GLENTOP

FIRST STEPS IN 68000 ASSEMBLY LANGUAGE

Robert Erskine

First Steps in Assembly Language for the 68000

Robert Erskine

Glentop Press Ltd

Dedicated to Tom and Margaret

MAY 1987

ISBN 1 85181 081 1

Published by: Glentop Press Ltd
Standfast House
Bath Place
High Street
Barnet
Herts EN5 5XE
Tel: (01) 441 4130

Originated directly from the publisher's w-p disks by
NWL Editorial Services, Tel (0458) 250834

Printed in Great Britain by Ashford Colour Press
Gosport, Hampshire

Contents

PART II

Introduction

You may already have had some experience of programming a computer in assembly language. If you have, the likelihood is that at some point you have lost the thread of understanding, either because the books you have read have been too technical or because the unfamiliar concepts of assembly language have not been clearly related to concepts which are already familiar to you.

It is one thing to learn a language like BASIC, in which instructions like PRINT and GOTO mean exactly what they appear to mean, and another thing altogether to deal with assembly language, in which numbers whose significance is often unclear are manipulated by strange and abstract instructions to produce further numbers whose purpose seems equally vague. Like a traveller without a map in a foreign land, you are stuck with a strange language and a strange currency and can find no means of orienting yourself.

It is important however, not to think of programming purely in terms of learning language instructions. Programming is mostly about using your imagination to see how a particular process or concept might work and how it might best be structured and manipulated in memory. The actual program instructions are merely a means to this end and you need not worry too much about trying to learn and memorize them all as most of them will tend to become familiar through experience. It is far more important to understand the key concepts of programming and the standard program structures which enable you to translate your ideas into easily manageable modules of code. For this reason, this book does not try to be a comprehensive text book covering every detail of each instruction code, although a large number of program instructions will be explained and illustrated in the text and a complete list of them, with descriptions of their functions, is given in Appendix B. At a later stage you may wish to purchase a more formal book containing comprehensive technical reference information, although for ordinary practical purposes you will find that this book contains most that the general applications programmer needs to know.

One thing that would come in very useful would be an assembly language reference manual relating specifically to your computer. Although 68000 assembly language works the same way for all 68000 based computers there are always significant differences between one machine and another; the major ones being the different operating systems which are used and secondly, the structure of the display screen.

The differences between operating systems are significant because they provide a means of accessing some of their subroutines directly from within your own assembly language programs. Different systems will provide different sets of routines and the methods of accessing them may vary between one computer and another. Additionally, some computer systems may have less 'transparent' operating systems than others. They may, for example, have a layer of user-interfaces such as BASIC or window and icon programs which can sometimes make the operating system difficult to get at directly.

The differences between screen structures are more obvious because the height and width of the display, the degree of graphic resolution, the number of colours used and the way in which colors and images are coded will vary considerably.

In the technical reference manual available for your computer you should find all the information you need for integrating these facilities and features in your programs. If this information is not supplied then it is worth checking your local bookstore for independently published books which relate to your particular machine or operating system.

Because of this wide variation in design, it is not possible in this volume to explain the operation of certain types of functions, such as line graphics, for different makes of computer. However, the insights into programming methods which you will acquire, together with the information contained in your own technical manuals, should enable you to construct graphics routines without much difficulty.

We are going to be taking things gradually, avoiding the technicalities of the computer's circuitry and concentrating on the most important aspect of programming: how to translate the ideas and concepts which are in your imagination into program structures which will enable them to be carried out.

In the first few chapters we shall be building up the broad outlines of

assembly language programming. Chapters 1 to 7 in Part I will mainly be general, illustrating topics common to most assembly language programming as well as facts and concepts relating to the 68000 chip in particular.

You will find it helpful to think of assembly language programming in terms of a group of key concepts which are common to all assembly language programs, such as data storage, data addressing and the use of registers, the use of flags, conditional branching, using stacks and referencing indexed tables of data. Chapters 1 to 6 are based on each of these concepts and in chapter 7 we shall look at some miscellaneous aspects of programming and system operation.

The corresponding chapters in Part II will deal in detail with the 68000, with much more emphasis on the use of its programming instruction set. Chapters 8 to 13 summarize the main concepts outlined in Part I and illustrate their applications using a number of complete and annotated programs. The multi-user, multi-tasking and protection capabilities of the 68000 will be outlined in Chapter 14, together with notes on program debugging and object code formats. An extensive appendix contains descriptions of the complete instruction set.

The programs in Part II are fairly simple, functional routines which are designed to help you learn to use instructions by example, in a meaningful context, rather than by the more common method of learning the functions of each group of instruction types in a more formal way.

They illustrate some of the more important programming functions such as setting up variables and arrays, printing characters and sentences to the display screen, arithmetical operations and processing stored data. They all follow a fairly similar structure so that they tend to reinforce understanding and most of the instructions used will appear frequently so that their functions will become familiar as you read through the chapters. If you experience any difficulty in understanding how a particular process works, don't worry too much. The more complex functions which you come across will be explained in the same chapter or later in the book and it is better to read on and come back to something than to become stuck over a point of detail.

Each chapter deals with a particular topic and on a first reading of Part II you will find it more useful to concentrate on the topics illustrated

by the example programs than to try to follow everything contained in them.

To begin with you may prefer to read through Parts I & II in sequence and then later, you can read the corresponding chapters in each part in conjunction in order to reinforce your understanding.

User and Supervisor Modes

The 68000 operates in two modes: user and supervisor mode. In practice you will normally only be concerned with user mode which is the mode in which ordinary user programs are executed. Supervisor mode is used by the computers's operating system in order to gain total supervisory control over the events taking place in the system. Supervisor mode is only mentioned in the text of this book in relation to special system functions and it is not necessary for you to have any detailed knowledge of its operation.

Part I

Chapter 1

Beginning Assembly Language

Assembly language, or 'machine code', consists of coded instructions which instruct the machine – or more accurately the processor – what to do. Machine code is a purely numeric form of assembly code, but when programs are written it is usual to use a set of non-numeric instructions called *mnemonics*, which are directly equivalent to machine code and are a lot easier for mere humans to understand.

Assembly language is the term used to describe this set of mnemonic instructions and the two terms will be used in their appropriate contexts throughout the book to distinguish the two forms of code. Programming in pure numeric machine code is possible and is very often done on old, 8-bit computers. With a complex processor like the 68000 you would need to be very fond of numbers to want to write your programs in this way and the programs in this book are presented in their assembly language format.

Although many of the concepts involved are similar to those used in BASIC, many are quite different and it is best to approach the subject without too many pre-formed notions. Think of the computer initially as a machine consisting of nothing but a keyboard, a processor, a screen and so many memory locations, say 256 000.

Starting from this uncluttered viewpoint, the principles of assembly language programming become extremely simple.

Consider the following facts:

1 All programs, data, colour and graphics must consist of numeric data.

2 Everything which the computer does, such as arithmetic calculations and the printing of letters, numbers and graphics on the screen, is performed on data taken either from memory or from some peripheral device such as the keyboard or a disc drive.

3 The CPU (central processing unit, or microprocessor) performs all arithmetic operations and controls and co-ordinates the movement of data between itself, memory, the keyboard, the VDU (visual display unit) screen and other peripheral devices.

These three facts represent the basic model of a processing system.

Unlike a high level language such as BASIC, which is designed to coordinate sets of general concepts, grouped under function names such as `PRINT`, `LIST`, `LOG`, `CLS` and so on, the function of assembly language is simply to direct the CPU to control the sequence of individual data movements around the system. In the next few chapters we shall be exploring not only how this is done but also how the movement of data relates to recognizable functions such as arithmetic calculation and the printing of characters and words to the screen. It is important, before you begin to learn these techniques, to acquire a general understanding of how data is stored and manipulated, because it is much easier to develop a program in assembly language if you are able to construct in your mind a mental model of the processes which are taking place.

To begin with, we shall be looking at memory and the formats in which programs and data are stored. We shall then go on to see how data is taken from memory and processed by the CPU during the execution of a typical machine code instruction.

Initially, we shall not be too concerned about the language we use for the instruction, nor about the precise way in which data is specified, or how we select the method by which it is sent to and received back from the CPU. Instead we shall concentrate on the general pattern of events; how the different elements of the system relate to each other and how the data is used to represent meaningful functions.

Memory, Addresses and Data

Firstly, we need to construct for ourselves a mental model of a computer's memory. A clear understanding of memory structure is essential to the understanding of how assembly language works and you will need to relate much of what you read in the following chapters to the events which you visualize taking place within the memory space.

We shall start by looking at the meanings of the terms *addresses* and *bytes*, because these are two of the main concepts from which our memory model will be constructed.

Memory consists of a sequence of separate, numbered locations in which items of information such as data and programs can be stored. There is no need for you to understand the actual physical structure of memory, as it exists on a chip. It is more helpful to imagine it simply as a series of numbered boxes into which programs and data can be placed. These can be pictured as a horizontal or vertical series of numbered pigeon holes, or, sometimes, as a two-dimensional matrix, whichever is most convenient for understanding a particular process.

Because memory locations are sequentially numbered, the number corresponding to each location is termed an *address*, in the same way that houses in a street are given address numbers. Figure 1.1 shows a number of possible representations of memory addresses.

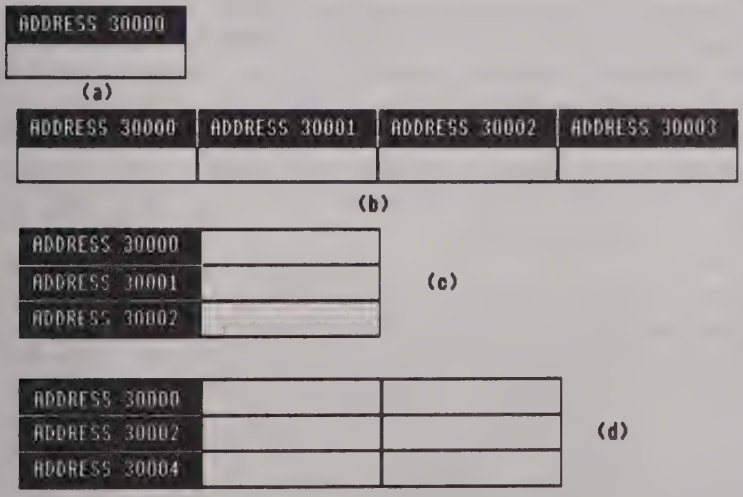


FIGURE 1-1. a) Single address
b) Row of consecutive addresses
c) Column of consecutive addresses
d) Two-dimensional array

The use of address numbers in assembly language is of vital importance because it is only by keeping track of addresses that the computers's CPU can find its way through a program. Just as in BASIC, where programs are given line numbers to indicate to the computer the order in which commands are executed, the CPU always needs to be aware of the address of the next machine code instruction which is to be executed. In the case of conditional branching operations, where execution is redirected to a subroutine for example, the CPU needs to keep a record of the address from which the branch was made so that it can return and pick up the sequence from where it left off. The programmer also needs to be aware of the address numbers of certain memory locations because particular items of data may be stored in specific places and there has to be a precise method of locating each one.

In our imaginary model, we shall assume that we are working with a computer which has a memory capacity of 256K. Since 1K of memory actually equals 1024 memory addresses, 256K therefore represents 262 144 addresses. We shall assume that address numbers 0 to 1023 are allocated to various items of data required by the computer's operating system (OS). Addresses 1024 to 66559 are allocated to the display memory, in which images which are visible on the VDU screen are stored. Addresses 66560 to 196607 are a free user area into which our own programs and data are loaded and addresses 196608 to 262143 are occupied by the computer's operating system. This model is greatly simplified but it illustrates the main areas into which memory is typically divided.

When we speak of 'addressing' a memory location, we mean that the CPU can have its attention directed to any one of the address locations in the system, including those in both RAM and ROM. In the case of the 68000, up to 16 million memory locations can be addressed, although in practice, many micros have much less memory available than this – usually 128 to 512K in total. The CPU can 'read' data from both RAM and ROM – that is, it can identify the contents of any addresses in these areas – and it can 'write' to any of the addresses in RAM – that is, it can insert fresh data into any of the RAM addresses.

Bytes

Data of any kind which is stored in a single memory address is recorded as a binary *byte*, representing an integer number between 0 and 255.

Each of the eight digits in a binary byte is termed a *bit* and a group of four bits, not surprisingly, is called a *nibble*. Each bit in a binary byte can represent either 0, in which case it has no value, or 1, in which case its value depends on its position within the byte. If the bit on the extreme right of the byte (the LSB or least significant bit) is set to 1 it has the value 1. A set bit in the second position has the value 2, the next 4, and so on through 8, 16, 32, 64 and finally 128, the value of the most significant bit (MSB).

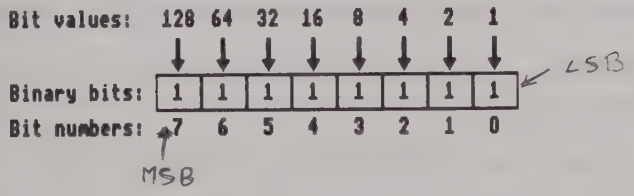


FIGURE 1-2.

The total value of a byte is found by adding the individual values of its set bits, as follows:

```
11001001 = 201 decimal (128+64+8+1)
```

The reason why numbers are stored in this integer binary form is that in a computer, each set bit in a binary number, that is, each bit which is a 1 rather than 0, constitutes a signal which is translated within the machine as a voltage. It is only in the form of an electric current that binary numbers can physically be transported around the system from one component to another; for example between the processor and a memory chip.

The consequence of using integer binary storage is that individual memory addresses can never contain data representing 'real' (floating point) numbers *as such*. The decimal number 3.76 might be represented by the value '3' in one address and the value '76' in the following address, although there are a number of other ways in which real numbers may be stored.

In Part 1 of this book we shall be using the binary representation of numbers extensively because in this format it is much easier to follow exactly what is happening to data when certain assembly language instructions are executed. It is a good idea to get into the habit of imagining your data in binary form in the earlier stages because it helps considerably in understanding some of the more complex concepts involved. In Part 2 we shall go on to use hexadecimal numbering, which will allow us to represent numbers in a much shorter form without straying too far from the clarity which binary numbers allow.

It is important to remember that *all* data contained in memory is in binary numeric form, including program instructions and textual data. If we were to list the data bytes contained in a block of addresses it would be difficult to distinguish between those values which represent program instructions and those which represent program data. How, then, is the processor able to distinguish between them?

The answer is that every assembly language instruction has a unique numeric code of its own, consisting of between two to ten bytes and, providing the processor begins by reading a program from the very first instruction byte, it is capable of decoding and distinguishing the form and function of every byte thereafter. However, if the program begins execution at the wrong address, or if an instruction has been coded incorrectly, the processor is no longer able to make sense of any of the code and the result is usually a complete system crash.

The following diagram shows the binary code of a 68000 addition command, `ADDQ #1,D2` which adds the value 1 to another operand which is contained in a temporary storage location called a register: in this case register 'D2'.

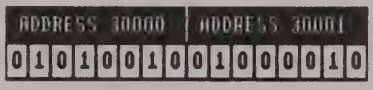


FIGURE 1-3.

This is a two-byte instruction consisting of the byte values 82 and 66; a code which uniquely represents the assembly language mnemonic `ADDQ #1,D2`. This object code consists of a 4-bit code representing '`ADDQ`' (0101), a 3-bit code representing the number to be added (001),

a zero, which is an additional part of the 'ADDQ' code, a 2-bit code specifying the (two-byte) size of the operation (01), a 3-bit code indicating the addressing mode of the instruction (000) and a 3-bit code indicating register 'D2' (010). Together these constitute two binary bytes: 01010010 (82) and 01000010 (66). The instruction `ADDQ #1,D2` is what you would actually write in your program and the values 82 and 66 are the object codes which are assembled for execution.

The number of bytes required for each possible variation of a particular instruction is fixed and the CPU therefore knows that the next byte it comes across will be the beginning of the following instruction.

Code representing data, such as variables and arrays, is stored in completely different areas of memory from program code and under normal circumstances the processor will never attempt to execute it by mistake.

Words and Long Words

Although a single byte has a maximum value of 255, you will obviously want to work with numbers which are much larger than this. In fact, the binary byte is only a basic unit of data; you can store binary numbers using several bytes if you wish. For example, a 2-byte integer represents a binary number of 16 binary digits, giving a decimal range of 0 to 65 535. A 16-bit value is termed a *word* and a 32-bit (4-byte) value is termed a *long word*, which can represent a decimal value between 0 and 16 777 215. When the 68000 is referred to as a '16-bit' microprocessor, this does not mean that you are limited to 16-bit numbers; it is simply a reference to the way in which the processor deals with numbers internally; it can 'read' or 'write' up to two bytes at a time, which does not actually affect the size of data which the programmer can deal with. A '32-bit' or a '64-bit' computer would be faster – which would make it more suitable for calculating a large prime number, predicting the world's weather or problems of a similar magnitude.

Data is stored in memory with the higher (most significant or 'hi') part of a binary value stored first, followed by the lower (least significant or 'lo') part. A single byte would, of course, be stored on its own in

a single address. A two-byte (word) value, such as 36829, would be expressed in binary as

```
1000111111011101
```

and would be stored in memory as follows:

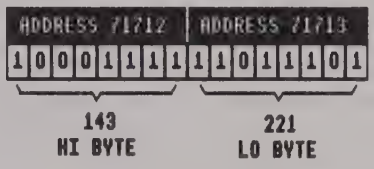


FIGURE 1-4.

The decimal value is found by multiplying the hi byte by 256 and adding the lo byte (143 * 256) + 221 = 36829

A 4-byte (long word) value, for example 131097, is expressed in binary, as

```
00000000 00000010 00000000 00011001
   0        2        0       25
```

and would be stored in memory as follows:

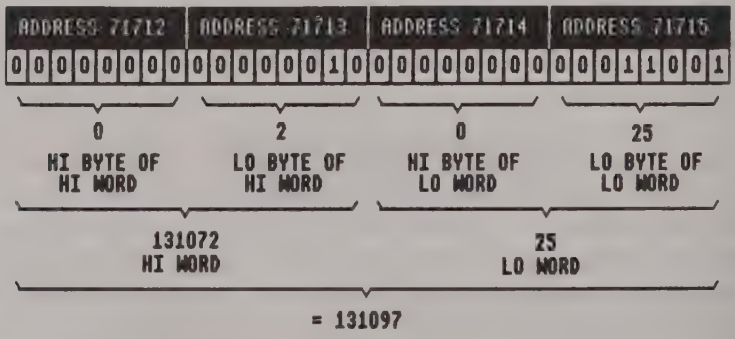


FIGURE 1-5.

In this case the decimal value is found by multiplying the hi word by 65536 and then adding the lo word: (2 * 65536) + 25 = 131097.

Note that although byte and word values consist of 8 and 16 bits respectively, there are occasions when they need to be stored in longer binary form, such as 32 bits. In these cases the values can simply be extended with zero bytes and stored as if they were 32 bit numbers, e.g.:

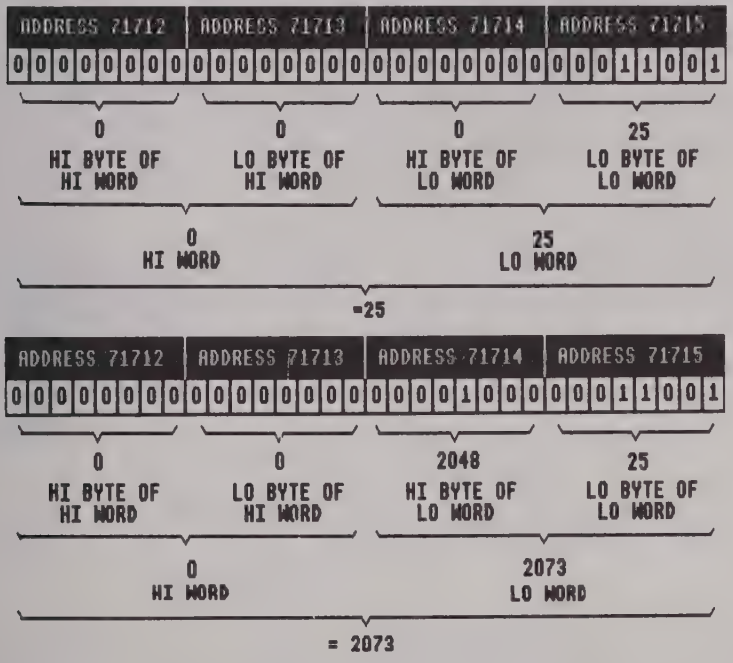


FIGURE 1-6.

Similarly, single byte values can be stored as words:

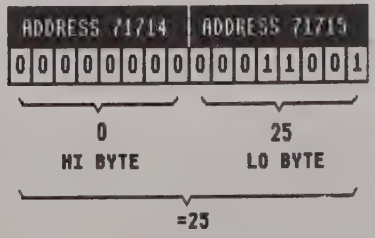


FIGURE 1-7.

It is important to note that *all word and long word sized data items must start at an even numbered address*, otherwise the system will signal an error condition. Byte data may be stored and accessed at both even and odd numbered addresses.

Here is a list of some of the different types of data which may be stored in memory addresses:

1 *instruction codes:* stored as 1–5 words

2 *byte data:* stored as 1 byte

3 *word data:* stored as 2 bytes

4 *long word data:* stored as 4 bytes

Before going on to look at how programs are organized in memory, we shall first look at how memory is typically arranged, so that we can imagine our program data in context.

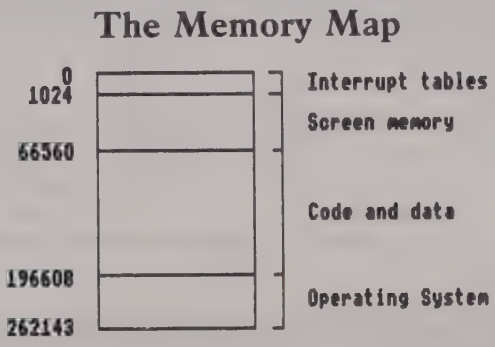


FIGURE 1-8. Memory map.

In this simplified map of a computer's memory, you can see clearly how the space allocated to data is divided into different sections. On different computers the addresses of these divisions will vary and other specialized memory areas will be reserved for special data and functions. In this model for example, there is no area reserved for a BASIC language interpreter, although many computers may incorporate this. Most machine manuals will include a memory map diagram similar to this to show you how memory space is allocated.

The first section, from addresses 0 to 1023 is reserved for special tables and other data used by the operating system. The second section, from 1024 to 66559 is reserved for the VDU display. Any data placed in one of these display memory locations will appear as an image on the screen. If a computer has colour capabilities, the data specifying the background and foreground colours of an image will also occupy this section.

The display area is best illustrated as a two dimensional array of memory addresses, with each row corresponding to the full width of the screen and the columns representing the vertical height of the screen. In this diagram, only the addresses corresponding to the top left hand corner of the screen are illustrated:

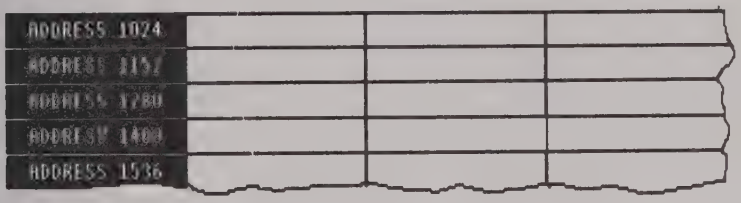


FIGURE 1-9. Display memory corresponding to top left-hand corner of screen.

In our imaginary computer the 'display' section is fixed and so the precise location of any point on the display screen can easily be calculated. On some computers, although the actual length of the display section remains constant, its position within memory may change constantly.

The 'code and data' section is the area which is reserved for programs and their associated data, whether they be written in BASIC, assembly language or any other language.

In BASIC programs the data area is used to store variables and arrays and their formats are organized by the BASIC interpreter. In assembly language, variables, arrays and other types of data are stored in reserved blocks of memory created by the programmer rather than by resident software and therefore it is necessary to be able to identify the locations of the addresses in this area in order to store and retrieve this information.

The operating system consists of sets of routines designed for structuring the way in which the system functions and includes control mechanisms for communicating with disc drives, error handling, input and output operations, keyboard and VDU communications and other administrative tasks. Customized extensions to the operating system may include complex arithmetic and graphics functions and other routines which, like most of the operating system, are accessible from user programs.

Program and Data Storage

We shall now look at a simple model of how a typical assembly language program and its data is stored in memory. We shall assume that our program occupies addresses 71680 to 71707. Henceforth the term 'address' will be used interchangeably to refer both to the number of a particular location and the physical location itself. The term 'content' will be used to refer to the data contained in an address, irrespective of whether it is code which forms part of a program instruction or whether it is data representing a character or a numeric value.

Thus our program code occupies 27 addresses and the contents of the first six of these can be pictured as follows.

ADDRESS 71680	116	Instruction
ADDRESS 71681	0	Data
ADDRESS 71682	24	Instruction
ADDRESS 71683	58	Instruction
ADDRESS 71684	0	Data
ADDRESS 71685	44	Data

FIGURE 1-10. Program code.

The instruction codes each occupy a varying number of addresses and in some cases are followed by addresses containing data. These data items are treated as being part of an instruction and may represent either constant data values or data representing the address of another location.

The variable data for our program, stored in the 'data' section of memory, is 5 bytes in length and occupies, say, addresses 71712 to 71716.

ADDRESS 71712	24
ADDRESS 71713	21
ADDRESS 71714	28
ADDRESS 71715	28
ADDRESS 71716	31

FIGURE 1-11. Program data.

You will notice that the 'program code' section also contains data as well as instruction codes, and you may be wondering why this data is different from the data in the 'data' section. The reason is that there is a distinction between 'immediate' data, which relates to a particular instruction, and array and variable data stored in the data area. For example, in the BASIC instruction `LET A=8*X` the value '8' is stored along with the BASIC instruction itself, whilst the value of the variable 'X' is fetched from the 'data' section of memory during execution. In other words, the '8' is always 8 and is part of the program command, whilst X may be one of a number of possible values, and would be stored in the variables section of memory.

Now consider what happens when our program is executed. It may be one which has been designed, for example, to add the value 48 to each of the contents of the addresses in the 'data' section and to print the results to the screen. It should be easy to follow the general sequence of events which take place.

Two of the instructions in the program area instruct the computer to take the constant 48 (termed the source operand) and add it to the contents of the first address in the data area (termed the destination operand). The next instruction then places the ASCII character corresponding to the sum in some of the addresses in the display section. Figure 1.12 illustrates part of the processes involved.

The ASCII characters in a computer are a standard set of characters which include all alphabetic letters, numerals, punctuation marks and

essential control codes such as line feed, carriage return and so on. The binary codes for all these characters are always positive numbers in the range 0 to 127. The remaining ASCII codes, 128 to 255, are assigned by different computer manufacturers to various other functions and are not standardized.

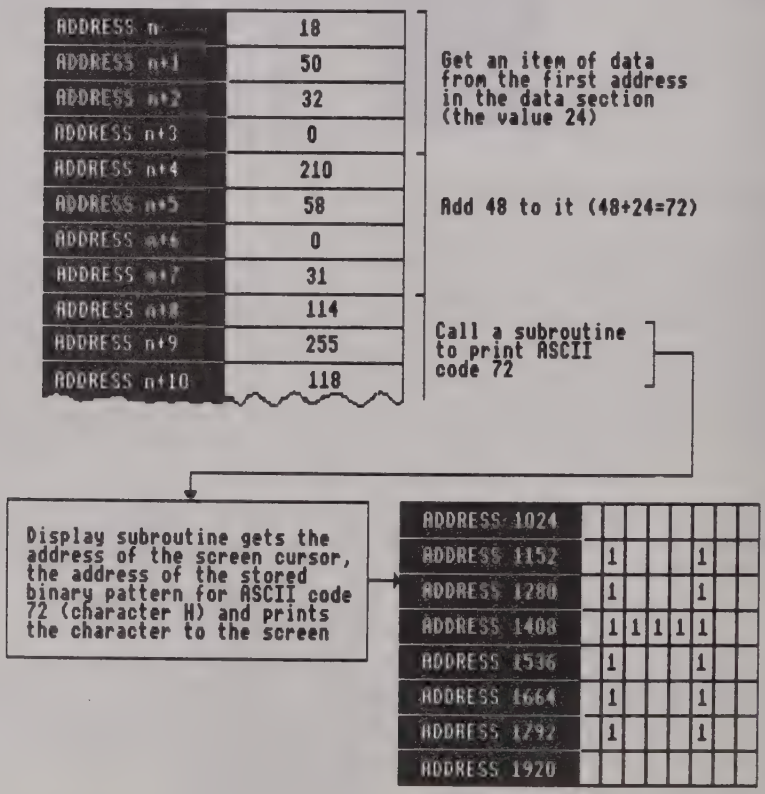


FIGURE 1-12.

In practice, the operation involves the *movement* and the *processing* of data in which all the relevant items, including the instruction and the data codes, are physically copied into the CPU, processed, and the resulting data transported to memory addresses in the display area.

The technical aspects of these operations need not necessarily be understood in detail by the programmer, since the CPU controls the sequence of events automatically.

When the first numbers have been added together and the corresponding ASCII character printed to the screen, the process can be repeated with the second item of data by looping back and repeating the first instruction. This process may then be repeated until all five additions have been completed and the characters printed to the screen.

The following example shows how the same operation might be performed in BASIC:

```
10 FOR count = 5 to 1 STEP -1
20 READ V
30 PRINT CHR$(48 + V)
40 NEXT count
50 DATA 24,21,28,28,31
```

This produces the values 72, 69, 76, 76 and 79, for which the corresponding printed ASCII characters are 'HELLO'.

The assembly language and BASIC processes have a number of similarities:

1 The instructions in both programs are executed sequentially, except for the loop sequence, which allows a section of the program to be repeated.

2 Both store their variable data separately and call up each item of data when it is needed.

3 Both require their instruction codes and program data to be sent to the CPU for processing during execution.

If we look at a flowchart for each version we shall also see some very important differences in the way in which the operations are carried out:

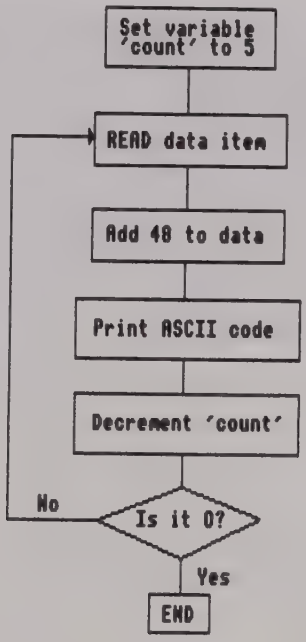


FIGURE 1-13. Flow chart for BASIC version.

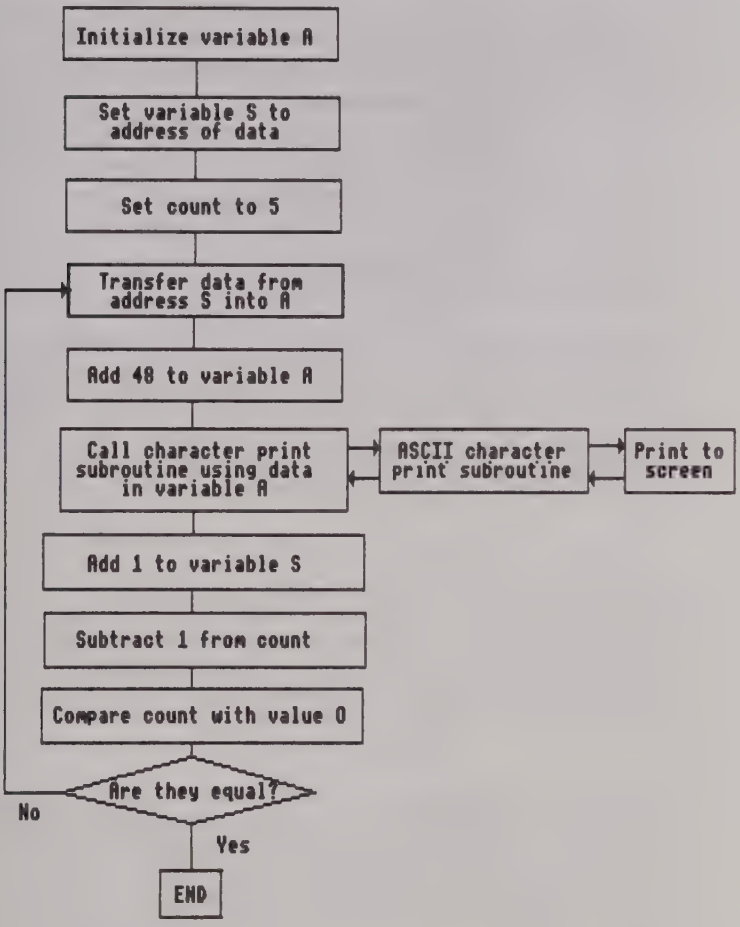


FIGURE 1-14. Flow chart of assembly-language version.

The first main difference is that the assembly language version is much longer, breaking down the individual steps in the operation to smaller, closely defined units. The BASIC program operates in almost the same way when it is executed but the individual steps are handled entirely by the machine's BASIC interpreter, leaving the programmer to structure the program using easily understood broad concepts such as the `FOR..NEXT` loop and the `READ..DATA` functions. Hence, the BASIC flowchart illustrates the flow of BASIC language mechanisms while the assembly language flowchart illustrates the *actual* flow of events between memory and the CPU.

The second difference is that although the BASIC version implies that data should be moved around in memory, the physical movement of the data is not specified. Although items of data are added and then transferred to the screen, the program itself does not concern itself with the precise locations of the data or where it should be moved to.

The assembly language version on the other hand, is very specific about where the different elements of the program are located and where they should be moved to. Thus an assembly language program consists not only of instructions specifying standard operations, such as addition and subtraction, but of instructions designed to locate and control the movement of different categories of data throughout the system.

Unfortunately, the range of operations which the CPU can perform is very limited and therefore it is not possible, for example, to write an instruction which transfers a numeric value into it and requests its cosine or logarithmic value as you would use `COS` or `LOG` in BASIC. In fact it deals entirely with integer values and is only capable of adding, subtracting and performing logical operations on them such as `AND` and `OR`. Although the CPU cannot handle multiplication and division directly, the 68000 system incorporates instructions which can perform these operations and therefore this is not a problem.

Should you require a more complex computation such as the cosine or logarithm of a number, then it is necessary to break down such an operation in terms of simple arithmetic and logical steps which the CPU can perform. There is no need to be deterred by this because with many computers it is possible to take a short cut by using ready made mathematical functions which are already programmed into the operating system. You can simply regard such functions as assembly language subroutines and call them up from within your own

program. In some cases a system will incorporate an arithmetic co-processor which is designed to handle transcendental functions and floating point numbers. These incorporate their own instruction set and allow you to specify the operands involved and the mathematical function required.

Assemblers

Assembly language, like any other computer language, consists of a set of instructions, representing specific program operations, and a syntax, which specifies the format in which instructions may be written. The reason why it is called assembly language is that it is designed to be used with an *assembler program*, which interprets the instruction mnemonics written by the programmer (the *source code*) and converts them into a set of numbers which can be interpreted and executed by the computer. The resulting numbers are called the *object code* or *machine code*.

A program which is written in assembly language cannot be run in the way that a BASIC program can; only the object code can be executed, and it is this code which is assembled and saved to tape or disc prior to being loaded and executed. The resulting machine code program is simply a list of numbers which represent particular instructions and some which represent data, as described previously.

When a BASIC program is run the instructions are *compiled* (translated) from BASIC to machine code at the time of execution. If we were to use a *disassembler* (i.e. a program for converting machine code back into assembly language) to examine the contents of a machine code program we would be given a listing which looked something like the following:

Address (hexadecimal)	*Object Code (hexadecimal)*	*68000 Instruction Mnemonics*
29CE8	7200	MOVEQ #00, D1
29CEA	7602	MOVEQ #02, D3
29CEC	41FA0038	LEA 38(PC)!29D26, A0
29CF0	7001	MOVEQ #01, D0
29CF2	4E42	TRAP #2
29CF4	7400	MOVEQ #00, D2
29CF6	183A002C	MOVE.B 2C(PC)!29D24, D4

```
29CFA       45FA0022        LEA 22(PC)!29D1E, A2
29CFE       12322000        MOVE.B 00(A2,D2.L), D1
29D02       D23A001F        ADD.B 1F(PC)!29D23, D1
```

You will see that we are given three different types of information. In the left hand column we have a list of addresses (in hexadecimal format) which represent the locations in which the machine code program is stored in memory. The right hand column contains the assembly language instruction mnemonics for the program and the centre column contains the hexadecimal machine code version of the assembly listing on the right. The machine code is divided into different groups of bytes, with each individual byte (2 hexadecimal digits) representing the contents of a single memory address.

The first instruction, `MOVEQ #00,D1` is translated into two bytes of object code, occupying addresses 29CE8 and $29CE9_{H}$. The second instruction also occupies two bytes while the third instruction, `LEA 38(PC)!29D26,A0` consists of 4 bytes, and occupies addresses 29CEC to $29CEF_{H}$.

The machine code program would be executed sequentially, starting with the first number of the code of the first instruction and continuing through each memory byte until a final instruction is reached. In between there may be program loops, subroutine calls and conditional jumps, just as there are in BASIC programs.

The program is initiated either by an auto start mechanism, as soon as it has been loaded into memory, or by calling it with a high level language instruction, for example from within a BASIC program or from the operating system. Alternatively, a program may be a *subroutine* which is called from within some other machine code program.

In Chapter 8 we shall be looking at assembler programs in more detail, showing how the translation from assembly language to object code is organized.

Chapter 2

Registers and Addressing Modes

Sources and Destinations

When an item of data is accessed and transferred from one location to another, for example when data in one location is added to another item of data, the the first item is termed the *source* operand and the second item, the *destination* operand. The terms source and destination are used extensively in assembly language to distinguish the status of the operands involved in an operation. As we shall see later, the possible source and destination locations for operands involved in different operations may be subject to entirely different rules. In the case of the addition instruction in the previous chapter (`ADDQ#1,D2`), the source operand, 1, is an immediate numeric constant, while the destination operand is located in a register. The register itself is the destination location, the contents of the register being the actual destination operand. When the two operands have been added, the result is automatically stored in the location which previously held the destination operand.

Registers

An operand which is being addressed by an assembly language instruction will be in one of two places; either in a memory location (in either the code or data sections) or in a *register*.

Registers are identified by alphabetical letters, like BASIC variables, and are literally temporary memory locations which are situated in the CPU rather than in RAM or ROM. A register may be treated in much the same way as any other memory location in that data may be loaded into one or moved out of one into another location and the contents of registers may have arithmetic operations performed on them. Some

are also used as temporary variables for holding the codes of memory addresses.

In the 68000 there are two main types of user registers; *data registers* (D0, D1, D2, D3, D4, D5, D6 and D7) and the *address registers* (A0, A1, A2, A3, A4, A5 and A6), each of which hold up to four bytes of data. The address registers are normally used for holding the addresses of memory locations.

Data Registers

Data registers are 32 bits in length and are capable of holding data of byte, word and long-word lengths. You can imagine them simply as labelled boxes divided into four byte-sized partitions which can be loaded with binary values. The least significant byte of a value occupies the right hand partition and the most significant byte occupies the left, corresponding to the way in which you would normally format a binary number.

As an example of the use of a general data register, suppose that you had a program in which repeated calculations were being performed on a set of figures and you wish to add up the totals. Just as you might use a variable in BASIC to accumulate the total, you could use a data register such as D2 in assembly language. As with a BASIC variable, the value contained in a data register can be used in subsequent program instructions.

FIGURE 2-1.

Address Registers

An address register is also 32 bits in length and, like a data register, is used as a source or destination for operands. However, whereas the purpose of a data register is simply to hold data which is being used in a program, *an address register is used specifically to hold address numbers*. Although an address register can actually hold a 32-bit value, only the least significant 24 bits (bits 0 to 23) of a value are used to specify an address, hence the maximum restriction in a 68000 based computer to 16 777 216 bytes (16 megabytes) of physical memory. Note that a 68008-based computer is restricted to a maximum of 1 megabyte, since only the lower 20 bits of an address register can be used to specify an address.

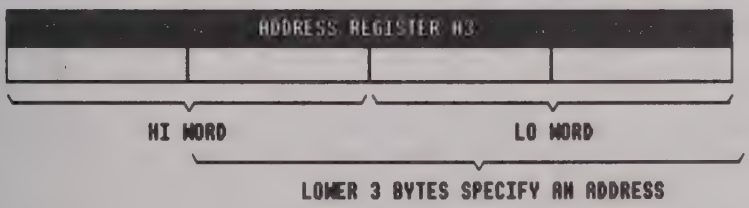


FIGURE 2-2.

Additional Registers

In addition to the address and data registers, the 68000 has a small number of other registers which are used for special purposes.

Program Counter

The PC (program counter) register is a 32-bit register whose lower 24 bits always contain the address of the program instruction that is currently being executed and is automatically updated by the system as each instruction is processed. Whenever a program branches to a subroutine or jumps to another point in the code, the new address is automatically loaded into the PC register so that the processor knows from where in memory to fetch the next instruction. The address contained in the PC register can be altered by the programmer to redirect execution to a different point in the program, but this is

normally only done in special circumstances, for example if there is an occasion when you want to return from a specific subroutine to a point other than the normal return address. Under normal circumstances, you would only need to refer directly to the PC register if you wish to refer to an address which is a specified number of bytes relative to the current execution address.

Status Register

The status register (SR) is a 16-bit register which is used to hold a number of bit-sized 'flags' which indicate the current status of the system. It is used to determine, for example, whether a computed value is positive or negative, whether it is less than, equal to or greater than some other value or whether it involves an arithmetic 'carry' or 'borrow'. The main functions of the status register are described in detail in the following chapter.

Stack Pointer

A special area of memory, termed the *stack*, is reserved for the storage of temporary data and variables. The current location of the 'top' of the stack; the point at which fresh data may be stored or old data removed, is at the address whose value is contained in address register A7. This register is therefore referred to as the *stack pointer* (SP), since it 'points' to the current stack top.

Addressing Modes

The available methods by which data may be accessed and moved around between memory locations, registers and the processor itself are termed *addressing modes* and these are clearly defined. It is not necessary to think too consciously about which mode to use in a particular situation, any more than it is necessary to think about the rules of grammar whenever you wish to speak. If you know where your data is and what you want to do with it then the appropriate addressing mode will come to mind automatically in most cases. However, it is useful to be aware of what is possible and what is not and the following section describes the formal structure of these modes. In this section we shall be starting to use some actual assembly language instructions, beginning with the `MOVE` instruction, which is used to copy operands from a source to a destination, and the `ADD` and `SUB` instructions, which add and subtract operands.

There are two main categories of addressing modes: *memory addressing*, in which operands contained in memory are addressed, and *register addressing*, in which operands located in registers are addressed. Although registers do not actually have address numbers of their own to identify their location, the term 'addressing' nevertheless includes register as well as memory references.

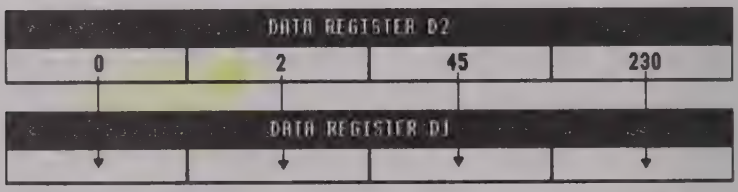
Implicit Addressing

Certain assembly language instructions involve the use of particular registers without explicitly stating which registers are to be used. In all cases, these instructions use one or more registers for the same reasons as you would choose to use one yourself: to store, retrieve, move, process or modify data. In these cases however, the particular register used is chosen for a specific purpose and therefore there is no need for the programmer to indicate which one is required. For this reason, this addressing mode is termed *implicit*, for the register to be used is implicit in the instruction itself. Such instructions include, for example, `RTS` (return from subroutine) which always implies the contents of the PC register. Some instructions are not only implicit, in that they imply the contents of SP and/or PC, but also involve the use of other addressing modes.

Register Direct Addressing

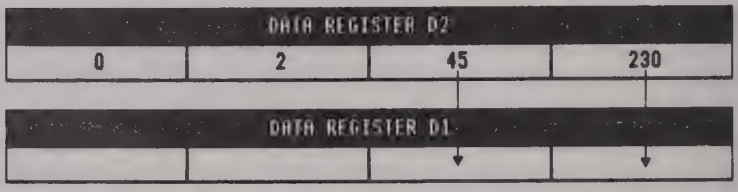
This addressing mode is used for operations performed on data contained in registers. For example, the contents of two registers may be added together or the contents of one may be transferred to another. In BASIC, equivalent instructions would include `LET A=B` or `LET A=A+B`.

In the first of these examples you will notice that the `MOVE` instruction has a '`.L`' after it, whilst in the second, it has a '`.W`'. The reason for this is that the first example is a 'long-word' operation, in which all four bytes of one register are copied into the other. In the second example, which is a 'word' operation, only the least significant word of a register is involved, leaving the most significant word of each register unchanged. This distinction has nothing to do with the fact that one is a `MOVE` operation and the other an `ADD` operation – we



OPERATION: MOVE.L D2,D1 (copy the entire contents of data register D2 into data register D1)

FIGURE 2-3.



