4	-	-
LREX	7	0	8	2	8	2	8	2	-	-
LST	5	0	6	2	6	2	7	4	-	-
LWP	4	0	5	2	6	2	6	4	-	-
LWPI	4	0	6	4	6	4	6	4	-	-
MOV	3	0	4	2	5	4	6	6	A	A
MOVB	3	0	4	2	4	3	4	4	A	A
All interrupt context switches	14 e	0 e	17 b	6 b	17 b	6 b	20 f	12 f	-	-

Instruction	Everything on chip		Everything but Src and Dst operands off chip		Everything but Dst operand off chip a		Everything off chip		Operand address derivation	
	C1	XM1	C1	XM1	C1	XM1	C1	XM1	Src	Dst
MPY	23	0	24	2	25	4	28	10	A	-
MPYS	(2 5	0	26	2	27	4	30	10	A	-
NEG	3	0	4	2	6	6	6	6	A	-
OR1	4	0	6	4	6	4	8	8	-	-
RSET	7	0	8	2	8	2	8	2	-	-
RTWP	6	0	7	2	7 g	2 g	10	8	-	-
S	4	0	5	2	6	4	8	8	A	A
SB	4	0	5	2	5	3	5	5	A	A
SBO	8	0	9	4	9	2	10	4	-	-
SBZ	8	0	9	2	9	2	10	4	-	-
SETO	3	0	4	2	5	4	5	4	-	-
SHIFT C≠0	5+C	0	6+C	2	6+C	2	8+C	6	-	-
C=0,R0=0	23	0	24	2 1 2 4		2 1 2 7		8	-	-
C=0,R0=N≠0	7+N	0	8+N	2	8+N	2	11+N	8	-	-
SOC	4	0	5	2	6	4	8	8	A	A
SOCB	4	0	5	2	5	3	5	5	A	A
STCR C=0	43	0	44	2	46	6	47	8	A	-
1<=C<=8	19+C	0	20+C	2	22+C	6	23+C	8	A	-
9<=C<=15	27+C	0	28+C	2	30+C	6	31+C	8	A	-
STST	3	0	4	2	4	2	5	4	-	-
STWP	3	0	4	2	4	2	5	4	-	-
SWPB	13	0	14	1 2 1 1 6		6	16	6	A	-
SZC	4	0	5	2	6	4	8	8	A	A
SZCB	4	0	5	2	5	3	5	5	A	A
TB	8	0	9	2	9	2	10	4	-	-
X h	2	0	3	2	4	4	4	4	A	-
XOP	15	0	16	2	18 b	6 b	22	14	A	-
XOR	4	0	5	2	6	4	8	8	A	-

a Registers for register-only instructions (STST, LST, STWP, LWP, shifts) and registers for instructions where an additional register is required (AI, ANDI, BL, CI, LDCR, LI, ORI, SBO, SBZ, STCR, TB, and shifts) are on chip.
b Trap vector off chip and new workspace on chip.
c Execution time is dependent upon the partial quotient after each clock cycle during execution. Clock cycles shown are for worse case operands.
d Will remain in Idle state until an unmasked interrupt request occurs (I= number of CLKOUT cycles until the request occurs).
e Trap vector and new workspace on chip (NMI only).
f Trap vector and new workspace on chip.
g Workspace on chip.
h Execution time shown does not include execution time of the instruction located at the source operand.

Operand Address Derivation Table (A)

Addressing Mode	Registers, index base addr, and symbolic address on chip		Registers on chip; index base addr, and symbolic address off chip		Registers off chip; index base addr, and symbolic address on chip		Registers, index base addr, and symbolic address off chip	
	C2	XM2	C2	XM2	C2	XM2	C2	XM2
Register	0	0	0	0	0	0	0	0
Indirect	1	0	1	0	2	2	2	2
Symbolic	1	0	2	2	1	1	2	2
Indexed	3	0	4	2	4	2	5	4
Indirect with autoincrement	3	0	3	0	5	4	5	4

$$T=tc[C1+C2+W*(XM1+XM2)]$$

T - Total instruction execution time
tc - CLKOUT cycle time
C1 - Base CLKOUT cycles
C2 - Additional CLKOUT cycles for operand address derivation (table 'A' above)
W - Number of wait states per off chip (byte length) memory cycle
XM1 - Base off chip (byte length) memory cycles
XM2 - Additional off chip (byte length) memory cycles for operand address derivation (table 'A' above)