OPERATION: ADD.W D2,D1 (add the low order word of register D2 to register D1)

FIGURE 2-4.

could just as easily have specified the instructions `MOVE.W D2,D1` and `ADD.L D2,D1`; the '`.W`' and the '`.L`' being the parts of the instructions which determine the size of the operands involved in the operations.

In both cases we could alternatively have used the instruction suffix '`.B`', signifying that only the least significant byte of the register contents be involved. If there is no '`.B`', '`.W`' or '`.L`' suffix after such an instruction, it is normally assumed by default that '`.W`' is intended although there are exceptions to this rule.

The two registers involved can both be data registers or one of them can be an address register. In the instruction `MOVE A2,D1` for example, the source operand is specified by address register direct addressing and the destination is specified by data register direct addressing.

The same type of operation performed using an address register as a *destination* is slightly different. In this case, only two sizes of data may be used in an operation – word and long-word. Therefore, the '`.B`'

suffix cannot be used. Some instructions indicate the use of an adress register destination by the addition of an 'A' to the instruction mnemonic, such a `MOVEA` and `ADDA`. The other important difference is that, if an address register is being used as the destination register for an operation, *all four bytes are affected*, unlike a data register where the unused bytes remain unaffected. If we were to transfer a word of data, say the value 300 (binary 0000000100101100) from register D1 to register A1, the most significant word of A1 would automatically be *sign-extended* to a full 32 bits – in other words, the highest bit of the least significant word, bit 15, would be copied into bits 16 to 31 of A1, as follows:

OPERATION: MOVEA.W D1,A1 (copy the low order word of data register D1 into the low order word of address register A1 and sign extend the value of bit 15 through bits 16 to 31)

FIGURE 2-5.

The implications of this are very important because it affects the way in which a memory address is specified. If we load a 4-byte (long-word) address into an address register, then the address which it represents will be the one specified by the least significant 24 bits of the register, as described above.

However, if we load a word value into an address register, the remaining two bytes of the register, the Hi word, will carry the *sign-extension* of the value and part of this will be incorporated in the 24-bit address specification. How does this affect the value of the address which we wish to specify?

There are two possible alternatives. If the value loaded into the address register is between 0 and 32767 decimal, the binary representation in the register will be as follows:

0111111111111111 binary
= 32767 decimal

which, after being sign-extended in the address register would be the same:

0000000000000000011111111111111 binary
= 32767 decimal

So far so good. This address is the highest in the bottom 32K of memory and is exactly the address we specified when we loaded it into the address register; sign-extension left it unchanged.

However, the alternative possibility is that we load a word value between 32768 and 65535 decimal into the address register. The binary representation would be as follows:

1000000000000000 binary
= 32768 decimal

which, after being sign-extended in the address register would become (counting only the lower 24 bits):

111111111111111110000000000000000 binary
= 16 744 448 decimal

This address happens to be the first address of the *top 32K* of memory – not the address we originally specified. This is not a fault but an advantage. If we actually want to access a memory address in the top or bottom 32K of memory then we only need to use a word (in the range 0 to 32767 or 32768 to 65535) to specify the address, which is a saving in both time and space. Hence, a word-sized address is termed a *short* address.

If we want to access an address *anywhere* in memory, we use *long* addressing; that is, we load a long word address value into the address register which, since it will not be sign-extended, will produce exactly the address which we specify, up to a maximum of 24 bits. This principle applies to any addressing mode which makes use of an address register.

One final point to note about address register direct addressing is that in some cases where the destination of an operation is an address register, for example in `ADDA` (add address), `SUBA` (subtract address) and `MOVEA` (move address), *the condition flags in the CCR register are not affected by the operation.*

Absolute Addressing

An absolute addressing operation is one in which an operand is identified by its actual memory address. In BASIC terms it is similar to the instructions `LET A=PEEK(20000)` or `POKE 20000,A`. In practice, the address is normally identified in the instruction by means of a user defined label rather than an actual number.

Direct addressing has *long* and *short* forms: a 'long' address, as already indicated, being specified by a 32-bit number and a 'short' address as a 16-bit number, sign-extended to 32 bits.

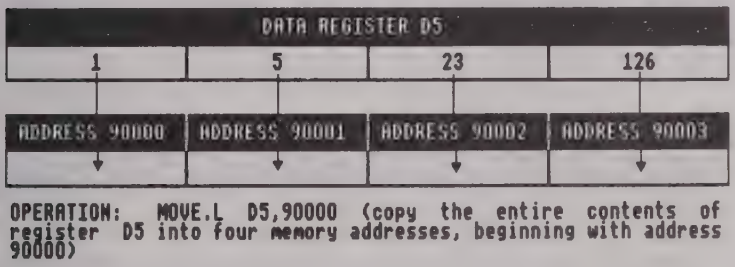


FIGURE 2-6.

This is the long form of direct addressing. The address value is a 3-byte number and therefore the system automatically recognizes it as a literal address. The '`.L`' suffix has nothing to do with the address size; it performs its usual function of denoting the size of the data which is to be transferred.

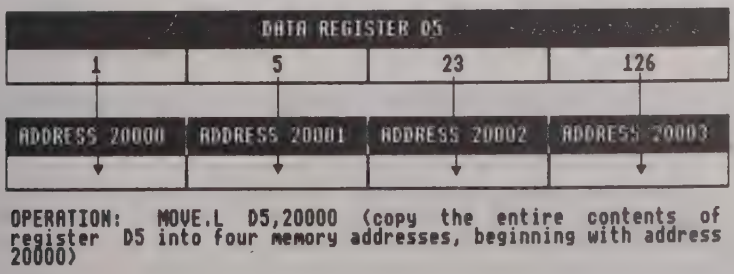
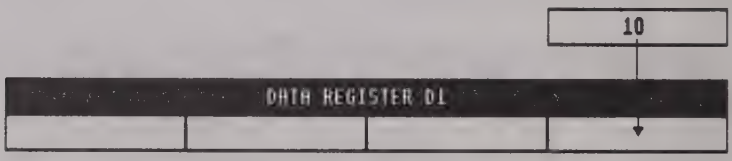


FIGURE 2-7.

This is the short form of direct addressing. The address value is a 2-byte number, which is automatically sign-extended to 32 bits. Since the 2-byte number is within the range 0 and 32767, it falls into the bottom 32K of memory and, after sign-extension, it therefore remains unchanged. As in the previous example, the '.L' suffix indicates that four bytes are to be transferred.

Immediate addressing

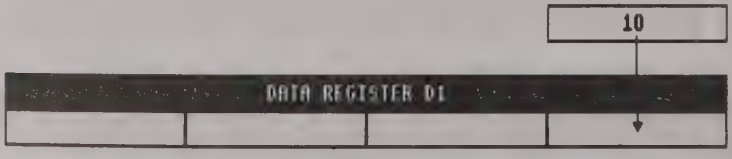
This is used for operations involving an immediate numeric constant. For example, a number may be transferred from or loaded into a register or, it may be added to or subtracted from a number which is already in a register. Equivalent BASIC instructions would be `LET A=10` or `LET A=A+10`.



OPERATION: MOVE.L #10,D1 (copy the immediate value 10 into the low order byte of register D1 and zero the three higher bytes of D1)

FIGURE 2-8.

Note that here, the '.L' suffix has been used to indicate that D1 should contain a 32-bit binary representation of 10. This would normally be done if we wish to ensure that the unused bytes of a data register are zeroed. The '#' sign is used to indicate that the value is an immediate constant.



OPERATION: MOVE.B #10,D1 (copy the immediate value 10 into the low order byte of register D1, leaving the three higher bytes unaffected)

FIGURE 2-9.

In this case the '.B' suffix has been used, specifying that the value 10 is only copied into the least significant byte of D1, leaving the rest unaffected.

Immediate Quick Addressing

Quick addressing is a form of immediate addressing in which operations can be formed faster than usual. Many programs, or certain routines within programs, involve a great deal of processing; especially those involving program loops or which move a lot of data from one location to another. It becomes vital, therefore, for programs to be executed in the shortest possible time and the saving of a few microseconds (millionths of a second) in the execution time of individual instructions can have an appreciable effect on the overall running of a program. You only have to consider the speed which is required to update the display screen in a fast moving graphics program, such as a flight simulator, to realise that every microsecond that is saved in a program can be of great significance.

The 'quick' instructions, therefore, are designed to take advantage of the fact that when very small values of data are being dealt with, it is worth providing a quicker method of processing them rather than rely on an operation which is designed to deal with larger and therefore more complex items of data.

There are only three of these instructions: `MOVEQ` allows you to set the entire 32 bits of a register to an 8-bit value, thus avoiding the use of a longer and less efficient operation such as `MOVE.L`. `ADDQ` is slightly different; it is designed for adding values in the range 1 to 8 to the contents of a register. `SUBQ` subtracts data from a register in the same numeric range.

Address Register Indirect addressing

This is equivalent to a BASIC instruction such as `LET A=PEEK(X)` or `POKE X,A`. It is used to access values which are stored in locations whose addresses are contained in registers.

Suppose that we have a byte value stored somewhere in memory which we wish to copy into data register D2. We may have no idea exactly where in memory the data is stored but we do know that its

address is currently contained in, say, address register A4. We can therefore get at it indirectly, via A4. To indicate indirection we enclose the address register in brackets, which tells the system that it is not the contents of A4 which we wish to load into D2 but the contents of *the address pointed to* by the contents of A4.

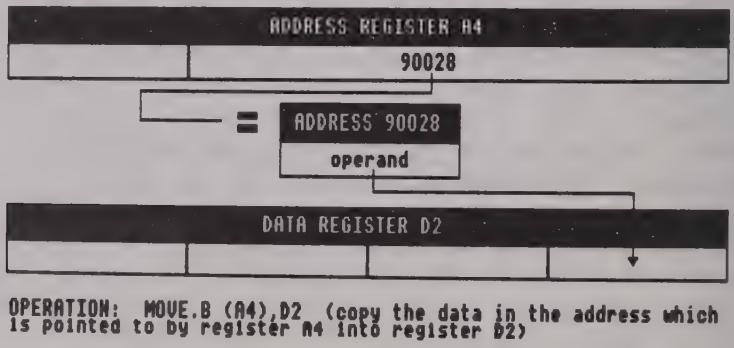


FIGURE 2-10.

The operation can, of course, be performed the other way round:

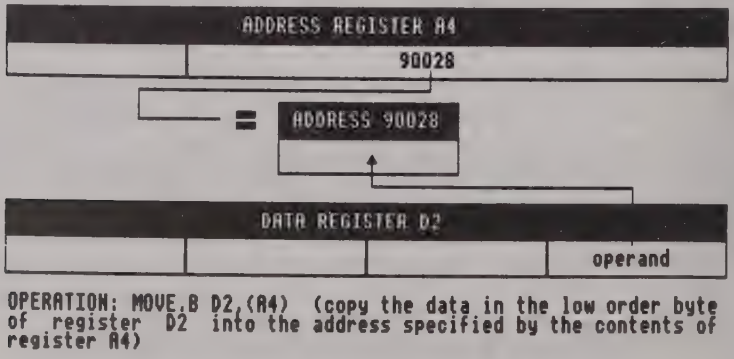


FIGURE 2-11.

This addressing mode indicates some interesting possibilities. Suppose, for example, you had a list of data somewhere in memory – say 12 bytes representing the number of gallons of gas you have purchased over a twelve month period, with each byte representing a month's total. You might want to use these figures in a program which is

designed to work out the average monthly figure for the year. If the address of the first month's gas consumption is contained in address 90028 then you could load this address into an address register so that it points to the required month. The arrangement would then look something like this:

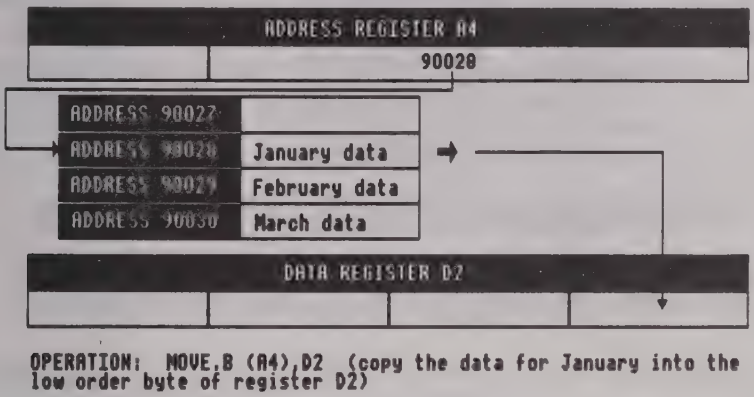


FIGURE 2-12.

If you want the figure for January for use in your program then you could load it into a data register using an instruction such as MOVE.B (A4),D2, as described above, and then use the data in the data register in your averaging program. It should also be clear that you could copy two or four month's figures in one go from the memory block into a data register, simply by using the suffixes '.W' or '.L' instead of '.B' in the instruction.

Having accessed your first item of data, you then need to access data for further months, which could easily be done by incrementing the value of A4 so that it points to the next required address and then using an ADD instruction such as ADD.B (A4),D2 to add its contents to the data register. However, there is one type of addressing mode which will perform this function for you: the *address register indirect with postincrement*, covered next.

Address Register Indirect with Postincrement

If we were working on the same block of data as before, we could substitute the following instruction for the one we used previously: `MOVE.B (A4)+,D2`. In this case we are doing exactly what was described before: the data contained at the address indicated by A4 is loaded into a data register and the address register is incremented by 1, 2 or 4 to point to the next required address, depending on whether we are transferring byte, word or long-word sized chunks of data. In this mode however, the incrementation of the address register is carried out automatically after every transfer of data; a function which is specified by the '+' sign in the instruction.

Address Register Indirect with Predecrement

This addressing mode works exactly like the one above except that it operates in reverse. The address register is decremented by 1, 2 or 4 and then the data is accessed, hence the term 'predecrement'. We might use this addressing mode to access items of data in memory in reverse order. In this case A4 would initially be loaded with the number of the address immediately *after* the December data; address 90040. The instruction `MOVE.B -(A4),D2` would then decrement A4 by 1, and then transfer the byte at 90039 into D2. Here, the predecrement mode is indicated by the '-' sign next to the brackets.

Address Register Indirect with Displacement

Although the regular address register indirect mode allows you to address individual memory locations via an address contained in an address register, it is not versatile enough to allow you to address locations which are situated *relative to* a particular address. If you have a list or table whose base address is contained in address register A3, how can you address a location within the table at, say, offset 3 from register A3? This kind of indirect memory access is achieved using the concept of a *displacement*, in which a constant value, contained in the instruction, is added to the address register used.

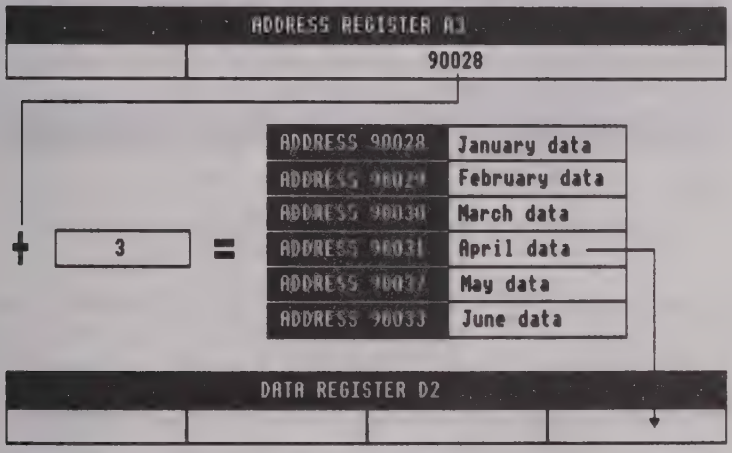


FIGURE 2-13.

In this example the displacement, 3, is added to the A3 register to enable access to the contents of the address 3 bytes on, relative to the address pointed to by register A3.

In this addressing mode, the 16-bit displacement is always sign-extended so that it represents a displacement in the range –32K to +32K. The concept of minus numbers will be discussed in the following chapter.

Address Register Indirect with Index and Displacement

Similar to the above mode is 'address register indirect with index and displacement', in which one register is used to point to the base of a block of data and another, either an address or a data register, is used to hold an index offset. An additional displacement value is placed outside the brackets as in the previous mode. The advantage of this over the previous mode is that the index register can be altered under program control to point to a number of items within a table, relative to the base register.

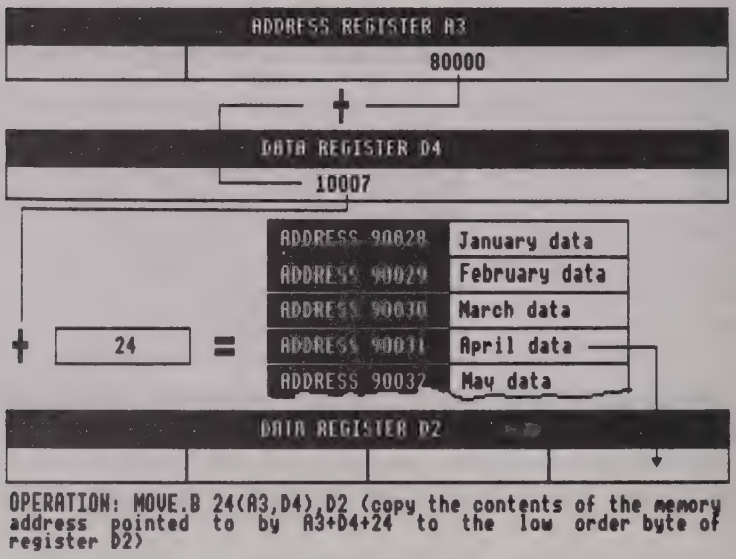


FIGURE 2-14.

In the above example the base register is A3, the index register, D4, contains the value 10007 and the displacement is 24. The address which is being accessed is therefore the sum of the contents of A3 and D4 plus the value 24.

It is also possible to specify the size of the index register. In the above example the byte at address A3+D4+24 is being copied into register D2. If we wanted to specify that the entire 32-bit contents of D4 were to be used as the index variable we could modify the instruction to `MOVE.B 24(A3,D4.L),D2`. In this case the '`.L`' length specifier defines the size of the contents of the index register and the '`.B`' defines the size of the operand which is to be loaded into D2.

This instruction could, for example, be used to access an operand which is in a block of data whose base address is contained in A3. Within this block is a sub-block whose base is at offset 24 from A3 (counting from zero). Within this sub-block is an address whose offset is contained in D4.

In this mode the immediate displacement is 8 bits in length and is sign-extended, giving a displacement in the range –128 to +127. If the index register is of size 'W', it is also sign-extended giving an index displacement in the range –32K to +32K, otherwise it is treated as a 32-bit positive value..

Program Counter Relative Addressing

The above addressing modes work very well if we happen to know the address, or at least the base address, of the data which we wish to access. However, if our program is one of several which might occupy memory at any one time we cannot guarantee that on every occasion it will be located into exactly the same region of memory. In this case, it does not make much sense for a program to refer to specific address numbers.

This problem is overcome by writing programs which are *position independent* or *relocatable*, which means that they can be loaded and run anywhere in memory. In this case, the location of a particular block of memory can be specified as being relative to a known point; the only known point being the location of the instruction currently being executed and whose address will be contained in the PC (program counter) register.

PC relative addressing instructions are formatted in exactly the same way as indirect addressing instructions, except that PC is substituted for the base register, as follows:

```
MOVE.B 4(PC,),D2          PC relative with displacement
MOVE.W 6(PC,),D5
MOVE.B 6(PC,D3.L),D2      PC relative with index and
MOVE.B 4(PC,D3.W),D1      displacement
```

The following table gives a summary of the addressing modes discussed in this chapter.

Addressing Mode	*Operand Location*
Implicit	Operands implicit in instruction
Absolute (short & long)	Operand at address specified in instruction
Register Direct	Operand contained in a register
Immediate	Operand contained in instruction
Address Register Indirect	Address of operand is in an address register
Address Register Indirect with Postincrement	As above
Address Register Indirect with Predecrement	As above
Address Register Indirect with Displacement	Address of operand is the contents of an address register plus a 16-bit signed displacement value
Address Register Indirect with Index and Displacement	Address of operand is the contents of an address register plus the contents of an index register plus an 8-bit signed displacement value
PC Relative with Displacement	Address of operand is PC plus a 16-bit signed displacement
PC Relative with Index and Displacement	Address of operand is PC plus an index register plus an 8-bit signed displacement

Chapter 3

Condition Flags

The key to any program, whether written in machine code or in any other language, is in the way in which it makes conditional decisions based on the status of various variables. In BASIC, typical decisions might be expressed as:

```
IF A$ = "Y" THEN GOTO 500
```

or

```
IF X>2 AND Y=3 THEN GOSUB 1000
```

or

```
IF count=10 THEN STOP
```

In the last example the variable 'count' is being used as a 'flag' in the sense that in the event of 'count' being equal to 10 then it may be regarded as a flag waved at the computer to indicate that a particular decision has to be made – the decision to `STOP` executing. If the flag is less than or greater than 10 then the need for an alternative decision is being flagged – the decision not to `STOP`.

The first two examples also involve the use of flags, although in these cases they are not so obviously labelled. The simplest way to look at a flag is to regard it as a proposition that is either true or false. Thus, during execution, the program flags the condition that it is true that A$="Y" , or that X>2 and Y=3, or it is false, and the program lines following these statements indicate the appropriate action to be taken.

What the program is doing, irrespective of whether it is originally written in assembly language or in BASIC, is using a 'bit flag' within the CPU to indicate either a true or false condition. Thus, when executing IF A$="Y" the computer is not looking at the similarity of

shape between one alphabetical letter and another and thinking to itself 'this character in variable A$ looks about as much like the letter "Y" as a bullfrog'. It is in fact comparing the binary codes which represent the two items of data and then setting a *bit flag* in a special purpose register to either 1 or 0, indicating either that it is true or false that the codes are identical. Once this bit has been *set* (i.e. it becomes a 1) or *reset* (i.e. it becomes a 0) the program merely needs to make a decision about what to do next.

In assembly language programs these decision flags need to be considered individually by the programmer, because there is no BASIC interpreter to determine automatically which flag is required in a particular situation. Are two items of data being compared to see if they are equal, which is greater than or less than the other, or whether they are positive or negative? In each case the answer obviously depends on the context and a separate bit flag is used in each case.

Let us select a few bit flags to see how they work. Since they only occupy 1 bit each they are all stored in a single register. In the 68000, this special register is termed the CCR (Condition Codes Register), which forms the lower half of a 16-bit register known as the SR (Status Register), which holds various flags indicating the current status of the system.

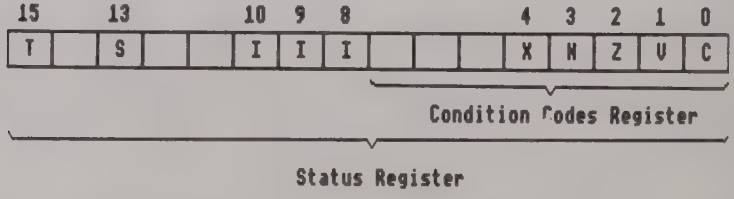


FIGURE 3-1.

Zero Flag (Z)

Taking the *zero flag* first, consider a typical decision. Suppose that a key has been pressed on the keyboard and that we have transferred its ASCII value to register D1. How would we do that? It depends on the computer we are using and on the particular method of keyboard scanning being used, but suppose for the present that our computer has an inbuilt keyboard scanning routine which can be called as a subroutine from our main program and which automatically returns the ASCII code of the last key to be pressed in the low byte of register D1.

What we need to determine is whether the byte code in the D1 register is the same as the ASCII code for Y In other words, is it true or false that the Y key has just been pressed? Naturally we need to know the ASCII code for Y which is 89 (or binary 01011001). The question takes the form: 'Is the byte contained in D1 equal to 89?' This question is technically a *comparison*, so we compare the immediate constant 89 with the contents of D1, using the instruction CMPI (which means compare immediate):

```
CMPI.B  #89,D1
```

or:

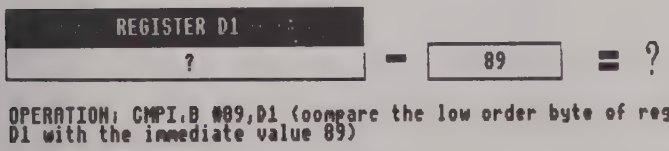


FIGURE 3-2.

How does a computer compare items of data? In this case it must subtract 89 from the contents of D1 (taking care not to actually alter the original value of D1 which we may sometimes want to preserve) and ask itself whether or not the answer is zero, in which case they must be equal, or greater or lesser than zero, in which case they are not equal. The answer is indicated by the *zero flag* which is automatically 'set' to 1 if they are equal or 'reset' to 0 if they are not.

This is an important point to remember about the zero flag: it does not necessarily indicate that the contents of a register or memory address equal zero; it indicates whether or not the *result of the previous operation* equals zero. In this case the previous operation was a comparison (in effect, a subtraction) and, since the result of the comparison was zero, the zero flag is automatically *set*.

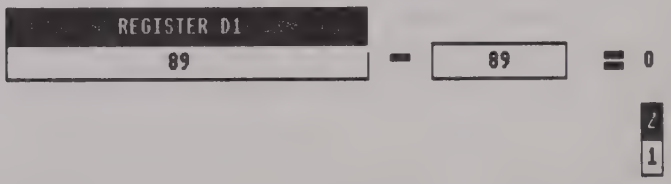


FIGURE 3-3.

You can see that any decision we may wish to take now, such as a call to a subroutine, can be made on the basis of testing the zero flag in the CCR register to see whether it contains 0 or 1.

Let us now look at how we might make such a test. Suppose that our program has asked us a question to which the input answer is either `Y` (yes) or `N` (no). If our answer is `Y` then perhaps the program might transfer execution to a routine labelled, say, '`FRED`'. If the answer is `N`, or in fact anything other than `Y`, execution should move on to the next instruction in the program sequence.

In assembly language terms the operation could be described as follows: If the ASCII code of the last key pressed is the same as the ASCII code for `Y` then set the "Z" flag. If the Z flag contains 1 then transfer execution to address '`FRED`', otherwise move to the next instruction in the sequence. What would this look like as part of a program?

```
;-----------------------------------------------------
;Instruction     Comment
;-----------------------------------------------------
 . . . . . .    ;ASCII code for last key pressed is in D1
CMPI.B #89.D1   ;Is it the same as ASCII code for "Y"?
BEQ FRED        ;'Branch if equal' to address 'FRED' if so.
                ;Otherwise continue to next instruction
;-----------------------------------------------------
```

That is all there is to it. The BEQ (branch if equal) instruction 'tests' the Z flag and transfers execution to the address labelled 'FRED' if it is set.

If we wished to restrict our answer to Y or N rather than Y or 'any other' key, we would simply perform two comparisons:

```
;-----------------------------------------------------------
;Instruction     Comment
;-----------------------------------------------------------
 . . . . . .    ;ASCII code for last key pressed is in D1
CMPI.B #89.D1   ;Is it the same as ASCII code for "Y"?
BEQ FRED        ;'Branch if equal' to address 'FRED' if so.
CMPI.B #78,D1   ;Or is it the same as ASCII code for "N" ?
BEQ MARY        ;Branch if equal to address 'MARY' if so.
Otherwise carry on to next instruction
;-----------------------------------------------------------
```

A separate comparison could, of course, be performed with every ASCII code in order to find out exactly which key had been pressed. This would be rather cumbersome however, and there are other techniques, on specific machines, for reading and cross-referencing large areas of the keyboard at once in order to isolate a particular key or set of keys.

Key testing is, of course, only one application but in all cases the principle is the same – the Z flag is used to indicate whether the result of a comparison or any other arithmetic or logical operation is zero. It also works with instructions such as MOVE, indicating whether a number moved into a certain location is zero or not.

Sign Flag (N)

Now let us look at the *sign flag*: N. This is used to indicate whether an item of data is a positive or a negative number.

Although in most circumstances a byte of data represents a value between 0 and 255, in some circumstances it is useful to regard it as representing a *signed* value in the range –128 to +127. This is very easily achieved by regarding bit 7 of a byte (the Most Significant Bit

or MSB) as being the *sign bit*. If it is set then the number is negative and if it is reset then the number is positive.

However, if we were to use bit 7 as the sign bit we would run into problems when performing arithmetic operations using signed values. For example, if we add together the signed representations of +10 and –5 (00001010 + 10000101), we would end up with 10001111 which is –15 as a signed number, and which is obviously incorrect. Binary addition is performed in a similar way to decimal addition, with any 'carry' being passed to the next column of the addition. The rules are that 0+0=0, 0+1=1, 1+0=1 and 1+1=0 carry 1, hence:

Binary *2's Comp*

```
 00001010    (+10)
+10000101    (- 5)   (bit 7 is set,
---------             indicating a minus value)
=10001111    (-15)
```

For the processor to have to correct this result would take up valuable processing time and so the solution is to represent signed numbers in *2's complement* form, in which all positive signed numbers keep their normal binary value and take the sign indicated by their eighth bit (bit 7), and *all negative numbers* are obtained by means of the following method:

1 All bits in the binary value are complemented (i.e. inverted)

2 The value 1 is added to the result

For example, number 5 is represented by 00000101. When we invert this, substituting zeroes for all the ones and vice versa, we get 11111010, which is the *one's complement*. We then add 1 to this number and end up with 11111011, which is the *2's complement* signed representation of –5.

If we now substitute this representation of –5 we get the correct answer to our addition:

Binary *2's Comp*

```
 00001010    (+10)
+11111011    (- 5)
---------
=00000101    (+ 5)
```

Two's complement arithmetic is recognized implicitly by a number of assembly language instructions and in most circumstances the conversion to 2's complement is performed automatically, without the programmer needing to specify this form of representation.

Now consider the following byte:

```
01111111
```

As a signed number this is, of course, +127: the highest positive number which can be represented by a signed byte. The value 11111111, in 2's complement, represents –1, which is the same number as +127 but with the sign bit set. Again, this is +1 (00000001) inverted to 11111110, with 1 added, which is 11111111.

Two's complement byte values therefore run as follows:

Binary		*2's Comp.*	*Unsigned*
00000001	=	+1	1
00000010	=	+2	2
00000011	=	+3	3

and so on up to:

Binary		*2's Comp.*	*Unsigned*
01111110	=	+126	126
01111111	=	+127	127

and then the next binary numbers in sequence are:

Binary		*2's Comp.*	*Unsigned*
10000000	=	-128	128
10000001	=	-127	129
10000010	=	-126	130

and so on up to:

Binary		*2's Comp.*	*Unsigned*
11111101	=	-3	253
11111110	=	-2	254
11111111	=	-1	255

The same principle applies to two-byte data representation. The value:

```
0111111111111111
```

represents +32767, its MSB being reset and which is the highest 2's complement positive word value. The value:

```
1000000000000000
```

represents –32768, its sign bit being set, which is the lowest 2's complement negative word value. 16 ones would, of course, represent –1 in a similar way to the byte form.

Suppose that we have performed an operation using byte-sized operands and we wish to know whether the resulting value is positive or negative in terms of 2's complement representation?

We could, of course, subtract 127 from it and see what the remainder is. It would be far simpler however, if we could just check a flag, and this is precisely how the N flag functions. After certain operations the eighth bit (bit 7) of the resulting byte, or the 16th bit (bit 15) of a word, is automatically copied into the N flag position in the CCR register and we can test this in much the same way that we tested the zero flag.

This would be useful, for example, if we were processing a block of data and wished to separate all those bytes of data which represented standard ASCII characters from those which do not.

It would be easy to separate standard from non-standard ASCII codes because, in 2's complement terms, the standard set is represented by codes +0 to +127 and the non-standard set, by –1 to –128. In other words, the distinction is revealed by the status of the sign bit. Therefore, we can load each of our bytes in sequence into a data register and test to see whether the N bit in the CCR register has been set or not, and then write program instructions to take appropriate action – for example to branch to one subroutine if the N bit is set or to another if it is reset.

Carry Flag (C)

Because the size of a byte, a word or a long word is limited to a certain number of bits, it frequently happens that an arithmetic operation produces a result which falls outside the range of the data size being used for the operation. Consider the following binary addition:

Binary	*Decimal*
00001010	10
+00000100	4

=00001110	=14

In this case there is no problem. The result is perfectly correct because it falls within the range 0 to 255. Compare this with the following addition:

Binary	*Decimal*
10001010	138
+10000100	+132
---------	---
=00001110	= 14
carry 1	

The problem here is that the addition has given the result 14, which is incorrect because the true sum of the two values is greater than 255, which is the highest value that a binary byte can represent. The binary addition has therefore resulted in a 'carry' from bit 7 which has nowhere to go. If this were a 16-bit addition it would go into bit 8 (the 9th bit) of the result, giving the correct decimal result of 270. In cases where the size limit of an operand has been exceeded in an operation, the carry bit is passed into the 'C' or *carry flag* in the CCR register, which becomes set, indicating that the operation has resulted in a value which is out of range. In our first example, which was within range, the carry flag would remain reset.

The same principle applies to subtraction operations. If we were to subtract 10 from 4 then the true result is negative. Since the lowest unsigned binary byte value is 0 then the actual result will be incorrect and a binary 'borrow' will be generated, which is copied into the C flag in the same way as a carry.

The C bit of the CCR register can therefore be regarded as the ninth bit (bit 8) of a byte value, the 17th bit (bit 16) of a word value or the 32nd bit (bit 31) of a double word value.

Wherever you perform an operation in which a carry or borrow is likely to occur, you can use an instruction which tests the C flag in order to determine whether or not the result is incorrect. Again, your program should specify the appropriate action to be taken.

Overflow Flag (V)

The 'V', or *overflow* flag is very similar to the carry flag except that it is used to detect binary overflow errors resulting from 2's complement arithmetic operations. Consider the following two addition operations:

Binary		*2's Comp.*
10001010		-118
+10001001		-119
---------		----
=00010011	=	+ 19

Binary		*2's Comp.*	*Unsigned*
01111000		+120	
+00111000		+ 56	
---------		----	
=10110000	=	- 80	
carry 1			

In both these cases, the signs of the numbers have been altered. In the first example, the change of sign was caused by the fact that, although there was no carry from bit 6 into bit 7, there was a carry from bit 7 into the carry flag, leaving a zero in bit 7. In the second example, there was no external carry into the carry flag but there was an internal carry from bit 6 into bit 7 of the byte. In both cases therefore, the value of the sign bit was altered causing an incorrect result. *Where an operation involving 2's complement values results in the alteration of the sign flag* the condition is termed an *overflow* and the V bit of the

CCR register is set, otherwise it remains reset. An overflow is usually an accidental error and a program can be designed to test for such an error and to redirect execution to a corrective subroutine.

Extend Flag (X)

The 'X' or *extend* flag performs the same kind of function as the carry flag, except that it is used in binary coded decimal and multiple precision arithmetic operations. This flag will be described in detail when we look at binary coded decimal arithmetic in Chapter 7.

Conditional Suffixes

All these flags, both singly and in combination, provide an extremely flexible means of making conditional branching decisions. The branching instructions of the 68000 can incorporate suffixes which use various combinations of these flags to provide every kind of conditional testing operation necessary for programming. These suffixes are appended to three types of instruction: `Scc` (set from condition), Bcc (branch on condition) and `DBcc` (decrement and branch on condition). An example of the '`EQ`' suffix was used earlier in this chapter in the form `BEQ` (branch if equal).

The table on the next page shows a complete list of the conditional suffixes:

Suffix	*Meaning*	*Conditions*
CC	if carry clear	if C = 0
CS	if carry set	if C = 1
EQ	if equal	if Z = 1
GE	if greater or equal	if either (N = 1 and V = 1) or (N = 0 and V = 0)
GT	if greater	if either (N = 1 and V=1 and Z = 0) or (N = 0 and V = 0 and Z = 0)
HI	if high	if C = 0 and Z = 0
LE	if less or equal	if N = 1 and V = 0) or (N = 0 and V = 1) or Z = 1
LS	if low or same	if C = 1 or Z = 1
LT	if less than	if either (N = 1 and V = 0) or (N = 0 and V = 1)
MI	if minus	if N = 1
NE	if not equal	if Z = 0
PL	if plus	if N = 0
VS	if overflow	if V = 1
VC	if not overflow	if V = 0

DBcc and Scc can be used with the additional suffixes T (true) and F (false).

The following examples show how the carry and overflow flags are affected by addition operations on various byte values:

Binary *Decimal*

```
 00001010     10
+00001010     10
---------     --
 00010100     20    V=0 and C=0
```

Binary *2's Comp.*

```
 01111000    +120
+00001010    + 10
---------    ----
=10000010    -126    V=1 and C=0 (overflow error)
```

Binary *2's Comp.*

```
 00001010   +10
+11111000   - 8
---------   ---
=00000010   + 2    V=0 and C=1 (correct 2's Comp.
carry 1                result.  Carry ignored)
```

Binary *2's Comp.*

```
 00001000   + 8
+11110110   -10
---------   ---
=11111110   - 2    V=0 and C=0
```

Binary *2's Comp.*

```
 11111000   - 8
+11110110   -10
---------   ---
=11101110   -18    V=0 and C=1 (Correct 2's comp.
carry 1                result.  Carry ignored)
```

Binary *2's Comp.*

```
 10001000  -120
+11110110  - 10
---------  ----
=01111110  +126    V=1 and C=1 (overflow error)
carry 1
```

Bit Rotation

Arithmetic operations are not the only ones which can alter the status of the flags. The rotation instructions for example, allow you to rotate the individual bits in a byte, word or double word of data to the right or to the left, altering the status of the flags accordingly. Why would you want to rotate bits to the left or right? The answer is not immediately obvious perhaps, because in normal, everyday programming we are used to thinking in terms of the 'face value' of numbers,

rather than thinking of them as a pattern of 0s and 1s. This is why we are using binary arithmetic a great deal in this part of the book: it is only by getting used to the idea of data as a binary number pattern that we can really appreciate how assembly language works and how an understanding of its structure can allow you to manipulate these patterns in creative and imaginative ways.

Imagine that you have a program which is feeding data out of the computer to some peripheral device – say a disc drive. You have 7 bytes of data to transfer and you need some method of counting off these bytes as they are transferred. You could, of course, use a register as a counter variable, subtracting one from it after each transfer and checking the zero flag until it is set, indicating that the counter has reached zero. As an alternative you could load a labelled address with the byte value 252 (binary 11111100). This contains a binary pattern of 6 ones (i.e. one less than the number you wish to count), followed by two zeros. Now see what happens when we 'rotate' the binary number to the left by 1 bit:

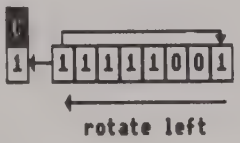


FIGURE 3-4. Bit rotation of a binary byte.

As the bits are rotated to the left, the 1 from the leftmost bit position replaces the zero which previously occupied the rightmost bit position. At the same time the leftmost 1 is also copied into the C flag. This is in fact a carry and effectively sets the carry flag.

We could rotate this byte to the left a total of seven times, each time following the transfer of a byte of data from the computer into the disc drive, thus using the 6 set bits in the rotating counter byte as counters for the amount of data transferred:

Carry	*Operand value*	*Flag*
0	11111100	original number
1	11111001	after first rotation
1	11110011	after second rotation
1	11100111	after third rotation
1	11001111	after fourth rotation
1	10011111	after fifth rotation
1	00111111	after sixth rotation
0	01111110	after seventh rotation

On the seventh rotation, the zero in the leftmost bit position is rotated into the carry flag which therefore becomes reset.

If each rotation is followed by an instruction which tests the carry flag, the condition of the flag can therefore be used to determine when the count has reached 7. This technique is illustrated in `PROG6` in Chapter 13.

There are a number of instructions for rotating and for 'shifting' data values and all affect some of the flags in various ways. Details of this group of instructions can be found in Appendix B and include `ROL` (rotate left with carry), `ROR` (rotate right with carry), `ROXL` (rotate left with extend), `ROXR` (rotate right with extend), `ASL` (shift arithmetic left), `ASR` (shift arithmetic right), `LSL` (shift logical left) and `LSR` (shift logical right).

Logical Operations

The final main group of instructions which affect the flags are the logical operators: `AND`, `OR` and `EOR`. `AND` takes a source and a destination operand and returns a result in which each bit is set if both the corresponding bits in the source *and* the destination are set. `OR` returns a result in which each bit is set if *either* of the corresponding bits in the source *or* the destination are set. `EOR` returns a result in which each bit is set if *either* of the corresponding bits in the source *or* the destination *but not both* are set. The following examples show the way in which the flags may be affected accordingly:

AND	OR	XOR
10101010	10101010	10101010
01101101	01101101	01101101
--------	--------	--------
00101000	11101111	11000111
V N Z X C	V N Z X C	V N Z X C
0 0 0 0 0	0 1 0 0 0	0 1 0 0 0

Note that the 'X' flag is not affected by these instructions and the 'C' and 'V' flags are always zeroed.

Specific Flag-altering Instructions

Most of the instructions on the 68000 affect various flags in one way or another as a program is executed and you sometimes need to be sure that a particular flag is in the required state before you use it.

One way of doing this is by performing a logical AND operation on the CCR register in order to reset all or some of the flags. The instruction for this is `ANDI #x,CCR` (`AND immediate to CCR`). The 'x' is a value whose bit pattern corresponds to the flag bits, X,N,Z,V and C of the CCR. Thus, for example, to reset all the flags the 'x' would be 0. To reset the carry and zero flags only, those two bits should be equal to zero and so 'x' would be 26 (i.e. binary 11010). The operation for setting flags is similar except that a logical OR operation is used: `ORI #x,CCR` (`OR immediate to CCR`) and instead of using reset bits to reset particular flags, we use set bits to set them. To set Z and C therefore, 'x' would be given the value 5 (i.e. binary 00101)

Flag Testing

It is possible to test data without previously having performed an operation on it. Suppose that you have an item of data in memory and you wish to know whether it is zero or negative. Using the `TST` (Test) instruction you can specify an operand and the 'Z' and 'N' flags will be set or reset according to its value. Since the value is not actually altered in any way, the 'V', 'C' and 'X' flags will not be altered either.

There are a number of other test instructions which specifically operate on individual bits within an operand and which affect the zero flag. These instructions are summarized as follows and a fuller description of their functions can be found in Appendix B. Note that these operations are normally used in multiprocessor operations where several processors have access to a shared memory resource. Testing an operand allows a program to determine whether another processor is accessing a particular part of memory and setting an operand bit informs other processors that your own processor is accessing it.

`BCHG`
(Test Bit and Change) This instruction tests a specified bit within an operand in memory or in a register. If the bit is reset then the zero flag is set and if the bit is set then the zero flag is reset. The specified bit is then complemented – in other words, if reset it is set or if set it is reset.

`BCLR`
(Test Bit and Clear) This is similar to `BCHG` except that instead of being complemented, the specified bit is left reset.

`BSET`
(Test Bit and Set) This is similar to `BCLR` except that the specified bit is always left set.

`BTST`
(Test Bit) This is similar to the above three instructions except that the specified bit is left unchanged.

`TAS`
(Test and Set) `TAS` tests a specified operand and the N and Z flags are set or reset according to its value, after which the high order bit of the operand is set. No other processor can access the operand while the `TAS` operation is taking place.

Chapter 4

Branching Operations

Relative Addressing

In a BASIC program it is frequently necessary to redirect execution from the current line number to another line number; either to `GOTO` another section of the program or to `GOSUB` to a subroutine.

In assembly language the same applies, except that in this case we would be redirecting execution not to another line number but to a memory address.

In BASIC, such an operation would normally involve a direct destination. In other words, we would specify something like `GOTO 300` or `GOSUB 800`. In assembly language we would do something similar, specifying that program execution should go to the instruction located at, say, address 70000 within the code segment for example, or to a subroutine starting at address 80000. The actual address value need not necessarily be known beforehand: it could be contained in one of the address registers, in which case we could specify it in the instruction using the register indirect addressing mode.

There will be occasions when we wish to redirect execution to an address which is *relative to* the instruction which is specifying the redirection. In BASIC this is an unnecessary operation but we would express such a command using an instruction such as:

```
100 GOTO 100+30
```

or

```
100 GOSUB 100+30.
```

In these cases we would be referring to line 130, which is 30 lines further on in the program relative to the `GOTO` or `GOSUB` commands.

The reason why we do not need to use relative addressing in BASIC is that the destination of a branch is explicit in the program. If we know that the target routine is at line 130 then we can refer to the line number directly. In assembly language there are no line numbers and therefore a target routine must be relative to some fixed point. If the target routine is positioned relative to the instruction which calls it then it is necessary to calculate the relative number of addresses between the branching command and the routine or subroutine address to which it refers.

If the fixed point is the address of the current instruction then the relative displacement between PC and the target routine must be calculated, since at any given time, PC contains the address of the instruction which is currently being executed. If PC is altered to point to a target routine relative to the branching instruction, the relative displacement is *added* to the current value of the PC register. If the address of the target routine is absolute rather than relative, then the absolute address must *replace* the value currently contained in PC.

In assembly language it is necessary to specify in the instruction itself the number of relative addresses between a branching command and the program or subroutine address to which it refers, or to specify or 'point' to the actual address of an absolute location. These two types of redirection instructions are distinguished in the 68000 by the use of two different types of instructions: *jump* and *branch* commands:

`JMP` (Jump) – jump to a specified address
`JSR` (Jump to Subroutine) – jump to a subroutine at a specified address
`BRA` (Branch) – branch to an address relative to the branch command
`BSR` (Branch to Subroutine) – branch to a subroutine address relative to the branch command.

Jump Operations

Taking the first of these, `JMP` is perfectly straightforward. We can jump to an address which is either absolute (i.e. the address number is specified in the instruction) or indirect (i.e. the address number can be specified using one of the forms of the address register indirect addressing mode). If you are unsure of the meanings of these modes you should refer back briefly to chapter two to refresh your memory.

The JSR command operates in almost exactly the same way except that it redirects execution to a subroutine located at a particular address. When the command is executed, the address immediately following the JSR command is temporarily stored away so that when a return is made from the subroutine, the program can recommence execution from the point where it left off.

Branch Operations

The Branch commands are a little more complex. In these cases we are transferring execution to an address a relative distance away from the branching command and if we were to look at the code of the command in program memory it would appear something like this:

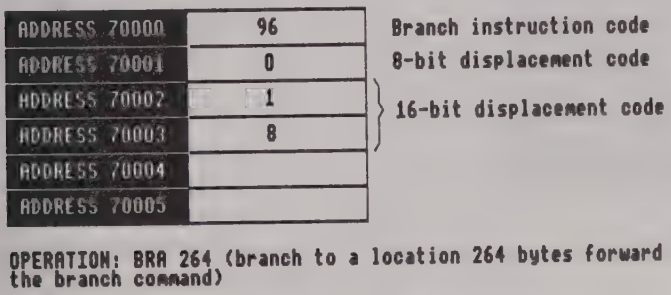


FIGURE 4-1.

The first byte of the command, in address 70000, contains the branching instruction itself. The following three addresses contain the data which specifies the distance of the displacement (i.e. the relative distance) between the branch command and the address to which execution will be transferred. This can either take the form of a single byte of data (in address 70001), giving a two's complement displacement in the range -128 to +127, or a word of data (in addresses 70002 and 70003) giving a two's complement displacement in the range –32K to +32K. If the byte in address 70001 is other than zero then a signed byte displacement will be used and if it is zero, then a signed word-sized displacement will be used.

Since the address of the program instruction currently being executed is always held in the PC register, any relative displacements will therefore be *relative to the value of the PC register*. However, since the PC register increments by 1 after every byte of an instruction has been interpreted, by the time it has interpreted the required displacement distance it will have moved on to an address two bytes further on from the beginning of the branch instruction. The displacement value, therefore, does not refer to the relative distance from the beginning of the branch instruction itself but from a point two bytes further on, as illustrated by the following diagram:

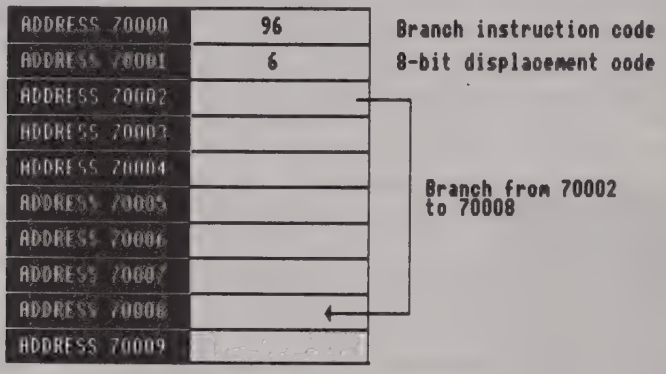


FIGURE 4-2.

This diagram illustrates the use of the instruction `BRA 6`, which means 'branch relative to the `BRA` instruction by 6 bytes'. If we wanted to branch backwards instead of forwards in memory we might use an instruction such as `BRA -6`.

Labelled Branching Operations

It will no doubt have occurred to you that if we had to calculate these relative values individually each time, then programming would be something of a nightmare. It would be easier if we could simply program an instruction such as `BRA ADDR1`, where `ADDR1` is a label or variable equivalent to the new execution address. This is precisely what an assembler program allows us to do. We can 'declare' the value of a label name of our own choosing to be the addres of a

particular program routine. Thereafter, when the program is compiled into object code, the relative distance between the BRA ADDR1 command and the actual destination address to which it refers would be calculated and coded automatically, as would the relative distances for any other instruction which refers to ADDR1. Apart from making the job of programming easier, this also means that a program can be loaded anywhere in memory since the label does not refer to a fixed address.

The same principle applies to BSR instructions. The command BSR ADDR2 would redirect execution to a subroutine whose start address is defined as ADDR2 in the source listing. Labels may also be used with JMP and JSR instructions.

Wherever a JSR or BSR instruction has redirected execution to a subroutine, a return is made by the inclusion in the subroutine itself of a return command, which serves the same function as the RETURN command in BASIC. In assembly language, the command is RTS (return from subroutine). There are some other return commands which are used in special circumstances and these will be described in later chapters.

As a summary of the principles described above, the following diagram shows how a BSR command redirects execution to a subroutine located at address 70000, and, after a return is made, how a JMP command is used to jump to address 80000.

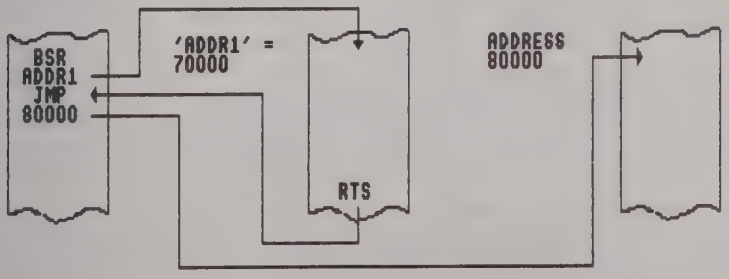


FIGURE 4-3.

Absolute and Indirect Branching

JMP and JSR jumps are either *absolute* or *indirect* and BRA and BSR branches involve an *absolute displacement*. An absolute branch is one which is made to a target location whose address is specified in the instruction. An indirect branch is one whose target address is contained not in the instruction but in a location pointed to by the instruction.

Absolute

JMP ROUT3 as an *absolute* instruction redirects execution to an address represented by the label ROUT3. This address is automatically loaded into the PC register and becomes the new execution address. In the case of JSR the target address is obtained in the same way.

Indirect

JMP (A4) is an *indirect* instruction which redirects execution to a target location whose address is contained in register A4. This address is automatically loaded into the PC register and becomes the new execution address. In the case of JSR the target address is obtained in the same way.

Relative

BRA ROUT3 is a PC relative branch because the value of the label is an absolute displacement which is added to the PC register. The same principle applies to BSR.

Note that it is easy to become confused about the notion of absolute addressing. Technically, an absolute address is a specified address number. When a label is used then the absolute address is represented by the label and the absolute address to which the label refers may differ, depending on what part of memory the program has been loaded into. In position dependent programs a label may be defined by the program as being equal to a particular address number. In position independent programs the label is assigned to the position of a particular instruction, whatever address that happens to be when the program is loaded. The assembler computer it as a PC relative location for most practical purposes you should simply regard the term absolute as referring to an operand which is not in a register and which is not referred to indirectly. Thus JMP ROUT3, JMP (A4) and

BRA ROUT3 may all refer to the same address but the first implies an absolute reference, the second an indirect reference and the third an absolute displacement.

The only difficulty which you are likely to run into in this respect is where a label is used which represents an address which has been computed as a program counter relative value, which is the case with programs which have been assembled as position independent code. *In this case an operand in a labelled address cannot be altered.*

Conditional Branching

The JMP, JSR, BRA BRS and RTS commands are termed unconditional redirection or 'program control' commands, corresponding to the BASIC instructions GOSUB, GOTO and RETURN. In most circumstances where we require redirection however, we need to use *conditional* instructions, corresponding to BASIC instructions such as IF A=B THEN GOTO .. or IF A<=B THEN GOSUB ... It is in these circumstances that we use the condition code flags in the CCR register, as described in the previous chapter. If you refer to the table on page 51 in the last chapter, you will see that certain condition code flags are affected by program operations which, in various combinations, correspond to conditions such as less than, equal to, greater than and so on. In branching operations we use these in a special form of the branch command: Bcc (Branch according to condition code), where 'cc' is one of the suffixes listed in the table. This was illustrated in our discussion of the zero flag, where the instruction BEQ (i.e. Bcc with 'EQ' substituted for the 'cc') was used to branch to a new execution address conditional upon the zero flag being set. We are now in a position to go a stage further and see how a conditional branching command and the zero flag can be used to simulate a BASIC FOR..NEXT loop.

Imagine that we have a routine which we wish to enclose within a program loop, which is to be iterated 10 times. We need to have some means of counting up to, or down from, 10 so that we know when the iterations have been completed. We can do this by selecting one of our data registers, say D4, as a loop control 'counter' and load it with the immediate value 10:

```
MOVEQ #10,D4
```

The low order byte of D4 now contains 10 and the higher three bytes are zeroed.

The routine which is to be iterated starts at address 80000 and ends at address 80007. Following this, starting at address 80008, we need to write a routine which tests to see whether the iteration has been completed. If it has, we want the program execution to carry on with the next instruction in sequence. If not, execution must loop back to address 80000 to repeat the routine. Here is the process in diagrammatic form:

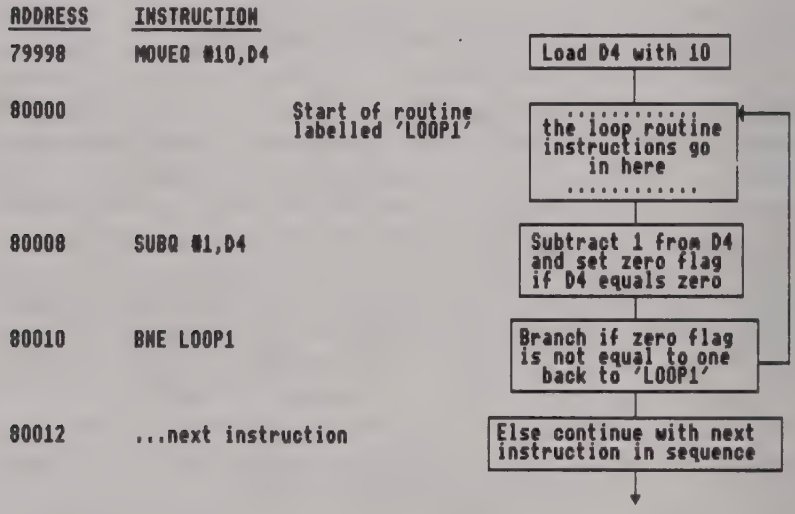


FIGURE 4-4.

Immediately before the routine, D4 is loaded with 10; a procedure which uses up 2 bytes from addresses 79,998 to 79,999. Then follows the loop itself, from addresses 80000 to 80007. Following this is a subtraction instruction: `SUBQ.B #1,D4` which means 'subtract quick' the value 1 from the least significant byte of D4'. In addition to subtracting 1 from D4, this operation will also set the zero flag if D4 now contains the value zero, or leave it reset otherwise. We are then in a position to test the zero flag and branch back to address 80000 if it is reset, which is achieved with the command `BNE LOOP1` (branch if 'not equal' back to `LOOP1`). The term 'not equal' refers to the status of the *zero flag*; if it is reset then the result of the previous `SUBQ` instruction must be 'not equal' to zero and execution is therefore redirected back to `LOOP1` (address 80000). If D4 has reached zero

after the SUBQ instruction then the zero flag is set. The branch is therefore not made and execution continues with the next program instruction.

The above example is, in effect, a FOR..NEXT loop and it is almost as simple to understand and as easy to program in assembly language as it is in BASIC. This kind of loop is so fundamental to programming that the 68000 has been provided with a mechanism for speeding up and simplifying the operation even further: DBcc (decrement and branch according to condition code). Again, DBcc instructions can take the form of branching instructions conditional upon the status of a range of flags.

This time, however, the process is a little more subtle. Suppose we have a loop which adds 1 to the contents of data register D2 up to 10 times until either register D2 is equal to register D3 or the count of ten has finished, whichever happens first. We could perform this operation by continually adding 1 to D2 and then using the comparison instruction, CMP, until the zero flag becomes set, which would indicate equality, or until our counter register has finished counting.

If we enclosed this routine within a DBcc loop we would use the form DBEQ (Decrement and branch until equal), as follows:

```
;-----------------------------------------------------------
;Label   Instruction       Comment
;-----------------------------------------------------------

        MOVEQ #9,D1       ;Load counter register with 9
LP1     ADDQ #1,D2        ;Add 1 to register D2

        CMP.L D2,D3       ;Compare D2 with D3

        DBEQ D1,LP1       ;If zero flag is set, go to NEXT
                          ;Otherwise decrement D1
                          ;If D1<>-1, loop back to LP1
                          ;Otherwise goto NEXT
NEXT ...
```

Firstly, we load our counter register with the value 9 (i.e. 1 less than the value of the required count). Then we add 1 to D2, using the ADDQ (add quick) instruction and then compare the contents of D2

with the contents of D3, using CMP. Next comes the DBEQ instruction. Firstly, it tests the zero flag to see if the previous comparison instruction resulted in equality. If D2 *is equal* to D3 (i.e. zero flag has been set) the loop is finished and execution passes to the next instruction. If D2 *is not equal* to D3, the D1 (counter) register is automatically decremented by 1. If D1 is not equal to –1, execution loops back to the instruction labelled 'LP1' (the ADDQ instruction). If D1 is equal to –1 the loop is finished and execution passes to the next instruction.

In BASIC the process could be coded as follows:

```
100 D1=9
200  REPEAT
300     D2=D2+1
400     IF D2=D3 THEN GOTO 700
500     D1=D1-1
600  UNTIL D1=-1
700 Next instruction ...
```

Using other condition suffixes, other types of loops can easily be constructed. For example DBMI can be used for a REPEAT/UNTIL MINUS loop and DBT corresponds to REPEAT/UNTIL TRUE. The DBF (Decrement and branch until false) form of this instruction, which on some assembler programs is expressed as DBRA (Decrement and branch) is equivalent to having no condition: an unconditional branch. In other words the loop will terminate only when the counter register has reached –1.

Subroutines

When a subroutine is called the following events take place:

1 The address of the instruction which follows the JSR or BSR instruction is automatically stored in a special reserved area of memory termed the 'stack.'

2 The new execution address specified by the branching instruction is loaded into the program counter (PC register) (or added to it in the case of BSR) and this therefore becomes the new execution address.

3 On completion of the subroutine, normally signalled by an RTS (Return from subroutine) instruction in the subroutine, the temporary address which was stored on the stack is transferred back into the PC register so that the program will recommence execution starting with the instruction which followed the branching instruction.

Where subroutines are nested, i.e. when a subroutine calls itself or calls another subroutine, the same sequence of events takes place, except that the RTS in every subsequent subroutine returns execution to the instruction following the branching instruction in the previous subroutine. The execution flow of a nested subroutine sequence is shown in the following diagram:

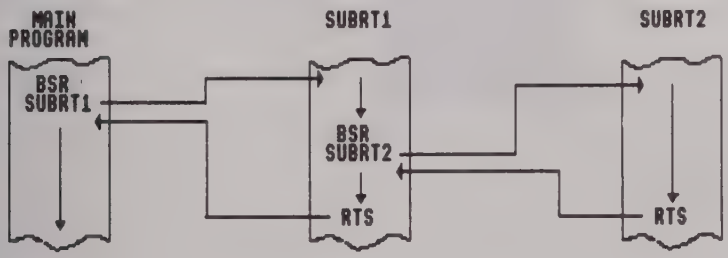


FIGURE 4-5.

Passing Parameters to Subroutines

It is frequently necessary to pass parameters to a subroutine; in other words, to transmit specific values to a subroutine which you are calling which it needs for performing calculations or for some other purpose.

For example, because of the complex structures of many computer systems, it is often difficult to write program sequences which directly perform tasks such as printing strings, numbers or graphic figures to the screen. On most computers it is possible to overcome these problems by using the same subroutines which the operating system itself uses, which are permanently located in memory. *The exact methods of calling these subroutines will vary between machines* but

the principle is always essentially the same: your program performs a branch to one of the system subroutines and passes *parameters*, which would be coded instructions which specify the subroutine required plus any data which the subroutine needs for its execution. The following diagram illustrates a typical example of this, showing how a user program passes parameters to the system specifying that a character is to be printed to the display screen.

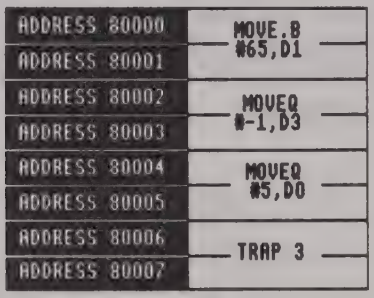


FIGURE 4-6.

In this case the ASCII code of the character to be printed is contained in register D1, the time allowed for the operation (e.g. delayed, infinite or specified) in register D3 a code (5) indicating that a single character is to be printed and in register D0. These values are the parameters for the operation and the subroutine will expect to find suitable values in these particular registers.

A mechanism such as this will be given different names by various manufacturers, such as a 'function despatch', 'operating system call' or 'trap' mechanism. You should refer to the specific documentation for your machine to establish the method of operating these functions, and the parameters and registers required. You will normally find that all important system control, device control, graphics and arithmetic operations have system calls which can be called by the user and they will save a great deal of programming time.

In the above example we considered the passing of subroutine parameters by means of registers, which is normally the case with simple subroutines or with system call mechanisms. However, when passing parameters to your own subroutines you may find that there are not enough free registers available. In this case, you can opt to

have your parameters stored elsewhere in a labelled data table and call them 'by name' when they are required.

The method used for passing parameters depends entirely on your requirements. You may wish to pass a 'one-off' series of parameters to a subroutine, in which case it makes sense to pass them via registers, or you may wish to pass a frequently used block of parameters, in which case it makes sense to keep them in a separate 'parameter block' and call them 'by name' whenever they are required.

If you pass parameters by name rather than via registers, it is up to you to include instructions within the subroutine which collect the parameters from their original locations before they are actually used in the subroutine.

With both methods, you will frequently need to pass parameters back from the subroutine to your main program. Again, these may be passed via registers or they may be placed in particular labelled areas of memory from where they can be retrieved by name after the subroutine has been completed. We shall be looking more closely at these methods in Chapter 11, where some program examples are given.

Chapter 5

The Stack

Every time you temporarily need to store some information, in memory or in registers, it can be a time consuming and complex business. You may, for example, have some data stored in several registers which you want to keep, and at the same time you want to use those registers for some other purpose. Perhaps you may wish to call a subroutine which uses these registers and therefore their current contents would be lost. What do you do to preserve this data? There may be no spare registers available and you have to think about finding some free space in memory where you can store the data away for later retrieval. It would be useful if there were a general purpose storage area where you could dump your data so that you could pick it up again later without too much difficulty.

Fortunately, all microprocessor systems have such a storage area, which is called a *stack*. You can picture the stack as a vertical series of memory locations, arranged in pairs. Its precise location in memory need not necessarily be known: all you need to remember is that the current address of the first available free space on the stack, the 'top' of the stack, is always held in a special register which is termed the *stack pointer* (SP) register.

The SP register is in fact address register A7, which plays a dual role. When the 68000 is operating in user mode, A7 is the stack pointer for a stack called the *user stack* and in supervisor mode, it points to the *supervisor stack*. In practice, the system organizes the stack status in each mode and you only need to remember that A7 points to the top of the stack.

There is no reason why you should not set up separate stacks for different purposes within your own user programs and that is simply done by moving the address of the top of your alternative stack into a free address register and using that register as a stack pointer in the same way that you would use A7.

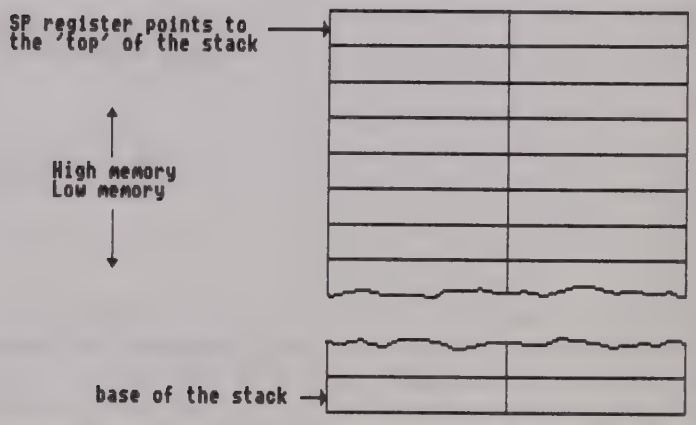


FIGURE 5-1.

A stack normally extends *downwards* in memory from the stack top. Every time some data is stored or 'pushed' on the stack, by means of a `MOVE` instruction (which simply means that data is into the free memory addresses at the current 'top' of the stack), then the SP register is decremented by two or four bytes (depending on whether you are stacking word or long word data) and the data is copied into the address pointed to by SP and SP–1 or SP to SP–3. For example, to stack the contents of D4 you would use the following instruction, which uses the 'address register indirect with predecrement' addressing mode: `MOVE.L D4,-(A7)`. When the data is subsequently removed from the stack, again by means of a `MOVE` instruction, the word or long-word data pointed to by SP is copied into the specified destination and then the SP register is automatically incremented by two or four. For example, to unstack or 'pop' the long word on top of the stack into D5 you would use the instruction `MOVE.L (A7)+,D5`. Note that the unstacking operation uses the 'address register indirect with postincrement' addressing mode.

You will notice that data which has been stored on the stack is removed in the opposite order. In other words, the last data to be stored on the stack is the first to be retrieved. This is known as a LIFO (last in, first out) arrangement and you should always take care that any information which you retrieve from the stack is popped in the correct order. In the following example we shall see how the stack might be used in a typical situation.

Suppose that you have a program which contains data in all the data registers, all of which you want to keep. Your program branches off into a subroutine in which you need to use two of these registers, say D1 and D2.

Your first action is to push onto the stack all four words contained in D1 and D2, say, 300, 287, 524 and 1176. The SP register originally pointed to address 70000 in the stack. After pushing, using the instructions `MOVE.L D1,-(A7)` and `MOVE.L D2,-(A7)`, the contents of D1 and D2 have been copied to addresses 69992 to 69998 and SP now points to address 69992.

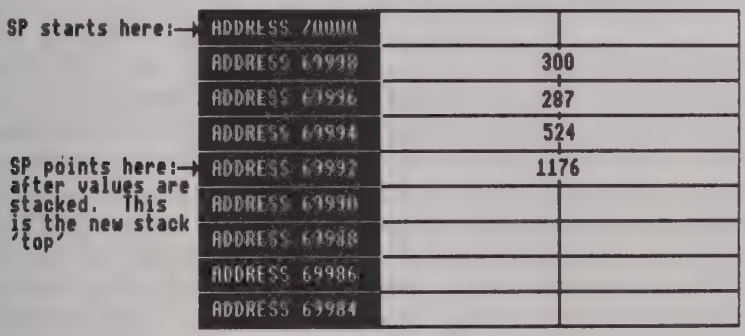


FIGURE 5-2.

Next you branch to the subroutine, which uses registers D1 and D2 for some purpose. After this, you return from the subroutine back to the main program and you are now ready to retrieve your four original values, which you can 'pop' back into D1 and D2 *in reverse order* using the instructions `MOVE.L (A7)+,D2` and `MOVE.L (A7)+,D1`. The SP register now points to address 70000, as it did originally.

During this whole operation the contents of the stack have grown downwards in memory by 8 bytes and then shrunk back again.

Note that although the values are popped off the stack in the reverse order to that in which they were pushed, they *need not necessarily be popped back into the same registers from which they were pushed.*

FIGURE 5-3.

An important point has been omitted from this description which is worth noting because it is essential to your understanding of how the system operates. Every time you call a subroutine, the program has to have some means of knowing where to return to when a return from the subroutine is made. The easiest method of doing this is to use the stack and in fact in the above example the system would have stored the return addresses from your subroutine on the stack in addition to the data which you stored there yourself. On returning from the subroutine, the return address would automatically be popped off the stack and transferred to the instruction pointer (IP) register, which indicates the current execution address of your program. Therefore, the order of data actually stored on the stack would have been as follows:

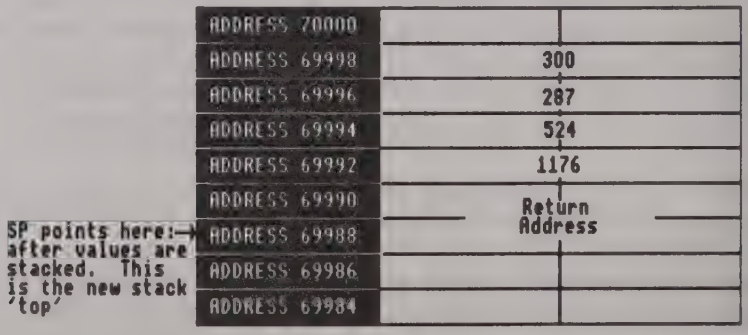


FIGURE 5-4.

Obviously the return address data stacked by the system would not interfere with your own data at all since it would be retrieved first.

You will notice that in the above diagrams the stack is depicted as columns of pairs of addresses. This is because only *word* and *long word* data can be pushed onto a stack. Obviously, if we were to push byte-sized data on to the stack as well there would be both odd and even sized items of information stored there which would cause some confusion. If you wish to store a single byte of data on a stack then the solution is simply to store it as part of a word, remembering that half the word is irrelevant when you come to retrieve it. For example, to store the value 6 on the stack it could first be loaded into the lower half of a data register and then pushed on to the stack. For example, `MOVE.W #6,D4` followed by `MOVE.W D4,-(A7)`.

In the above example we saw how the stack can be used to store temporary items of information when we need to free one or more registers for some other purpose. What happens when we want to free *all* the registers whilst retaining their data? Rather than having to stack the contents of each register separately, the 68000 has a single instruction, `MOVEM` (Move Multiple), which makes it possible to push or pop the contents of all or some of the registers at once. This essentially works in much the same way as the stacking process for the contents of a single register. With the `MOVEM` instruction the data in each of the registers is transferred to the stack in the order in which you list them in the instruction and the SP register is decremented accordingly. On retrieval, again using `MOVEM`, the data is popped back into the registers. For example, the instruction

```
MOVEM.L D1-D4/D6/D7/A3-A6,-(A7)
```

pushes all of the contents of `D1`, `D2`, `D3`, `D4`, `D6`, `D7`, `A3`, `A4`, `A5` and `A6` onto the stack. The instruction

```
MOVEM.L (A7)+,D1-D4/D6/D7/A3-A6
```

retrieves them.

Reverse Stacks

A reverse stack works in exactly the same way as a normal stack except that it grows *upwards* in memory rather than downwards. Again, data is pushed and popped on a LIFO basis but the SP register is of course incremented *after* every push operation and *decremented before* every pop operation.

Queues

The stack principle is used in one other useful kind of data structure in the 68000: the queue. In a queue, data is pushed in much the same way as it is in a stack except that it is popped on a first in, first out (FIFO) basis. You can imagine a queue as looking rather like a stack except that it is open at both ends. As fresh data is added to the head of the queue, earlier data is left further and further behind. A queue needs two stack pointers, which consist of two address registers: one to point to address of the first free space at the head of the queue (the 'put' pointer) and one to point to the back of the queue (the 'get' pointer). The put pointer works in a similar way as the SP pointer in a reverse stack: it always points to the current free space and increments according to the size of the data which is put on the queue. The get pointer always points to the earliest item of data contained in the queue and therefore can be used to retrieve it. The get pointer is also incremented when data has been retrieved from the queue.

In the following diagram you can see how this works. The data has been put in the queue starting from address 60000 and the last item to be entered is at address 60010 The put pointer points to address 60012, the next free address, while the get pointer points to the earliest item of data at address 60000. If we wanted to retrieve the first word which was entered in the queue we would take it from the address pointed to by the get pointer using, say, `MOVE.W (A3)+,D1` and the pointer would then move up to the next item of data in sequence at 60002. If we wanted to add a word to the head of the queue we would place it at the address pointed to by the put pointer, which would then increment by two bytes to point to the next free space at 60014. e.g. `MOVE D6,(A4)+`.

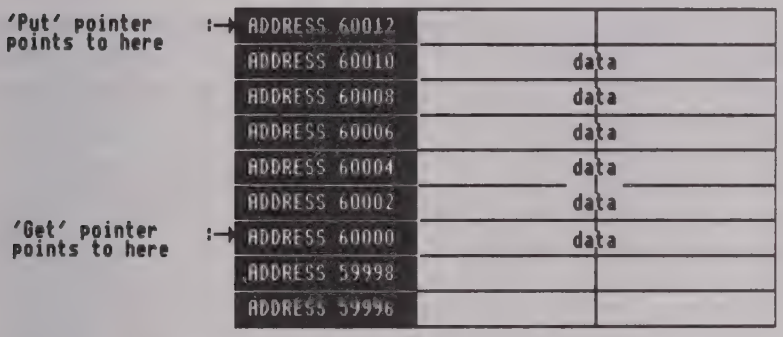


FIGURE 5-5.

Like stacks, queues can extend upwards or downwards in memory, the only difference in operation being that the put and get pointers *both* either auto-increment or auto-decrement, according to the direction of the queue.

Since both pointers either increment or decrement in the same direction for both types of queue, the problem arises that as data is stored and retrieved, the queue begins to creep through memory from the point where it started and if we are not careful it can move through memory like a caterpillar and devour any program or data code in its path. It is necessary, therefore, to create 'circular' queues, in order to restrict their movement within predefined limits. This is done by checking the put pointer to ensure that it has not moved beyond a predetermined address. If it has then the entire queue can be moved and the get and put pointers adjusted accordingly.

Altering Return Addresses

As we saw earlier in this chapter and in Chapter 4, when a branch is made to a subroutine the address of the instruction immediately following the branching instruction is automatically saved on the stack for subsequent retrieval when the subroutine has been completed, so that the original program sequence can be recommenced from the point at which the temporary branch was made.

It is sometimes necessary in a program to alter return addresses on the stack, either to redirect a return to a point other than the address following the original calling instruction or because you have used a programming method which causes the return address to be incorrect.

For example, suppose that you branch from your main program to a subroutine. Within the subroutine you have a conditional decision routine, after which control will return to the main program either at the point where you left off or at some other point, depending on the outcome of the decision operation. To return to the point where you left off you would simply use an RTS instruction and the return address would be retrieved from the stack and execution would recommence from that point. To return to a different address however, the return address would have to be altered so that it corresponds to the new location. One method of doing this would be to store the new return address on the stack by over-writing the old one. For example, if the old return address is currently on top of the stack and the new return address is in register A4, you can simply load the contents of A4 into the address pointed to by A7 without actually altering the value of A7:

```
MOVE.L A4,(A7)
```

The RTS instruction would then cause a return to be made to the new address.

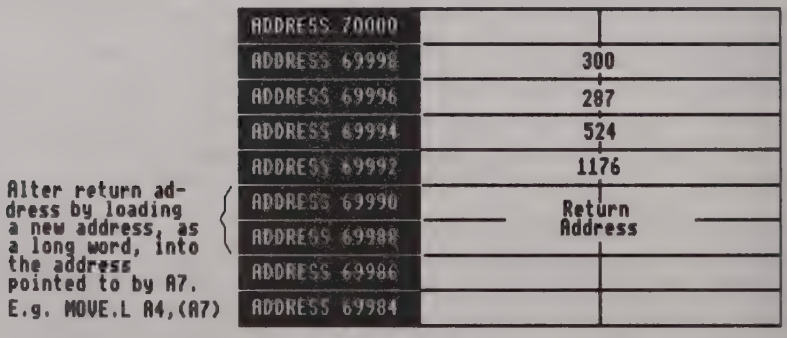


FIGURE 5-6.

Passing Parameters via the Stack

In Chapter 4 we looked at how parameters can be passed to subroutines 'by register', where they are simply passed via registers, or 'by name', where the address label of the data block in which they are stored is passed to the subroutine. It is also possible to pass them using the stack. In this case, the parameters are simply pushed on to the stack prior to the subroutine call and then retrieved from the stack by the subroutine as required. As you will see from the following diagram, which uses the same data parameters as in our previous example, the four words are placed on the stack before the return address, which is always pushed onto the top. The data must therefore be retrieved by by-passing the return address using an offset value, e.g. `MOVE.W 4(A7),D1`. The stack pointer stays unchanged and the word, above the return address is copied into D1.

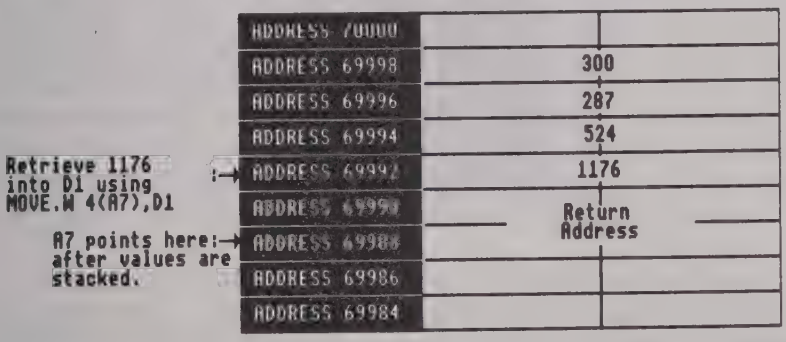


FIGURE 5-7.

When the subroutine has finished, and a return has been made to the main program, the four values should be popped back off the stack in the normal way. This last operation is performed to tidy up the stack so that it does not have any redundant data left in it.

The reason why the stack should be kept tidy is that when it contains a great deal of mixed data such as temporary variables, return addresses and parameters, it is very easy to lose track of its contents. At best you will end up by popping incorrect data from the stack and using it in your programs and at worst you will pop incorrect return addresses, rendering your programs inexecutable. Everything pushed onto the stack should therefore be removed as soon as it is no longer needed.

Stack Frames

You can create a temporary area within a stack for use by a particular subroutine. This is done by creating a second stack pointer (any one of the address registers), which points to the current top of the stack, and then moving the normal stack pointer, register A7, further down in memory. The stack space between the two pointers is reserved space which can be used exclusively by a particular subroutine and then cancelled after the subroutine has been executed, thus restoring the stack to its original state.

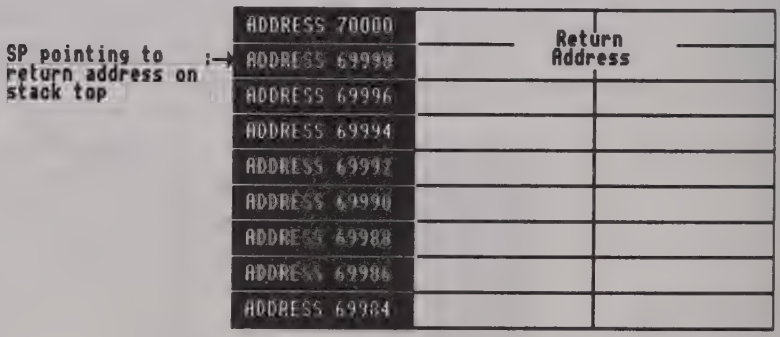


FIGURE 5-8.

The above diagram shows the condition of the stack on entry to a subroutine with SP pointing to the return address which is on top of the stack at address 69998. If you copy SP into an address register, say A4 for example, and then subtract 10 from SP, the stack pointer will then point to address 69988 as shown in the following diagram.

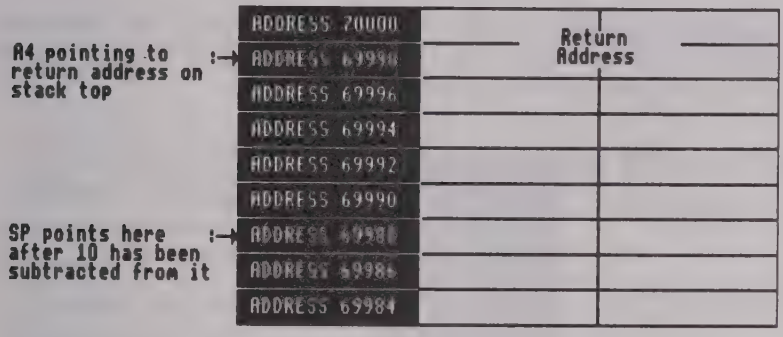


FIGURE 5-9.

This creates a free local storage area between addresses 69996 and 69989 into which 5 temporary word variables created by the subroutine can be stored. This area is termed a *stack frame* and individual data items can be accessed within this frame using A4 as a base pointer. For example, a word stored in addresses 69996 and 69997 can be addressed as `-2(A4)` and a word stored in addresses 69994 and 69995 can be addressed as `-4(A4)`. Before returning from the subroutine, the stack frame can be cancelled and SP returned to its original address 69998 simply by copying A4 back into SP with `MOVEA.L A4,A7` and then a straightforward return can be made.

In practice, the construction of stack frames can sometimes become fairly complicated such as in the implementation of high level languages in 68000 machine code, where a series of multiple linked stack frames may be needed. To make the job simpler, two special instructions are provided which allow you to create and remove stack frames: `LINK` and `UNLK`.

When the `LINK` instruction is used, the following three actions are performed automatically. In this example we shall assume that register A4 is to be used as the reserved space pointer:

1 The current contents of register A4 are pushed on to the stack

2 The SP register (A7) is copied into register A4

3 The SP register is decremented by an appropriate amount, specified by a 16 bit displacement integer. This is a two's complement value and, since a stack normally extends downwards in memory, it will usually be a negative value between –2 and –32768.

For example, to reserve eight bytes on the stack, the following instruction might be used: `LINK A4,#-8`. The effect of this instruction is illustrated in the following diagram:

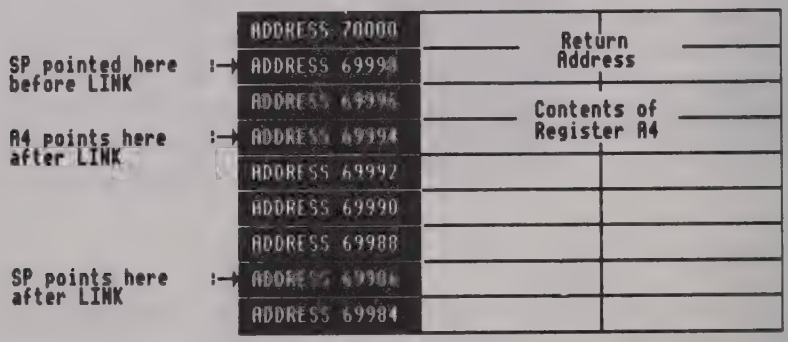


FIGURE 5-10.

The original contents of the stack were the return address for the subroutine. The current long-word length contents of A4 are then pushed on to the stack and the SP register is decremented by four bytes in the normal way. SP is then copied into A4 and is then decremented by 8 bytes, as specified in the instruction. We now have two stack pointers. SP can be used in the normal way, as the user stack pointer, and A4 is the pointer for the stack frame of eight reserved addresses.

Data can be entered into the stack relative to A4. The instruction `MOVE.L D4,-4(A4)` for example, will load the long-word contents of D4 into the stack frame at addresses 69990 to 69993.

When the subroutine has been completed, the entire stack frame can be removed, and the stack returned to its former state, by reversing the Link process: The contents of A4 are automatically copied back into A7 and the original contents of A4 are popped off the stack into A4. By this means, the entire stack frame effectively disappears and the stack returns to its original condition. This unlinking process is achieved by using the `UNLK` instruction, which simply specifies the name of the stack frame pointer being used, e.g. `UNLK A4`.

Chapter 6

Data Handling

In Chapter 2 we looked at the addressing modes used by the 68000. We saw how, by use of the indirect addressing modes, we could use address registers to contain addresses pointing to items in blocks of data stored in memory.

In this chapter we shall be looking in more detail at the uses of the indirect addressing modes, showing how they can be used to access complex arrangements of data.

In the earlier example we had a set of data, occupying 12 consecutive bytes in memory, which represented 12 month's totals of petrol consumption. Suppose we wished to go further and include the consumption figures for a three year period. We might want to use these, for example, in a program which compared our petrol consumption for corresponding months in each of the three years. Our data now occupies 36 bytes and the first item of data, our petrol consumption for the first month of year 1, is located at address 80000, which is labelled '`DAT_1`'.

Initially, we might want to access the data for January in each of the three years, then for February and so on. To do this, we need to have some method of pointing to the required months.

The simplest method is to use *direct* or 'absolute' addressing, in which the data is addressed by reference to its actual address number or to a label representing the actual address number. The instruction `MOVE.B DAT_1,D4` for example, copies the first item of data into the low order byte of register D4. Other individual items can be addressed by using their address value or an equivalent label.

This addressing method works very well for individual items of data but for addressing multiple items in complex structures of data the *indirect* addressing modes provide much more flexibility.

If we use address register indirect, loading an address register with the value 80000, we can use the address register indirect with postincrement mode to move sequentially through memory, accessing the data for each month, say, one byte at a time. However, this may not be exactly what we want to do: instead of moving sequentially we may want to access the data for January of the first year, January of the second and third years and then February of the first, second and third years and so on. To do this, we need to have some method of pointing not only to the required month but to the required month in any of the three years.

The data for January of the first year is located at address 80000 (`DAT_1`), so we load this address into an address register, say, A3, using an instruction such as `MOVEA #80000A3` or `LEA DAT_1,A3`. `LEA` means 'load effective address' and calculates the physical address of the labelled operand and loads it into a specific address register. It is now easy to access the data for this month, using an instruction such as `MOVE.B (A3),D1` to move the January data into register D1. This is a straightforward application of *address register indirect* addressing. January of the following year is 12 months, and therefore 12 bytes, further on. We can access this by means of an instruction such as `ADD.B 12(A3),D1` which adds the value 12 to the value of A3 (leaving A3 itself unaffected) and then adds the data contained in address `DAT_1+12` (i.e. the value of A3+12) into D1. The data for January of the third year would, of course, be accessed and added to the total by means of an instruction along the lines of `ADD.B 24(A3),D1`. Figure 6.1 shows this in diagrammatic form.

This type of addressing corresponds to the *address register indirect with displacement* addressing mode described in chapter 2.

The number which is added to the A3 register in this mode is termed the *displacement* and must always be no larger than a word-sized value. The displacement is always sign-extended prior to being added to an address register and therefore any 16 bit value which has its most significant bit set (i.e. numbers in the range 32768 to 65535) will become negative. The displacement, therefore, is effectively a value in the range –32K to +32K.

This is not the end of the story. It would be awkward if we always had to have a separate instruction for each of the displacement values we used in such an operation. It would be a lot easier if we could

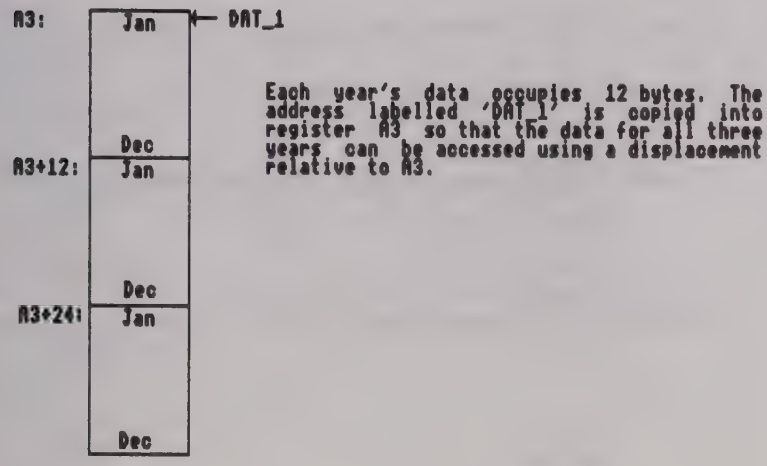


FIGURE 6-1.

use a pointer to indicate the start of our data and have another variable pointer which could be altered within our program to index each subsequent item of data automatically.

If we were to load an address register such as A3 with the start address of the data as before and one of the other registers, say D4, with a zero: `MOVEQ #0,D4` then register A3 points to the base of the data, as before, and D4 can be used as an *index register* to point to locations which are relative to the beginning of `DAT_1`. If A3=80000 and D4=0 then January of the first year can be addressed as 0(A3,D4), the zero being a zero displacement. So we could retrieve the January data with an instruction such as `MOVE.B 0(A3,D4.L),D5`. By altering the value of D4 within a program loop we can easily point to any individual item of data in the table.

The above `MOVE` instruction is given the suffix '`.B`' because it is byte-sized data which we are transferring and the index register is given the suffix '`.L`', indicating that it is the entire 32 bits of D1 which forms the index value.

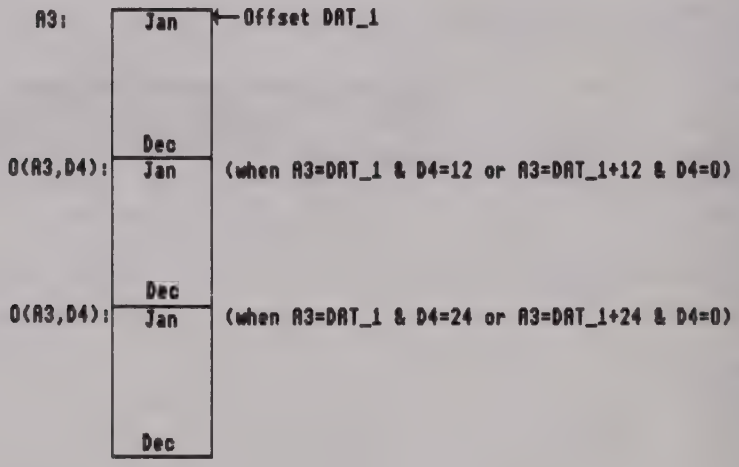


FIGURE 6-2.