8.14.10.5 SBP9989

Address Modification Table A

Addressing Mode	Clock Cycles	Memory Access
Register	0	0
Indirect	4	1
Indexed	6	2
Symbolic	6	1
Indirect with autoincrement	6	2

Instruction	Clock Cycles	Memory Access	Add. Mod Table Source	Dest
A	12	4	A	A
AB	12	4	A	A
ABS Msb=0	10	2	A	-
Msb=1	14	3	A	-
AI	14	4	-	-
ANDI	14	4	-	-
B	6	1	A	-
BL	10	2	A	-
BLWP	24	6	A	-
C	12	3	A	A
CB	12	3	A	A
CI	12	3	-	-
CKOF	10	1	-	-
CKON	10	1	-	-
CLR	8	2	A	-
COC	12	3	A	-
CZC	12	3	A	-
DEC	10	3	A	-
DECT	10	3	A	-
DIV ST4 Set	20	4	A	-
ST4 Reset	56	6	A	-
DIVS ST4 Set	56	4	A	-
ST4 Reset	60	6	A	-
IDLE	10	1	-	-
INC	10	3	A	-
INCT	10	3	A	-
INV	10	3	A	-
JUMPs - All	6	1	-	-
LDCR C=O	48	3	A	-
1<=C<=15	16+2C	3	A	-
LI	12	3	-	-
LIMI	12	2	-	-
LREX	10	1	-	-
LST	10	2	-	-
LWP	10	2	-	-
LWPI	12	2	-	-
MOV	10	3	A	A
MOVB	12	4	A	A
MPY	52	5	A	-
MPYS	56	5	A	-
NEG	12	3	A	-
~RESET function	20	5	-	-
~LOAD function	20	5	-	-
Interrupt context switch	20	5	-	-

Instruction	Clock Cycles	Memory Access	Add. Mod Source	Table Dest
ORI	14	4	-	-
RSET	10	1	-	-
RTWP	16	4	-	-
S	12	4	A	A
SB	12	4	A	A
SBO	12	2	-	-
SBZ	12	2	-	-
SETO	8	2	A	-
SHIFT C≠0	12+2C	3	-	-
C=0,R0=0	52	4	-	-
C=0,R0=N≠0	20+2N	4	-	-
SOC	12	4	A	A
SOCB	12	4	A	A
STCR C=0	56	4	A	-
1<=C<=8	40	4	A	-
9<=C<=15	56	2	A	-
STST	8	2	-	-
STWP	8	2	-	-
SWPB	10	3	A	-
SZC	12	4	A	A
SZCB	12	4	A	A
TB	12	2	-	-
X [a]	4	1	A	-
XOP [b]	28	7	A	-
XOR	12	4	A	-
Undefined opcodes	24	6	-	-

a Execution time is added to that of the instruction located at the source address

b Execution time includes time to perform a context switch resulting from XIPP being inactive

T=tc[C+(W1*M)] + tc(W2*R)

T - Total instruction execution time
tc - Clock cycle time
C - Number of clock cycles for instruction execution plus address modification
W1 - Number of required wait states per memory access for instruction execution plus address modification
M - Number of memory accesses
R - Number of CRU operations
W2 - Number of required wait states per CRU operation

8.14.10.6 TMS99000 Family

Instruction	Machine States	Memory Access	Add. Mod Source	Table Dest
A	4	4	A	A
AB	4	4	A	A
ABS Msb=0		3	A	-
Msb=1	3	3	A	-
AI	4	4	-	-
AM	12	7	A	A
ANDI	4	4	-	-
B	3	1	A	-
RIND	4	2	A	-
BL	5	2	A	-
BLSK	7	5	-	-
RLWP	11	6	A	-
C	4	3	A	A
CB	4	3	A	A
CI	4	3	-	-
CKOF	7	1	-	-
CKON	7	1	-	-
CLR	3	2	A	-
COC	4	3	A	-
CZC	4	3	A	-
DEC	3	3	A	-
DECT	3	3	A	-
DIV ST4 Set	10	4	A	-
ST4 Reset a	31	6	A	-
DIVS ST4 Set	10 or 13	4	A	-
ST4 Reset a	35	6	A	-
IDLE	7+2N	1	-	-
INC	3	3	A	-
INCT	3	3	A	-
INV	3	3	A	-
JUMPs - All	3	1	-	-
LDCR C=0,serial	40	3	A	-
C≠0,serial	8+2C	3	A	-
parallel	5	3	A	-
LI	3	3	-	-
LIMI	5	2	-	-
LREX	7	1	-	-
LST	5	2	-	-
LWP	3	2	-	-
LWPI	3	2	-	-
All interrupt context switches	14	6	-	-