If we wish to compare different months in each year we can alter both the address register and the index register as required. A3 then A3+12 followed by A3+12 again will base the address register at the first month of each of the three years in succession, whilst D4 can be incremented or decremented to point to any month within a year as required.

This is an example of the *address register indirect with displacement and index addressing* described in Chapter 2.

If we use the displacement as well as the index in this mode, it increases addressing flexibility considerably. Again the displacement is a sign extended value but when used with an index it represents a displacement of only –128 to +127 bytes. The index value is either long (32-bit) as in the previous example, or a 16-bit sign extended value in the range –32K to +32K.

Suppose that in addition to petrol consumption data for a three year period, you also had blocks of data giving mileage information for the three year period as well. The data starts at address `DAT_2` and extends for 72 bytes. The data is arranged in groups with the 12 consumption figures for year 1 followed by the 12 mileage figures for year 1 and with the same arrangement for the two following years.

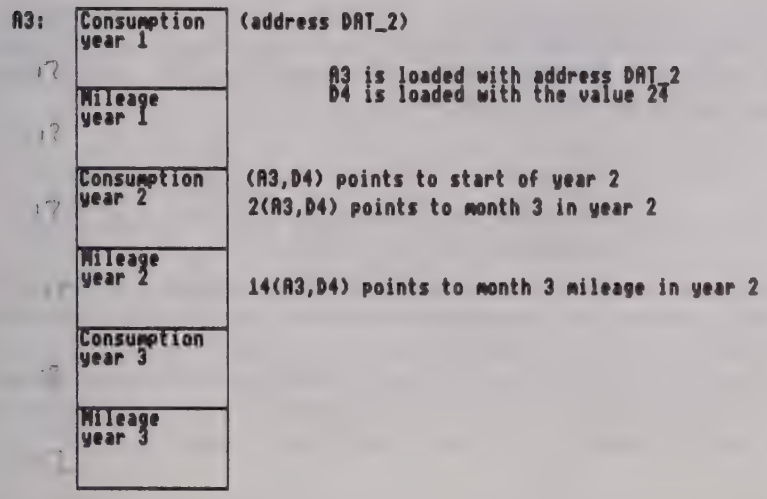


FIGURE 6-3.

Your base register (A3) can point to the block of data required, your index register (D4) to the year required and your displacement constant to the data required. Referring to figure 6.3 you can see that you can get the consumption data and the mileage data from the third month of the second year by means of the instructions:

```
LEA.L DAT_2,A3       ;address of base of data table.
MOVEQ #24,D4         ;offset of 2nd year (Jan. consumption)
MOVE.B 2(A3,D4),D5   ;consumption for year 2, month 3 into D5
MOVE.B 14(A3,D4),D6  ;mileage for year 2, month 3 into D6
```

In this case D4 points to the offset of the year and the displacement points to the offset of the month (counting from zero). Alternatively you could use the displacement to point to the year and the index to point to the month, depending on how you want to access the data.

The same data for month 3 of year 3 can be obtained with the same instructions but after adding 24 to D4 to point it to the start of the third year's data. Different months can be addressed by using different displacement values and, if sequential data needs to be accessed then the value of D4 can be altered within a loop structure so that it is incremented to point to each item in turn.

There are many different ways in which indexed blocks of data might be used in a program. On a simple level they can be used in much the same ways as multidimensional arrays in BASIC and the above examples illustrate this kind of usage.

Indexing Lookup Tables

Another common form of indexed data is the lookup table, in which a list of numeric or textual data is stored in a single dimensional array for reference by programs. Suppose that you are writing a conversion program in which the multiplication factors for converting the source data are stored in a separate area of memory. A base address register, such as A2, points to the base of the table: `LEA MYTABLE,A2`.

You may to convert feet into metres and the necessary 16-bit conversion factor is stored at offset 7 in the table (counting from zero). The data may be retrieved from the table using an instruction such as `MOVE.W 7(A2),D3` and then used in your calculation.

Another use for lookup tables is where you have a series of subroutines stored in memory and you need to select any one of these during the execution of a program.

The address of each of these subroutines could be stored in a lookup table so that they can be used to redirect execution to any of the subroutines required. The advantage of this method is that all the required addresses are stored consecutively as 32-bit numbers in the table and each one can be accessed using an indexed offset. You can then use the table base address with the indexed offset to access any of the address numbers listed in the table and then redirect execution to the target subroutine.

For example, a peripheral device such as a joystick might feed a particular value into the computer which indicates that a subroutine for shooting at a space invader should be called. If this value is 8 (contained in, say D3) then the address of the subroutine is at offset 8 in the subroutine table. The base of the table might be contained in register A6. The instruction `MOVEA.L 0(A6,D3),A2` loads the address of the subroutine, located at A6+D3, into A2 and the appropriate subroutine can then be called using an instruction such as `JSR (A2)`. Alternatively, you could achieve the same objective by substituting the single instruction: `JSR 0(A6,D3)`.

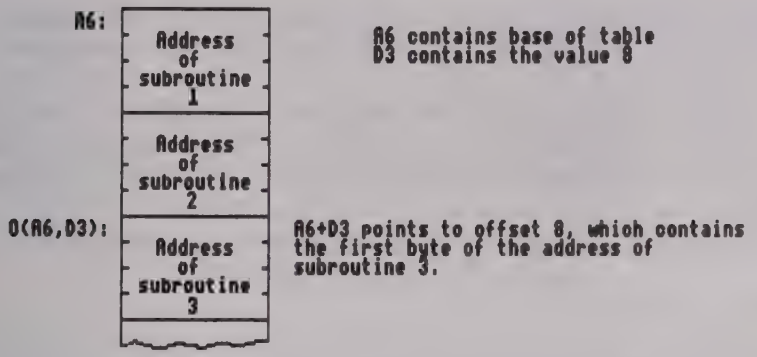


FIGURE 6-4.

Block Instructions

Another important use for indexing is where you have blocks of data which need to be compared, input from an external device, output to an external device, processed in sequence, transferred from one part of memory to another or stored sequentially in memory as they are generated by a program.

For example, in a games program you might have a block of data containing the graphics bit-pattern for an entire screen image which periodically will be copied on to the VDU screen. Alternatively, you might be writing a printer dump program which copies the bit-pattern of the screen image to a printer, one byte or one word at a time.

Again, you might have a block of data in memory representing the ASCII codes of a text which you wish to print to the screen and which needs to be loaded one byte at a time into a particular register before being passed to a display subroutine.

In cases where blocks of data need to be addressed in this kind of way, the address register indirect with postincrement or address register in direct with predecrement addressing modes are used.

As an example of a block operation, suppose you wished to copy 100 items of consecutive data from one memory area to another. In order to transfer the data it is necessary to use at least three registers: one pointing to the first address of the destination of the data (say, a

VDU screen address location), one pointing to the first address of the source data and one to count the quantity of data transferred.

The source register, say A2, is loaded with the address of the first byte of the source data: `MOVEA.L £80000,A2`. The destination register, say A3, is loaded with the address of the first byte of the area of memory into which the data will be moved: `MOVEA.L #90000,A3`. A data register is then loaded with the number of bytes to be moved, less 1:

```
MOVE.L #99,D4.
```

The entire operation can then be executed in a program loop with a `DBRA` (decrement and branch) instruction being used to decrement D4 and branch back to the instruction labelled '`LOOP`' until D4 equals –1:

```
LOOP    MOVE.B (A2)+,(A3)+
        DBRA D4,LOOP
```

The `MOVE` instruction automatically copies the data from the location addressed by A2 to the location addressed by A3 and then increments both A2 and A3 so that they point to the next locations in sequence. The `DBRA` instruction automatically subtracts 1 from D4 and if it is greater than –1, branches execution back to '`LOOP`' to repeat the operation.

The operation could also be performed backwards with A2 and A3 initially pointing to the *last* addresses in the source and destination blocks, plus 1. In this case the address register indirect with predecrement mode would be used:

```
LOOP    MOVE.B -(A2),-(A3)
        DBRA D4,LOOP
```

Instead of moving a block of data from one area of memory to another, we could equally well compare two separate blocks of data. This is frequently done, for example, where text has been input at the keyboard and you wish to compare it with text stored in memory. In this case we would use a method similar to the above, using registers to point to the start addresses of the two blocks of data to be compared. Instead of using the `MOVE` instruction we would use `CMPM` (compare memory): `CMPM.B (A2)+,(A3)+` and instead of using `DBRA` as a loop control instruction we could use `DBNE` (decrement and

branch until not equal). The `CMPM` instruction will set the zero flag for each data pair of equal value and therefore, as long as each compared item is equal, `DBNE` will decrement D4 and loop back to '`LOOP`'. If two compared items are not equal the zero flag will be reset and execution will then pass to the next instruction in sequence. In any case the loop will terminate when the counter register equals –1.

Altering Indexed Blocks

There will be occasions when you wish to modify the contents of a block of data: to change, add or remove items for example. Again, this is done using indexing methods to access the required items in the block. How would we do this? Obviously, we need some method of defining the size and structure of the list so that a program can add, subtract or modify items accurately. Suppose we had a list of unstructured items – in other words, the data in the list does not correspond to any external structural concept such as years or months. You may, for example, be programming a whole series of mathematical calculations, the results of which you wish to store in a variable length list for later reference. In this case you could structure the list as follows:

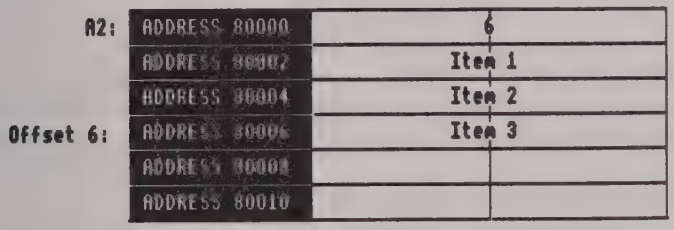


FIGURE 6-5.

Here the list is of variable length and is comprised of a series of word-sized data. You may wish to add or subtract items, either at the end of the list or within the body of it. The first word on the list is a value corresponding to the offset value of its last item. Therefore, to reference the end of the list, all you need to do is to load this value into an index register and use it to reference the last entry. The list starts at an address offset within the data section labelled '`LIST_1`' so we load this into a base register.

```
LEA.L LIST_1,A2
```

Address 'LIST_1' contains the offset value of the last item in the list, 6, so we load this into a data register to use as an index:

```
MOVE.W (A2),D4
```

Item 3 at offset 6 in the list can now be retrieved by an indirect index instruction, e.g.:

```
MOVE.W 0(A2,D4),D6
```

and 'removed' from the list by decrementing D4 by 2, which implies that item 2 is now the last item in the list at offset 4:

```
SUBQ #2,D4
```

The fact that item 3 still physically exists does not matter since nothing is now pointing to it.

Suppose that we now wish to add a new item 3 to the list, which is currently in data register D3. The index needs to be incremented again:

```
ADDQ #2,D4
```

And the new item can be added:

```
MOVE.W D3,0(A2,D4)
```

Then to add a fourth item, from D5:

```
ADDQ #2,D4 (D4 now points to offset 8)
MOVE.W D5,0(A2,D4)
```

Now we have finished with the list for the time being. D4 contains the offset value of item 4 which is now the last item in the list. This must be copied into the first word location in the list for future reference:

```
MOVE.W D4,(A2)
```

And so the operation has been completed. This same technique can be modified so that items elsewhere in the list can be added, subtracted or modified in some way. Suppose that we wished to take the second item in the list, add 10 to it and replace it. Assuming that A2 is still pointing to address 'LIST_1', we must first check that there is a second item in the list. So we retrieve the offset value of the last item, as before:

```
MOVE.W (A2),D4
```

The offset of item 2 will be 4 (2 times 2) since we are dealing with word length data. We therefore need to *compare* the contents of D4 with the value 4 to make sure that we are not looking for an item beyond the boundary of the list:

```
CMPI.W #4,D4
```

The comparison effectively subtracts 4 from D4 without actually altering the value of D4. However, certain flags may be affected by the operation and can be used to test the result.

If the result is negative (i.e. D4 $<$ 4) then the carry flag will be set because the operation will cause a binary borrow, otherwise 4 will be either less than or equal to D4 and the carry flag will be reset. In either case we can take some appropriate action.

If the carry flag is set then there cannot be an item at offset 4 and so the operation should be abandoned. At this point there would be an instruction such as `BCS NEXT` (branch if carry set to a point in the program labelled 'NEXT').

If the carry flag is not set, there must be an item at offset 4 in the list and so we can access it using the instructions:

```
MOVE.W 4(A2),D6   ;get item at offset 4 into D6
ADDI.B #10,D6   ;add 10 to it
MOVE.W D6,4(A2)   ;replace it at offset 4
```

Alternatively, the above three instructions could be replaced by the single instruction `ADDI.B #10,4(A4)`.

There is no need to update the end of list offset value because we have done nothing to alter it.

Now let's go back to our petrol consumption data and find out whether we can see more clearly how this kind of arrangement allows us to access the information in whatever way we wish. We shall consider just twelve months consumption data as we did originally, except that this time our array, labelled DAT_1, is arranged in word lengths, with an extra word at the beginning of the table which holds the offset of the last item in the table. We know of course that there are twelve months in a year but the computer doesn't know that and so addresses DAT_1 and DAT_1+1 must contain the value 24; the offset of the month 12 data. By loading the offset of the last item in the table into an index register we can point to the address of the data for the twelfth month and we can alter this and use it as we wish.

```
;-----------------------------------------------------------
;Instruction     Comment
;-----------------------------------------------------------
LEA.L DAT_1,A2  ;Get the address of the start of
                ;the table in register A2
MOVE.W (A2),D4  ;Get the offset value of the
                ;last item of the table in D4
MOVEQ #0,D1     ;Set D1 to offset value 0
```

What might you want to do with this data? If you want to work out your average consumption for the year you can run your D1 index register through the data, adding the value of each item of data to, say, D6 and comparing D4 and D1 after each addition to make sure that D1 does not exceed the boundary of your data:

```
;-----------------------------------------------------------
;Label   Instruction            Comment
;-----------------------------------------------------------
        MOVEQ #0,D6            ;Clear the D6 register
LOOP1   ADD.W 0(A2,D1),D6      ;Add an item of data to D6
        ADDQ #2,D1             ;Update the index pointer
        CMP D1,D4              ;Compare the index offset
                               ;with the end of list offset
        BCC LOOP1              ;If carry clear (end of list
                               ;not reached), branch back
                               ;to 'LOOP1'
;-----------------------------------------------------------
```

When you have the final total you divide the contents of the D6 register by the number of items in the list and you have your average. This division operation is illustrated in example program PROG6 in Chapter 13.

You might want to know the highest consumption figure for any one month. In that case you need a data register, say D6, to hold the highest item value. You run through the data table as before and compare each sequential item of data with the highest item value found so far. If any item is higher than the current content of D6 then the value of the item becomes the new D6 value and is MOVEed into the D6 register, otherwise you carry on to the next item until the Bcc instruction indicates that you have finished.

If you feel that you cannot work out the exact sequence of instructions required then don't be too concerned. It is only essential at this stage to establish a firm mental picture of what it is you are trying to do, how your data might be efficiently arranged and the kinds of addressing methods you need to use to access the data. Once you have properly understood the structure of your task, you will eventually be able to code the instructions standing on your head – and often, with complex structures of data, it will feel that way.

Sorting Data

It is fairly easy to sort the items in a list so that, for example, its data is rearranged in ascending numeric order. The following BASIC program illustrates a very simple bubble sort, in which a FOR/NEXT loop is used to compare each item in a single dimensional array A(6) with the next item. If the value of an item is greater than the next item then a flag is set to indicate that the values are out of order and the values are then swapped. The process continues until all the items are in the correct order, as indicated by the 'FLAG' variable becoming equal to zero at the end of a reiteration of the loop:

```
10 FLAG=0
20 FOR X=1 TO 5
30 IF A(X) <= A(X+1) THEN GOTO 80
40 FLAG=1
50 TEMP=A(X)
```

```
60 A(X)=A(X+1)
70 A(X+1)=TEMP
80 NEXT X
90 IF FLAG=1 THEN GOTO 10 ELSE END
```

We shall be looking at a full assembly language version of a bubble sort in Part II. At this point it is only necessary to think about the structure of the array of data in assembly language and to see how the indirect addressing modes we have been using could be used to form an equivalent method of sorting to the BASIC method above.

If you examine the following diagram of the data list, you should be able to work out how the assembly language version of the program might be constructed. There is always more than one way of doing something so there is no 'right' answer – you should think about the simplest and most efficient way of doing it.

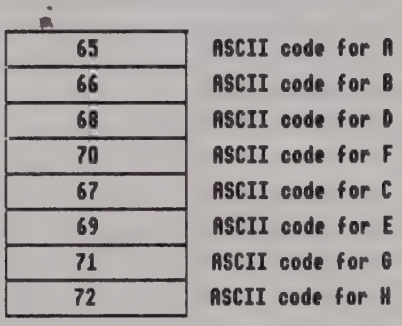


FIGURE 6-6.

Program Positioning and Labelling

The problem with addressing and branching is that, at the time a program comes to be executed, its position within memory may not be known. When a program is loaded into a computer it can either be loaded into a specific block of memory or, more commonly, it is loaded into the first available memory area. The former is called *position dependent* and the latter, *position independent* code.

It is important when planning a program to decide which of these it is going to be. Preferably, programs should be position independent because you may not know beforehand the circumstances in which a program will be used. At some point it may have to share memory with other programs or it may be transported and run on an entirely different type of 68000 based computer from the one on which it was coded.

If a program is position dependent, the addresses of its various data blocks, subroutines and other code sections will always be the same. In position independent programs the addresses may differ each time the programs are loaded, depending on how many other programs are loaded and the amount of space which they occupy. The main thing which concerns the programmer however, is that the positions of all the elements of a program are constant relative to each other. When you wish to access a particular data table amongst several which occupy memory, you need to be confident that the address value of the table relative to the position of the current instruction is correct.

This is why the use of labels to identify the locations of code and data sections is so useful. By identifying each separate section of a program by means of labels in your assembly source program, you ensure that the assembled object code contains the correct relative offset values for all the key elements of the program which you will need to access. If your program is position independent, the assembler will automatically calculate all relative branches and the relative positions of data items using the PC relative addressing mode.

Your petrol consumption table might be assigned the label `GASTAB1`. When the program is finally assembled into object code, the assembler knows the exact position of your table in memory relative to the start of your program. Any instruction in the program which refers to this table by means of its label, such as `LEA,L GASTAB1,A2` will automatically be assembled in such a way that the relative displacement between the current program instruction (held in the PC register) and the required data will be computed automatically. Likewise, a table called `GASTAB2` will be given a different address value, which again will be obtained by reference to the label.

Chapter 7

Exceptions, I/O and Arithmetic Operations

Exceptions

When a computer is in operation, even if it is not currently executing programs, it has a number of routine tasks to perform. The keyboard needs to be checked for input and other devices may be sending signals or data which require the processor's attention.

These kinds of tasks are performed by *exception* mechanisms, whereby the CPU literally interrupts whatever it is doing and services any exception task which need to be performed.

An exception operates in a similar way to a subroutine call. The CPU halts execution at whatever point in a program it has reached and recommences execution with an exception procedure, after which it normally returns and carries on with its original task. An exception differs from a subroutine call in that it is designed to cope with a set of necessary and sometimes urgent tasks for which it would be impractical or impossible to incorporate `BSR` or `JSR` instructions in a program. Many of these tasks have nothing to do with the program currently being executed.

Exception calls may be initiated by a timer, in the case of the keyboard scan for example, by signals from an external device or by program errors such as attempts to divide by zero or to address non-existent memory locations. Exceptions may also be initiated by certain program instructions.

There are basically two types of exceptions: *external* exceptions, associated with peripheral devices, with bus errors and reset signals, and *internal* exceptions, which are associated with the execution of program instructions.

Internal exceptions may be initiated by certain 'privileged' instructions attempted in user mode, by attempts to address word or long word data at odd addresses, by attempts to use illegal or unimplemented instuctions, by using the trace facility and by certain other instructions which will be discussed presently.

Some exceptions may be masked or *disabled* by setting the three I (interrupt) flags in the CCR register. These are termed *maskable* exceptions and may include, for example, the operating system exception which checks the keyboard. By disabling exceptions you can inhibit keyboard entries until a certain routine is completed and then exceptions can be re-enabled by resetting the I flags. Whenever an exception takes place then further exceptions are automatically disabled until the exception routine has been completed.

Some exceptions are *non-maskable* – they will operate no matter what the status of the interrupt flags. Obviously if some catastrophic event has occurred in the system it is desirable that its associated exception mechanism should be able to override other events which are taking place.

Operation of Exceptions

Whenever an exception takes place, the program which is currently being executed is halted, usually temporarily, and execution is then diverted to one of a series of procedures which are designed to 'service' the exception, that is, to do whatever is necessary for the type of exception which is taking place and then return execution back to the program. Some service procedures will do no more than print an error message while others may perform fairly complex tasks – it depends on what the system programmers have designed for the particular computer which you are using.

These service procedures are accessed by means of a *vector table*, situated in lower memory, which contains the addresses of all the exception routines provided by the system. The vector table is indexed by a 'vector number', which is either calculated within the system or which is supplied by an external device connected to the system. The address of the service routine is then read from the vector table at an offset derived from the vector number, and execution is redirected to it.

Since a program may be interrupted at any point by an exception, it is vital that the current parameters of the program, such as the contents of the status register and the current PC contents, are saved for later retrieval. This is performed automatically during an exception and other parameters may also be saved, depending on the type of exception taking place.

Return from an exception is effected by the inclusion of an RTE (return from exception) instruction at the end of the exception service routine.

Exception Priority System

Exceptions are not all of equal priority. Clearly an exception resulting from an attempt to divide by zero is less important, in terms of urgency, than a sudden voltage loss. In order to differentiate different levels of priority, the 68000 has an exception priority mechanism which determines a priority level for each category of exception. Priority group 0 exceptions occur immediately, whatever else the processor is doing at the time. Priority group 1 exceptions are delayed until the current instruction is completed. Priority group 2 exceptions are those which only occur when particular instructions are being executed. The exceptions within each group are are as follows:

Group	*Exception type*
0	Reset
	Bus error
	Address Error
1	Trace
	Interrupt request
	Illegal instruction
	Unimplemented instruction
	Privilege violation
2	TRAP instruction
	TRAPV instruction
	CHK instruction
	Division by zero

Internal Exceptions

Internal exceptions may be caused by the following:

Addressing Errors Since word and double word operands may only be aligned with even numbered addresses, an attempt to address a word or long word at an odd numbered address will result in an address error exception.

Privilege Violation Attempts to use certain 'privileged' instructions whilst in user mode will result in a privilege violation exception. These instructions include:

```
AND.W immediate to SR
OR.W immediate to SR
EOR.W immediate to SR
MOVE USP
MOVE to SR
RESET
RTE
STOP
```

Illegal commands An illegal instruction is one which does not belong to the 68000 and therefore has no opcode which is intelligible to the processor. All illegal opcodes will will cause an exception. Unimplemented instructions are similar except that they are a special case. Any instruction code whose higher four bits consist of the binary digits 1010 or 1111 cause a special type of exception which allows system designers to simulate instructions which are not implemented on the standard 68000.

Trap exceptions The `TRAP` instruction is used to divert execution to particular system subroutines. Trap exception types 0–15 occur only when the `TRAP` instruction is used. The `TRAPV` instruction causes an exception if the overflow flag in the status register is set when the instruction is used.

`CHK` generates a trap exception if the contents of the destination data register to which it

refers are less than 0 or greater than the contents of the source operand.

DIVS and DIVU instructions cause trap exceptions if they involve an attempt to divide by zero.

Trace exceptions When the 'T' (trace) flag in the status register is set, a trace exception is performed after every single instruction. The trace exception service routine is used in Chapter 8 to obtain a listing of the register contents during the execution of an instruction.

External Exceptions

External exceptions are generated as a result of events outside the immediate processor environment and may be caused by the following:

Bus Errors A bus error exception is caused by an attempt to address incorrect destinations such as non-existent addresses.

Reset A reset is an event in which the entire system is re-initialized, either when it is first powered up or when some event has caused catastrophic system failure.

Interrupts Interrupts are a type of exception resulting from a signal which is input from an external device. The exception vector to which execution is diverted is obtained as usual from the vector table but the vector number for the service routine is supplied by the interrupting device itself.

The external devices may each be assigned a priority level between 0 and 7 which, when an interrupt request is detected, is automatically compared with the processor priority level set by the three interrupt mask bits in the system byte of the status register. If the processing priority code is greater than or equal to the priority level of the

requesting device then the interrupt request is left pending while the next instruction is processed. If the requested interrupt is of a higher priority level then the interrupt exception is serviced immediately.

Exception Vector Table

The exception vector table occupies 1024 bytes, containing 256 exception vectors, each consisting of 32-bit exception routine address pointers. The first 64 vectors are dedicated to certain types of exceptions and the remainder are user defined vectors which are used by system designers for pointing to customized exception routines for particular operating systems.

Vector Number	Address Number	Exception Type
0	0	Reset
1	4	Reset
2	8	Bus error
3	12	Address error
4	16	Illegal instruction
5	20	Division by zero
6	24	`CHK` instruction
7	28	`TRAPV` instruction
8	32	Privilege violation
9	36	Trace
10	40	Line 1010 emulator
11	44	Line 1111 emulator
12–23	48–95	Reserved
24	96	Spurious interrupt
25	100	Level 1 interrupt autovector
26	104	Level 2 interrupt autovector
27	108	Level 3 interrupt autovector
28	112	Level 4 interrupt autovector
29	116	Level 5 interrupt autovector
30	120	Level 6 interrupt autovector
31	124	Level 7 interrupt autovector
32–47	128–192	`TRAP` vectors
48–63	192–255	Reserved
64–255	256–1023	User interrupt vectors

Input and Output Operations

On most processors, data may be input from or output to peripheral devices by means of I/O 'ports' which are addressed in a similar way to memory addresses, using instructions such as 'IN' and 'OUT'.

The 68000 does not implement these instructions and I/O operations must be performed via peripheral hardware devices such as the 6821 Peripheral Interface Adaptor (PIA) and the 6850 Asynchronous Communications Interface Adaptor (ACIA).

Communication via these devices can be fairly complex and a detailed description of their operation is beyond the scope of a book of this kind. In practice, many operating systems will provide a simple method of communicating through these devices by linking them to a trap mechanism so that data may be input or output by loading parameters into certain data registers and then initiating an appropriate trap mechanism, using the TRAP instruction.

Binary Arithmetic

Performing binary arithmetic on the 68000 is an easy matter because special instructions are provided for binary addition and subtraction and for signed and unsigned binary multiplication and division.

The instructions allow multiple precision arithmetic operations – that is, operands several words in length can be operated on, yielding multiple word results where necessary. Where there is a binary carry or borrow between operands, the 'X' (extend) flag is set and certain arithmetic instructions automatically pass the carry from one operand to the next. For this reason, it is advisable to ensure that the extend flag is reset before you start, using MOVE £0,CCR. With some arithmetic instructions the Z flag is set by a zero result but *unchanged* by a non-zero result. In this case it is necessary to ensure that the Z flag is in the desired condition before the instruction is executed if the flag is to be tested afterwards.

64-bit binary addition and subtraction

As a first example, let us suppose that we wish to add together two 64-bit numbers. There are several forms of the ADD instruction: ADD

(add binary), where one of the operands has to be in a data register, `ADDA` (add address), where the destination operand has to be in an address register; `ADDI` (Add immediate), where the source operand has to be an immediate value; `ADDQ` (Add quick), where the source operand has to be an immediate value in the range 1 to 8; and `ADDX` (Add extended), where the value of the extend flag is incorporated in the result. Since this is a multiple precision operation we shall be using `ADDX`.

The two 64-bit numbers are initially located in binary form in memory and will be transferred, one byte at a time into data registers, starting with the least significant byte of each number, and added together one at a time. If we store the bytes in data register D2 and D3 the addition instruction would take the form `ADDX.B D2,D3`. After each byte addition the result would be stored in D3 and may be transferred from there to a separate memory location. The program would then loop back and add the next bytes in sequence until all 64 bits have been added:

```
;-------------------------------------------------------
;A2 points to address beyond end of first 64-bit number
;A3 points to address beyond end of second 64-bit number
;D4 holds the number of additions to be performed, less 1
;D2 & D3 will hold each byte to be added

        MOVE #0,CCR       ;Clear flags in CCR
LOOP    MOVE.B -(A2),D2   ;Decrement A2 and move byte
                          ;from first number into D2.
        MOVE.B -(A3),D3   ;Decrement A3 and move byte
                          ;from second number into D3.
        ADDX.B D2,D3      ;Add bytes together  (including
                          ;extend flag  value).
        MOVE.B D3,(A2)    ;Replace sum in memory space
                          ;of first number.
        DBRA D4,LOOP      ;Subtract 1 from D4 and loop back
                          ;to 'LOOP' if D4 greater than -1
;-------------------------------------------------------
```

If you imagine the binary operands printed horizontally, you can see how this process corresponds to the manual addition process, from right to left, with the binary carries being passed automatically from a less significant to a more significant byte of the sum at each stage via the extend flag.

The addition instruction may be performed on both signed and unsigned operands and no separate instruction is needed for each case.

The same operation can be performed without removing the operands from memory, as follows:

```
        MOVE #0,CCR             ;Clear flags in CCR
LOOP    ADDX.B -(A3),-(A2)      ;Add two bytes
        DBRA D4,LOOP
```

Binary subtraction follows exactly the same principles as binary addition, the subtraction instructions being `SUB` (Subtract Binary), `SUBA` (Subtract Address), `SUBI` (Subtract Immediate), `SUBQ` (Subtract Quick) and `SUBX` (Subtract with Extend). If the above examples were subtraction operations we would, of course, be using `SUBX`.

Binary Multiplication

Multiplication is performed on either signed or unsigned operands and a separate instruction is used for each case: `MULS` (Multiply Signed) and `MULU` (Multiply Unsigned). In both cases the multiplicand (the number being multiplied) is held in a data register and the multiplier may be immediate, in memory or stored in another data register. The following example shows how a 16-bit multiplicand in register D2 is multiplied by a 16-bit multiplier in D1, with the 32-bit result being stored automatically in D2, using the instructions `MULS D1,D2` or `MULU D1,D2`, depending on whether signed or unsigned numbers are being used. Note that the original operands may not be larger than 16 bits. The multiplication instructions have no size specifier after them because operations are always of word-size.

FIGURE 7-1. MULU D1,D2.

Binary Division

Division is also performed on signed or unsigned operands and the instructions used are `DIVS` (Divide Signed) and `DIVU` (Divide Unsigned). In this case the dividend (number to be divided) is a 32-bit number in a data register and the divisor is a 16-bit number which may be immediate, in memory or in another data register. In the following example the unsigned dividend in D2 is divided by the immediate value 10 and the result is automatically stored in register D2, with the quotient occupying the low word and the remainder occupying the high word.

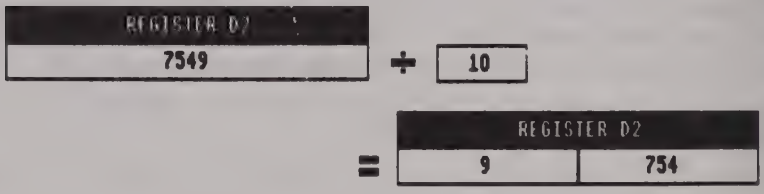


FIGURE 7-2. DIVU #10,D2.

If you need to swap over the quotient and remainder words in the register then the `SWAP` instruction is used, as demonstrated in `PROG6` in Chapter 13, e.g. `SWAP D2`.

Binary Coded Decimal Arithmetic

In some circumstances, binary representation is an inconvenient way or storing and transmitting data. If the computer is exchanging floating point numeric data, such as financial information, with some peripheral device then the format in which the data is represented by the two machines may be incompatible.

The solution to this problem is to use *binary coded decimal* (BCD) arithmetic, which is a technique whereby binary numbers are used to represent the decimal digits 0 to 9, so that numeric values can be stored, exchanged and processed in regular decimal form. This is done by using four bits of a byte (a nibble) to represent each decimal digit, as follows:

Binary	BCD	Binary	BCD
0000	0	1000	8
0001	1	1001	9
0010	2	1010	—
0011	3	1011	—
0100	4	1100	—
0101	5	1101	—
0110	6	1110	—
0111	7	1111	—

You will notice that, of the fourteen number combinations possible using four bits, nine are used to represent decimal digits and the rest are unused. This slightly complicates the performance of arithmetic operations as we shall see presently.

From this table you will see that it is possible to represent decimal digits directly in memory. The decimal digit 8, for example, could be stored in a memory byte as 00001000; exactly the same as it would be stored as a binary value. This representation is termed *unpacked* binary coded decimal (BCD). Since the remaining four bits of the byte are wasted, they too can be used to hold a BCD digit, so that a single byte can hold a decimal value between 0 and 99. The number 48, for example, would be represented as 01001000 (`BCD 4` followed by `BCD 8`). This form of BCD representation is termed *packed* BCD.

From this you will be able to see that there is no limit to the type of value which can be stored in this way. By using some of the unused four-bit codes in the above table, you can devise BCD data structures which incorporate signs, decimal points and other mathematical symbols. The decimal number –2834.85, for example, might be stored as follows:

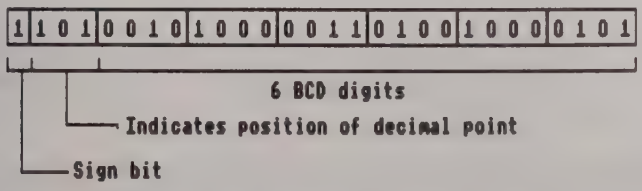


FIGURE 7-3.

The left hand bit indicates the negative sign of the number and the next three digits, 101, indicate that the decimal point comes just before the fifth digit of the value.

The problem with BCD arithmetic, however, is that because some of the possible bit combinations in a nibble are unused, this leads to inaccuracies when arithmetic operations are performed on them. For example, in *binary* arithmetic the addition of 4 and 8 yield the following result:

```
 00000100    = 4
+00001000    = 8
---------    ---
=00001100    =12
```

In BCD however, this is an inaccurate result since there is no representation of the number '12'. In BCD, the result required is '1' and '2' (0001 0010) which would be 18 in binary (i.e. a difference of six). This is, in fact, how adjustments to binary numbers are carried out in BCD arithmetic in order to convert then to correct BCD values. The least significant nibble in the binary result is 12 (1100). If we add 6 to this the result is modified as follows:

```
 0000 1100    binary 12
+0000 0110    binary 6
----------
=0001 0010
  1    2      = '12' in BCD
```

Whenever addition or subtraction operations are performed in which BCD rather than binary values are involved, a special set of instructions are provided which automatically convert the results into BCD format. These are `ABCD` (Add Decimal with Extend) and `SBCD` (Subtract Decimal with Extend).

These work in much the same way as binary addition and subtraction operations and may be performed between BCD values in memory or in data registers.

`ABCD`, like `ADDX`, takes the value of the extend flag into account during each stage of a calculation and multiple-precision BCD operations can be performed in the same way as binary operations, substituting `ABCD` for `ADDX` and `SBCD` for `SUBX`.

There are no specific multiplication or division instructions provided for BCD values.

Part II

Chapter 8

Assembling Programs

In the first chapter we took a general overview of the system, looking at the sizes of data which can be used in programming and the ways in which the data is stored in memory. We then looked at part of a simple program, seeing how the code and data sections relate to each other and how the processor executes a program by moving data from one location to another and processing it. Finally, we took a brief look at the general principles of assembler programs.

In this chapter we shall revise some of the key points about program and data storage. We shall also be starting to use hexadecimal numbering rather than binary representation. Finally, we shall examine the use of assembler programs in more detail and produce a complete coding of the example program which we looked at in Chapter 1.

Data Sizes

In Chapter 1 we saw how the basic unit of data is the binary byte and how data can also be represented in word and long word lengths. By way of revision, the following table lists the primary data lengths which are commonly used:

Length	*Name*	*Comment*
1 bit	Bit	The primary unit of binary numbering.
4 bits	Nibble	Used in BCD arithmetic to represent decimal digits 0 to 9.
8 bits	Byte	The primary unit of data storage.
16 bits	Word	Two bytes.
24 bits	—	The number of bits used to represent a memory address value.
32 bits	Long word	Four bytes.

Although these are standard, named data lengths there is no reason why you should not store data in any size or format you wish, according to the requirements of your application. You might choose, for example, to represent floating point numbers in 6 bytes: 2 for the mantissa (the whole part) and 4 for the exponent (the fractional part), with one or more bits reserved to indicate the number of decimal places required when the value is displayed on the screen. In this case you would need to write routines which encode and decode the data in accordance with the format you have chosen.

Hexadecimal Numbering

So far we have been using the decimal and binary numbering systems: the decimal system because we are familiar with it and the binary system because it constitutes the actual representation of numeric values in a computer system. Another numbering system is also often used in computing: the hexadecimal, or base-sixteen system. Its advantage over the binary system is that it is a convenient way of representing binary values without needing to go to all the trouble of writing down a large number of 0s and 1s. Its advantage over the decimal system is that every byte of data can be uniformly represented by two hex digits, thus it is easy to arrange data in a regular tabular format and to identify byte-sized units of data in a program listing.

There are 16 hexadecimal digits: 0 to 9, equivalent to the decimal values 0 to 9, and A to F, equivalent to the decimal numbers 10 to 15 as shown in the table on the next page:

Decimal	Hex	Decimal	Hex
0	00	16	10
1	01	17	11
2	02	18	12
3	03	19	13
4	04	20	14
5	05	21	15
6	06	22	16
7	07	23	17
8	08	24	18
9	09	25	19
10	0A	26	1A
11	0B	27	1B
12	0C	28	1C
13	0D	29	1D
14	0E	30	1E
15	0F	31	1F

If we look at a 16-bit binary number you will be able to see easily how each hexadecimal digit corresponds to a 4-bit section of it:

```
0000 1010 1100 1001 = 2761 decimal ((10 * 256) + 201)
 0    A    C    9   = 0AC9HEX
```

Each separate 4-bit section is read as if it were the low order four digits of a binary number, so that each will represent a value on the range 0–15 decimal (0–F hexadecimal).

For conversions between binary, hex and decimal values, you may find the conversion table in Appendix C useful. All three numbering systems can be used in programming and in an assembler listing you can usually specify the *radix* (the numbering system) of the data as being hexadecimal simply by appending a '$' to it. To move the immediate hexadecimal value 3F into register D4 for example, you might use an instruction such as `MOVE.B #$3F,D4` – the # indicating that it is an immediate value and the $ indicating the radix. To move a value contained in an absolute address, such as $2AFB_{HEX}$, into D4, then you would use an instruction such as `MOVE.B $2AFB,D4`.

Assembler Programs

In Chapter 1 we briefly looked at assembler programs, which enable the programmer to enter the assembly language source program as a listing and from which the object, or machine code version of the finished program is compiled.

In this chapter we shall be going into the operation of assembler programs in more detail. Taking the simple program example from chapter 1 we shall produce a complete source code listing and then examine the object code which is produced from it.

Assembler Structure

An assembly language 'source' program consists of a listing of a number of separately identifiable blocks of commands which define the sections of a program in a structured fashion. The general structure is not imposed and therefore it is up to the programmer to arrange the sections of the program in an orderly and consistent sequence so that it is easy to refer to it during the debugging phase. Typically, the listing will contain definitions of all labelled constant values, the labelled addresses of data items and reserved memory blocks and labelled blocks of code, including any subroutines.

The listing consists of four main elements:

1 Assembler directives, or 'pseudo-ops', which are commands which are part of the assembler program rather than actual assembly language instructions.

2 Labels – which are user-defined names which the programmer gives to the various elements of the program.

3 Comments – which serve the same function as `REM` statements in BASIC.

4 Assembly language instructions – which are the actual 68000 program commands.

Since different assembler programs vary in their formats and in the facilities which they provide, we shall not be going into a full examination of the operation of any particular package. Readers should therefore refer to the documentation provided with their own

assembler, which will normally contain extensive and detailed instructions on how they should be used.

In this book the assembler formats and directives used to illustrate the various example programs are typical of those used in most assemblers and each feature will be fully explained wherever it is introduced. By the end of this chapter the general structure of an assembler listing will be reasonably well understood and you should have no difficulty in following all the program examples listed in the chapters which follow.

The first program, like all the others in this book, will be relocatable (i.e. position independent). It is good practice to make programs relocatable as a matter of course because you cannot always anticipate the circumstances under which they will be executed. If you are only likely to run one program at a time on your computer then it does not matter much – your relocatable program will always be loaded into the same memory area. If you write a non-relocatable program then you are free to refer to absolute addresses. For example, it would be permissible to use an instruction such as `MOVE.B 80000,D2` (load the contents of address 80000 into D2.

A relocatable program would have to use a label to achieve the same ends because the data to which you are referring is not always likely to be at address 80000. As you will see in the example program, these labels are assigned to code and data sections in the listing and the assembler program calculates their relative offset values during assembly. In practice, it is common to use labels in non-relocatable programs as well so that, for example, at the beginning of your source listing you would have a statement such as `TABLE_1 EQU £80000`. Subsequently you can refer to this address using its label.

A non-relocatable program is defined by using the assembler directive `ORG` at the beginning of the program, followed by the address at which you want your program to be loaded.

Separate `ORG` statements may be used to define the beginning of your code section, the beginning of your data section and also the base address of the stack if the default stack is not large enough and you wish to assign more space for it.

A relocatable program may be defined with a `RORG` directive and the system will subsequently work out where the program is to be loaded.

In many cases relocatable assembly will be the default condition and therefore RORG will be unnecessary.

A program may be executed in a variety of ways: it may be called as a subroutine from another program, from the operating system prompt or from a high level language such as BASIC. The precise method of calling a program will differ between operating systems but in most cases you will find that various parameters have to be passed to the operating system at the beginning of a program in order to ensure that the correct channels are opened (i.e. to the screen, a screen window, keyboard, printer etc) and that the program is correctly integrated with any other 'tasks' which the computer is running. This information may include a priority code, indicating the level of priority which the current task has in relation to the other tasks. The example programs will incorporate some of these parameters in order to demonstrate the kinds of operations which your operating system may require you to perform. Since these are operating system specific, they may be coded completely differently on your own computer but bear in mind that their inclusion is normally necessary. The technical manual for your machine should provide you with the information required by your own operating system.

Example Program 1

Our first example program will be documented in detail and we shall go through all the stages of assembly from source code to object code, so that you can see how the different elements of the program and the assembler relate to each other.

As a reminder, here is the program once again in BASIC:

```
10 FOR count = 5 to 1 STEP -1
20 READ V
30 PRINT CHR$(48 + V)
40 NEXT count
50 DATA 24,21,28,28,31
```

As you will recall, this merely adds 48 to each of the data items and prints the ASCII characters corresponding to the results to the screen. In the following assembler version, all user-defined labels are printed

in italics, assembler directives are printed in normal type and the assembly language instructions are printed in bold type. Comments are preceded by the ';' symbol. This format will be used throughout the remainder of the book. After the listing a full commentary on the program will be given. At the end of the chapter, a literal translation of each instruction mnemonic is also given.

> *Note:* On a first reading of this listing the main things you should be observing are the assembly format and the structure of the program as a whole. Although the workings of the actual assembly language program are explained there may be much that will appear confusing at this stage but the details will become clearer as you read through later chapters.

```
;-------------------------------------------------------
;            DATA ADDITION PROGRAM ENTITLED PROG1
;            ADDS 48 TO A SET OF VALUES AND PRINTS
;             THE CORRESPONDING ASCII CHARACTERS
;-------------------------------------------------------
;FIRST THE PROGRAM CODE SECTION BEGINS
;-------------------------------------------------------
;OPEN DEVICE CHANNELS, JOB ID ETC.
;-------------------------------------------------------
GO      MOVEQ   #0,D1           ;Job ID
        MOVEQ   #2,D3           ;Exclusive device
        LEA.L   DEVICE,A0       ;Address of device code
        MOVEQ   #1,D0           ;Code for opening channel
        TRAP    #2              ;Trap for opening channel
;-------------------------------------------------------
;THEN THE PROGRAM REGISTERS ARE INITIALIZED
;-------------------------------------------------------

        MOVEQ   #0,D2           ;D2 will index data
        MOVEQ   FOUR,D4         ;D4 will count off data
        LEA.L   MYDATA,A2       ;A2 points to base of
                   'MYDATA'
;-------------------------------------------------------
;THEN THE ADDITION IS PERFORMED
;-------------------------------------------------------
LOOP1   MOVE.B  0(A2,D2),D1     ;Move an item of data to D1
        ADD.B   ADVAL,D1        ;Add 48 to it
```

```
;-------------------------------------------------------
;THEN AN OPERATING SYSTEM TRAP IS CALLED TO PRINT
;THE RESULT TO THE SCREEN
;-------------------------------------------------------

        MOVEQ   #-1,D3          ;Timeout code in D3
        MOVEQ   #5,D0           ;Transfer display
                                ;function code into D0
        TRAP    #3              ;Call operating system
                                ;display function trap
;-------------------------------------------------------
;THEN UPDATE INDEX AND COUNTER
;-------------------------------------------------------

        ADDQ    #1,D2           ;Increment index pointer
        DBRA    D4,LOOP1        ;Loop back to LOOP1 if
                                ;D4 is > -1
;-------------------------------------------------------
;IF ALL ADDITION COMPLETED (D4 = -1) THEN TERMINATE PROGRAM
;-------------------------------------------------------

        MOVEQ   #2,D0           ;'close channel' code
        TRAP    #2              ;Close channel
        MOVEQ   #-1,D1          ;Job ID
        MOVEQ   #0,D3           ;Error code
        MOVEQ   #5,D0           ;'remove task' code
        TRAP    #1              ;Remove task
;-------------------------------------------------------
;THEN THE DATA IS DEFINED
;-------------------------------------------------------
MYDATA  DC.B  24,21,28,28,31    ;Defines and names the set
                                ;of data values in 5
                                ;reserved bytes
ADVAL   DC.B  48                ;Defines and names the
                                ;addition value in 1
                                ;reserved byte
FOUR    DC.B  4                 ;Defines and names length of
                                ;MYDATA, less 1, in 1
                                ;reserved byte
DEVICE  DC.W  4                 ;Number of characters in
                                ;device name in one reserved
                                ;word

        DC.B  'CON_'            ;Device name
```

```
;-------------------------------------------------------
;AND THE PROGRAM IS TERMINATED
;-------------------------------------------------------

     END
;-------------------------------------------------------
```

The 'rem' statements (;) at the start of the program simply name it and define what it is intended to do.

Following this, the actual program begins and the first 68000 instruction is given the label 'GO'. This arbitrary name is merely to indicate where the actual program starts and is not obligatory.

The first instruction, `MOVEQ #0,D1`, sets the whole of D1 to zero. This is a parameter which will instruct the operating system to assign an ID code to the program. The next instruction, `MOVEQ #2,D3` sets D3 to the value 2. This is a parameter which will inform the operating system that an exclusive device will be required by the program. The instruction `LEA.L DEVICE,A0`, loads the address containing the length (4) of the device specification (`CON_`) into the A0 register. `CON_` stands for 'console' and indicates that communication channels to the screen and keyboard should be opened. The label `DEVICE` is defined in the data section at the end of the program. The `MOVEQ #1,D0` instruction moves the value 1 into D1. This is a parameter to inform the operating system to schedule the current program as a new 'task' to be performed.

Then the `TRAP #2` instruction calls a specific operating system trap routine, passing the above parameters to it. The trap routine interprets the parameters and performs the requested operations before returning to the program.

Note that all the above operations relate to an operating system specific procedure and are likely to be different on your own computer.

Next the program proper begins. The instruction `MOVEQ #0,D2` sets the whole 32 bits of the D2 register to zero. This register will be used as an index offset to access the data in the data section and initially the offset will be zero.

The D4 register will be used to count off the five bytes of data and so it is initialized with the count value (5) less 1: `MOVEQ FOUR, D4`.

The assembler will recognize the label FOUR' as being the name of the data address in which the actual value 4 has been stored and during execution it will be transferred from there into D4.

Again, FOUR is an arbitrarily chosen label which is defined in the data section at the end of the program. It is *the value contained in* address FOUR which is loaded into D4 rather than the data address number itself.

The instruction LEA.L MYDATA,A2 then loads the base address of the data labelled 'MYDATA' into address register A2. MYDATA is the data to which we shall be adding 48 and is defined at the end of the program.

The label LOOP1 marks the address to which program execution will loop back after each addition and printing operation has been completed.

The instruction MOVE.B 0(A2,D2),D1 moves the data stored at the address named 'MYDATA' (pointed to by A2) plus the index offset in D2 (initially 0) into D1. The instruction ADD.B ADVAL,D1 then adds 48 (the number contained in address ADVAL) to the data contained in D1 so that D1 contains the result of the operation. ADVAL is defined at the end of the program.

The following three instructions cause the program to branch to an operating system trap procedure which prints the ASCII code of the value contained in D1 to the screen. Different computers will have different ways of doing this and you should refer to the documentation for your own machine for details. In effect, three parameters: the contents of D1, the value –1 and the value 5 are passed to an operating system trap procedure which performs the printing operation and returns control back to the program. D1 already holds the ASCII code for the character to be printed, –1 is a timeout parameter, indicating how long (if at all) the routine should wait to output its information to the screen if the console channel is being used by some other program, and 5 is a parameter indicating that a byte of data is to be sent to the screen.

The D2 index register is then incremented by 1 to point to the next data item (ADDQ #1,D2) and the DBRA D4,LOOP1 command automatically decrements the D4 register, containing the loop count, and if the operations have not been completed (i.e. if D4 >= 0) then

execution is looped back to the address which has been given the label 'LOOP1'.

Finally, another operating system trap is called which closes the console channel. The parameter 2 specifies this function and trap 2 performs it. Then another trap is called which tells the operating system that the current task is finished and may be removed from the execution schedule. The parameter –1 refers to the current task ID code, 0 refers to an error code and 5 is a task termination code. TRAP 1 performs these operations and returns control to the operating system, to any program from which the current program was initiated or to some other task which is waiting in the schedule queue.

Following this, the data segment of the program is defined. This section will eventually follow the program code when the assembled program is loaded and run. The fact that the labels associated with the data have already been referred to in the body of the program, prior to their definition, does not matter. The assembler will run through the source code twice. On the first pass it will take note of all labels used and on the second pass it will replace all label references with appropriate values.

MYDATA is an arbitrarily named label which we choose to assign to the five bytes of data which will be used in the program. When the program is assembled, the label MYDATA becomes a numeric variable pointing to the address of the first of the five data items. The directive 'DC.B' means 'define byte constant'; in other words, the assembler is told to reserve five byte-sized memory spaces for the following items of data and to insert the data in the reserved addresses.

The same is then done for ADVAL, which is the name we choose to give to the constant value 48 which will eventually be added to each of the data items. Then we give the name FOUR to the constant value 4, which will be used to count off each item of data after it has been added.

Finally, the value 4, labelled 'DEVICE', is stored as a word using DC.W (define word constant). This represents the number of characters in the following data item, 'CON_'. These two items of data are used at the beginning of the program to set up a communications channel for the console. Note that 'CON_' is defined as bytes (DC.B). This means that the ASCII code of each individual character in the word 'CON_' should be stored as separate bytes in four consecutive addresses.

The program is then terminated with the statement 'END', indicating to the assembler that there is no more code or data to assemble.

If we were to type this listing into our computer, using a word processor or screen editor utility, it can then be read and converted into object code using an assembler program. The assembler will use the various elements of the program and calculate the relative positions in memory between all the labelled elements of the program. If there are any errors, these will be printed to the screen with appropriate error messages so that you can amend the listing if necessary and re-assemble it.

The resulting object code, which is written to disc, is a relocatable machine code file with all the assembler directives and labels stripped from it. The assembler assembles the code in such a way that when the program is loaded, the operating system is free to assign the code, stack and data sections to whatever free areas of memory are available and in different circumstances, for example where memory space is being shared with other programs, the assigned base address of the program will vary. However, the relationships between the various data blocks and code sequences of a program will remain constant because the assembler translates all labelled points in a program into relative offsets rather than fixed locations. In some cases these offsets are located relative to the assigned base address of the program and in some cases, for example in program loops, the offsets are located relative to the branch instructions.

Linking Program Segments

A program which we have assembled into object code may be just one of several modules belonging to a larger program, or may be a program which is intended to interact with other programs which may share memory space with it at the time it is run. For this reason, it may be necessary in some cases to run the assembled program through a LINK program, which gathers together all associated modules and segments and arranges them in an efficient structure, ensuring that all code and data shared by more than one program module are properly linked. The final linked code is usually termed an 'executable' object file: that is, it is finally in a condition in which it can be loaded and executed.

If we pass our source listing through the assembly and link processes, the final machine code file can then be read by a 'disassembler', 'monitor' or 'debug' program, which gives a complete listing of the assembly language mnemonics, a hexadecimal 'dump' of the actual object code for each instruction and the addresses into which each instruction has been loaded, as follows:

```
;-------------------------------------------------------
;Address Object Code      68000 Instruction
;       (hexadecimal)     Mnemonics
;-------------------------------------------------------
29CE8    7200             MOVEQ  #00, D1
29CEA    7602             MOVEQ  #02, D3
29CEC    41FA0038         LEA    38(PC)!29D26, A0
29CF0    7001             MOVEQ  #01, D0
29CF2    4E42             TRAP   #2
29CF4    7400             MOVEQ  #00, D2
29CF6    183A002C         MOVE.B 2C(PC)!29D24, D4
29CFA    45FA0022         LEA    22(PC)!29D1E, A2
29CFE    12322000         MOVE.B 00(A2,D2.L), D1
29D02    D23A001F         ADD.B  1F(PC)!29D23, D1
29D06    76FF             MOVEQ  #FF, D3
29D08    7005             MOVEQ  #05, D0
29D0A    4E43             TRAP   #3
29D0C    5242             ADDQ   #1, D2
29D0E    51CCFFEE         DBRA   D4,29CFE
29D12    7002             MOVEQ  #02, D0
29D14    4E42             TRAP   #2
29D16    72FF             MOVEQ  #FF, D1
29D18    7600             MOVEQ  #00, D3
29D1A    7005             MOVEQ  #05, D0
29D1C    4E41             TRAP   #1
;-------------------------------------------------------
```

From this you can see that the program code has been loaded into an area of memory beginning at address $29CE8 (171240 decimal). You will recall from Chapter 1 that code must always be loaded at an even address so that it can be 'word aligned'. The first instruction has been located in memory at address $29CE8. The object code is listed in hexadecimal and if you examine the listing you will see that each pair of hex digits represent 1 memory byte. The second instruction, for example, is located at $29CEA (decimal 171242) and its object

code occupies 2 bytes; therefore, the third instruction starts two bytes further on at $29CEC (decimal 171244).

If you look at the mnemonics in this listing carefully and compare them with those in the original source file, you will notice that there are a number of differences. The third instruction for example, `LEA.L 38(PC)!29D26,A0`, was originally `LEA.L DEVICE,A0`. The reason for this is that the actual physical address of the data labelled '`DEVICE`' is $38 bytes further on in memory, *relative to the value which will be held in the PC register at the time this instruction is executed.* The assembler has noted the fact that this program is relocatable and has specified the location of '`DEVICE`' to be PC relative rather than absolute. The monitor program which loaded the program and which produced the above assembly listing has introduced a separation symbol: '!' following which it has inserted the actual address at which '`DEVICE`' has been located: $29D26. Had the entire program been loaded at a different address then '`DEVICE`' would still be located at an offset of $38 bytes relative to the third instruction but of course its actual physical address would be different. The seventh physical instruction, `MOVE.B 2C(PC)!29D24,D4` at address $29CF6 was originally `MOVE.B FOUR,D4` and has been interpreted similarly, as has `LEA.L 22(PC)!29D1E,A2` (originally `LEA.L MYDATA,A2`) at address $29CFA. When the program is executed the data contained in the address labelled '`FOUR`', in this case the data contained in address $29D24, is moved into low order byte of register D4. The actual address of '`MYDATA`' (in this case $29D1E) is loaded into address register A2.

Note that the instruction at address $29D06, `MOVEQ #FF,D3` was originally `MOVEQ -1,D3`. FF is of course the hexadecimal 2's complement representation of –1.

The `DBRA D4,29CFE` instruction at address $29D0E was originally `DBRA D4,LOOP1`. The monitor program has simply substituted the physical address of the instruction labelled `LOOP1` for the label.

Often a number of small details will be altered automatically when a source program is assembled. One example in this case can be seen in the instruction at address $29CFE: the index register D2 has acquired an '`.L`' size specifier, indicating that the entire 32 bits of D2 are used as the index value. In the source listing no size specification was given but its inclusion in the object listing can be useful since it can help to identify possible errors resulting from the use of an incorrect data size. In this particular case it does not matter whether

16 or 32 bits of D2 are used because the index value will never be greater than 16 bits and the earlier `MOVEQ` instruction would have set the hi word of D2 to zero.

In addition to the executable object file, the assembler may also have generated a 'list' file, containing a copy of the original source code with line numbers and error codes added to aid the debugging process. It may also have generated a 'symbol table' which gives information about all the labels contained in your program, indicating whether they relate to code or data addresses and, if they are data labels, their defined size and the initial values contained in them. The table would normally indicate the total size of the program and the sizes of the code and data sections.

Tracing a Program

Once our program has been assembled and listed we can test it by executing it one instruction at a time using a toolkit program which 'traces' the status of the registers as each instruction is executed. A trace listing for the second program instruction, `MOVEQ #2,D3`, appears as follows. This shows the status of the registers *immediately before* the instruction is executed:

```
29CEA 7602                    MOVEQ  #02, D3
D0=0      D1=0      D2=0      D3=0      D4=0      D5=0      D6=0
D7=0      A0=0      A1=0      A2=0      A3=0      A4=0      A5=0
A6=0
A7=3DBC6        Status= Z T Imask=0 Program Counter =29CEA
```

From this you can see that the top of the stack, whose address is in register A7, is located at $3DBC6 and the instruction, `MOVEQ £02,D3` is located at address $29CEA as indicated on the top line of the first trace. Since the program counter is pointing to this instruction it also contains the same value. The 'T' after the word 'Status' indicates that the trace flag is set in the status register because the program which prints out the listing is making use of the processor's trace facility. The Z flag is set because the previous instruction moved an 0 into register D1. The IMASK (interrupt mask) in the status register is at zero, indicating that interrupts have not been masked.

```
29CEC 41FA0038                LEA   38(PC)!29D26, A0
D0=0     D1=0     D2=0     D3=2     D4=0     D5=0     D6=0
D7=0     A0=0     A1=0     A2=0     A3=0     A4=0     A5=0
A6=0
A7=3DBC6      Status= T Imask=0 Program Counter =29CEC
```

The second trace shows what happens when the MOVEQ instruction is executed. Register D3 now contains the value 2 and, since the previous instruction occupied two bytes of memory, the program counter has been incremented by 2 to point to address $29CEC, ready for the LEA instruction to be executed.

```
29CF0 7001                    MOVEQ  #01, D0
D0=0     D1=0     D2=0     D3=2     D4=0     D5=0     D6=0
D7=0     A0=29D26 A1=0     A2=0     A3=0     A4=0     A5=0
A6=0     A7=3DBC6      Status= T Imask=0 Program Counter =29CF0

29CF2 4E42                    TRAP   #2
D0=1     D1=0     D2=0     D3=2     D4=0     D5=0     D6=0
D7=0     A0=29D26 A1=0     A2=0     A3=0     A4=0     A5=0
A6=0     A7=3DBC6      Status= T Imask=0 Program Counter =29CF2
```

The third trace shows what happens after the LEA.L 38(PC)!29D26,A0 instruction has been executed. The value of the address $38 bytes relative to the PC counter, address $29D26, has been loaded into register A0 and the program counter has been incremented by 4 bytes to $29CF0.

Data Dumps

A further debugging aid is normally provided in a monitor program: a hexadecimal 'dump' of a program and its data can be obtained, together with a listing of the printable ASCII codes which correspond to the data contained in the dumped addresses. This gives a broad overview of the space occupied by your program and can be used to check the operation of individual instructions against the data addresses to which they refer. For example, if your program is designed to load a string of calculated values into a certain block of memory, you can run your program through once under control of the monitor program and then obtain a dump of the data area concerned to check that your program is inserting the correct values.

The following listing shows a dump for PROG1:

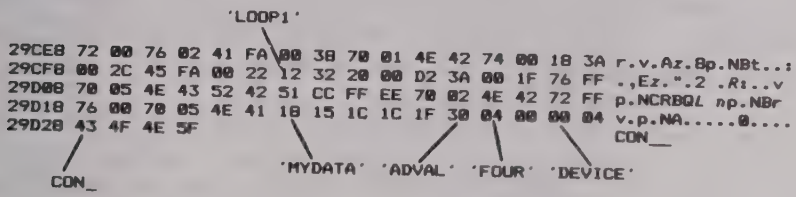


FIGURE 8-1.

The four digit numbers in the left hand column are memory addresses, starting with the first address of the program at $29CE8. The centre block shows the hexadecimal contents of the individual bytes of the program, arranged in rows of 16 bytes. The block on the right shows the printable ASCII codes corresponding to the values contained in each of the 16 rows. Where a code has no printable ASCII character associated with it, it is represented by a dot. Obviously the program code itself results in a meaningless jumble of character codes, as does most of the data section except for the codes representing the name 'CON_'. With some programs, where the data section contains lines of text, the words of the text will appear in the ASCII block and this can be useful for checking and correcting textual data. In PROG1 there are no texts in the original data and so the ASCII block is largely irrelevant.

Note that the data section of the program follows on immediately from the end of the code, exactly as it was positioned in the original source program. Had we chosen to define our data *before* the code in the source program, which is the normal practice with some assembly languages, we might have ended up with an odd number of data bytes which would cause code alignment problems, with the code starting illegally at an uneven address. For this reason, 68000 program data is conventionally placed after the code.

Executing a Machine Code Program

The program may be executed in a number of different ways. It can be loaded and run by keying in its disc file name from the operating system prompt, from within a monitor program, from within another machine code program or from within a high level language program.

For example, to execute it from the operating system we would simply begin with the operating system screen prompt, type in the name under which the program has been filed on disc and then press return. The program would automatically load and run, displaying the word 'HELLO' on the screen.

In comparison with the BASIC version of the program, there seems to be a very great deal of code involved for such a modest result. If you go back and compare the source listing with the object code listing you will see that some of the original material had little to do with the actual program at all, consisting mostly of assembly program labels and directives rather than actual assembly language. The source listing is extensively padded out with short explanatory notes, as all programs should be. When programs need to be debugged or altered and combined with other programs they can be very difficult to follow unless the listing has been carefully documented in this way.

If you look at the object listing you will see that much of it is concerned with setting up and closing channels and so on, which in many cases would not be necessary. The heart of the program, starting with the instruction `MOVEQ #0,D2` and ending with `DBRA D4,LOOP1` consists of only 10 assembly language mnemonic instructions which are assembled into 30 bytes of machine code. Obviously in a larger program the proportion of 'real' program code in relation to all the formal definition statements and rem statements would be much greater and if you build up a disc library of commonly used assembler subroutines, the amount of physical work involved in keying in an assembly language program can compare favourably with that required for a BASIC or other high level language program.

The following list is a brief reference key to the meanings and functions of the assembly language mnemonics used in the above program:

Mnemonic	*Meaning*
`MOVE #0,D1`	`MOVE` Quick the immediate value 0 into register D1, resetting the entire 32 bit of the register to zero.
`MOVEQ #2,D3`	`MOVE` Quick the immediate value 2 into register D3.

Mnemonic	*Meaning*
The low order byte of D3 will contain the 2 and the remaining 3 bytes will be zeroed.	
`LEA.L DEVICE,A0`	Load the Effective Address which is labelled '`DEVICE`' into the whole of register A0.
`MOVEQ #1,D0`	`MOVE` Quick the immediate value 1 into register D0.
`TRAP #2`	Call operating system trap number 2.
`MOVEQ #0,D2`	`MOVE` Quick the immediate value 0 into register D2.
`MOVE.B FOUR,D4`	`MOVE` the Byte of data contained in the address labelled '`FOUR`' into the low byte of register D4, leaving the three higher bytes of D4 unchanged.
`LEA.L MYDATA,A2`	Load the Effective Address which is labelled '`MYDATA`' into the whole of register A2.
`MOVE.B 0(A2,D2),D1`	`MOVE` the Byte of data at the address which is the sum of (0+reg A2 contents+reg D2 contents) into the low byte of register D1.
`ADD.B ADVAL,D1`	`ADD` the Byte contained in the address labelled '`ADVAL`' to the contents of the low byte of D1.
`MOVEQ #-1,D3`	`MOVE` Quick the immediate value minus 1 into register D3.
`MOVEQ #5,D0`	`MOVE` Quick the immediate value 5 into register D0.
`TRAP #3`	Call operating system trap number 3.
`ADDQ #1,D2`	`ADD` Quick the immediate value 1 to register D2.
`DBRA D4,LOOP1`	Decrement register D4 by 1 and `BRA`nch to the address labelled '`LOOP1`' if D4 is greater than –1.

Mnemonic	*Meaning*
`MOVEQ #2,D0`	`MOVE` Quick the immediate value 2 into register D0
`TRAP #2`	Call operating system trap number 2.
`MOVEQ #-1,D1`	`MOVE` Quick the immediate value minus 1 into register D1.
`MOVEQ #0,D3`	`MOVE` Quick the immediate value 0 into register D3.
`MOVEQ #5,D0`	`MOVE` Quick the immediate value 5 into register D0.
`TRAP #1`	Call operating system trap number 1.

Chapter 9

Addressing Modes

In Chapter 2 we looked at some of the different types of registers used in the 68000 and examined the addressing modes used for addressing data located in both registers and in memory.

In this chapter we shall be looking at the registers in greater detail and then the addressing modes illustrated in Chapter 2 will be summarized, along with some sample instructions. Finally, we shall look at another complete program which illustrates some of these modes in a practical context.

Register Model

Figure 9.1 shows all the 68000 registers, arranged in their different categories, as follows:

1 **Data Registers**
The eight 32-bit registers which are used for holding program data.

2 **Address Registers**
The seven 32-bit registers which are used to hold memory addresses for accessing data which is located in memory.

3 **Stack Pointer**
Used to point to the 'top' of the stack. The stack pointer is register A7.

4 **Program Counter**
The PC register holds the address of the instruction currently being executed.

5 **Status Register**
The SR register contains bit flags indicating the current status of the system.

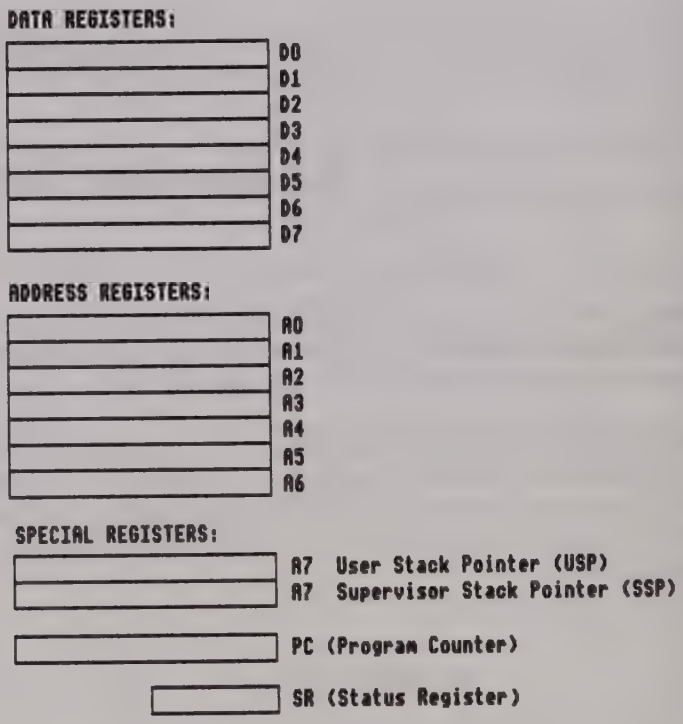


FIGURE 9-1.

Register Descriptions

Registers: D0 D1 D2 D3 D4 D5 D6 D7

The eight 32-bit registers in this group are mainly used to hold data for transfer, for temporary storage and for arithmetic and logical operations. The main features of data registers are as follows:

1 Data may be copied into or out of data registers in byte, word or long word lengths. When byte or word data is copied into a data register it is copied into the low order byte or word of the register leaving the higher order 24 or 16 bits unaltered.

2 Data registers may be used as sources or destinations in program instructions.

3 Data registers may be used as counter registers, containing, for example, count values for loop operations. The DBRA (decrement and branch) and DBcc (decrement and branch according to condition code) instructions decrement the values of nominated data registers automatically.

4 A data register may be used as an index register, containing an offset value which is added to the value of an address register to obtain the full effective memory address of an operand.

Address Registers: A0 A1 A2 A3 A4 A5 A6

Address registers are used to contain the addresses of code or data contained in memory. They may also be used to contain data, providing that it is held in the form of word or long word sized values. Their principal features are as follows:

1 When a memory address is contained in an address register, only the low order 24 bits of the value are used to specify the address. The remaining high order byte, if any, is ignored.

2 Address registers may only contain word or long word sized values. When a word sized value is loaded into an address register its most significant bit (the sign bit) is copied (sign extended) into the 16 high order bit positions of the register. Thus 0000000000001010 (binary) loaded into an address register will automatically become:

00000000000000000000000000001010

1000000000001010 (binary) will become:

11111111111111111000000000001010

Thus a 16-bit address value with its sign bit zeroed will represent an address within the bottom 32K of memory. A 16-bit address with its sign bit set will represent an address in the top 32K of memory. A 32-bit address value loaded into an address register (of which only 24 bits are significant) may represent any address within a 16 megabyte range.

3 Address registers may be used as both sources and destinations in program instructions. When an address register is a *destination* operand for certain instructions such as MOVEA, ADDA and SUBA, none of the flags are altered by the operation.

4 An address register may be used as an index register, containing an offset value which is added to the value of another address register to obtain the full effective memory address of an operand.

Special Registers: SP PC SR

The registers described above are the ones which are most commonly used in programming and which are referred to by name in source listings. The three special registers described below may also be specified in certain instructions but for the most part they are used implicitly, which means they are used automatically in some operations.

Program Counter: PC Register

As explained in Chapter 2, the program counter is used by the system to point to the address of the instruction which is currently being executed. In the last chapter we saw this in action in the trace listing, where the PC register was pointing to the address in the code segment at which the traced instruction was located. The PC register is incremented automatically by the system as each byte of code is accessed and executed. The address of the instruction currently being executed, or more accurately the address of whichever byte of the instruction is currently being executed, is thus always contained in PC. The location of an address which is to be referenced by the instruction may be addressed as being relative to the value of PC, as illustrated in the program in the previous chapter.

A branch to a different point in the program involves the alteration of the PC register to point to the new execution address. In the previous program for example, the address of the instruction `DBRA D4,29CFE` was $29DOE. By the time this four byte instruction has been decoded by the processor, PC is pointing to $29D12 (the address following the end of the instruction). If the branch to $29CFE is made then this becomes the new value of the PC register, otherwise execution continues from $29D12.

Status Register: SR

The status register consists of two bytes: the CCR (condition codes register) and the system byte. The CCR contains the bit flags which provide information about the result of the operation instruction: X (extend), N (negative), Z (zero), V (overflow) and C (carry). The system byte contains bit flags indicating the current status of the system: T (trace flag), S (supervisor bit) and I (3 maskable interrupt bits). These flags will be discussed in detail in the next chapter.

Stack Pointer: SP Register

The top of the stack; the location at which data can be added to or removed from the stack, is contained in the SP register which is address register A7. When the system is operating in user mode, A7 is referred to as the USP (user stack pointer) and in supervisor mode, A7 is referred to as the SSP (supervisor stack pointer). In either case the register is normally referred to in program instructions simply as A7 and no distinction is made. However, there is a privileged instruction, `MOVE USP`, which specifies the stack pointer directly and can only be used in supervisor mode to move an address value to or from register A7. This is used to set an initial value for the stack pointer or to store it temporarily elsewhere when a new stack is being set up.

Addressing Modes

In Chapter 2 an overview of the addressing modes was given. It was explained that an operand may be addressed implicitly, where it is contained in a register which is automatically used by a specific program instruction; it may be in a register (register addressing); it may be an immediate data value (immediate addressing); it may be in a specified memory address (absolute addressing); it might be in an address pointed to by one or more of the registers (indirect addressing) or it might be positioned relative to the program counter (PC relative addressing).

The following summary of each of these modes shows how the location of an operand is determined in each case. The actual physical location of an operand is called the *effective address*, and in the following examples the elements required for the calculation of an effective address are shown where relevant. If the operations are not entirely clear, it may help to refer back to Chapter 2 where the addressing modes are shown in diagrammatic form.

Implicit addressing

The operands are implicit in the instruction. The instruction `RTS` (return from subroutine) for example, implicitly refers to the PC and SP registers. The return address is automatically taken from the top of the stack, which is pointed to by SP, and loaded into PC. SP is automatically incremented.

Register Direct Addressing

This mode involves operands which are contained in data and/or address registers. Data registers may have byte, word or long word values loaded into or copied from them and address registers are confined to word and long word sized values. When an address register is the destination of a word operand in a register addressing operation then the value contained in the register is sign extended to 32 bits. Instructions which specify an address register destination usually have an 'A' suffix: for example `MOVEA`, `ADDA`, `SUBA` etc. These are essentially the same as the `MOVE`, `ADD` and `SUB` instructions but in their 'A' format they serve as a reminder that with some instructions none of the flags will be altered by the operation.

Examples:

`MOVE.B D2,D3`	Byte transfer operation
`MOVEA.W D3,A5`	Word transfer operation
`MOVEA.L A4,A5`	Long word transfer operation
`ADD.W A2,D6`	Word addition operation
`ADDA.L D5,A6`	Long word addition operation

Absolute Addressing

In this mode an actual address value is given for the operand and is included in the instruction itself. The number is expressed as a label representing the address containing the required operand or as an actual address number. The calculation of the effective address, in hexadecimal notation, is as follows:

Operation:

`MOVE DATA,D4`	Move the operand contained in the address indicated by the label '`DATA`' into register D4.

DATA	$2294C
EFFECTIVE ADDRESS	=$2294C = address of operand

Examples:

`MOVE.L D5,ANADDR`	Copy 4 bytes of data from D5 to the address labelled '`ANADDR`' and the three following addresses).

`MOVE.W ANADDR,D3`	Copy 2 bytes of data from the address labelled 'ANADDR' (and the following address) to D3.
`ADDA.L 80000,A2`	Add the contents of address 80000 to A2.

If the absolute address is in the top 32 or the bottom 32K of memory then it can be addressed using a *short* instruction. For example `MOVE $1000,D4`. The hexadecimal value $1000 is the low order 16 bits of the address which is sign extended to 32 bits to give the full address. The instruction `MOVE $20000,D4` contains a 'long' address; the hexadecimal value $20000 being interpreted as a 32 bit address, of which the lower 24 bits are relevant.

Immediate Addressing

Immediate addressing involves operands which are numeric constants and which are stored as part of the instruction rather than in the data section or in a register. Some instruction types have a special form of mnemonic for operations involving immediate operands, such as `ADDI`, `SUBI` and `CMPI`.

Examples:

`MOVE £408,D4`	Load the word value 408 into D4
`CMPI.L £22,D4`	Compare contents of D4 with 22
`MOVEQ £1,D6`	Move the value 1 into D6

Address Register Indirect Addressing

This is where the effective address is contained in one of the address registers.

Operation:

`MOVE.L (A3),D4`	Move the operand in the address pointed to by A3 into D4.

A3	$29D1E
EFFECTIVE ADDRESS	=$29D1E

= address of operand (the remaining 3 bytes of the long word operand are in $29D1F, $29D20 and $29D21)

Examples:

`ADD.W (A2),D4`	Add the contents of the address pointed to by A2 (and the following address) to the low word of D4.
`MOV.W D4,(A2)`	Copy contents of low order word of D4 into the address pointed to by A2 (plus the following address).
`CMP.W (A0),D4`	Compare word value in address pointed to by A0 (plus the following address) to the value of D4

Address Register Indirect with Predecrement

This mode is similar to the address register indirect mode except that the value of the address register is decremented by 1, 2 or 4 bytes prior to the operation, depending on whether a byte, word or long word operand is involved.

Operation:

`MOVE.W D4,-(A6)` Subtract 2 from register A6 and copy the low order word of register D4 into the address pointed to by the new value of A6 (and the following address)

A6	$29D1E
– 2	= $29D1C
EFFECTIVE ADDRESS	= $29D1C

= address of operand. The low order byte from D4 will be copied into $29D1D

Examples:

`MOVE.L D0,-(A5)`	Subtract 4 from A5 and copy the whole of D0 into the address pointed to by the new value of A5 and the 3 following addresses.
`ADD.B -(A3),D1`	Subtract 1 from A3 and add the value contained in the address pointed to by A3 into the low order byte of D1.

Address Register Indirect with Postincrement

This mode is similar to the address register indirect with predecrement mode except that the value of the address register is *incremented* by 1, 2 or 4 bytes *after* the operation, depending on whether a byte, word or long word operand is involved.

Operation:

`MOVE.W D4,(A6)+`	Copy the low order word of register D4 into the address pointed to by A6 (and the following address) then add 2 to register A6.
A6	$29D1E
EFFECTIVE ADDRESS	= $29D1E
	= address of operand. The low order byte from D4 will be copied into $29D1F. Then A6 = A6 + 2

Examples:

`MOVE.L D0,(A5)+`	Copy the whole of D0 into the address pointed to by A5 and the 3 following addresses. Then add 4 to A5.
`ADD.B (A3)+,D1`	Copy the value contained in the address pointed to by A3 into the low order byte of D1. Then add 1 to A3.

Address Register Indirect with Displacement

This is a form of indirect addressing in which an address register, used as a base, is combined with a displacement value to give the effective address of the operand.

Operation:

`MOV.B 12(A4),D4`	Move the byte operand in the address pointed to by the sum A4 plus 12 into the low order byte of D4.
A4	$29D1E
+ 12	$0C

EFFECTIVE ADDRESS	=$29D2A
	= address of operand

Examples:

`ADD.W 2(A6),D1`	Add the word contents of the address pointed to by A6+2 (and the following address) to the low order word of D1.
`MOV.B D3,6(A4)`	Copy the low order byte of D3 into the address pointed to by the sum of A4+6.

The displacement is limited to 16 bits and is automatically sign extended, giving a displacement offset value in the range plus or minus 32K. Displacements greater than or equal to $8000 are negative. The displacement constant may be labelled, so that an instruction such as `MOVE.B OFFSET(A6),D4` may be used. (Effective address is value of A6 plus the value of '`OFFSET`').

Address Register Indirect with Index and Displacement

In this form of indirect addressing a displacement constant is combined with an index register (any of the address or data registers) to give the effective address of the operand.

Operation:

`MOVE.B 6(A1,D2.L),D4`	Move the word operand in the address pointed to by A1 plus the constant value 6 plus the value of D2 into register D4.

A1	$29D1E
+ 6	$06
+ D2	$0A
EFFECTIVE ADDRESS	=$29D2E
	= address of operand

Examples:

`ADD.B 12(A0,A2.W),D5` Add the contents of the address pointed to by the sum of A0 plus the constant value 12 plus the value contained in the low order word of A2 to D5.

`MOVE.L D4,2(A4,D2.L)` Copy the entire contents of D4 into address pointed to by the sum of A4 plus constant value 2 plus value contained in D2. (Contents of the low order 24 bits of D4 go into the next three addresses).

In this addressing mode the displacement constant is always a byte value which is automatically sign extended, giving a displacement in the range plus or minus 127 bytes. The index register value may be of either word or long word size. If the index is a word value then it is automatically sign extended giving an index offset in the range plus or minus 32K.

Note that the index register is given a size indicator (`.W` or `.L`) of its own, in addition to the specification suffix for the operand size.

PC relative addressing

Program counter relative addressing is very similar to the address register indirect addressing modes except that the PC register is substituted for the address register in the instructions. PC relative addressing is normally used in the writing of position independent code such as the program in the previous chapter. In that program there was no need to specify the PC register directly since the assembler, being aware that position independent code was required, calculated all PC relative offsets automatically.

The following examples show typical PC relative instructions:

Program Counter Relative with Displacement

`ADD.W 2(PC),D1` Add the word contents of the address pointed to by the sum of PC+2 (and the following address) to the low order word of D1.

`MOV.B D3,6(PC)` Copy the low order byte of D3 into the address pointed to by the sum of PC+6.

Program Counter Relative with Index and Displacement

`ADD.B 12(PC,A2.W),D5` Add the contents of the address pointed to by the sum of PC+12 plus the value contained in the low order word of A2 to D5.

`MOVE.L D4,2(PC,D2.L)` Copy the entire contents of D4 into the address pointed to by the sum of PC+2+D2. (The contents of the low order 24 bits of D4 go into the three following addresses).

If PC relative addressing is used with labels, the assembler will automatically work out the relative displacement between the PC register and the address of the operand, as illustrated in the program object code listing in the previous chapter. If PC relative addressing is performed using constant values, care must be taken to ensure that the operand address is calculated as being relative to the value contained in PC at the *start* of the instruction. This is done by using a '`*`' symbol to force the adjustment. For example, `ADD.B *+8,D6` (add the byte contained in the address 8 bytes relative to the PC register into the low byte of register D6). This type of instruction is seldom used because the labelling of addresses is standard practice and saves a good deal of displacement calculation when using PC relative addressing.

Addressing Mode Classification

For each 68000 instruction, the addressing modes which can be used vary a great deal, both for the source and the destination operands. To simplify matters it is useful to classify the addressing modes according to their reference types so that the mode which can be used for any given type of instruction can be expressed as a simple code. These reference types are as follows:

Data — A data referencing addressing mode is one which addresses data contained either in data registers or in memory but not in address registers.

Memory — Memory referencing addressing modes are those which address operands contained in memory rather than in any kind of register.

Control — A control reference is one which is the destination of a jump or branch.

Alterable — Alterable references refer to those operands which are capable of being altered by an operation. This therefore excludes the immediate addressing mode. It also excludes PC relative addressing.

These classifications overlap so that it is possible, for example, to refer to an addressing mode as being 'control alterable' or 'data alterable'. The various combinations are codified as follows:

<ea>	Effective Address – any addressing mode can be used.
<aea)	Alterable Effective Address
<cea>	Control Effective Address
<dea>	Data Effective Address
<caea>	Control Alterable Effective Address
<daea)	Data Alterable Effective Address
<maea>	Memory Alterable Effective Address

The addressing modes themselves can be codified by using the following symbolic representations:

An	Any address register
Dn	Any data register
Rn	Any register
Ri	Any register being used as an index
d8	8-bit displacement constant
d16	16-bit displacement constant
<imm>	Immediate data
rl	Register list – as used with `MOVEM` instruction

From this we can construct a table showing the addressing modes along with their permissible reference types:

Mode	*Symbol*	*Data*	*Mem*	*Control*	*Alterable*
Data reg direct	Dn	X			X
Addr reg direct	An				X
Absolute	nnnnn	X	X	X	X
Immediate	<imm>	X	X		
Addr reg indirect	(An)	X	X	X	X
with predecrement	-(An)	X	X		X
with postincrement	(An)+	X	X		X
with displacement	d16(An)	X	X	X	X
with index	d8(An,Ri)	X	X	X	X
PC relative	d16(PC)	X		X	X
with index	d8(PC,Ri)	X	X	X	

From this we can construct a symbolic representation for any given instruction. For example, the two possible addressing modes for the `BTST` instruction (bit test) can be represented by `BTST Dn,<daea>` and `BTST #<imm>,<dea>`, meaning that the source operand can either be in a data register or immediate. The destination operand can only be a data such as `Dn`, `(An)`, `d16(An)`, `d8(An.Ri)`, `-(An)`, `(An)+` reference, or absolute. The `BSET` (bit test and set) instruction would actually alter the destination operand and therefore would be a data alterable reference <daea> which would exclude the PC relative modes. This form of representation is used in the 68000 instruction glossary in Appendix B.

Example Program 2

The following program illustrates some of the above addressing modes. The purpose of this program is to illustrate how to implement a memory buffer in assembly language and to use it to print a text to the screen. A buffer is simply a block of memory locations which is of a fixed length and which can be used to store a measured length of data before transferring it elsewhere. Buffers are frequently used with printers where data is passed, a fixed number of bytes at a time, to a print buffer. From there the buffered text is output to the printer and the next chunk of data is then loaded into the buffer and so on. The reason for such a buffer is that certain peripherals may only be able to handle certain quantities of data within a certain length of time and the buffer provides a means of measuring and controlling the output.

On a more general level the buffer concept is extremely useful in programming because there are many occasions on which a set block of memory is required. For example, you may wish to add together two sets of values and store the results in a separate area of memory before dealing with them. Very often you may need to set up a keyboard buffer to isolate and identify a fixed number of characters input from the keyboard.

In this example we are going to transfer a text which is 49 bytes long onto the screen via a buffer which will be only 26 bytes in length. The text will therefore be printed in two separate stages, although it will appear on the screen as one continuous sentence.

> *Note:* On the first reading of this listing you should primarily be observing the ways in which the data is addressed. Again, do not worry too much if you cannot follow all the details of the program. You will be able to go over these listings again at a later stage when some of the more difficult concepts have been explained. It is a good idea to look at the data section at the end of the listing first so that the labels referred to in the program will make more sense.

```
;-------------------------------------------------------
;        PROGRAM ENTITLED PROG2
;   LOADS TEXT INTO A BUFFER BEFORE DISPLAYING
;-------------------------------------------------------
;FIRST, DEFINE CONSTANT VALUE WITH LABEL
;-------------------------------------------------------
BUFLEN   EQU   25      ;Buffer length minus 1
;-------------------------------------------------------
;THEN ASSIGN TASK ID AND OPEN CONSOLE CHANNEL
;-------------------------------------------------------

MOVEQ    #0,D1

MOVEQ    #2,D3

LEA.L    #DEVICE,A0

MOVEQ    #1,D0

TRAP     #2
;-------------------------------------------------------
```

Initially the length of the buffer (minus 1) is assigned the label BUFLEN using the EQU (=) assembler directive. This constant will be used several times during the program to count off character codes as they are loaded into the buffer and it can therefore be loaded into a count register using its label. There is no reason why it should not be an unlabelled immediate constant but it is good practice to label as many constants as possible so that the finished program is easy to follow during the debugging phase.

Following this the program ID and console channel are initialized as explained in the previous chapter.

Next the main part of the program begins:

```
;-----------------------------------------------------------
;SET UP ROUTINE TO COPY DATA TO BUFFER
;-----------------------------------------------------------

        MOVE.B  COUNT,D6    ;D6 to count blocks printed

        LEA.L   MYDATA,A2   ;A2 points to base of data
LOOP1   LEA.L   BUFF,A3     ;Address of buffer in A3

        MOV.L   #BUFLEN,D4  ;Length of buffer in D4
LOOP2   MOVE.B  (A2)+,D5    ;Copy item of data to D5
                            ;and increment A2

        CMPI.B  #42,D5      ;Compare D5 with ASCII code
                            ;for '*'

        BEQ     NEXT        ;Branch if same to 'NEXT'

        MOVE.B  D5,(A3)+    ;Else copy data to buffer
                            ;and increment A3

        DBRA    D4,LOOP2    ;Loop back to 'LOOP2' if
                            ;D4 > -1
NEXT    BSR.S   PRNT        ;Call 'PRNT' subroutine

        SUBQ    #1,D6       ;Subtract 1 from count reg.

        BEQ     EXIT        ;Branch to 'EXIT' if D6 = 0
;-----------------------------------------------------------
```

The first instruction of the program proper; `MOVE.B COUNT,D6`, loads the value contained in the address labelled `COUNT`, as defined at the end of the program, into the low order byte of register D6. The buffer will be filled and emptied twice and D6 will keep track of how many times this has been done.

The instruction `LEA.L MYDATA,A2` loads the address of the start of the text, defined as `MYDATA` at the end of the program, into register A2.

The next instruction; `LEA.L BUFF,A3` loads the address of the first byte of the buffer space into A3. This instruction is located at an address labelled '`LOOP1`' because we shall need to loop execution back to this point later in the program.

The instruction `MOVE.L #BUFLEN,D4` loads the length of the buffer less 1 (defined at the beginning of the program as 25) into register D4 so that it can be used to count off the number of characters which are entered in the buffer. `BUFLEN` is effectively a constant and therefore it appears in the instruction as `#BUFLEN`, indicating that it represents a numeric constant rather than a value contained in an address labelled `BUFLEN`. `MOVE.B (A2)+,D5` is labelled '`LOOP2`' because a subsequent instruction will loop execution back to this point. This is an address register indirect with postincrement addressing mode instruction which copies the data contained in the address pointed to by A2 (initially address '`MYDATA`') into register D5. A2 is then automatically incremented by 1, ready to point to the next character code within `YDAT`.

The first section of the text to be copied into the buffer will terminate at the end of '`FF`' in the word '`BUFFER`'. The second part of the text is shorter than the buffer and so we need to ensure that the buffer filling process terminates as soon as the 'stop' symbol, '`*`', is reached. At this point therefore, we compare the contents of D5 with the ASCII code for '`*`' (42) using the instruction `CMPI.B #42,D5` (immediate addressing mode). `CMPI` stands for 'compare immediate'. If they match the Z flag will be set and the `BEQ NEXT` instruction will redirect execution to the address of the instruction labelled '`NEXT`'. If they do not match (i.e. Z=0) then execution carries on as follows.

The data which is now in D5 is the ASCII code for one of the text characters (initially 84, the code for '`T`'), and this needs to be copied into the buffer, which is indirectly addressed by register A3. This operation is performed by the instruction `MOVE.B D5,(A3)+`. A3 is

afterwards automatically incremented by 1 to point to the next free address in the buffer. We have now effectively transferred a character from the memory block labelled `MYDATA` to the memory block labelled `BUFF` via the D5 register and the previous two autoincrement mode instructions have set A2 and A3 so that they point to the address of the next text character and to the address of the next free buffer space respectively.

The `DBRA D4,LOOP2` instruction automatically decrements the D4 register by 1 and if the buffer is now full, D4 will hold the value –1. If D4 is greater than –1 (i.e. the buffer is not yet full) then execution loops back to the point labelled '`LOOP2`' so that a further character can be copied into D5.

If D4 equals zero then execution continues with the next instruction in sequence: `BSR.S PRNT` (branch short to subroutine). This is an instruction which calls the subroutine located at an address labelled `PRNT` which will handle the printing of the text contained in the buffer. The suffix '`.S`' is optional, specifying that this is a 'short' branch, the destination address being within plus or minus 127 bytes from the branching instruction. This results in a slightly faster execution speed.