Instruction	Machine States	Memory Access	Add. Mod Source	Table Dest
MOV	3	3	A	A
MOVB	4	4	A	A
MPY	24	5	A	-
MPYs	26	5	A	-
NEG	3	3	A	-
ORI	4	4	-	-
RSET	7	1	-	-
RTWP	6	4	-	-
S	4	4	A	A
SB	4	4	A	A
SBO	7	2	-	-
SBZ	7	2	-	-
SETO	3	2	-	-
SHIFT C≠0	5+C	3	-	-
C=0,R0=0	22	4	-	-
C=0,R0=N≠0	7+N	4	-	-
SHIFT DOUBLE C≠0	13+C	5	A	-
C=0,R0=0	30	6	A	-
C=0,R0=N≠0	14+N	6	A	-
SM	11	7	A	A
SOC	4	4	A	A
SOCB	4	4	A	A
STCR C≠0,serial	13+2C	4	A	-
C=0,serial	45	4	A	-
parallel	9	4	A	-
STST	3	2	-	-
STWP	3	2	-	-
SWPB	3	3	A	-
SZC	4	4	A	A
SZCB	4	4	A	A
TB	7	2	-	-
TEST MEMORY BIT	28	3	-	-
X [b]	2	1	A	-
XOP	15 [c]	8	A	-
attached proc.		10	A	-
XOR	4	4	A	-
Undefined opcodes	14 [c]	6	-	-
external proc.		8	-	-

a Execution time is dependent upon the partial quotient after each clock cycle during execution
b Execution time is added to that of the instruction located at the source address
c Exceution time does not include the time required by software or an attached processor to emulate the instruction

Address Modification Table A

Addressing Mode	Clock Cycles	Memory Access
Register	0	0
Indirect	1	1
Indexed	3	2
Symbolic	1	1
Indirect with autoincrement	3	2

$$T=tc[C+(W*M)]$$

T - Total instruction execution time
tc - Machine state time (four times the external input clock period)
C - Number of machine states 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

8.14.11 Pin Assignments

8.14.11.1 TMS9900

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	ICO	58	NC
15	A3	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	DO	63	~MEMEN
20	A4	42	D1	64	~HOLD
21	A3	43	D2		
22	A4	44	D3		

NC - No internal connection

8.14.11.2 TMS9980A

Pin	Function	Pin	Function	Pin	Function
1	'"HOLD	15	A2	29	D3
2	HOLDA	16	A1	30	D4
3	IAQ	17	A0	31	D5
4	A13/CRUOUT	18	DBIN	32	D6
5	A12	19	CRUIN	33	D7
6	A11	20	vcc	34	CKIN
7	A10	21	Vbb	35	vss
8	A9	22	~Ø3	36	Vdd
9	A8	23	INT 2	37	CRUCLK
10	A7	24	INT 1	38	~WE
11	A6	25	INT 0	39	READY
12	A5	26	DO	40	~MEMEM
13	A4	27	D1		
14	A3	28	D2		

8.14.11.3 TMS9981

Pin	Function	Pin	Function	Pin	Function
1	~HOLD	15	A2	29	D4
2	HOLDA	16	A1	30	D5
3	IAQ	17	A0	31	D6
4	A13/CRUOUT	18	DBIN	32	D7
5	A12	19	CRUIN	33	OSCOUT
6	A11	20	vcc	34	CRIN
7	A10	21	Ø3	35	VSs
8	A9	22	INT 2	36	Vdd
9	A8	23	INT 1	37	CRUCLR
10	A7	24	INT 0	38	''WE
11	A6	25	DO	39	READY
12	A5	26	D1	40	''MEMEM
13	A4	27	D2		
14	A3	28	D3		