On returning from the `PRNT` procedure, the program must now check to see whether the buffer has been filled and emptied twice, in which case the main routine has effectively finished.

The variable which records this was earlier copied into register D6. We need to subtract 1 from it: `SUBQ #1,D6`.

The `BEQ EXIT` instruction tests the zero flag to see whether the previous subtraction operation resulted in a zero. If it did then the job is finished and execution is redirected to a point in the program labelled '`EXIT`'.

If the result of the subtraction was not zero then the buffer must be refilled and printed again so execution passes on to the next instruction.

We now have a problem whose solution is found in the next five instructions. If the first chunk of text has been printed then the buffer is still full of its ASCII codes. The second chunk of text will be

shorter than the length of the buffer and so some unwanted leftover characters will be printed at the end of the text. To avoid this we must *flush* the buffer by filling each of its addresses with the value 32: the ASCII code for a blank space.

```
;-------------------------------------------------------
;FLUSH THE BUFFER
;-------------------------------------------------------

        LEA.L   BUFF,A3       ;Base addr. of buffer in A3

        MOVE.L  #BUFLEN,D4    ;Length of buffer in D4
LOOP3   MOVE.B  #32,(A3)+     ;Transfer ASCII code for
                              ;space into buffer and add
                              ;1 to A3

        DBRA    D4,LOOP3      ;Loop back to 'LOOP3' if
                              ;D4 > -1

        JMP     LOOP1        ;Else jump back to LOOP1
;-------------------------------------------------------
```

To flush the buffer the A3 register is loaded with the address of the beginning of the buffer, `BUFF`. Then the length of the buffer is moved into our counter register: `MOVE.L #BUFLEN,D4`. The ASCII code for a space, 32, is then loaded into the buffer using the instruction `MOVE.B #32,(A3)+`. This is done 26 times, according to the count in D4, using `DBRA D4,LOOP3` and thus every address in the buffer is loaded with the space code. It is then necessary to jump back and fill the buffer with the next batch of text using the instruction `JMP LOOP1`. This occurs after the first occasion on which the buffer is filled because the variable in D6 would not then be equal to zero. On the second pass the previous `BEQ EXIT` instruction diverts execution directly to '`EXIT`'.

At this point the main part of the program terminates and so we round it off with the set of instructions labelled `EXIT`:

```
;-------------------------------------------------------
;TERMINATE MAIN PROGRAM
;-------------------------------------------------------
EXIT    MOVEQ   #2,D0

        TRAP    #2

        MOVEQ   #-1,D1

        MOVEQ   #0,D3

        MOVEQ   #5,D0

        TRAP    #1
;-------------------------------------------------------
```

This is the same set of termination operations which we used in the previous program, closing the console channel and informing the operating system that the task is completed.

Following the main program is the subroutine 'PRNT', as follows:

```
;-------------------------------------------------------
;PRNT SUBROUTINE
;-------------------------------------------------------
PRNT    LEA.L   BUFF,A3         ;Base addr. of buffer in A3

        MOVE.L  #BUFLEN,D4      ;Length of buffer in D4
LOOP4   MOVE.B  (A3)+,D1        ;Copy data to D1 and
                                ;add 1 to A3

        BSR.S   DISP            ;Call display subroutine

        DBRA    D4,LOOP4        ;Loop back to 'LOOP4' if
                                ;D4 > -1

        RTS                     ;Otherwise return to PROG2
;-------------------------------------------------------
```

The first subroutine in the program, PRNT, is designed to take the contents of the buffer and print it to the screen. This is simply done by running an address register pointer through the buffer, loading each character into D1 and then using our previous character display

routine to print each character of the text to the screen. This time, the character display routine is defined as a separate subroutine called 'DISP' which is called from within the PRNT subroutine.

Since the A3 register is currently unused, we can use this to point to the start address of the buffer: LEA.L BUFF,A3.

D4 can again be used as a counter to count off each printed character and so it is loaded with the length of the buffer: MOVE.L BUFLEN,D4.

Next we set up a program loop labelled 'LOOP4' to transfer data from the buffer into D1: (MOVE.B (A3)+,D1) and from there to the DISP subroutine (BSR.S DISP). The A3 register is auto-incremented after the MOVE instruction to point to the next character in the buffer and so the D4 register must be checked to see if it has reached –1 (end of printing operation) or is greater than –1, in which case execution loops back to LOOP4. This is achieved with the instruction DBRA D4,LOOP4.

If D4 has in fact reached –1 then a return is made back to the main program by means of the RTS instruction. This effectively returns execution back to the instruction which follows the BSR.S PRNT instruction in the main program.

The final block of program code is the character display sequence:

```
;-------------------------------------------------------
;DISP SUBROUTINE
;-------------------------------------------------------
DISP     MOVEQ  #-1,D3

         MOVEQ  #5,D0

         TRAP   #3

         RTS             ;Return to 'PRNT'
;-------------------------------------------------------
```

Here, the DISP subroutine is defined, which is simply the character print routine from the last chapter, elevated to the status of a subroutine in its own right, labelled 'DISP'. The RTS at the end returns execution back to the instruction following the BSR.S DISP instruction within the PRNT subroutine.

```
;-------------------------------------------------------
;THEN DEFINE DATA
;-------------------------------------------------------
MYDATA   DC.B   "THIS TEXT GOES IN THE BUFFER "
         DC.B   "BEFORE BEING PRINTED","*"
BUFF     DS.B   26          ;26 undefined bytes
                            ;for the buffer
COUNT    DC.B   2           ;count variable
DEVICE   DC.W   4
         DC.B   'CON_'
         END                ;End program
;-------------------------------------------------------
```

The data section contains the text which is to be printed, which is defined as a series of byte constants (`DC.B`). The start address of the text will be at a location in memory labelled '`MYDATA`' and therefore the first letter of the text, '`T`', will actually be located at '`MYDATA`' itself. The '`*`' symbol which follows the text is to be used as a 'stop' code to indicate where the text ends and will be detected during the course of the program.

The actual buffer is labelled '`BUFF`', which again represents the address of the start of the buffer. Initially the buffer is empty and is initialized as a set of 26 undefined bytes using the `DS` (define storage) assembler directive.

The label '`COUNT`' refers to the address of a single memory byte whose value is 2: the number of times the buffer will be filled and printed when the program is executed. When we refer to '`COUNT`' in the program we are implying not the address of the byte labelled '`COUNT`' but its contents; the value 2.

The listing terminates as before with the assembler directive '`END`'.

The following object code listing shows the program in assembled form, which should help you to follow the processes described above.

```
;-------------------------------------------------------
;Address   Code                Mnemonic
;-------------------------------------------------------
29CE8     7200                MOVEQ #00, D1
29CEA     7602                MOVEQ #02, D3
29CEC     41FA00B8            LEA B8(PC)!29DA6, A0
29CF0     7001                MOVEQ #01, D0
```

```
29CF2     4E42              TRAP #2
29CF4     1C3A00AE          MOVE.B AE(PC)!29DA4, D6
29CF8     45FA0062          LEA  62(PC)!29D5C, A2
29CFC     47FA008C          LEA  8C(PC)!29D8A, A3
29D00     283C00000019      MOVE.L  #19, D4
29D06     1A1A              MOVE.B  (A2)+, D5
29D08     0C05002A          CMPI.B  #2A, D5
29D0C     67000008          BEQ 29D16
29D10     16C5              MOVE.B  D5, (A3)+
29D12     51CCFFF2          DBRA  D4,29D06
29D16     6128              BSR 29D40
29D18     5346              SUBQ #1, D6
29D1A     67000018          BEQ 29D34
29D1E     47FA006A          LEA  6A(PC)!29D8A, A3
29D22     283C00000019      MOVE.L  #19, D4
29D28     16FC0020          MOVE.B  #20, (A3)+
29D2C     51CCFFFA          DBRA  D4,29D28
29D30     4EFAFFCA          JMP FFCA(PC)!29CFC
29D34     7002              MOVEQ #02, D0
29D36     4E42              TRAP #2
29D38     72FF              MOVEQ #FF, D1
29D3A     7600              MOVEQ #00, D3
29D3C     7005              MOVEQ #05, D0
29D3E     4E41              TRAP #1

PRNT:-

29D40     47FA0048          LEA  48(PC)!29D8A, A3
29D44     283C00000019      MOVE.L  #19, D4
29D4A     121B              MOVE.B  (A3)+, D1
29D4C     6106              BSR 29D54
29D4E     51CCFFFA          DBRA  D4,29D4A
29D52     4E75              RTS

DISP:-

29D54     76FF              MOVEQ #FF, D3
29D56     7005              MOVEQ #05, D0
29D58     4E43              TRAP #3
29D5A     4E75              RTS
;-------------------------------------------------------------
```

The following data dump shows the arrangement of the initial data in memory, prior to the program being executed. The reserved buffer space occupies the area from the address immediately following '*' (hex code $2A) to the address immediately before the '02' on the second line from the bottom.

```
29D5C -- -- -- -- 54 48 49 53 ....THIS
      20 54 45 58 54 20 47 4F .TEXT GO
29D68 45 53 20 49 4E 20 42 55 ES IN BU
      46 46 45 52 20 42 45 46 FFER BEF
29D78 4F 52 45 20 42 45 49 4E ORE BEIN
      47 20 50 52 49 4E 54 45 G PRINTE
29D88 44 2A 00 00 00 00 00 00 D*......
      00 00 00 00 00 00 00 00 ........
29D98 00 00 00 00 00 00 00 00 ........
      00 00 00 00 02 00 00 04 ........
29DA8 43 4F 4E 5F 00 00 00 00 CON_....
```

The following list contains explanations of the functions of those assembly language instructions used in PROG2 which did not feature in PROG1:

Mnemonic	*Meaning*
CMPI.B #42,D5	CoMPare Immediate the contents of the low order byte of D5 with the byte value 42 and set the Z flag if they are the same.
BEQ NEXT	Branch if EQual (i.e. Z=1) to the program instruction labelled 'NEXT'.
BSR.S PRNT	Branch to the SubRoutine (using a Short address) labelled 'PRNT'.
SUBQ #1,D6	SUBtract Quick the value 1 from the value contained in register D6. SUBQ is essentially the same as SUB.L but is quicker to execute.
JMP LOOP1	JuMP to the program instruction labelled 'LOOP1'.
RTS	ReTurn from Subroutine to the program or subroutine which called it.

In PROG2 we have not used the more complex indirect addressing modes and these will be covered in later chapters. In the next chapter we shall return to the subject of flags, which are essential to the understanding of the programs which follow.

Chapter 10

Status and Condition Flags

In Chapter 3 we discussed the functions of the various condition flags in the CCR (condition codes register) byte of the status register and examined how and why various arithmetic and logical operations affect them. In this chapter we shall briefly revise the condition flag functions and look at some trace listings which will help you to relate some of the flag settings to specific operations and to the contents of the registers as the instructions are executed.

We shall then examine the functions of the status flags which occupy the 'system byte' of the status register. Finally, a number of instructions will be listed which can be used specifically to alter the values of various flags.

The Status Register

The status register in the 68000 is structured as follows:

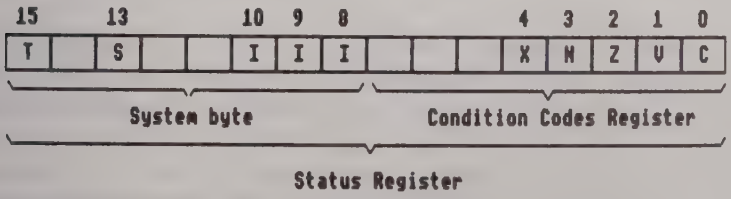


FIGURE 10-1.

To begin with, we will only be concerned with the condition flags: V (overflow flag), N (negative or sign flag), Z (zero flag), X (extend flag)) and the C (carry flag).

Overleaf is a summary of the functions of each of these flags:

V Set when an operand's sign flag is altered by an operation, indicating an overflow condition in terms of 2's complement arithmetic

N Set when a value is negative in terms of 2's complement arithmetic (high order bit = 1) and reset when the value is positive (high order bit = 0)

Z Set when the result of an operation is zero, otherwise reset

X Set when a carry or borrow occurs in 'extended' arithmetic operations such as `ADDX` and `ABCD`.

C Set if an operation results in a binary carry or borrow

We shall now go on to look at some of these flags in action, using a number of program fragments as illustrations.

Zero Flag

Firstly, we shall deal with the *zero flag*. In the following operation, we shall take the example which was used in Chapter 3, where we took a keyboard entry and used a routine to discover whether the 'Y' key had been depressed. The code for this operation is as follows: (Code for last key pressed is contained in D1)

```
CMPI.B #89,D1
NOP
```

If we execute this operation using a trace utility, the following register and flag values will be printed out for the instructions. Note that because a trace listing shows the status of the flags and registers *immediately before* the corresponding instruction has been executed, we need to have a trace listing for the next instruction in sequence so that the effect of the `CMPI` instruction can be shown. In this example the dummy instruction `NOP` (No Operation) had been added. When executed, `NOP` has no effect although it occupies space in the program code and advances the PC register.

The information in which we are specifically interested is printed in italic:

```
29CEE 0C010059              CMPI.B   #59, D1
D0=0   D1=59   D2=0    D3=0    D4=0    D5=0    D6=0
D7=0   A0=0    A1=0    A2=0    A3=0    A4=0    A5=0    A6=0
A7=3DBC6        Status= T Imask=0 Program Counter =29CEE

29CF2 4E71                  NOP
D0=0   D1=59   D2=0    D3=0    D4=0    D5=0    D6=0
D7=0   A0=0    A1=0    A2=0    A3=0    A4=0    A5=0    A6=0
A7=3DBC6       Status= Z T Imask=0 Program Counter =29CF2
```

In this case the D1 register originally contained $59, which is the hexadecimal equivalent of 89 decimal. The comparison operation therefore set the Z flag, indicating that D1 is equal to 89. Note that after the CMPI operation, the original value of D1 has been left unchanged.

Sign Flag

The sign flag value is always a copy of the most significant bit of a binary number, regardless of its size. In the following example, decimal 10 is added to decimal 120 which has the effect of setting the sign flag. The result in unsigned arithmetic is 130 decimal ($82) whilst the 2's complement value of the result is –126. The value 20 is then subtracted from the result, giving an unsigned decimal result of 110 ($6E) and a 2's complement value of +110; the sign flag having been reset again.

```
MOVEQ #120,D1
ADDI.B #10,D1
SUBI.B #20,D1
NOP
```

The trace printouts for the above operations are as follows. Note the way in which the value of register D1 alters in each case, showing the hex values $78, $82 and $6E:

```
29CF8 7278                  MOVEQ  #78, D1
D0=0   D1=0    D2=0    D3=0    D4=0    D5=0    D6=0
D7=0   A0=0    A1=0    A2=0    A3=0    A4=0    A5=0    A6=0
A7=3DBC6       Status= T Imask=0 Program Counter =29CF8
```

```
29CFA 0601000A              ADDI.B   #A, D1
D0=0   D1=78   D2=0    D3=0    D4=0    D5=0    D6=0
D7=0   A0=0    A1=0    A2=0    A3=0    A4=0    A5=0    A6=0
A7=3DBC6       Status= T Imask=0 Program Counter =29CFA

29CFE 04010014              SUBI.B   #14, D1
D0=0   D1=82   D2=0    D3=0    D4=0    D5=0    D6=0    D7=0
A0=0   A1=0    A2=0    A3=0    A4=0    A5=0    A6=0
A7=3DBC6       Status= V N T Imask=0 Program Counter =29CFE

29D02 4E71                  NOP
D0=0   D1=6E   D2=0    D3=0    D4=0    D5=0    D6=0
D7=0   A0=0    A1=0    A2=0    A3=0    A4=0    A5=0    A6=0
A7=3DBC6       Status= V T Imask=0 Program Counter =29D02
```

Overflow Flag

In the previous example, the overflow flag became set after the `ADDI.B £$0A,D1` instruction was executed. This is because the sign of the value was altered by the addition. The sign is changed again by the `SUBI` instruction so that V is set again. In 2's complement terms the above operations correspond to the following:

	+	120	
plus	+	10	
=	–	126	(incorrect because the sign flag was altered, causing an overflow error)
minus	+	20	
=	+	110	(final result correct)

Carry Flag

In the next example, the 2's complement value +10 is added to –1, which is contained in the D1 register. The result of this addition is +9 which is correct. In this case however the *carry flag* is set because there has been a binary carry from the most significant bit of D1 into the carry flag and therefore *in decimal terms* the result is incorrect. –1 as a 32-bit *unsigned* value is 4 294 967 295 ($FFFFFFFF) and therefore in decimal terms the calculation is 4 294 967 295 + 10 = 9:

```
MOVEQ #-1,D1 (-1 = FFFFFFFF = 4294967295 decimal)
ADD.L #10,D1 (+10 =0000000A = 10 decimal)
NOP
```

Result is $00000009_{HEX} = +9$ in 2's complement and 9 in decimal.

```
29D08 72FF                    MOVEQ  #FF D1
D0=0  D1=0    D2=0    D3=0    D4=0    D5=0    D6=0
D7=0  A0=0    A1=0    A2=0    A3=0    A4=0    A5=0    A6=0
A7=3DBC6        Status= T Imask=0 Program Counter =29D08

29D0A D2BC0000000A          ADD.L   #A, D1
D0=0  D1=FFFFFFFF     D2=0    D3=0    D4=0    D5=0    D6=0
D7=0  A0=0    A1=0    A2=0    A3=0    A4=0    A5=0    A6=0
A7=3DBC6      Status= N T Imask=0 Program Counter =29D0A

29D10 4E71                    NOP
D0=0  D1=9    D2=0    D3=0    D4=0    D5=0    D6=0    D7=0
A0=0  A1=0    A2=0    A3=0    A4=0    A5=0    A6=0
A7=3DBC6    Status= C X T Imask=0 Program Counter =29D10
```

Extend Flag

In the next example, the decimal value 8 is added to the decimal value 255. If you look at this operation in binary first of all, you will see that a carry is generated from the operation:

```
 00001001   (9)
+11111111   (255)
 --------
=00001000   (8)
carry 1
```

The result of this is that the *extend and carry* flags are set, as follows:

```
MOVEQ #9,D1
ADDI #255,D1
NOP

29D16 7209                    MOVEQ  #09, D1
D0=0  D1=0    D2=0    D3=0    D4=0    D5=0    D6=0    D7=0
A0=0  A1=0    A2=0    A3=0    A4=0    A5=0    A6=0
A7=3DBC6        Status= T Imask=0 Program Counter =29D16
```

```
29D18 060100FF            ADDI.B   #FF, D1
D0=0  D1=9    D2=0    D3=0    D4=0    D5=0    D6=0    D7=0
A0=0  A1=0    A2=0    A3=0    A4=0    A5=0    A6=0
A7=3DBC6        Status= T Imask=0 Program Counter =29D18

29D1C 4E71                NOP
D0=0  D1=8    D2=0    D3=0    D4=0    D5=0    D6=0    D7=0
A0=0  A1=0    A2=0    A3=0    A4=0    A5=0    A6=0

A7=3DBC6        Status= C X T Imask=0 Program Counter =29D1C
```

The extend flag signifies the same condition as the carry flag and is used in multiple precision operations to ensure that each carry from an operation is carried automatically over to the next byte, as explained in Chapter 7.

Status Flags

The status flags, T, S and I, are contained in the system byte of the status register and are used to indicate the following conditions:

Trace Flag (T)

In this chapter we have been using a 'trace' facility to 'single-step' through individual instructions, providing a print-out of the status of all the registers and flags. The assembler toolkit program which provides this facility is using one of the exception functions incorporated in the 68000 to achieve this. When the T flag is set, execution is redirected to a special exception routine which in this case is programmed to print out the status of the flags and registers for each individual instruction and then return execution to the main program. When the T flag is reset, program execution functions normally.

Interrupt Flag (I)

Interrupt exceptions are described in Chapter 7 and if you are not clear about their function you may wish to refer to this section of the book. The interrupt flag , consisting of three bits, is used selectively to disable (mask) external maskable interrupts or to enable them. When interrupts have been disabled, maskable interrupts from external devices cannot interrupt the flow of execution. Interrupts are given a priority code from 0 (lowest) to 7 (highest), hence the allocation of three interrupt flag bits in the status register. Any device

whose interrupt priority code is less than or equal to the priority level set by the interrupt flags will be unable to interrupt the processor.

Supervisor Flag

The S flag determines whether the system is operating in user mode (S = 0) or supervisor mode (S = 1). Note that when an exception takes place, the processor enters supervisor mode automatically for the duration of the exception, regardless of the current setting of the S flag.

Flag Control Instructions

For the purpose of familiarizing yourself with the ways in which the flags can be altered during program operation, it is useful to divide the flag altering instructions into four main groups:

1 The first group of flag altering instructions are the routine instructions such as `MOVE`, `ADD`, `SUB`, `CMP` etc. which alter the flags according to the results of the operations being performed. This group includes the logical instructions, `AND`, `OR` and `XOR`, all arithmetic instructions, shift and rotate instructions and a number of miscellaneous instructions such as `CHK`, `CLR`, `EXT` and `SWAP`. Notable instructions which *do not* alter any flags are jump and branch instructions and some of the instructions which specifically use an address register as a destination, such as `ADDA`, `MOVEA` etc. In the instruction glossary in Appendix B the flags which are affected by any specific instruction are listed under each instruction heading.

2 The second group of instructions which affect the flags are those which are specifically designed to set, reset or copy one or more of the condition flags. These include `ANDI to CCR`, `EORI to CCR`, `MOVE to CCR`, `MOVE from CCR` and `ORI to CCR`. For example, `MOVE to CCR` can be used to move a source operand into the CCR register. The flags will be set according to the binary pattern in the source operand. `MOVE #0,CCR` would reset all the flags for example. Individual flags may be specifically set or reset by using the logical instructions. `ORI #1,CCR` for example will set the carry flag without affecting any of the other flags.

3 The third group of flag altering instructions are privileged instructions which can only be used from within supervisor mod-

These are normally used by an operating system to set initial values for the status and condition flags or to alter them for some specific task and include ANDI to SR, EORI to SR MOVE to SR and ORI to SR.

4 The fourth group of flag altering instructions are those which are designed to test an individual bit in an operand, set the condition flags accordingly, and in some cases to alter the value of the original bit. This group includes BCHG, BCLR, BSET, BTST and TST and TAS. The TST instruction tests an entire operand in memory or in a register and alters the status flags without altering the operand. TAS is intended for use in a resource sharing system where one program can temporarily test and set a bit in the destination operand in order to exclude access by other processors whilst it accesses data. This group of flags is described in more detail in chapter 3 and under the individual instruction headings in Appendix B.

One further instruction which is used in relation to the condition codes is Scc (set according to condition) which uses the same conditions as the Bcc and DBcc instructions. The effect of using Scc is that if the condition tested is true, a specified data alterable destination byte is set to the value 255, otherwise it is set to zero.

For example, suppose that you reserve a special address for use with the Scc instruction and give it the label TESTBYTE. At some point in the program you may wish to test a flag, such as the zero flag, after a certain operation has been performed but you may not wish to make a conditional decision at that point. In order to keep a record of the status of the Z flag for later use you can use the instruction SEQ TESTBYTE. If the zero flag is set then the byte in address TESTBYTE will be set to 255, otherwise it will be zeroed. Later on you can refer to the TESTBYTE operand to determine the result of the earlier operation and make a conditional decision accordingly. Note that in programs which have been assembled relative to PC, you cannot alter the contents of a labelled address and it will be necessary to use the low byte data register as a destination for Scc instead. From there it can be pushed onto the stack until you need it.

Chapter 11

Conditional and Unconditional Branching Operations

In Chapter 4 we examined the conditional branch instructions, Bcc and DBcc and the unconditional jump and branch instructions, BRA, BSR, JMP and JSR. In this chapter we shall review these instructions and then look at a program illustrating how these branching functions are used.

Short and Long Branching

An absolute address within a 16 megabyte memory space needs to be specified using a 24-bit address value. An absolute address specified within an instruction in the source listing, such as `JMP $29CE8`, is interpreted by the system as being a 24-bit number: $029CE8. If the address is contained in an address register then it is held as a 32-bit number: $00029CE8, even though only the lower 24 bits are relevant. This is termed a *long* address because it contains all the bits necessary for specifying an address anywhere in memory.

A *short* absolute address can be represented by only 16 bits which, as a signed value in an address register, can represent any address within the top 32K or bottom 32K of memory, for reasons which were explained in Chapter 2.

A *relative* address, as used with `BRA`, `DBRA`, `DBcc`, `Bcc` and `BSR` instructions, may also be short, being expressed as a signed 8-bit displacement *relative to the instruction which specifies it* and giving a branch in the range ±127 bytes. A 16-bit or 'long' relative branch displacement gives a branch in the range ±32K.

For most practical purposes the time and space gained by the use of short addressing will be negligible and it is not something which you

need normally be concerned about unless your program is time-critical.

Conditional Branches

The conditional branching instructions are Bcc (branch according to condition code) and DBcc (decrement and branch according to condition code). A conditional branch is made to a location indicated by a positive or negative displacement relative to the PC register. A branch specified by an 8-bit displacement value gives a branch in the range ±127 bytes and a 16-bit displacement value gives a branch in the range ±32K. In practice, the address to which a conditional branch is made is normally designated by a user-defined label and the actual relative displacements are calculated automatically by the assembler program.

Obviously if your program is unusually long or if one program branches to another and you wish to branch conditionally to an address in another part of memory then it may not be possible since the destination cannot be more than 32K bytes away from the branch instruction. In this case a JMP or JSR command would be needed.

The DBcc instruction is similar to Bcc except that it automatically decrements a nominated data register after the condition flags are tested. If the specific condition is met or if the nominated register equals –1 after decrementation then the branch is not made.

The conditional branch instructions are as follows. Note that the same conditions apply both to Bcc and DBcc so that for DBEQ (decrement and branch if equal) for example, the same flag conditions apply as those for BEQ (branch if equal). The Bcc conditions should be interpreted as 'if' conditions – e.g. BCC means 'branch *if* carry flag reset'. The DBcc conditions should be interpreted as 'until' conditions. For example, DBCC means 'branch *until* carry flag reset (or data register = –1)'.

Note that the DBcc instructions can be used with the additional conditions T(true) and F(false).

Instruction	*Condition*	*Flag Status*
BCC	if carry reset	C = 0
BCS	if carry set	C = 1
BEQ	if equal	Z = 1
BNE	if not equal	Z = 0
BPL	if plus	N = 0
BMI	if minus	N = 1
BVC	if overflow clear	V = 0
BVS	if overflow set	V = 1
BHI	if high	C = 0 & Z = 0
BLS	if low or same	C = 1 or Z = 1
BHS	if high or same	C = 0
BLO	if low	C = 1
BGT	if greater than	(N = 1 & V = 1 & Z = 0) or (N = 0 & V = 0 & Z = 0)
BGE	if greater or equal	(N = 1 & V = 1) or (N = 0 & V = 0)
BLE	if less or equal	Z = 1 or (N = 1 & V = 0) or (N = 0 & V = 1)
BLT	if less than	(N = 1 & V = 0) or (N = 0 & V = 1)

Unconditional Branches and Jumps

The unconditional branch operations, BRA (branch always), BSR (branch to subroutine) and DBRA (decrement and branch) work similarly to Bcc and DBcc except of course that there are no conditions attached. Again the branches are PC relative and the signed displacement values may be of 8 or 16 bits. The unconditional jumps, JMP (jump) and JSR (jump to subroutine) use absolute or indirect addresses as their destinations – in other words the destination address replaces the current value of the PC register rather than being a displacement value which is added to it. It is important, however, that you should try as far as possible to avoid using absolute address numbers with JMP and JSR instructions. If you jump to a numbered absolute address and then subsequently alter the source code then the destination address may have to be changed. By labelling jump destinations you can ensure that if the program is altered and then re-assembled, the destination address will automatically be adjusted if necessary by the assembler.

If your assembled program is position-dependent then a JMP instruction which refers to a specific numbered address will not work if the program has to be loaded into some other part of memory. In this case you would be quite free to jump out of your program to some other program at a known address but any attempt to jump into another part of your own program will fail because the destination address will have changed. In order to avoid this kind of problem it is best to make your programs position independent and to use PC relative branches wherever possible.

Care should be taken when using JMP and JSR instructions because some assemblers are less sophisticated than others and may not code the necessary relocation information.

The BRA and JMP instructions never retain the old value of the PC register after the jump is made. The BSR and JSR instructions, equivalent to the BASIC GOSUB instruction, automatically store the current value of the PC register on the stack so that when a return from the subroutine is made, the old value of the PC register can be retrieved from the stack and the program can re-commence from where it left off.

The advantage of the JMP over the BRA instruction is that whereas BRA is confined to the PC relative addressing mode, the destination of a JMP instruction can be specified using any memory addressing mode except indirect with predecrement and indirect with postincrement. The same advantage applies to the JSR instruction over BSR. Additionally, JMP and JSR permit a longer jump range then BRA and BSR.

Conditional Branching to Subroutines

The BSR and JSR instructions are always unconditional, although a conditional BSR or JSR instruction can easily be simulated by preceding it with a conditional branch or jump instruction which by-passes the branch if the condition is not met. This is most easily illustrated in BASIC, as follows:

```
30 A=10
40 B=C
50 IF A=B THEN GOTO 70: REM conditional test
```

```
60 GOSUB 210:           REM unconditional GOSUB
70 next instruction
```

In assembly language these operations might be expressed as:

```
;-----------------------------------------------------------
;Label     Mnemonic          Comment
;-----------------------------------------------------------
          MOVEQ #10,D2      ;Load D2 register with 10
          MOVE D4,D3        ;Load D3 register from D4
          CMP.L D2,D3       ;Compare D2 with D3
          BEQ NEXTINS       ;Branch if equal (Z=1) to NEXTINS
          BSR SUBRT1        ;Else call subroutine 1
NEXTINS   next instruction  ;Continue program
;-----------------------------------------------------------
```

Returning from Subroutines

A return from a subroutine is normally made using an RTS (return from subroutine) instruction. RTS automatically retrieves the old PC value from the stack and uses it as the return address.

A variation on this is provided by the RTR (return and restore condition codes) instruction. This is used when you wish to call a subroutine and return not only to the point where you left off but with the condition flags restored exactly as they were before the subroutine was called. As soon as the subroutine is entered, the current contents of the SR register are saved on the stack using: MOVE SR,-(A7).

When the subroutine has been completed the RTR instruction automatically retrieves the stored flag values from the stack and places the lower 5 bits in the CCR register before retrieving the return address from the stack and loading it into PC.

A similar instruction, RTE (return from exception) is used for returning from an exception service routine, although this is a privileged instruction which cannot be used in user mode. Like RTR, it automatically retrieves the old flag values and loads them back into CCR. Unlike RTR however, it also loads the old status flag values back into the system byte of the status register.

Example Program 3

In the next example program, PROG3, four types of branching operations are illustrated: a call to a subroutine, a conditional branch, a loop and an RTS.

The purpose of this program is to sort through a list of byte values and identify those which are valid standard ASCII codes; in other words, those which have a value between 0 and 127. The characters which correspond to the valid ASCII codes will be printed to the screen and the non-valid codes, of which there are two, will be ignored.

In this program, the character display interrupt routine has been defined as a separate subroutine in its own right, as it was in the previous program.

> *Note:* on a first reading of this program you should be looking particularly at the ways in which the branching operations transfer execution from one point in the program to another.

As before, the data section is at the end of the listing and it will be helpful to have a brief look at it before you begin the code listing. The 7 single byte ASCII codes have been defined in the data section at an address labelled ASCDATA and, as you will see, two of them are invalid codes having a value greater than 127. The COUNT variable contains the number of items contained in the ASCDATA array, less 1.

```
;-------------------------------------------------------
;          ASCII SORTER PROGRAM CALLED PROG3
;          SORTS &  PRINTS VALID ASCII CODES
;-------------------------------------------------------
;ASSIGN TASK ID AND OPEN CONSOLE CHANNEL
;-------------------------------------------------------
          MOVEQ    #0,D1
          MOVEQ    #2,D3
          LEA.L    #DEVICE,A0
          MOVEQ    #1,D0
          TRAP     #2
```

```
;-----------------------------------------------------------
;THEN THE PROGRAM REGISTERS ARE INITIALIZED
;-----------------------------------------------------------
          MOVE.B  COUNT,D4     ;D4 will count data
          LEA.L   ASCDATA,A2   ;A2 points to data
;-----------------------------------------------------------
;THEN DATA ITEM IS LOADED AND TESTED FOR VALID ASCII RANGE
;-----------------------------------------------------------
LOOP1     CLR.L   D1           ;Clear register D1
          MOVE.B  (A2)+,D1     ;Copy an item of data to D1
          EXT.W   D1           ;Byte in D1 is sign-extended
                               ;into a word
          EXT.L   D1           ;Word in D1 is sign-extended
                               ;into a long word
          BMI     NEXTDATA     ;Branch if sign negative
                               ;N = 1) to 'NEXTDATA'
          BSR     DISP         ;Otherwise call 'DISP
                               ;subroutine
NEXTDATA  DBRA    D4,LOOP1     ;If D4>-1 loop to 'LOOP1'
;-----------------------------------------------------------
;TERMINATE MAIN PROGRAM
;-----------------------------------------------------------
          MOVEQ   #2,D0
          TRAP    #2
          MOVEQ   #-1,D1
          MOVEQ   #0,D3
          MOVEQ   #5,D0
          TRAP    #1
;-----------------------------------------------------------
```

The main routine initially copies the count value into D4 and loads the address of the first item of data into A2.

In the main loop the first step is to clear the whole of the D1 register, ready to hold a data item. Then we load a data byte from `ASCDATA` using `MOVE.B (A2)+,D1`. This also increments A2 by 1 byte to point to the next data item. We then need to test the D1 register to see whether it contains a value less than 128. The easiest way to do this is to test the *sign* of the value, since, if it is greater than 127 it will have its sign bit set, which in turn will cause the sign flag to be set.

Firstly the low order byte of D1 is sign extended into a word using `EXT.W D1`. Then the low order word of D1 is sign extended to a long

word using EXT.L D1. The byte in D1 is now fully sign extended so that we can test the whole of D1. If the original byte in D1 was less than 128 then it will be positive and the three high order bytes of D1 will have been extended with zeroes, otherwise it will be negative and D1 will have been extended with 1s. We could of course have simply planned to test the low order byte of D1 but it is useful to see how a value in a register can be modified in this way.

D1 can now be tested by a conditional branching instruction, in this case BMI (branch if minus). If the sign (N) flag was set by the EXT operations then BMI will force execution to branch to the address of the NEXTDATA routine, thus ignoring the code in D1 and continuing with the rest of the program. If the sign flag is not set, the code in D1 must be a valid ASCII code and so a branch is made to the DISP subroutine, which prints the character to the screen and then returns execution to the instruction following BSR at NEXTDATA. The DBRA instruction decrements the counter register, D4, and branches back to LOOP1 if D4 is not yet equal to –1. The loop is executed 7 times, after which the main program terminates.

Following this we have our display subroutine:

```
;-------------------------------------------------------
;'DISP' SUBROUTINE
;-------------------------------------------------------
DISP      MOVEQ   #-1,D3
          MOVEQ   #5,D0
          TRAP    #3
          RTS               ;Return to main program
;-------------------------------------------------------
;THEN THE DATA SECTION IS DEFINED
;-------------------------------------------------------
ASCDATA   DC.B    65,83,200,67,73,129,73     ;Data
COUNT     DC.B    6         ;Defines and names length of
                            ;ASCDATA (less 1) in 1
                            ;reserved byte
DEVICE    DC.W    4
          DC.B    'CON_'
          END
;-------------------------------------------------------
```

When the program is executed the valid ASCII codes, 65, 83, 67, 73 and 73 are printed consecutively to spell out the word 'ASCII'.

The following object code listing derived from *PROG3* should help you to follow the above description.

```
;-------------------------------------------------------
;Address    Code              Mnemonic
;-------------------------------------------------------
29CE8       7200              MOVEQ #00, D1
29CEA       7602              MOVEQ #02, D3
29CEC       41FA003E          LEA 3E(PC)!29D2C, A0
29CF0       7001              MOVEQ #01, D0
29CF2       4E42              TRAP #2
29CF4       183A0035          MOVE.B 35(PC)!29D2B, D4
29CF8       45FA002A          LEA 2A(PC)!29D24, A2
29CFC       4281              CLR.L  D1
29CFE       121A              MOVE.B (A2)+, D1
29D00       4881              EXT.W  D1
29D02       48C1              EXT.L  D1
29D04       6B000006          BMI  29D0C
29D08       61000012          BSR  29D1C
29D0C       51CCFFEE          DBRA  D4,29CFC
29D10       7002              MOVEQ  #02, D0
29D12       4E42              TRAP  #2
29D14       72FF              MOVEQ  #FF, D1
29D16       7600              MOVEQ  #00, D3
29D18       7005              MOVEQ  #05, D0
29D1A       4E41              TRAP  #1
DISP:-
29D1C       76FF              MOVEQ  #FF, D3
29D1E       7005              MOVEQ  #05, D0
29D20       4E43              TRAP  #3
29D22       4E75              RTS
;-------------------------------------------------------
```

The conditional branch instruction, `BMI 29D0C`, was `BMI NEXTDATA` in the source listing. This has therefore been assembled to incorporate the `ADDRESS` value of `NEXTDATA` so that execution branches to 29D0C, which is the address of the `DBRA` instruction.

Similarly, the `BSR` to the `DISP` subroutine has been interpreted as `BSR 29D1C`. This causes execution to branch to 29D1C which is the start address of the display subroutine. The `RTS` instruction automatically returns execution from 29D23, the end of the `DISP` subroutine, back to 29D0C which is the instruction immediately following the `BSR` instruction.

The DBRA instruction automatically decrements D4 by 1 and if it is greater than –1, transfers execution back to 29CFC which was labelled LOOP1 in the original source listing. If D4 = –1, execution would continue from 29D10 instead.

Could this program have been coded more efficiently using a single DBMI instruction rather than a combination of BMI and DBRA? You might find it useful to work out an alternative coding along these lines.

Passing Parameters to Subroutines

In Chapter 4 we saw how parameters can be passed to subroutines. The two methods described were passing 'by register', where the parameters are moved into registers before the branch to the subroutine is made, and passing 'by name', where the parameters are stored in a data table starting at a named base address. Parameters may also be returned from a subroutine, either by value or by name.

The following program, PROG4, illustrates the use of both the 'by register' and 'by name' methods.

This program is considerably more complex than the preceding ones and will need careful study. The main program is a short routine which passes a message number to a subroutine labelled PRNT, which organizes the printing of both the output message and a message header, which is incorporated in the subroutine. PRNT calls a second subroutine called CHAR which prepares the characters which are to be printed and which in turn calls a third subroutine, DISP, which is our old character print trap routine.

The execution flow is thus as follows:

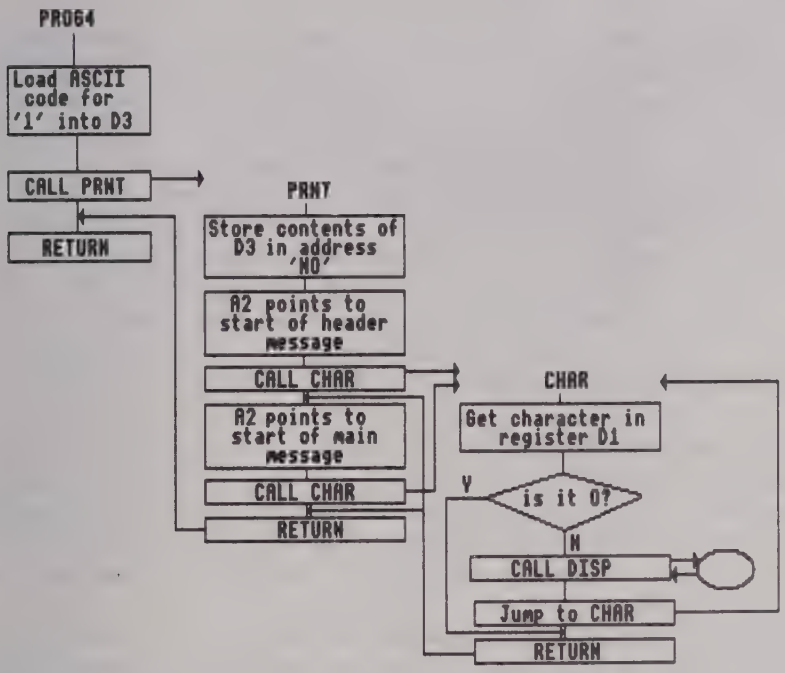


FIGURE 11-1. Structure of PROG4.

The screen display output of the program consists of the words:

```
MESSAGE NUMBER 1   (this is the message header)
BUGS ARE LETHAL    (this is the message)
```

The source listing for the main program, PROG4, is as follows:

```
;-------------------------------------------------------
;           PROG4: PARAMETER PASSING PROGRAM
;           DISPLAYS A MESSAGE
;-------------------------------------------------------
;ASSIGN TASK ID AND OPEN CONSOLE CHANNEL
;-------------------------------------------------------
    MOVEQ    #0,D1
    MOVEQ    #2,D3
    LEA.L    DEVICE,A0
    MOVEQ    #1,D0
    TRAP     #2
```

```
;-------------------------------------------------------
;DEFINE CONSTANTS
;-------------------------------------------------------
CR   EQU     13              ;Define carriage return code
LF   EQU     10              ;Define line feed code
;-------------------------------------------------------
;MAIN PROGRAM
;-------------------------------------------------------
    MOVE.B   #'1',D3         ;ASCII code for '1' in D3
    BSR      PRNT            ;Call subroutine PRNT
;-------------------------------------------------------
;TERMINATE MAIN PROGRAM
;-------------------------------------------------------
    MOVEQ    #2,D0
    TRAP     #2
    MOVEQ    #-1,D1
    MOVEQ    #0,D3
    MOVEQ    #5,D0
    TRAP     #1
;-------------------------------------------------------
```

The D3 register is loaded with the ASCII code of the character '1', which is the number of the message and which will be a parameter passed 'by register' to a subroutine. Note that the '1' in the `MOVE.B #'1',D3` instruction refers neither to the absolute address number 1 (in which case it would be expressed as `MOVE.B 1,D3`) or to the value 1 (in which case it would be `MOVE.B #1,D3`). The quote marks indicate that it is the ASCII code for the physical character 1 which is required.

Then a branch is made to the subroutine 'PRNT' which functions as follows:

```
;-------------------------------------------------------
;SUBROUTINE PRNT
;-------------------------------------------------------
PRNT     LEA.L   NO,A2        ;Address of NO in A2
         MOVE.B  D3,(A2)      ;Get passed code from
                              ;PROG4 in address 'NO'
         LEA.L   INTRO,A2     ;Address of INTRO in A2
         BSR     CHAR         ;Branch to CHAR
         LEA.L   MESS,A2      ;Address of MESS in A2
         BSR     CHAR         ;Branch to CHAR
         RTS                  ;Return to main
                              ;program
```

```
;-----------------------------------------------------
;SUBROUTINE CHAR
;-----------------------------------------------------
CHAR     MOVE.B  (A2)+,D1    ;Get character code
                             ;from array
         CMPI.B  #0,D1       ;Is it zero? (stop code)
         BEQ     EXIT        ;Branch to exit if so
         BSR     DISP        ;Else call subroutine
                             ;DISP
         BRA     CHAR        ;Loop back to CHAR
EXIT:    RTS                 ;Return to PRNT
;-----------------------------------------------------
```

In the PRNT subroutine the address labelled 'NO' is moved into the A2 register. This address is the one in which the ASCII code for '1' will be loaded. This code, which is currently in the low byte of register D3, is then loaded into NO using the instruction MOVE.B D3,(A2) so that it becomes an extension to the header message.

The A2 register is then loaded with the address of the data block headed INTRO, which marks the start of the header message (LEA.L INTRO,A2) and a call is made to the subroutine CHAR which will organize the printing of the header (BSR CHAR).

On returning from CHAR, the A2 register is loaded with the address of the message: LEA.L MESS,A2. Again, CHAR is called which prints the message and then the PRNT subroutine concludes with RTS, returning execution to the main routine.

The CHAR subroutine will call the subroutine DISP and so each character in turn must be loaded into the low byte of the D1 register. On each occasion when CHAR is called the A2 register will be pointing to the base address of the text which is to be printed: firstly INTRO and secondly MESS. The characters are therefore copied to D1 using MOVE.B (A2)+,D1 which automatically postincrements A2 to point to the next character in sequence. The character code in D1 must then be compared with the immediate value 0 to check whether it is the 'stop' code placed at the end of each message. If it is equal to zero the Z flag will be set and the BEQ (branch if equal) instruction diverts execution to the location labelled EXIT. If it is not equal (Z = 0) then D1 must contain one of the message characters and a call is made to the subroutine DISP which will display the character on the screen. On returning from DISP an unconditional branch is made back to the

location labelled CHAR (BRA CHAR) so that the next character code can be processed.

The EXIT location contains an RTS instruction which returns execution back to the previous subroutine, PRNT.

Finally, the DISP subroutine is coded as follows. This is the operating system character display trap exception which was used in the previous programs.

```
;-----------------------------------------------------------
;DISP SUBROUTINE
;-----------------------------------------------------------
DISP    MOVEQ   #-1,D3
        MOVEQ   #5,D0
        TRAP    #3
        RTS             ;Return to main program
;-----------------------------------------------------------
```

On completion, the RTS in this subroutine returns control back to the CHAR subroutine.

The data for the program is defined as follows:

```
;-----------------------------------------------------------
;DEFINE DATA
;-----------------------------------------------------------
MESS      DC.B      'BUGS ARE LETHAL',CR,LF,0
DEVICE    DC.W      4
          DC.B      'CON_'
INTRO     DC.B      'MESSAGE NUMBER '
NO        DS.B      1
          DC.B      13,10
          DC.B      0
          END
;-----------------------------------------------------------
```

The message which is to be sent is defined as a character string and the base address of this is passed, as a 'by name' parameter, to the subroutine CHAR. By name, in this case, means that the message will be addressed by the name 'MESS'. The two strings immediately following the message, CR and LF stand for 'carriage return' and 'line feed' and their ASCII values were defined at the beginning of the

program, using the EQU (equal to) directive. The '0' is a stop code, indicating the end of the message, and is used in the same way as the stop code '*' in PROG1. Note that the address N0 is defined as a single reserved byte *storage* area (DS.B 1) which is initially empty.

The following object code print outs show the four assembled modules of the program:

Main program:

```
29CE8 7200              MOVEQ  #00, D1
29CEA 7602              MOVEQ  #02, D3
29CEC 41FA0062          LEA    62(PC)!29D50, A0
29CF0 7001              MOVEQ  #01, D0
29CF2 4E42              TRAP   #2
29CF4 163C0031          MOVE.B  #31, D3
29CF8 6100000E          BSR  29D08
29CFC 7002              MOVEQ  #02, D0
29CFE 4E42              TRAP   #2
29D00 72FF              MOVEQ  #FF, D1
29D02 7600              MOVEQ  #00, D3
29D04 7005              MOVEQ  #05, D0
29D06 4E41              TRAP   #1
```

PRNT program:

```
29D08 45FA005B          LEA    5B(PC)!29D65, A2
29D0C 1483              MOVE.B   D3, (A2)
29D0E 45FA0046          LEA    46(PC)!29D56, A2
29D12 6100000C          BSR  29D20
29D16 45FA0024          LEA    24(PC)!29D3C, A2
29D1A 61000004          BSR  29D20
29D1E 4E75              RTS
```

CHAR program

```
29D20 121A              MOVE.B   (A2)+, D1
29D22 0C010000          CMPI.B   #0, D1
29D26 6700000A          BEQ  29D32
29D2A 61000008          BSR  29D34
29D2E 6000FFF0          BRA  29D20
29D32 4E75              RTS
```

DISP program:

```
29D34 76FF                  MOVEQ  #FF, D3
29D36 7005                  MOVEQ  #05, D0
29D38 4E43                  TRAP   #3
29D3A 4E75                  RTS
```

The data for these modules has been grouped under the base address $29DC3, with the the main routine message running from $29DC3 and the header message in PRNT running from $29D56, as shown in the following hexadecimal data 'dump':

```
29D3C 42 55 47 53 20 41 52 45 BUGS ARE
      20 4C 45 54 48 41 4C 20  LETHAL
29D4C 0D 0A 00 00 00 04 43 4F ......CO
      4E 5F 4D 45 53 53 41 47 N_MESSAG
29D5C 45 20 4E 55 4D 42 45 52 E NUMBER
      20 31 0D 0A 00 00 00 00 .1......
```

Note that $31, the code for '1', has been inserted by the PRNT subroutine into the correct memory location at $29D65. This was not, of course, done during assembly but during a test run of the program.

Subroutine Returns

As an example of what happens when a call is made to a subroutine, it will be useful to examine the contents of the stack at the point just after the CHAR subroutine has been called from PRNT. The ten addresses at the top of the stack contain the following data:

```
;-----------------------------------------------------
;   STACK TOP    DATA     COMMENT
;-----------------------------------------------------
     3DBC6      00 00    ;Original top of stack
     3DBC4      9C FC    ;Return address from PRNT
     3DBC2      00 02
     3DBC0      9D 16    ;Return address from CHAR
     3DBBE      00 02
;-----------------------------------------------------
```

As execution passes from the main routine to PRNT, the return address to the main program ($29CFC) is automatically placed on the stack: A7 first decrements by two bytes and $9CFC is stacked. A7 decrements by another two bytes and $0002 is stacked so that the complete return address is stacked in four bytes as $00029CFC. At that point the stack pointer, A7, is pointing to stack address $3DBC2. The stacked return address is the address of the instruction in the main routine which immediately follows the branch to PRNT. When a return (RTS) is made to the main routine from PRNT, the return address is automatically popped from the stack and loaded into the PC register so that the main routine will continue executing from address $29CFC.

Before that however, PRNT in its turn calls the CHAR subroutine and so the return address from CHAR to PRNT, $29D16, is added to the stack and the stack pointer is altered to point to $3DBBE. CHAR in its turn will call DISP and so the return address from DISP to CHAR will be stacked below the above data.

The return addresses are unstacked in reverse order and by the time execution has finally returned to the main program, the stack pointer is again pointing to the original top of stack at $3DBC6. Note that values removed from the stack are not physically removed. It is the position of the stack pointer which determines the top of the stack and when it has returned to $3DBC6 the return addresses, although still there, are effectively redundant because as far as the system is concerned the stack extends no further than the current 'top of stack' address. When further values are stacked they will overwrite any redundant data already stored there.

Linking Programs

This program is a particularly difficult one to follow on a first reading and could, of course, have been programmed much more simply. Its function however, is to show the relationships between separate program modules, demonstrating how parameters can be passed by register and by value from one subroutine to another.

When you have been able to follow the flow of execution you will appreciate how a multi-program system based on the 68000 can be implemented. Separate programs, subroutines, and sets of data

belonging to one or more different users can be loaded into memory and shared, so that the code for individual programs can be simplified. If commonly used sets of data, or utility subroutines such as text printing routines are stored as commonly accessible library items on disc, then any new program which needs to use them can access them without having to duplicate them in its own listing. If a particular program needs to make use of its own local parameters, these may be passed to a common library routine by one of the methods illustrated above.

Suppose for example that the above program had been assembled not as one complete program but as four, separately assembled modules. The main program contains the message and the message number in its data section and PRNT contains the message header in its own data section. To make code and data in separate programs and subroutines accessible to each other the assembler directives XDEF and XREF are used. The main program would contain the assembler directives: XREF, PRNT and XDEF MESS. XREF PRNT means that any reference to the subroutine labelled PRNT refers to a separately assembled external program module. XDEF MESS means that the message data defined in the main program may be accessed by other, external programs.

Likewise, PRNT would contain the directive XREF MESS meaning that any reference to MESS in the PRNT program refers to an address in a program module external to PRNT. It would also contain the directive XREF CHAR, meaning that the subroutine CHAR is an external reference. The CHAR routine would contain the directive XREF DISP.

These cross references will be be resolved by a linker program so that, when the main program is loaded from disc, any other modules cross referenced with the main program will also be loaded so that all the separate modules can function together as if they are all part of a single block of code and data, even if they are loaded into completely separate areas of memory.

Instead of the main program used above, suppose that you have written some other program which contains a list of numbered error messages, any one of which might need to be displayed on screen at some point. If PRNT, CHAR and DISP already exist as standard library programs then all you would need to do is to link them to your main program with a linker program, rather than incorporating their code in your main program. PRNT might contain the header message 'ERROR MESSAGE' rather than 'MESSAGE NUMBER'. When your program needs

to print one of its error messages then all you need to do is to call PRNT, passing the number and location of the appropriate message as parameters. You would then get a display message on the screen such as 'ERROR MESSAGE 23 FILE NOT FOUND'.

A major advantage of this is that PRNT, CHAR and DISP may also be available to other users or to other programs running in the system simultaneously. Any one of them can pass their own parameters to PRNT and obtain an appropriate message display.

Chapter 12

Stack Operations

In Chapter 5 it was explained that a stack is an area of memory in which items of data can temporarily be stored and which is also used to contain return addresses from subroutines. We looked at a number of stack operations including the stacking of registers, the passing of parameters by means of stacks and the setting up of stack frames. It was explained that the SP (stack pointer) register, which is register A7, points to the current 'top' of the stack and that other address registers may be used to point to data within the body of the stack. It was also explained that a stack normally extends *downwards* in memory from the stack top, although upward extending stacks can also be created if required, as can circular stacks (queues).

In the following program, one of the most common uses of the stack is demonstrated, in which the values contained in several of the data registers are pushed on to the stack prior to a call to a subroutine, so that they can later be retrieved and loaded back into the registers.

Registers D3, D4, D5 and D6 are first loaded with arbitrary byte values. These values are then preserved on the stack by the `MOVEM` (move multiple) instruction.

Following this the text of a message is stored on the stack which will then be passed to a subroutine called `READ` which will unstack it and print it to the screen.

Finally, a return is made to the main program and the values originally stacked from D3, D4, D5 and D6 are retrieved, in reverse order, by the `MOVEM` instruction.

```
;-------------------------------------------------------
;                    PROGRAM PROG5
;            PASSES A MESSAGE VIA THE STACK
;-------------------------------------------------------
;DEFINE CARRIAGE RETURN AND LINE FEED ASCII CONSTANTS
;-------------------------------------------------------
CR        EQU      13
LF        EQU      10
;-------------------------------------------------------
;OPEN CONSOLE CHANNEL ETC
;-------------------------------------------------------
          MOVEQ        #0,D1
          MOVEQ        #2,D3
          LEA.L        DEVICE,A0
          MOVEQ        #1,D0
          TRAP         #2
;-------------------------------------------------------
```

Initially, the ASCII codes for for carriage return and line feed (13 and 10) are assigned to the labels 'CR' and 'LF' by means of an EQU (=) directive.

Following this, the usual task ID and console channel operations are performed and then the main program starts:

```
;-------------------------------------------------------
;STACK SOME INITIAL DATA VALUES, TRANSFER THE MESSAGE ONTO
;THE STACK AND CALL THE UNSTACKING PROCEDURE
;-------------------------------------------------------
          MOVEQ     #1,D3        ;Load D3 with 1
          MOVEQ     #2,D4        ;Load D4 with 2
          MOVEQ     #3,D5        ;Load D5 with 3
          MOVEQ     #4,D6        ;Load D6 with 4
          MOVEM     D3-D6,-(A7) ;Store registers
                                 ;D3 to D6 on stack
          MOVEA.L   A7,A3        ;Copy stack pointer into
                                 ;register A3
          MOVEA.L   A7,A4        ;Copy stack pointer into
                                 ;register A4 as well
          LEA.L     MESS,A2      ;A2 points to address of
                                 ;the message
LOOP      MOV.B     (A2)+,D1     ;Load a message character
                                 ;code into D1
```

```
        CMPI.B    #0,D1         ;Is it zero (stop code)?
        BEQ       NEXT          ;Jump to NEXT if so (Z=1)
        MOVE      D1,-(A7)      ;Else store character
                                ;code on stack as a
                                ;word value
        BRA       LOOP          ;Jump back to 'LOOP'
NEXT    MOVEQ     #0,D1         ;Clear the D1 register
        MOVE      D1,-(A7)      ;Push it on  stack
        BSR       READ          ;Call the unstacking
                                ;subroutine
        MOVEA.L   A4,A7         ;Restore original SP
                                ;value
        MOVEM     (A7)+,D3-D6   ;Then retrieve
                                ;initial data
;-------------------------------------------------------
;MAIN PROGRAM TERMINATION ROUTINE
;-------------------------------------------------------
        MOVEQ     #2,D0
        TRAP      #2
        MOVEQ     #-1,D1
        MOVEQ     #0,D3
        MOVEQ     #5,D0
        TRAP      #1
;-------------------------------------------------------
```

Firstly, the D3 ,D4, D5 and D6 registers are loaded with some arbitrary initial values, representing data which you might wish to store for retrieval when the main operation is completed. These are pushed onto the stack as four words from the low order words of the four registers using the `MOVEM D3-D6,-(A7)` instruction. This instruction stacks each of the four registers in turn (from D6 to D3), automatically adjusting the stack pointer before each word is stacked by default, `MOVEM` without a size specifier implies `MOVEM.W` and if all four bytes of each register need to be stacked, `MOVEM.L` should be used. If necessary, all the contents of all the registers may be stacked using the `MOVEM` instruction, for example when you wish to call a subroutine and return with all the registers containing the same values as they did before the call. Again, if you begin to run out of spare registers, the `MOVEM` command can be used to stack the contents of some of them, freeing them for other purposes.

When we come to retrieve the message from the stack, instead of using the normal method of popping the stacked ASCII codes, using register

A7, we shall be retrieving them using another address register, A3. A3 therefore needs to be loaded with the current top of stack value from A7: `MOVEA.L A7,A3`. Note the use of the 'A' in `MOVEA`, indicating that the destination of the `MOVE` is an address register. A3 now points to the address in the stack which will eventually contain the first message character data. This will become clearer later on when we look at the destacking subroutine.

When the message has been printed, the stack pointer will no longer be anywhere near the data which we stacked from registers D3 to D6. If we plan to retrieve this data then we need to make a further copy of the current A7 value so that we can restore the stack pointer later on: `MOVEA.L A7,A4`.

Following this the A2 register is loaded with the address of the start of the message data and the following instruction, labelled 'LOOP', transfers a message character into the low byte of D1 and increments A2 to point to the next message character: `MOVE.B (A2)+,D1`.

D1 is then checked to see if it contains the stop code, 0. If it does (i.e. Z=1) then execution moves on to the instruction labelled 'NEXT' because of the `BEQ NEXT` instruction. If D1 does not contain 0 then it must contain a message code and can therefore be stacked.

The `MOVE D1,-(A7)` instruction stacks the code as a word value and adjusts the stack pointer. The `BRA LOOP` instruction then returns execution back to the instruction labelled 'LOOP'.

When all the message characters have been pushed onto the stack in this way, execution moves on to the instruction labelled 'NEXT', which moves the value 0 into the whole of D1. D1 is then stacked so that its contents can be used as a stop code when the stack contents are retrieved.

Following this the unstacking subroutine 'READ' is called and on return from this subroutine the original contents of D3, D4, D5 and D6, which were stacked at the beginning of the program, are retrieved back into those registers using the `MOVEM (A7)+,D3-D6` instruction in the order D3 to D6. However, since the stacking of the message text the A7 register no longer points to this data. We therefore need to restore A7 to its original value by loading it with the original SP value of which a copy is held in A4: `MOVEA.L A4,A7`.

Finally, the usual termination routine rounds off the main section of the program.

At the point where the `READ` subroutine has just been called from the main program the stack contains the following values:

	Stack address	*Codes*	*ASCII*
Original values	3DBC4:	00 04	
from registers	3DBC2:	00 03	
D3, D4, D5 and D6	3DBC0:	00 02	
A3 and A4 point here:	3DBBE:	00 01	
Message starts here:	3DBBC:	00 4D	M
	3DBBA:	00 45	E
	3DBB8:	00 53	S
	3DBB6:	00 53	S
	3DBB4:	00 41	A
	3DBB2:	00 47	G
and so on down to:			
	3DB80:	00 54	T
	3DB7E:	00 41	A
	3DB7C:	00 43	C
	3DB7A:	00 4B	K
	3DB78:	00 0D	[cr]
	3DB76:	00 0A	[lf]
	3DB74:	00 00	[stopcode]
Return address to			
main program:	3DB72:	9D 20	
A7 points here:	3DB70:	00 02	

The first four words on the stack contain the values 1, 2, 3 and 4 which we stacked at the beginning of the program. They were stacked in predecrement mode in the order D6 to D3 and are unstacked in post increment mode in the order D3 to D6. The next word in the stack at address $3DBBC contains the ASCII code of the first character in the message and is pointed to by the A3 register which was set up in the main program above. Next come the ASCII codes for the next characters in the message, finishing with the carriage

return, line feed and stop codes. Lastly, the return address back from 'READ' to the main program is at stack addresses $3DB70 to $3DB73. A7 points to $3DB70 since this was the address of the last word stacked.

It should be clear from this that if we were to unstack the data pointed to by A3 (effectively A3–2) into D1, print the character in D1 to the screen, subtract 2 from A3, load D1 again from the address pointed to by A3 and so on all the way down the stack, we can retrieve and print the whole message without altering the value of A7. In other words, the message is effectively in a *stack frame* within the stack and can be referenced via its base pointer, A3.

The individual operations of the READ subroutine are as follows:

```
;-------------------------------------------------------
;UNSTACK THE MESSAGE, TRANSFER EACH CHARACTER TO THE DISPLAY
;PROCEDURE & RETURN TO PROG5 WITH MESSAGE REMOVED FROM STACK
;-------------------------------------------------------
READ    MOVE        -(A3),D1    ;Retrieve a character from
                                ;the stack
        CMPI.B      #0,D1       ;Compare it with the value
                                ;0 (i.e. is it the stop
                                ;code?)
        BEQ         EXIT        ;Jump if so (Z=1) to 'EXIT'
        BSR         DISP        ;Else call display
                                ;subroutine
        BRA         READ        ;Branch back to 'READ'
EXIT    RTS                     ;Return to the main program
;-------------------------------------------------------
;DISP SUBROUTINE
;-------------------------------------------------------
DISP    MOVEQ       #-1,D3
        MOVEQ       #5,D0
        TRAP        #3
        RTS                     ;Return to main program
;-------------------------------------------------------
```

The first instruction in the READ subroutine, MOVE -(A3),D1, copies a character code into D1 from the word location in the stack which is indexed by A3. A3 is first autodecremented by two bytes. In other words, just as A7 was used in predecrement mode to stack the message characters, the A3 register, starting at exactly the same position, is being used in predecrement mode to retrieve them in the same order.

The next instruction, `CMPI.B #0,D1`, compares the contents of D1 with zero and if the result is zero (Z=1) then D1 contains the stop code, otherwise it contains a character code. The next instruction therefore, `BEQ EXIT`, determines whether execution jumps to the end of the routine or carries on with the following instruction.

Then a call to the subroutine '`DISP`' is made, which will display the character in D1 on the screen.

On returning from this call a jump is made back to the instruction labelled '`READ`'.

Finally, when all the characters have been printed, the `RTS` instruction will return execution back to the main program.

At this point the program data is defined:

```
;-----------------------------------------------------------
;DATA SECTION
;-----------------------------------------------------------
MESS      DC.B    'MESSAGE NUMBER 2',CR,LF
          DC.B    'PASSED VIA STACK',CR,LF,0
DEVICE    DC.W    4
          DC.B    'CON_'
          END
;-----------------------------------------------------------
```

The message is defined as a series of bytes representing the ASCII codes for the individual characters of the message, terminating with a carriage return, line feed and stop code: '0'. The base address of the message is labelled '`MESS`'.

The object code listing for `PROG5` is as follows:

```
;-----------------------------------------------------------
;ADDRESS  CODE                MNEMONICS
;-----------------------------------------------------------
29CE8 7200                   MOVEQ  #00, D1
29CEA 7602                   MOVEQ  #02, D3
29CEC 41FA0086               LEA    86(PC)!29D74, A0
29CF0 7001                   MOVEQ  #01, D0
29CF2 4E42                   TRAP   #2
29CF4 7601                   MOVEQ  #01, D3
29CF6 7802                   MOVEQ  #02, D4
```

```
29CF8 7A03              MOVEQ  #03, D5
29CFA 7C04              MOVEQ  #04, D6
29CFC 48A71E00          MOVEM.W  /D3 /D4 /D5 /D6 ,-(A7)
29D00 264F              MOVE.L   A7, A3
29D02 284F              MOVE.L   A7, A4
29D04 45FA0048          LEA   48(PC)!29D4E, A2
29D08 121A              MOVE.B   (A2)+, D1
29D0A 0C010000          CMPI.B   #0, D1
29D0E 67000008          BEQ  29D18
29D12 3F01              MOVE.W   D1,  -(A7)
29D14 6000FFF2          BRA  29D08
29D18 7200              MOVEQ  #00, D1
29D1A 3F01              MOVE.W   D1,  -(A7)
29D1C 61000014          BSR  29D32
29D20 2E4C              MOVE.L   A4, A7
29D22 4C9F0078          MOVEM.W (A7)+,/D3 /D4 /D5 /D6
                                              29D26 7002
                        MOVEQ  #02, D0
29D28 4E42              TRAP  #2
29D2A 72FF              MOVEQ  #FF, D1
29D2C 7600              MOVEQ  #00, D3
29D2E 7005              MOVEQ  #05, D0
29D30 4E41              TRAP  #1

READ:-

29D32 3223              MOVE.W    -(A3), D1
29D34 0C010000          CMPI.B   #0, D1
29D38 6700000A          BEQ  29D44
29D3C 61000008          BSR  29D46
29D40 6000FFF0          BRA  29D32
29D44 4E75              RTS

DISP:-

29D46 76FF              MOVEQ  #FF, D3
29D48 7005              MOVEQ  #05, D0
29D4A 4E43              TRAP  #3
29D4C 4E75              RTS
;-------------------------------------------------------
```

Again, READ could be a separately assembled library subroutine which can unstack and print any message passed to it. The only parameter required by READ is the base of the stacked message (in A3).

Chapter 13

Data Structures and Data Processing

In Chapter 6 we looked at some of the ways in which blocks of data such as arrays can be accessed by means of the indirect addressing modes. This is a crucial aspect of programming and in this chapter we are going to work through a complex example in close detail.

This is a fairly long program which shows how data stored in an array can be accessed and processed in a number of different ways. We shall be using our earlier petrol consumption model, since you will already be familiar with the nature of the data we shall be dealing with, and in the process we shall be covering a number of new topics including multiplication, division and bit shifting and rotation.

In this program the fuel consumption and mileage figures for a two year period are stored in the data section in the following order:

Fuel consumption for year 1 (12 separate months)
Mileage for year 1 (12 separate months)
Fuel consumption for year 2 (12 separate months)
Mileage for year 2 (12 separate months)

The program will add up the fuel and mileage figures for both years and print the totals to the screen. The average monthly consumption and mileage will also be worked out, together with the overall miles per gallon calculation. The consumption and mileage for May in each year will then be added and printed out and finally, there will be a procedure which allows you to key in the first three letters of any month in order to obtain the total mileage for the month over the two year period. The format of the final, printed output is as follows:

```
TOTAL CONSUMPTION:   524.0
TOTAL MILEAGE:       10327.0
AVERAGE CONSUMPTION: 21.83
AVERAGE MILEAGE:     430.29
MILES PER GALLON:    19.70
MAY CONSUMPTION:     40.0
MAY MILEAGE:         853.0
JAN    780
FEB    807
JUN    762
JUL    944
AUG    898
```

The general structure of the program is as follows:

1 A pointer register will be used to access each of the values in the four data arrays. The added totals will be placed in a separate array labelled `TOTALS`.

2 The consumption and mileage totals will be used to work out the monthly averages and the miles per gallon figure. These will also be placed in the array `TOTALS`. The totals for May will then be added and placed in `TOTALS`.

3 The totals contained in `TOTALS` will then be converted from binary values to floating point decimal values and pushed on to the stack.

4 The text messages will be transferred from the data area to the screen, each followed by the calculated results which are popped from the stack, converted to ASCII characters and displayed on the screen along with the messages.

5 The month by month mileage totals will be calculated and transferred to an array labelled `SUBMIL`.

6 The data in `SUBMIL` will be accessed and printed in response to month names input from the keyboard.

```
;-------------------------------------------------------
;             PROGRAM ENTITLED PROG6
;    ANALYSIS OF CONSUMPTION AND MILEAGE DATA
;-------------------------------------------------------
;FIRST, DEFINE CONSTANT VALUES WITH LABELS
;-------------------------------------------------------
MAY       EQU    8          ;Offset for may figures
CR        EQU    13         ;Carriage return code
LF        EQU    10         ;Line feed code
POINT     EQU    "."        ;Decimal point code
;-------------------------------------------------------
;OPEN CONSOLE CHANNEL ETC
;-------------------------------------------------------
          MOVEQ          #0,D1
          MOVEQ          #2,D3
          LEA.L          DEVICE,A0
          MOVEQ          #1,D0
          TRAP           #2
;-------------------------------------------------------
```

Initially, the ASCII codes for carriage return and line feed are defined as labelled constants, along with the decimal point code and an index offset for the May data.

May is the fourth month, counting from zero, and since the data is stored in word lengths its offset will therefore be 8. `CR`, `LF` and `POINT` are the labels given to the ASCII character codes for carriage return, line feed and the decimal point. Note that the decimal point is entered in character form and the assembler will work out its ASCII code automatically.

The usual initialization instructions come after this and then the main program begins as follows:

```
;-------------------------------------------------------
;GET CONSUMPTION TOTAL FOR ALL MONTHS OF BOTH YEARS
;-------------------------------------------------------
          LEA.L     GALLS1,A2    ;Base of consumption data
                                 ;(address of GALLS1)
          MOVEQ     #11,D4       ;Month count (less 1)
          CLR.L     D5           ;Clear D5 (set it to zero)
LOOP1     ADD       (A2)+,D5     ;Add data item from GALLS1
                                 ;to D5
```

```
        DBRA    D4,LOOP1    ;Repeat LOOP1 while D4>-1
        LEA.L   GALLS2,A2   ;A2 Points to next year
                (GALLS2)
        MOVEQ   #11,D4      ;Month count (less 1)
LOOP2   ADD     (A2)+,D5    ;Add data item from array
                            ;to D5.
        DBRA    D4,LOOP2    ;Repeat LOOP2 while D4>-1
        LEA.L   TOTALS,A4   ;A4 points to 'TOTALS'
                            ;array
        MOVE.W  D5,(A4)     ;Store consumption total
                            ;in TOTALS array
        MOVE.W  D5,-(A7)    ;Stack consumption total
;-------------------------------------------------------
;GET MILEAGE TOTAL FOR ALL MONTHS OF BOTH YEARS
;-------------------------------------------------------
        LEA.L   MILES1,A2   ;Base of mileage data
                            ;(address of MILES1)
        MOVEQ   #11,D4      ;Month count (less 1)
        CLR.L   D5          ;Clear D5 (set it to zero)
LOOP3   ADD     (A2)+,D5    ;Add data item from array
                            ;to D5
        DBRA    D4,LOOP3    ;Repeat LOOP3 while D4>-1
        LEA.L   MILES2,A2   ;A2 Points to next year
                (MILES2)
        MOVEQ   #11,D4      ;Month count (less 1)
LOOP4   ADD     (A2)+,D5    ;Add data item from array
                            ;to D5.
        DBRA    D4,LOOP4    ;Repeat LOOP4 while D4>-1
        ADDQ    #4,A4       ;A4 points to 'TOTALS+4'
        MOVE.W  D5,(A4)     ;Store mileage total
                            ;in TOTALS array
;-------------------------------------------------------
```

In this section the consumption and mileage figures for the two years are added together and stored in the TOTALS array. A2 is loaded with the first address of the data and is used as an indirection register to retrieve each item of data and add it to the D5 register. D4 is loaded with 11, to count off each month (11 to –1), and A2 is auto-incremented by two after each addition because each item of data is stored as a word and the operations are of length '.W'.

After the first 12 items, A2 is repositioned to point to the next year and the data continues to be added to D5 as before. Finally, the 24

month total is loaded into the first two addresses of the array TOTALS. Since the low order word of D5 contain the total its contents are automatically loaded by MOVE.W D5,(A4) into both the address pointed to by A4 plus the one following. The total consumption is also stored temporarily on the stack by the MOVE.W D5,-(A7) instruction. The above steps are then repeated for the mileage figures, which are stored in the fifth and sixth addresses of the TOTALS array (TOTALS+4 and TOTALS+5), for reasons which will be made clear later on.

In the next section the average consumption and mileage figures will be calculated:

```
;-------------------------------------------------------
;CALCULATE AVERAGE CONSUMPTION
;-------------------------------------------------------
    ADDQ      #4,A4             ;Point A4 to next TOTALS
                                ;location (TOTALS+8)
    MOVE.W    (A7),D6           ;Retrieve consumption total
                                ;from stack into D6
    MOVE.W    D6,-(A7)          ;Store a copy back on stack
    MOVE.W    D5,-(A7)          ;Stack mileage total, which
                                ;is still in D5
    DIVU      DNUM1,D6          ;Divide D6 by 24
    MOVE.W    D6,(A4)           ;Store quotient from lo word
                                ;of D6 in TOTALS
    MOVEQ     #16,D2            ;D2 holds shift count
    LSR.L     D2,D6             ;Shift D6 right according to
                                ;count in D2
;
;Now convert the remainder to a decimal fraction
;
    MULU      CENT,D6           ;Multiply D6 by 100
    DIVU      DNUM1,D6          ;Divide by 24 again
    MOV.W     D6,2(A4)          ;Store in TOTALS array
;-------------------------------------------------------
;CALCULATE AVERAGE MILEAGE
;-------------------------------------------------------
    ADDQ      #4,A4             ;Point A4 to next free
                                ;TOTALS (TOTALS+12)
    DIVU      DNUM1,D5          ;Divide D5 by 24
    MOVE.W    D5,(A4)           ;Store quotient from lo word
                                ;of D5 in TOTALS
    MOVEQ     #16,D2            ;D2 holds shift count
```

```
    LSR.L     D2,D5             ;Shift D5 right according to
                                ;count in D2
;
;Now convert the remainder to a decimal fraction
;
    MULU      CENT,D5           ;Multiply D5 by 100
    DIVU      DNUM1,D5          ;Divide by 24 again
    MOV.W     D5,2(A4)          ;Store in TOTALS array
;-------------------------------------------------------
```

Firstly, A4 is incremented to point to the offset of the next free space in the TOTALS array (TOTALS+8) to store the average consumption figure. At this point we have the total mileage in TOTALS+4 and in D5. The total consumption is both in TOTALS+0 and on the top of the stack. The consumption figure is popped from the stack into D6. It does not matter that it was originally pushed from D5 because stacked data can be popped into any of the general registers. We still need a copy of the consumption figure in the stack, so it is popped using the address register indirect mode *without* postincrement. Now the consumption figure is in TOTALS, in D6 and on top of the stack.

Next the mileage total is pushed onto the stack from D5. We now have the mileage and consumption figures in the TOTALS array, on the stack and in the D5 and D6 registers. To calculate the average consumption we need to divide the consumption total in D6 by the constant located at the address labelled DNUM1, which is defined in the data section as 24. This is performed by DIVU DNUM1,D6. The quotient of the division operation ends up in the low order word of D6 and this is transferred directly to address TOTALS+8 and TOTALS+9 by the MOVE.W D6,(A4) instruction. The remainder of the division ends up in the high order word of D6, but, since we normally prefer our remainders to be expressed as decimal fractions we need to convert it before storing it away. This is done by multiplying D6 by 100 and dividing by 24. However it is first necessary to shift the division remainder from the high to the low order word of D6 and to ensure that the high order word is zeroed. The bits in D6 need to be shifted to the right 16 times with zeroes being passed into the high order word during each shift. The shift count is loaded into D2: MOVEQ £16,D2 and the shift is performed by the logical shift right instruction LSR: LSR.L D2,D6. The multiplication by 100 is then performed by MULU CENT,D6 and the division by 24 by DIVU DNUM1,D6. CENT is defined in the data section as 100.

Ignoring the remainder from this second division, we load the contents of the low order word of D6 (the quotient from the second division) into the TOTALS array at TOTALS+10 and TOTALS+11, pointed to by A4+2. This is in fact the remainder of the average consumption calculation.

If this process is not clear, the following calculation example shows what has just been performed. We shall assume that the actual consumption total is 205:

Instruction Mnemonic	Register D6 High	Register D6 Low	Function
	0	205	
DIVU DNUM1,D6	13	8	D6/24=8 rem'dr 13
MOVE.W D6,(A4)	13	8	store 8 in TOTALS
MOVEQ #16,D2	13	8	Shift count in D2
LSR.L D2,D6	0	13	Shifts remainder over
MULU CENT,D6	0	1300	100 times D6 = 1300
DIV DNUM1	4	54	D6/24=54 remainder 4
MOVE.W D6,2(A4)			store 54 in TOTALS
	Therefore 205/24=8.54		

The figure we end up with in TOTALS in this example is 8.54: the '8' occupying 2 addresses and the '54' occupying the next two addresses.

The A4 register is now incremented by 4 to point to the next free space in the TOTALS array (TOTALS+12), which will be used for storing the average mileage.

The next stage is to convert the mileage total, currently in D5, into an average in the same way as the consumption total above. The result is again deposited in the TOTALS array at TOTALS+12, TOTALS+13, TOTALS+14 and TOTALS+15.

We are now ready to calculate the miles per gallon figure.

```
;-------------------------------------------------------
;CALCULATE MILES PER GALLON
;-------------------------------------------------------
    CLR.L    D6              ;Clear register D6
    ADDQ     #4,A4           ;Point A4 to next free
                             ;TOTALS (TOTALS+16)
    MOVE.W   (A7)+,D6        ;Retrieve total mileage from
                             ;stack to D6
    MOVE.W   (A7)+,D5        ;Retrieve total consumption
                             ;from stack to D5
    DIVU     D5,D6           ;Divide miles by consumption
                             ;(D6 by D5)
    MOVE.W   D6,(A4)         ;Store quotient in TOTALS
    LSR.L    D2,D6           ;Shift D6 right according to
                             ;count in D2

;Now convert remainder to a decimal fraction

    MULU     CENT,D6         ;Multiply by 100
    DIVU     D5,D6           ;Divide by consumption
                             ;(D6 by D5)
    MOVE.W   D6,2(A4)        ;Store in TOTALS array
;-------------------------------------------------------
```

In the above section D6 is zeroed to ensure that it contains no superfluous data and the A4 register is again incremented by 4 to point to the next free TOTALS space (TOTALS+16). The mileage total is popped from the stack into D6 and the consumption total is popped into D5. This time we need to divide the mileage total by the consumption total to give the miles per gallon figure. The same division operations as before are performed except that this time, D5 is used as the divisor. Finally, the result is placed in the TOTALS array at TOTALS+16, TOTALS+17, TOTALS+18 and TOTALS+19.

Next, the consumption figures for May in each year are retrieved, added together and stored in the TOTALS array:

```
;-------------------------------------------------------
;CALCULATE TOTAL MAY CONSUMPTION FOR BOTH YEARS
;-------------------------------------------------------
    LEA.L    GALLS1,A2       ;Start of data
    MOVE.W   MAY(A2),D5      ;Get GALLS1 May in D5
    ADDA     #48,A2          ;Point A2 to GALLS2 May
```

```
    ADD       MAY(A2),D5    ;Add GALLS2 May to D5
    ADDQ      #4,A4         ;Point A4 to next free
                            ;TOTALS (TOTALS+20)
    MOVE.W    D5,(A4)       ;Store May galls in TOTALS
;-------------------------------------------------------
;CALCULATE TOTAL MAY MILEAGE FOR BOTH YEARS
;-------------------------------------------------------
    LEA.L     MILES1,A2     ;Start of data
    MOVE.W    MAY(A2),D5    ;Get MILES1 May in D5
    ADDA      #48,A2        ;Point A2 to MILES2 May
    ADD       MAY(A2),D5    ;Add MILES2 May to D5
    ADDQ      #4,A4         ;Point A4 to next free
                            ;TOTALS (TOTALS+24)
    MOVE.W    D5,(A4)       ;Store May miles in TOTALS
;-------------------------------------------------------
```

A2 is loaded with the GALLS1 address and the first year May figure is copied into D5 from the address pointed to by A2+MAY. MAY, you will recall, is a labelled constant equal to the value 8, since the May data is at offset 8 in GALLS1, counting from zero. A2 is then incremented to point to the second year May figure. Note that because the original data is stored as word values, A2 is incremented by 48 rather than 24. Both years' May totals are added and stored as a word value in the TOTALS array at TOTALS+20 and TOTALS+21.

The May mileage figures are added and stored in the same way as the consumption figures, except that A2 is initially given the offset value of MILES1 and the results are stored in TOTALS+24 and TOTALS+25.

We are now nearly ready to take the stored totals and print them to the screen with the appropriate text messages. Firstly, the figures are transferred from TOTALS onto the stack:

```
;-------------------------------------------------------
;MOVE CONTENTS OF 'TOTALS' ARRAY  ON TO THE STACK
;-------------------------------------------------------
        LEA.L     TOTALS,A2     ;Point to base of TOTALS
RES     CLR.L     D5            ;Clear register D5
        MOVE.W    2(A2),D6      ;Get a decimal fraction
                                ;from TOTALS

;
;First convert binary data from TOTALS into individual
;digits so that they can be converted to ASCII and printed:
```

```
;
LOOP5   DIVU      DNUM2,D6      ;Divide by 10
        SWAP      D6            ;Swap halves of D6
        MOVE.W    D6,-(A7)      ;Stack remainder
        ADDQ      #1,D5         ;Increment digit count
        CLR.W     D6            ;Clear lo word of D6
        SWAP      D6            ;Swap halves of D6
        CMPI      #0,D6         ;Is D6=0?
        BNE       LOOP5         ;Repeat LOOP5 if not (ZF=0)
        MOVE.W    POINT,D6      ;Get ASCII code for decimal
                                ;point
        SUBI.W    48,D6         ;Subtract 48 from it
        MOVE.W    D6,-(A7)      ;Stack it after remainder
        MOVE.W    (A2),D6       ;Get a quotient from TOTALS
        SWAP      D6            ;Swap halves of D5
LOOP6   DIVU      DNUM2,D6      ;Divide D6 by 10
        SWAP      D6            ;Swap halves of D6
        MOVE.W    D6,-(A7)      ;Stack the remainder
        ADDQ      £1,D5         ;Increment digit count
        CLR.W     D6            ;Clear lo word of D6
        SWAP      D6            ;Swap halves of D6
        CMPI      £0,D6         ;Is D6=0?
        BNE       LOOP6         ;Repeat LOOP6 if not (ZF=0)
        MOVE.W    D5,D6         ;Copy digit count into D6
        SWAP      D5            ;Swap halves of D5
        ADD.W     D5,D6         ;Rest of digit count added
        MOVE.W    D6,-(A7)      ;Stack total digit count
        LEA.L     RESPNT,A6     ;Address of RESPNT in A6
        MOVE.L    A2,(A6)       ;Store current TOTALS offset
                                ;in reserved address RESPNT
;-------------------------------------------------------------
```

Firstly, A2 is loaded with the first address of the TOTALS array. At this point the totals are stored in the TOTALS array in 4-byte chunks: the first 2 bytes being the quotient of each result and the second 2 bytes being the decimal remainder (if any) of each result. Our objective is to break down each number into separate values so that, for example, quotient 32 remainder 85 (i.e. 32.85) would become 5 separate values: '3', '2', '.', '8' and '5'. These values will be pushed separately on to the stack (in reverse order) and a count will be kept of the total number of values stacked.

D5 is cleared so that it can be used to count the individual values and the first total first is loaded into D6 from TOTALS. Firstly, we load the remainder value of the total because the result will be pushed onto the stack in reverse order so that it can be popped off and printed in the correct order.

The problem we now have is that the first result, the consumption total remainder in D6, is a 16-bit binary integer, whereas we actually want to print it as a floating point decimal number. The conversion is easily achieved by repeatedly dividing the number in D6 by 10 (stored at address DNUM2) and pushing the *remainders* on to the stack. This process continues, in program LOOP5, until D6=0. The BNE (branch if not equal) instruction detects this and continuously loops back to LOOP5 until D6=0.

Every time D6 is divided by 10 the remainder goes into its high word and the quotient into its low word. The SWAP D6 instruction swaps over these two words so that the MOVE.W D6,--(A7) instruction can be used to stack the remainder. The D5 (counter register) is incremented by 1 after each division and swap and the low order (remainder) word of D6 is cleared. Another SWAP then transfers the quotient back into the low order word so that another division can take place. The following example shows how this process works, with D6 initially containing the remainder value '85':

Instruction Mnemonic	Register D6 High	Register D6 Low	Function
	0	85	
DIVU DNUM2,D6	5	8	D6/10=8 remainder *5*
SWAP D6	8	5	Words swapped over
MOVE.W D6,-(A7)	8	5	Stack the value 5
CLR.W D6	8	0	Clear low byte
ADDQ #1,D5	8	0	Increment counter
SWAP D6	0	8	Words swapped over
CMPI #0,D6	0	8	Does D6=0? (no)
BNE LOOP5 (this causes a repeat if D6>0, as follows):			
DIVU DNUM2,D6	8	0	D6/10=0 remainder 8
SWAP D6	0	8	Words swapped over
MOVE.W D6,-(A7)	0	8	Stack the value 8
CLR.W D6	0	0	Clear low byte

`ADDQ #1,D5`	0	0	Increment counter
`SWAP D6`	0	0	Words swapped over
`CMPI #0,D6`	0	0	Does D6=0? Yes Therefore the stack holds '5' and '8' which will be printed out in reverse as '85'.

The decimal point needs to be stacked now and this is currently stored as an ASCII value at the address pointed to by the label `POINT`. This is moved into D6 and 48 is subtracted from it before it is pushed onto the stack. The 48 will be added to it again later when all the converted digits are translated to ASCII codes before printing them.

The total number of digits stacked is now in the low order word of the counter register D5 and this is temporarily transferred to the high order word: `SWAP D5`.

After this the *quotient* of the consumption total is moved into D6 and the above conversion process is repeated in `LOOP6`, dividing D6 by 10 until it equals 0 and pushing each remainder on to the stack.

After `LOOP6`, the total count of quotient digits is in the low order word of D5 and the total number of remainder digits is in the high order word. The low order word of D5 is transferred to D6 and the high order word of D5 is swapped with its low order word: `SWAP D5`. Then the low order word is added to D6 so that D6 now holds the count of the total number of stacked digits in the decimal number. This digit count is pushed onto the stack for later use.

After printing these newly stacked digits we will need to come back and repeat the above process with the next result in the `TOTALS` array. We therefore need to store the offset of the current result (contained in A2), so the contents of A2 are copied into a reserved memory address labelled '`RESPNT`' for later retrieval. First the address of `RESPNT` is loaded into A6 and the instruction `MOVE.L A2,(A6)` stores the contents of A2 in the `RESPNT` address.

At this point in the program the contents of the stack are as follows:

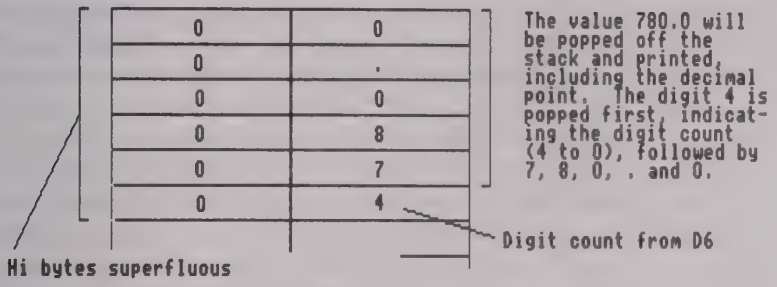


FIGURE 13-1. First result on stack, prior to printing.

The next task is to print the first message to the screen, which is "TOTAL CONSUMPTION: " stored at address MES1:

```
;-------------------------------------------------------
;PRINT A HEADER MESSAGE
;-------------------------------------------------------
       LEA.L     MES1,A2       ;A2 points to base of text
       MOVE.L    MESSOFF,D2    ;Offset of current message
                               ;into D2 from 'MESSOFF'
       MOVEQ     #20,D4        ;Text length count (less 1)
                               ;in register D4
LOOP7  MOVE.B    0(A2,D2),D1   ;Get text character in D1
       BSR       DISP          ;Display character on screen
       ADDQ      #1,D2         ;Point to next character
       DBRA      D4,LOOP7      ;Repeat LOOP7 while D4>-1
       LEA.L     MESSOFF,A6    ;Get address of MESSOFF
       MOVE.L    D2,(A6)       ;Store offset of next
                               ;message in reserved
                               ;address 'MESSOFF'
;-------------------------------------------------------
```

The address offset of 'MES1' is copied into A2 and D2 is loaded with the index offset contained in the address pointed to by the label MESSOFF (initially zero). D4 is then loaded with the message length (less 1), which is 20. Each message will be defined in the data section as 21 bytes, including spaces.

LOOP7 copies each of the ASCII codes of the message into D1 using MOVE.B 0(A2,D2),D1 and calls the DISP subroutine to print them one

by one to the screen. After they have all been printed, the index offset currently contained in D2 is stored back in address MESSOFF for use with the next message. Having printed the message heading we are now ready to transfer the actual consumption total to the screen.

```
;-------------------------------------------------------
;PRINT THE CORRESPONDING FIGURES
;-------------------------------------------------------
       MOVE.W    (A7)+,D4     ;Retrieve result digit total
                              ;from stack into D4
LOOP8  CLR.L     D1           ;Clear register D1
       MOVE.W    (A7)+,D1     ;Get a digit from stack
       ADDI.B    #48,D1       ;Convert it to ASCII
       BSR       DISP         ;Display it
       DBRA      D4,LOOP8     ;Repeat while D4>-1
       MOVE.B    CR,D1        ;Carriage return code goes
                              ;in D1
       BSR       DISP         ;Print it
       MOVE.B    LF,D1        ;Line feed code goes in D1
       BSR       DISP         ;Print it
       LEA.L     MESCNT,A6    ;Address of MESCNT in A6
       ROL       (A6)         ;Rotate header message count
                              ;which is stored in 'MESCNT'
       BCC       MONTHS       ;Branch to MONTHS routine if
                              ;all results printed
       LEA.L     RESPNT,A2    ;Else retrieve TOTALS offset
                              ;into A2 from RESPNT
       ADDQ      #4,A2        ;Point to offset of next
                              ;result in TOTALS array
       BRA       RES          ;Branch back to 'RES'
;-------------------------------------------------------
```

The first step is to retrieve the count of the number of digits in the consumption total, which is popped off the stack into D4. This will be used as a loop count value.