8.14.11.4 SBP9900A

Pin	Function	Pin	Function	Pin	Function
1	GND	23	A1	45	D4
2	GND,	24	A0	46	D5
3	WAIT	25	NC	47	D6
4	~LOAD	26	INJ	48	D7
5	HOLDA	27	GND	49	D8
6	~RESET	28	GND	50	D9
7	IAQ	29	DBIN	51	D10
8	CLOCK	30	CRUOUT	52	D11
9	INJ	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	INJ
14	A10	36	IC0	58	NC
15	A9	37	NC	59	~CYCEND
16	A8	38	NC	60	CRUCLK
17	A7	39	NC	61	''WE
18	A6	40	INJ	62	READY
19	A5	41	DO	63	~MEMEN
20	A4	42	D1	64	~HOLD
21	A3	43	D2		
22	A4	44	D3		

8.14.11.5 TMS9995

Pin	Function	Pin	Function	Pin	Function
1	XTAL1	15	~INT 1	29	A5
2	XTAL2/CLKIN	16	IAQ/HOLDA	30	A6
3	CLKOUT	17	~DBIN	31	VSs
4	D7	18	'HOLD	32	A7
5	D6	19	-WE/-CRUCLK	33	A8
6	D5	20	~MEMEM	34	A9
7	D4	21	~NMI	35	A10
8	D3	22	~RESET	36	A11
9	D2	23	READY	37	A12
10	Vcc	24	A0	38	A13
11	D1	25	A1	39	A14
12	DO	26	A2	40	A15/CRUOUT
13	CRUIN	27	A3		
14	~INT 4/~EC	28	A4		

8.14.11.6 SBP9989

Pin	Function	Pin	Function	Pin	Function
1	GND	23	A1	45	D4
2	GND	24	A0	46	D5
3	WAIT	25	~MPEN	47	D6
4	~LOAD	26	INJ	48	D7
5	HOLDA	27	GND	49	D8
6	~RESET	28	GND	50	D9
7	IAQ	29	DRIN	51	D10
8	CLOCK	30	CRUOUT	52	D11
9	INJ	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	INJ
14	A10	36	ICO	58	~XIPP
15	A9	37	INTACK	59	~CYCEND
16	A8	38	NC	60	CRUCLK
17	A7	39	MPILCK	61	~WE
18	A6	40	INJ	62	READY
19	A5	41	DO	63	~MEMEN
20	A4	42	D1	64	~HOLD
21	A3	43	D2		
22	A4	44	D3		

8.14.11.7 TMS99000 Family

Pin	Function	Pin	Function	Pin	Function
1	~WE/~CRUCLK	15	Vcc	29	A13/D13
2	~DEN	16	A0/D0/CRUIN	30	A14/D14
3	~RESET	17	A1/D1	31	~ST8/D15/CRUOUT
4	~APP	18	A2/D2	32	ALATCH
5	~HOLD	19	A3/D3	33	Vss
6	WAITGEN	20	A4/D4	34	CLKOUT
7	READY	21	A5/D5	35	XTAL2
8	~INTREQ	22	A6/D6	36	XTAL1/CLKIN
9	~NMI	23	A7/D7	37	BST3
10	IC0	24	A8/D8	38	BST2
11	IC1	25	A9/D9	39	BST1
12	IC2	26	A10/D10	40	~MEM
13	IC3	27	A11/D11		
14	~INTP	28	A12/D12		

8.14.12 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
SO	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
ETR	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		

8.14.13 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

8.15 BIBLIOGRAPHY

TI Publications

TMS9900 Microprocessor Assembly Language Programmer's Guide (943441-9701)

TMS9901 Programmable Systems Interface (MP003)

TMS9902 Asynchronous Communications Controller Data Manual (MP004)

TM990/100M Microcomputer User's Guide (MP321)

TM990/101M Microcomputer User's Guide (MP337)

TM990/302 Software Development Board User's Guide (MP343)

TM990/402 Line-by-Line Assembler User's Guide and Listing (MPB07)

Component Software Handbook (MP918)

Realtime Executive User's Manual (MP373)

Model 990 Computer Terminal Executive Development System (TXDS) Programmer's Guide (946258-9701)

Model 990 Computer AMPL Microprocessor Prototyping Laboratory Operation Guide
AMPL I (946244-9701)
AMPL II (946275-9701)

Model 990 Computer DX10 Operating System Release 3 Reference Manuals Volumes:
II Production Operation (946250-9702)
III Application Programming Guide (946250-9703)
IV Developmental Operation (946250-9704)

Time of Day Clock Application Sheet