Next, the first result digit is popped off the stack into D1, after which 48 is added to it to convert it into its ASCII form. Note that the decimal point code, which is stacked among the digits, will also have 48 added to restore it to its original ASCII value when its turn comes to be popped. The ASCII digit is now held in D1 and a call is made to the DISP subroutine to print it to the screen. This process is repeated until D4=–1, after which the carriage return and line feed codes are also passed to DISP for printing.

At this point we need to check to see whether all the results and messages have been printed and this is done by *rotating* the bits in the address pointed to by the label MESCNT to the left, using the instructions LEA.L MESCNT,A6 to obtain the address of MESCNT and ROL (A6) to perform the rotation. This rotates the binary number in MESCNT once to the left, depositing the high order bit into the carry flag and also transferring it to the low order bit position. If a zero bit is thus rotated into the carry flag, then all the results have been printed and the BCC instruction (branch if carry clear) transfers execution to MONTHS; the start of the next routine. If a set bit is rotated into the carry flag then there are further results to print and the BRA instruction loops execution back to RES, where the next message and result will be processed. Before branching to RES, the value contained in the address pointed to by the label RESPNT is copied into A2. This contains the index offset within the TOTALS array of the result which has just been printed. This offset was stored in RESPNT earlier in the program.

A2 is then incremented by 4 to point to the next result within the TOTALS array.

Finally, when all the results have been printed, the program moves on to the next routine, MONTHS, which will allow the monthly totals to be printed in response to key inputs.

In this section we first need to add the month by month mileage figures from year 2 to those from year 1 and stores them in a separate array called SUBMIL, as follows:

```
;-------------------------------------------------------
;FIRST ADD MONTHLY MILEAGE TOTALS AND STORE IN SUBMIL ARRAY
;-------------------------------------------------------
MONTHS LEA.L     MILES1,A2     ;Point A2 to MILES1 data
       LEA.L     MILES2,A5     ;Point A5 to MILES2 data
       LEA.L     SUBMIL,A4     ;Point A4 to SUBMIL array
       MOVEQ     #11,D4        ;D4 to count off months
LOOP9  CLR.L     D5            ;Clear D5 register
       ADD.W     (A2)+,D5      ;Add word pointed to by A2
                               ;to D5 and add 2 to A2
       ADD.W     (A5)+,D5      ;Add word pointed to by A5
                               ;to D5 and add 2 to A5
       MOVE.W    D5,(A4)+      ;Transfer total to SUBMIL
       DBRA      D4,LOOP9      ;Repeat LOOP9 while D4>-1
;-------------------------------------------------------
```

This operation is very simple. The addresses of MILES1 and MILES2 are loaded into A2 and A5 respectively, while the destination address, SUBMIL, is loaded into A4. D4 will count off each of the 12 monthly subtotals and so it is loaded with the value 11 to count from 11 to –1.

D5 is cleared and then the word data at the address pointed to by A2 is added to it, after which A2 is automatically incremented by 2. The word data pointed to by A5 is then added to D5 so that D5 now contains the sum of the mileage figures for a particular month for each of the two years. A5 is also autoincremented by 2.

The contents of D5 are then copied into SUBMIL by MOVE.W D5,(A4)+ and A4 is also autoincremented by two. A2 and A5 now point to the next month's figures for each of the two years and A4 points to the next free space in SUBMIL.

The DBRA instruction decrements and checks the D4 counter and loops back to LOOP9 until all the data has been added and transferred.

Now that the mileage subtotals are stored in the SUBMIL array, we can consider what method we might use for accessing any one of these subtotals by inputting the name of a particular month from the keyboard and outputting the corresponding subtotal to the screen.

Like the DISP procedure used in the previous programs, the particular method of inputting text from the keyboard will depend on which operating system your computer uses. In this case we shall assume that it is done by using an operating system TRAP routine which transfers characters from the keyboard into a buffer whose base address is contained in register A1, as follows:

```
;-------------------------------------------------------
;GET 4 CHARACTERS INPUT FROM THE KEYBOARD
;-------------------------------------------------------
KEY     CLR.L     D5            ;Clear register D5
        LEA.L     KEYBUF,A1     ;Address of buffer in A1
        MOVEA.L   A1,A3         ;Copy into A3
        MOVEQ     #4,D2         ;Character count in D2
        MOVEQ     #2,D0         ;O/S function code in D0
        TRAP      #3            ;O/S trap call
        MOVEQ     #2,D4         ;Count of required
                                ;characters in D4 (less 1)
KEYTOT  ADD.B     (A3)+,D5      ;Add character code to D5
```

```
        DBRA      D4,KEYTOT     ;Repeat loop KEYTOT while
                                ;D4>-1
        CMPI.B    #215,D5       ;Has 'END' been entered?
        BEQ       EXIT          ;Branch to finish if so
        MOVEQ     #32,D1        ;ASCII code for space
        BSR       DISP          ;Display it
;-------------------------------------------------------
```

In this routine the number of characters we want from the keyboard is 3, since this will suffice to identify the name of any particular month. The fourth character will be the carriage return code following the input of the three characters.

D5 is first cleared because we shall need to use its lower byte for an addition operation. A1 is then loaded with the address of the four byte keyboard buffer which is labelled `KEYBUF` in the data section. A copy of this base address is loaded into A3 so that we can later retrieve the buffered characters.

Next, the operating system `TRAP` is invoked by loading the number of characters to be fetched, 4, into register D2 and a function parameter, 2, into D0 and then initiating a `TRAP #3` exception which causes the program to wait until four characters have been typed in at the keyboard.

At this point there would be three characters printed on the current screen line: the ones which would have just been input on the keyboard. The carriage return used to enter the characters would have placed the screen cursor on the line below. The ASCII codes for these would also be stored in the `KEYBUF` array with A3 pointing to the address of the first character.

Then the accumulated totals of the ASCII values of the three characters keyed in need to be obtained. This is done by the loop starting at the instruction labelled `KEYTOT` which adds the ASCII values of the three characters into D5.

The next two instructions, `CMPI.B #215,D5` and `BEQ EXIT`, test the input to see if the program is to be terminated and this procedure will be explained later.

Before we print the corresponding mileage total for the required month we might wish to insert a space so that the result will be

indented from the left margin of the screen by one character. This is done by loading D1 with 32, the ASCII code for a space, and calling the DISP routine to print it.

Next, we are ready to retrieve the mileage totals from SUBMIL and print them:

```
;-----------------------------------------------------------
;USE TOTALLED ASCII CODES OF 3 INPUT CHARACTERS TO INDEX A
;LOOKUP TABLE POINTING TO MILEAGE TOTALS
;-----------------------------------------------------------
        CLR.L     D6            ;Clear register D6
        LEA.L     DATES,A4      ;Point to DATES table
        MOVEQ     #23,D4        ;Length of DATES table
                                ;less 1
MATCH   MOVE.B    (A4)+,D6      ;Copy data fron DATES
                                ;into D6 and add 1 to A4
        CMP.B     D5,D6         ;Compare with input code
        DBEQ      D4,MATCH      ;If codes match or D4=-1
                                ;then go to next instruction
                                ;else loop back to MATCH
        LEA.L     SUBMIL,A2     ;Address of SUBMIL in A2
        MOVE.B    (A4),D6       ;Get index value from DATES
        MOVE.W    0(A2,D6),D5   ;Copy total from SUBMIL
                                ;into D5
        MOVEQ     #-1,4         ;Set counter to -1
LOOP10  DIVU      DNUM2,D5      ;Divide D5 by 10
        SWAP      D5            ;Swap halves of D5
        MOVE.W    D5,-(A7)      ;Stack remainder
        ADDQ      #1,D4         ;Update counter
        MOVEQ     #16,D2        ;Shift count in D2
        LSR.L     D2,D5         ;Shift D5 according to
                                ;count in D2
        CMPI      #0,D5         ;Is D5=0?
        BNE       LOOP10        ;Repeat LOOP10 if not
LOOP11  MOVE.W    (A7)+,D1      ;Else retrieve a digit
        ADDI.B    #48,D1        ;Convert it to ASCII
        BSR       DISP          ;Display it
        DBRA      D4,LOOP11     ;Repeat while D4>-1
        MOVE.B    CR,D1         ;Carriage return code
        BSR       DISP          ;Print it
        MOVE.B    LF,D1         ;Line feed code
        BSR       DISP          ;Print it
```

```
        BRA       KEY             ;Jump to get next key
                                  ;input
;-------------------------------------------------------------
;TERMINATION ROUTINE
;-------------------------------------------------------------
EXIT    MOVEQ     #2,D0
        TRAP      #2
        MOVEQ     #-1,D1
        MOVEQ     #0,D3
        MOVEQ     #5,D0
        TRAP      #1
;-------------------------------------------------------------
;DISP SUBROUTINE
;-------------------------------------------------------------
DISP    MOVEQ     #-1,D3
        MOVEQ     #5,D0
        TRAP      #3
        RTS                       ;Return to main program
;-------------------------------------------------------------
```

Firstly, register D6 is cleared, after which the address of the array labelled 'DATES' is loaded into A4. The length of the DATES array, less 1, is loaded into D4 as a count variable.

The DATES array has a strange structure, as you can see if you refer to the data section. Its first value, 217, is the sum of the ASCII codes for capital 'J', 'A' and 'N', representing January. The number which follows, 0, is the offset of the January mileage totals in the SUBMIL array. Similarly, the third value, 205, is the sum of the ASCII character codes for 'F', 'E' and 'B', representing February. The fourth value, 2, is the offset within the SUBMIL array of the mileage totals for February, and so on through the year. It so happens that the total value of the ASCII codes for the first three capital letters of each month in the year is a unique number in each case so that we can use these totals to identify every individual month.

It follows that if we were to take the total of the ASCII values for the three letters which have just been keyed in at the keyboard and compare it with each byte in the DATES array then eventually there will be a match. The value immediately following the matched number will be the value of the corresponding mileage result offset within the SUBMIL array. For example, if we input 'APR' at the keyboard then we shall get the code 227 from the 'KEY' loop. Comparing this with each

byte in DATES we find a match at the 7th byte. The value following this in DATES is 6, so the mileage total for April is at offset 6 within the SUBMIL array.

In practice it works like this: The ASCII total of the entered characters is currently in register D5 and the DATES array is pointed to by A4. The instruction MOVE.B (A4)+,D6 copies a byte from DATES into D6 and autoincrements A4 by 1. The CMP.B D5,D6 instruction compares this byte with the ASCII total in D5 and the DBEQ D4,MATCH instruction loops back to the start of the MATCH loop if there is no match between the contents of D5 and D6 and D4 is greater than –1. This continues until either D4=–1 (no match found) or a match is found, in which case the Z flag will become set and the DBEQ instruction terminates the loop, with the A4 register pointing to the data immediately following the matched values, which will represent the index offset into SUBMIL. In the case of April this would be 6.

A2 is then loaded with the address of the SUBMIL array and the index offset from DATES is loaded into D6. The next instruction, MOVE.W 0(A2,D6),D5 copies the value contained in SUBMIL at the address represented by A2+D6 into D5.

The value now in D5 is the required mileage total and we are ready to convert it into individual character codes before displaying it on the screen. LOOP10 performs a similar function to LOOP6, dividing the total in D5 by 10 until D5=0 and pushing the remainder values onto the stack after each division. At the end of LOOP10 the result has been divided into separate digits which have been pushed onto the stack, the total digit count being held in the D4 register.

LOOP11 pops one digit at a time from the stack into D1, adds 48 to it to obtain its ASCII code and calls the DISP routine. After all the digits have been displayed (D4=–1) it then prints the carriage return and line feed codes. At this point the result has been displayed on the screen, indented below the month name, and the program can then loop back to KEY, using the BRA KEY instruction, so that a further keyboard entry can be intercepted.

In practice, the program keeps looping back from the BRA KEY instruction to 'KEY', printing the mileage total for every month which is entered at the keyboard. The only way to break this loop is to input the characters 'END' from the keyboard, whose ASCII codes add up to 215. If you go back and look at the instructions which immediately

follow the 'KEYTOT' label, you will see that the three characters entered at the keyboard each have their ASCII codes added to the D5 register. If the total of these codes becomes 215, as would be the case if 'END' were keyed in, the BEQ EXIT instruction would direct execution to the end of the program.

```
;-----------------------------------------------------------
;THEN DEFINE DATA
;-----------------------------------------------------------
GALLS1  DC.W   22,23,18,20,16,15          ;Gallons year 1
        DC.W   25,22,20,23,20,19
MILES1  DC.W   400,450,350,425,375,280    ;Mileage year 1
        DC.W   479,423,398,416,423,368
GALLS2  DC.W   18,18,26,27,24,25          ;Gallons year 2
        DC.W   22,23,21,25,24,28
MILES2  DC.W   380,357,496,501,478,482    ;Mileage year 2
        DC.W   465,475,423,489,470,524
TOTALS  DS.W   14                         ;Reserve memory space
                                          ;for calculated data
MES1    DC.B   "TOTAL CONSUMPTION:   "    ;Headings for the
        DC.B   "TOTAL MILEAGE:       "    ;output of the
        DC.B   "AVERAGE CONSUMPTION: "    ;calculated data
        DC.B   "AVERAGE MILEAGE:     "
        DC.B   "MILES PER GALLON:    "
        DC.B   "MAY CONSUMPTION:     "
        DC.B   "MAY MILEAGE:         "
SUBMIL  DS.W   12                         ;Reserve memory space
                                          ;for mileage totals
DATES   DC.B   217,0,205,2,224,4,227      ;Date codes and
        DC.B   6,231,8,237,10,235,12      ;offsets for each
        DC.B   221,14,232,16,230,18       ;month
        DC.B   243,20,204,22
MESSOFF DC.L   0                          ;Reserve memory for
                                          ;current header
                                          ;message address
MESCNT  DC.W   %1111110000000000          ;Define counter for
                                          ;counting off each
                                          ;printed heading
RESPNT  DC.L   0                          ;Reserve memory
                                          ;for storing index
                                          ;offset of data in
                                          ;TOTALS array
DNUM1   DC.W   24                         ;Store divisor no.1
```

```
DNUM2   DC.W   10                      ;Store divisor no.2
CENT    DC.W   100                     ;Store multiplier
KEYBUF  DS.B   4                       ;Buffer for inputs
DEVICE  DC.W   4
        DC.B   'CON_'
        END                            ;End of program
;-------------------------------------------------------------
```

The first four lines of the data section contain the original data figures, with the gallons and mileage for year 1 followed by those for year 2. Each set of data is separately labelled.

Following this is TOTALS, a reserved array of 14 words which is used for storing the results as they are calculated. All calculated results will occupy *two words*, irrespective of their values, because it is normally helpful to store data in a uniform size format. A minimum of two words are needed because some results are floating point decimal values and therefore require an appropriate amount of memory space.

The seven text messages, with their base offset labelled 'MES1', form part of the output of the results at the end of the program and these have all been made the same length by padding them out with spaces so that it is easier to format the output neatly on the screen later on.

SUBMIL is the label of another array reserved for results: this one being for the total mileage figures for each corresponding month of the two years. The data which goes in here is used to print out results in response to the keyed in month names.

DATES is an array containing coded data representing the names of months of the year, together with the offsets of the mileage totals, which are used for indexing the results contained in SUBMIL.

The label MESSOFF refers to the address of a message offset value which is initially zero. It is used to keep a record of the offset address of the current message text as each result is printed out.

MESCNT is the address of a word which is used to count off each result as it is displayed and this has been entered directly as a *binary* value, as specified by the '%' at the beginning.

RESPNT is the label of the address containing the offset of the current data being referenced in the TOTALS array and which is initially zero.

DNUM1, DNUM2 and CENT are the addresses of the numeric constants 24, 10 and 100 which are used in the division and multiplication operations.

KEYBUF consists of 4 bytes as a buffer for keyboard inputs.

Sorting Data

In Chapter 6 a simple BASIC bubble sort was shown. The following assembly language program is an adaptation of this, in which several characters are sorted into alphabetic order and displayed on the screen.

This listing, which is presented with only brief rem statements, should help you to test your understanding of the programs in this book. It consists of instructions which have all been previously demonstrated and follows the principles applied in the BASIC version fairly closely. Once you have worked out how this program operates, you will no doubt be able to expand it and adapt it for your own purposes. Note that it has been coded as a subroutine so that it can be called from other programs, therefore it terminates with an RTS instruction.

```
;-----------------------------------------------------------
;        A SORT PROGRAM DESIGNED TO SORT A SET OF
;         ASCII CHARACTERS INTO ALPHABETIC ORDER
;-----------------------------------------------------------
;FIRST THE PROGRAM REGISTERS ARE INITIALIZED
;-----------------------------------------------------------
LOOP1   CLR.L  D6              ;D6 will count swaps
        LEA.L  CHARS,A2        ;Offset of data in A2
        MOVEQ  #6,D4           ;Length of data (less 2)
                               ;in D4
;-----------------------------------------------------------
;COMPARE TWO ITEMS OF DATA AND EXCHANGE IF OUT OF ORDER
;-----------------------------------------------------------
LOOP2   MOVE.B (A2),D1         ;Move an item of data to D1
        MOVE.B 1(A2),D2        ;Move the next one into D2
        CMP.B  D1,D2           ;Compare the two items
        BCC    NEXT            ;Branch if C=0 to NEXT
        EXG    D1,D2           ;Otherwise exchange the
```

```
                                ;contents of D1 and D2
        MOVE.B D1,(A2)          ;Replace in memory
        MOVE.B D2,1(A2)         ;Replace in memory
        ADDQ   #1,D6            ;Increment exchange counter
;---------------------------------------------------------
;POINT TO NEXT ITEM AND REPEAT THE LOOP
;---------------------------------------------------------
NEXT    ADDQ   #1,A2            ;Point to next data item
        DBRA   D4,LOOP2         ;Loop back to LOOP2 while
                                ;D4>-1
;---------------------------------------------------------
;CHECK TO SEE IF ALL ITEMS SORTED (D6=0)
;---------------------------------------------------------
        CMPI   #0,D6            ;Any swaps in this loop?
        BNE    LOOP1            ;If so, go back to LOOP1
;---------------------------------------------------------
;PREPARE SORTED LIST FOR DISPLAY
;---------------------------------------------------------
        MOVEQ  #7,D4            ;Otherwise get character
                                ;count (less 1) in D4
        LEA.L  CHARS,A2         ;Point A2 to start of
                                ;sorted data
LOOP3   MOVE.B (A2)+,D1         ;Transfer data item to D1
        BSR    DISP             ;Call display subroutine
        DBRA   D4,LOOP3         ;Go back to LOOP3 while
                                ;D4>-1
        RTS                     ;Return from sort subroutine
;---------------------------------------------------------
;DISP SUBROUTINE
;---------------------------------------------------------
DISP    MOVEQ  #-1,D3
        MOVEQ  #5,D0
        TRAP   #3
        RTS
;---------------------------------------------------------
;DEFINE THE DATA
;---------------------------------------------------------
CHARS   DC.B   'ABDFCEGH'       ;Define the data to be
;              sorted

        END
;---------------------------------------------------------
```

Chapter 14

Debugging, Instruction Formats & Supervisor Mode Operation

Program Debugging

The process of debugging a machine code program can sometimes be difficult and time consuming.

With a BASIC program only two things can happen if a program is faulty: either the program stops and an identifying error message appears, or the program runs but produces erroneous results. Either way it is usually a simple matter to go through the listing to identify and correct the faults.

A machine code program is much more difficult to debug because minor errors can be very difficult to detect and an examination of the original source listing, however well it might be annotated, will not necessarily reveal the problem.

There is no universal law, other than Murphy's Law, which determines that a machine code program will fail at the first attempt. Murphy's Law is nevertheless powerful enough to ensure that most programs of any degree of complexity will surely fail, not only on the first attempt but probably on the second, third and fourth as well. In machine code bugs appear to propagate: a successful attempt to eliminate one type of bug seems to cause some kind of genetic mutation process which spawns more virulent strains of bug which are inured to most kinds of systematic treatment. Nevertheless, tried and tested methods are available which will enable you eventually to coax your programs into a stable condition, or even into a complete state of perfect health.

Ninety percent of errors are easily traceable, since they are of a common and almost inevitable kind. These include:

- Confusing absolute addresses with immediate data – the former require only an address value or label name such as `80000`, `$29D1C` or `YLABEL`. The latter must be preceded by a '#' sign such as `#80000`, `#$29D1C` or `#MYLABEL`.
- Destination errors – such as forgetting to use `ADDA`, `MOVEA` etc. when the destination is an address register. An assembler may insist that you use an `A` suffix although some disassemblers may remove the `A` when they produce a listing. Check the source code, not the object code for such errors.
- Size errors – such as forgetting whether the data in a data register is to be regarded as a byte, word or long word operand.
- Address register errors – such as forgetting that a 16-bit operand in an address register is automatically sign extended or that address registers cannot accept byte operands.
- Alignment errors – aligning word and long word operands at odd numbered addresses.
- Flag errors – forgetting that some operations do not affect any and some do not affect all of the flags; and that some operations, such as `ABCD`, do not change the zero flag if the result is zero.
- Conditional branching errors – a loop count variable may be incorrect, the loop may branch to the wrong location, or the wrong conditions may be specified. `DBcc` instructions are particularly problematic in this respect until you are used to them.
- Loop register errors – some loops require a register to be cleared before the beginning of the loop, for example when the loop is being used to add separate items of data to the register on each iteration. Always clear a register before use if you are unsure that it is already clear since it may contain redundant data which will affect your calculations.
- Positioning – if specific addresses are referred to in a program, destinations may have changed during editing or a non-relocatable program may have been loaded from the wrong address.

These are only a few examples but they will serve to demonstrate some of the simple errors which can render an otherwise perfectly designed program unexecutable.

To identify and correct these and other errors, there are a number of diagnostic procedures which can be performed. The following paragraphs describe the most elementary of these.

Assembly Errors

The first line of defence is the assembly process itself. Any program instructions which are formally incorrect will be identified by the assembler as it converts the assembly language mnemonics into object code. A good assembler will produce an annotated source listing of its own, numbering all the program lines and identifying any incorrect instructions with an error message such as 'Line 8: second operand cannot be an address register'. Only when all such errors have been eliminated will the assembler produce an executable object file and issue a message such as '`Minor errors 0, Major errors 0`'. This reassuring message merely means that there are no formal errors left in the program. Any structural or functional errors will be revealed later.

Trial Run

The second step is to load and run your program, at which point your program may work perfectly, you may simply get an error message such as '`Error 23: division by zero`', the system may go into limbo, leaving you with nothing on the screen and no response from the keyboard, or the entire system may crash. This trial run will give you some idea of the scale of the problem and in some cases certain errors will be identified immediately. If your program was supposed to print a message for example, and gets no further than the first character, then the problem probably lies in a program loop mechanism.

Debugging Monitor

The next stage is to reload the program under the control of a monitor program. A typical monitor provides a number of useful debugging tools which can be used in a systematic way to test a program thoroughly. The most important of these are as follows:

- Disassembly listing – this produces an output similar to the object code listings used in earlier chapters and is useful among other things for checking that jumps and branches transfer execution to the correct locations and that data accesses refer to the correct locations.

- Breakpoints – very often it is not obvious where a fault is located in a program. A breakpoint feature allows you to assign 'breakpoints' to a number of key locations such as the final instructions of important routines. The program may then be run under monitor control (not necessarily from the beginning) and will automatically stop and return to the monitor when it encounters a breakpoint. At this stage you can obtain a listing of the current register contents and flag settings to check whether they contain the correct values.

- Trace – a single step trace mechanism, which allows you to execute a program one instruction at a time, will give a listing of all register and flag contents after the execution of every instruction. This is not much use for extensive tracing because the trace will follow every twist and turn your program takes. If you have a lot of branches to common subroutines, `TRAP` instructions or loops which iterate hundreds or even thousands of times then you will produce several miles of printer paper containing information which you can never hope to analyze. A trace is very useful however in cases where you have identified a possible error source and need to run through a short sequence of code to determine exactly what occurs during execution.

- Dump – a hexadecimal 'dump' of the contents of a specified block of memory can be useful to check that data is being entered or modified correctly. A dump will normally show the byte contents of about sixteen memory addresses per line and will normally include a listing of the corresponding printable ASCII characters.

Systematic use of these facilities will normally be sufficient to track down most types of error, although you must be prepared to exercise a lot of patience in some cases. The important thing is to adopt a clear and logical approach. If object code listings, breakpoint runs, traces and dumps reveal no obvious errors but a particular routine will still not run correctly, remember the obvious fact that an error must exist *somewhere*. It may be that the error is due to faulty programming rather than a lack of accuracy or else you are subconsciously looking for one kind of error and therefore overlooking another.

`PROG6` in Chapter 13 took two days to debug, despite all evidence that the code was operating correctly. The fault lay not in the program but in a reference table in the assembler documentation, which contained a misprinted ASCII value! It is advisable to adopt the maxim of Sir Arthur Conan Doyle's Sherlock Holmes: "When you have eliminated

the impossible, whatever remains, however improbable, must be the truth."

Instruction Opcode Formats

Although it is usual to program using the 68000 instruction mnemonics, it is useful to be aware of the way in which these mnemonics are translated by an assembler into a machine language opcode. It is unlikely that you would ever want to program directly using opcodes, but frequently you may need to alter the opcodes in an assembled program. For example, if you are debugging a program and have loaded it into a debugging monitor, you may wish to alter a few of the opcodes in order to test or fine tune your object program.

Suppose, for example, that there is some unidentified minor fault somewhere in your program which prevents it from operating correctly. On of the most effective ways of tracing faults is to use a 'trace' utility to obtain a listing of all register contents as each instruction is executed one at a time. If you have any TRAP instructions in the program, the trace utility will divert execution to an operating system trap routine and you will have to trace all the way through that before returning to your own code – a time consuming and unnecessary procedure. It would be a lot simpler if you could temporarily convert every TRAP code into a NOP code so that the trap routines are not actually called during debugging.

You may also wish to insert additional instructions into the program, without having to go to all the trouble of altering the source listing and re-assembling the entire program.

A TRAP instruction is coded as a two byte opcode:

```
01001110 0100XXXX    ($4E4?)
```

The most significant 12 bits of this opcode represent the TRAP instruction itself and the four least significant bits (represented by 'XXXX' are reserved for the code of the type of trap required (i.e. traps 0 to 15). The opcode for a NOP instruction is:

```
01001110 01110001    ($4E71)
```

From this you can see that whatever the value of the low byte of a TRAP instruction might be, the higher byte is exactly the same as the higher byte of a NOP instruction. A TRAP instruction can therefore be transformed into a NOP instruction simply by changing the lower byte of the TRAP to $71.

Most other instructions are more complicated than this although some other instructions which you are likely to want to change to NOP during debugging, such as JMP, JSR and TRAPV are also two byte instructions with their higher byte set to $4E.

If you look at the first instruction used in the programs in the preceding chapters, MOVEQ #0,D1 you will see from the object code listings that this is assembled as $7200.

In this case the object code follows the pattern:

```
0111RRR0 DDDDDDDD
```

in which 0111 and the single 0 is the unique opcode for any MOVEQ instruction, RRR is a three bit code for the register referred to in the instruction and the eight Ds represent the single byte binary number which is loaded into the register. The three byte code for D1 is 001 and the immediate value is 00000000, so the complete opcode for this instruction is:

```
01110010 00000000    ($7200)
```

More complex instructions are encoded in a similar way but incorporating a data size specifier and an effective address field, indicating the operands containing the values which collectively specify the physical address or addresses of the instruction operands. The effective address field incorporates a mode field, indicating the addressing mode used by the instruction. For example, the instruction MOVE.B 2(A2),D4 moves a byte from the effective address specified by the sum of the contents of A2 plus 2 into the low order byte of register D4. The opcode for this instruction consists of four bytes:

```
0001 000 100 101 010 00000000 00000010  ($112A0002)
```

The highest four bits, 0001, are the opcode for a *byte move* operation. The next three bits, 000, are a code indicating the addressing mode used for the *destination* operand (data register direct), the next three

bits, 100, represent the code number for the destination register, D4. The next three bits (101) represent the addressing mode used for the *source* operand (address register indirect with displacement) and the next three bits (010) are the code for address register A2. The final 16 bits are the opcode for the displacement (00000000 00000010).

It will be clear from this that instruction encoding is a complex business and it is unfortunately not possible, because of the way in which opcodes are irregularly split up into varying sized bit fields in different instructions, to construct a simple one-for-one table showing how various opcodes correspond to their assembly language mnemonics.

In the dark ages of computing it was usual to program in assembly language by hand, carefully working out the opcodes for each instruction and recording them on paper before copying them into the computer.

With the availability of relatively inexpensive assemblers and disassemblers the necessity for all this effort has disappeared so that it is possible not only to encode the original source listing using mnemonics but also to edit the assembled object code in this way. Most monitor programs will allow you to create spaces in assembled code and to insert or modify existing instructions using standard 68000 mnemonics as well as hex values.

Supervisor Mode Operation

In the preceding chapters, numerous references have been made to differences in operation between supervisor and user mode on the 68000.

Supervisor mode is something which you would normally have no direct contact with since it always operates automatically at 'management' level whenever it is required. Supervisor mode is initiated automatically when the system is powered up and allows the operating system to use certain 'privileged' instructions to allocate memory, to establish contact with the console and to set various pointers such as the stack pointer, the PC register and the initial contents of the status register. Once control is handed over to user mode (by zeroing the 'S' flag in the status register) it usually remains there, switching

temporarily back to supervisor mode whenever certain events take place such as TRAP instructions and other exceptions, I/O operations, multi-processor communications and access by user programs to operating system routines. From user mode it is impossible to select supervisor mode directly since the instructions which can do this are privileged. When user programs initiate a mode switch, such as with a call to a TRAP routine, supervisor mode is automatically selected for the duration of the operation and then returns to user mode immediately afterwards.

Where a user program is running in a multi-user environment in which a number of different programs are competing for the processor's attention, the system must arbitrate between the claims of different programs, ensuring that no program can use, modify or destroy the contents of other programs without pre-determined authority. This protection extends to the execution of certain key operations such as I/O transactions. It could be catastrophic if one user were able to mask interrupts or communicate freely with external devices if these operations were to interfere with the efficient running of other processes. The operating system must therefore control and coordinate these operations so that the system as a whole runs smoothly and continuously.

The visible consequence of any attempt by a program to display anti-social tendencies is that the operating system, functioning in supervisor mode, will intercept and prevent such actions, either redirecting execution to some corrective mechanism which smoothly maintains law and order or to some customized system exception routine which displays an error message and excludes the offending program from current system operation.

Certain 68000 instructions are specifically designed for use by the operating system running in supervisor mode. These are mostly instructions which are concerned with loading pre-determined values into system registers such as the system byte of the supervisor register during system initialization and for obvious reasons may not be used in applications programs. In user mode, any attempt to use these privileged instructions is interrupted by an exception mechanism. These special instructions are ANDI to SR, EORI to SR, MOVE to SR, MOVE USP, ORI to SR, RESET, RTE and STOP.

You will notice that most of these privileged instructions are ones which can be used directly to modify the contents of the SR register.

The lower byte of the SR register, the CCR, can be accessed freely in user programs and a separate set of corresponding instructions is provided for this purpose.

Memory Management System

Under normal circumstances there are very few problems associated with addressing memory locations. When you need to execute a program, address data or call a subroutine which is external to your own program you simply program the appropriate instruction and the system itself works out the correct address for the required destination.

When the system is supporting a multi-user environment however, the situation is considerably more complex. You may not be the only user of the system and your current program may not be the only one active at any one time.

You can imagine a situation in which your precious program, which may not be considered by the system to have the highest priority, may be located somewhere amongst millions of bytes of memory, fighting for its existence amongst bigger and more important programs which might at any moment invade your stored data and wipe out your own program.

Furthermore, there is the danger that your own program may alter vital flags or memory pointers and disrupt the functioning of other processes in the system.

With this in mind you may feel that you could lead a far less traumatic existence if you could write straightforward programs and expect the system to work out for itself exactly where and when your code should reside in physical memory and be executed; at the same time making sure that no other users have access to your own, private, code and data.

The memory management unit (MMU) of the 68000 is designed to do precisely this. The ordinary applications programmer need have no detailed understanding or knowledge of the way in which a multi-user, multi-tasking system is configured.

Essentially the MMU enables the operating system to allocate memory into discrete blocks called memory spaces – some only being available in supervisor mode, some in user mode, some containing data, some containing code and some containing code and data.

The MMU is selectively able to separate these spaces so that a 'task' operating within one address space may be unable to access either the code or the data belonging to another task. The operating system itself can be protected in such a way that although it cannot be hijacked or altered by a user program, its code can be accessed by user programs under supervisor mode control so that key subroutines such as peripheral control and communication routines can be used without causing any harm.

Since the system has more extensive responsibilities than to concern itself exclusively with the operation of one single user's particular program, it is designed to be *task* oriented rather than program oriented. In simple terms, a task is simply a complete and coherent set of instructions which together perform a particular function. The memory management system holds a table containing details of the position and status of all the tasks currently resident in the system, enabling the operating system to maintain a smooth flow of execution throughout the system, switching back and forth between different tasks and between user and supervisor mode where required.

Afterword

The aim of this book has been to explain and demonstrate the main concepts of assembly language programming and to provide you with enough knowledge and understanding to be able to write some fairly sophisticated programs of your own.

The subject of assembly language is a large and complex one and there is a great deal more to learn. The next step is to go through the instruction glossary in Appendix B, which will help to reinforce your understanding of the instructions which you have learned and to discover some new ones which have not been included in the previous chapters.

After that, you may wish to go a stage further and buy a more technical book which covers the subject in a more formal way and which will introduce you to some of the finer points of programming.

It is important that you should be able to relate the material in this book to the operation of you own system and therefore it is essential to obtain some documentation which explains how to load and execute machine code programs and how to access the graphics and other service routines provided in your system.

There are other topics which have not been covered here and which may be of special interest to you. These include interfacing with high level languages such as Pascal, programming associated processors such as the 68008, 68010 and 68020 and using devices such as the 6820 PIA and the 6850 ACIA. You may also wish to go much more deeply into the subject of system architecture than we have here and if you ultimately wish to become involved in system design then you will need much more precise and extensive technical details of system operation.

For complete, detailed technical information on most aspects of 68000 processor operation then the essential reference book is Motorola's own *16-bit Microprocessor User's Manual*, available from bookshops.

You will by now appreciate that aside from the technical aspects of assembly language programming, there is scope for a lot of 'creative' development. Any potter will tell you that no matter how academically expert you are on the molecular behaviour of clay, the shape in which it finally ends up depends on the skill and creativity of the craftsman.

The kinds of programs which you fashion with a computer language depend a great deal on the way in which a language is structured. Artificial intelligence programmers mostly use programs like LISP and Prolog because those languages are structured in such a way that they suit that particular kind of application. The languages themselves virtually suggest the way in which you set about a problem. Assembly language on the other hand is about as structured as the potter's lump of clay, and there is plenty of scope for twisting your programs into such weird and ugly shapes that even your computer will refuse to have anything to do with them. The worst thing you can do is to sit down and start a program without the slightest idea of how it is supposed to end up. Assembly language is not like that.

Ultimately, any kind of program is an expression of what is going on in your imagination. The careful definition of the problem which you are working on usually suggests to you the kinds of data structures which you need for its solution. If you can get into the habit of thinking in structures, of imagining a real-world problem as sets of efficiently interrelated sets of data, then you can virtually formulate your programs before you even sit in front of the keyboard. Assembly language itself is simply a means of manipulating and expressing those

structures and with practice and familiarity you should have no difficulty in achieving a high level of proficiency in a relatively short time.

Appendices

Appendix A

Instructions by Category

Data Movement

EXG	Exchange registers
LEA	Load Effective Address
LINK	Link and Allocate
MOVE	Move data
MOVEA	Move address
MOVEM	Move multiple
MOVEP	Move data to peripheral
MOVEQ	Move quick
PEA	Push effective address
SWAP	Swap register halves
UNLK	Unlink

Integer Arithmetic

ADD	Add binary
ADDA	Add address
ADDI	Add immediate
ADDQ	Add quick
ADDX	Add extended
CLR	Clear operand
CMP	Compare
CMPA	Compare address
CMPI	Compare immediate
CMPM	Compare memory
DIVS	Signed division
DIVU	Unsigned division
EXT	Sign extend
MULS	Signed multiplication
MULU	Unsigned multiplication
NEG	Negate
NEGX	Negate with extend

SUB	Subtract binary
SUBA	Subtract address
SUBI	Subtract immediate
SUBQ	Subtract quick
SUBX	Subtract with extend
TAS	Test and set
TST	Test

Logical

AND	Logical and
ANDI	And immediate
OR	Logical or
ORI	Or immediate
EOR	Logical exclusive or
EORI	Eor immediate
NOT	Logical complement

Shift & Rotate

ASL	Arithmetic shift left
ASR	Arithmetic shift right
LSL	Logical shift left
LSR	Logical shift right
ROL	Rotate left
ROR	Rotate right
ROXL	Rotate left with extend
ROXR	Rotate right with extend

Bit Manipulation

BTST	Bit test
BSET	Bit test and set
BCLR	Bit test and clear
BCHG	Bit test and change

BCD Operations

ABCD	Add decimal
SBCD	Subtract decimal
NBCD	Negate decimal

Program Control

Bcc	Branch on condition
DBcc	Decrement, test and branch
DBRA	Decrement and branch
Scc	Set from condition
BRA	Unconditional branch
BSR	Branch to subroutine
JMP	Unconditional jump
JSR	Jump to subroutine
RTR	Return & restore CCR
RTS	Return from subroutine

System Control

ANDI to SR	AND immediate to SR
EORI to SR	EOR immediate to SR
MOVE to SR	Move to SR
MOVE USP	Move user stack pointer
ORI to SR	OR immediate to SR
RESET	Reset
RTE	Return from exception
STOP	Load SR and stop
CHK	Check register
TRAP	Trap
TRAPV	Trap on overflow
ANDI to CCR	AND immediate to CCR
EORI to CCR	EOR immediate to CCR
MOVE to CCR	Move to CCR
MOVE from SR	Move from SR
ORI to CCR	OR immediate to CCR

Appendix B

Instruction Glossary

Key to Abbreviations

Instruction Mnemonics

The following abbreviations are used in the glossary to represent registers operands:

An	Any address register
Dn	Any data register
Rn	Any register
Ri	Any register being used as an index

Addressing Modes

The following table indicates how the various addressing modes are classified under the effective address categories: *Data*, *Memory* (shown as Mem in the table), *Control* and *Alterable* (shown as Alt).

Mode	*Symbol*	*Data*	*Mem*	*Control*	*Alterable*
Data reg direct	Dn	X			X
Addr reg direct	An				X
Absolute	nnnnn	X	X	X	X
Immediate	<imm>	X	X		
Addr reg indirect	(An)	X	X	X	X
with predecrement	-(An)	X	X		X
with postincrement	(An)+	X	X		X
with displacement	d16(An)	X	X	X	X
with index	d8(An,Ri)	X	X	X	X
PC relative	d16(PC)	X		X	X
with index	d8(PC,Ri)	X	X	X	

The following codes, based on these classifications, are used in the glossary to specify the effective address of certain operands:

<ea>	Effective Address – any addressing mode can be used
<aea)	Alterable Effective Address
<cea>	Control Effective Address
<dea>	Data Effective Address
<caea>	Control Alterable Effective Address
<daea)	Data Alterable Effective Address
<maea>	Memory Alterable Effective Address

Operand Sizes
The operand sizes applicable to the instructions are coded as `B.` (byte), `W.` (word) and `L.` (long word).

Flags
The N, Z, V, C and X flags are listed in the glossary under each instruction heading. The codes used to indicate the effect of individual instructions on the flags are as follows:

0	flag reset
1	flag set
A	flag affected by instruction.
?	flag affected but setting undefined

Blank spaces under the flags indicate that they are not altered by the instruction.

ABCD (decimal addition)

Addressing Modes:

```
ABCD Dn,Dn
ABCD -(An),-(An)
```

Mnemonic	*Operation*	*Size*	Flags:	*N*	*Z*	*V*	*C*	*X*
`ABCD`	Decimal addition	`B.`		?	A	?	A	A

Description:
`ABCD` is a `BCD` addition operation which adds a binary coded decimal

source operand and the value of the extend flag to a destination operand with the result being stored in the destination. Note that the zero flag is zeroed if the result is greater than zero, otherwise it is unchanged.

ADD, ADDA, ADDI, ADDQ and ADDX (binary addition)

Addressing Modes:

```
ADD <ea>,Dn
ADD Dn,<maea>
ADDA <ea>,An
ADDI #<imm>,<daea>
ADDQ #<imm>,<aea>
ADDX Dn,Dn
ADDX -(An),-(An)
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
ADD	Add binary	B.W.L.	A A A A A
ADDA	Add address	W.L.	
ADDI	Add immediate	B.W.L.	A A A A A
ADDQ	Add quick	B.W.L.	A A A A A
ADDX	Add extended	B.W.L.	A A A A A

Description: ADD adds a source to a destination operand and stores the result in the destination.

The ADDA form of ADD specifies that the destination operand must be an address register and that the data size must be either word or long. Word sized results are sign extended. No flags are affected.

The ADDI form of ADD specifies that the source operand must be an immediate value.

The ADDQ form of ADD specifies that the source operand must be an immediate value in the range 1 to 8.

The ADDX form of ADD specifies that the extend bit is added to the source operand before it is added to the destination.

Note that with ADDQ the flags are not affected if the destination is an address register. With ADDX the zero flag is reset if the result is greater than zero, otherwise it is unchanged.

AND, ANDI, ANDI to CCR and ANDI to SR (Logical AND)

Addressing Modes:

```
AND <dea>,Dn
AND Dn,<maea>
ANDI #<imm>,<daea>
ANDI #<imm>,CCR
ANDI #<imm>,SR
```

Mnemonic	*Operation*	*Size*	Flags:	*N Z V C X*
AND	Logical AND	B.W.L.		A A 0 0
ANDI	AND immediate	B.W.L.		A A 0 0
ANDI to CCR	AND immediate to CCR	B.		A A A A A
ANDI to SR	AND immediate to SR	W.		A A A A A

Description: AND performs a logical AND operation between a source and a destination operand with the result being stored in the destination.

ANDI performs the same function but the source operand must be an immediate value.

ANDI to CCR ANDs an immediate operand with the low order byte of the status register.

ANDI to SR is a privileged instruction which ANDs an immediate operand with the entire 16 bits of the status register.

Note that with ANDI to CCR and ANDI to SR the flags are affected according to the bit values of the immediate value.

ASL & ASR (arithmetic bit shifts)

Addressing Modes:

```
SL Dn,Dn
ASL #<imm>,Dn
ASL <maea>
ASR Dn,Dn
ASR #<imm>,Dn
ASR <maea>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
ASL	Arithmetic shift left	B.W.L.	A A A A A
ASR	Arithmetic shift right	B.W.L.	A A A A A

Description: ASL shifts the bits in an operand to the left, moving the most significant bit of the operand into both the carry and extend flags and moving a zero into the least significant bit position.

ASR shifts the bits in an operand to the right, moving the least significant bit of the operand into the carry and extend flags. The high order (sign) bit is replicated into its original position.

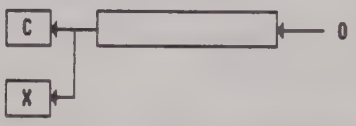


FIGURE B-1. ASL.

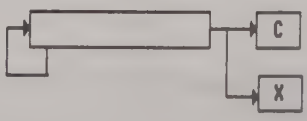


FIGURE B-2. ASR.

Bcc, BRA & BSR (branch instructions)

Addressing Modes:

```
BCC <label>
BRA <label>
BSR <label>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
Bcc	Branch conditionally	B.W.	
BRA	Branch always	B.W.	
BSR	Branch to subroutine	B.W.	

Description: BRA is a relative branch instruction which redirects execution to a location relative to its own position, indicated by a label representing a signed 8- or 16-bit displacement value.

BSR is similar to BRA except that it redirects execution to a subroutine. The return address is automatically stacked so that a return can be made to the instruction immediately following the BSR instruction.

Bcc is a conditional relative branching instruction which branches to a destination location only if the specified conditions are true.

The conditions are incorporated in the instruction name: e.g. BEQ, BNE etc. and are as follows:

Condition	*Meaning*	*Flags*
CC	Carry clear	C=0
CS	Carry set	C=1
EQ	Equal	Z=1
F	False	0
GE	Greater or equal	(N=1 & V=1) or (N=0 & V=0)
GT	Greater than	(N=1 & V=1 & Z=0) or (N=0 & V=0 & Z=0)
HI	High	C=0 & Z=0
LE	Less or equal	Z=1 or (N=1 & V=0) or (N=0 & V=1)
LS	Low or same	C=1 or Z=1
LT	Less than	(N=1 & V=0) or (N=0 & V=1)
MI	Minus	N=1
NE	Not equal	Z=0

PI	Plus	N=0
T	True	1
VC	Overflow clear	V=0
VS	Overflow set	V=1

BCHG, BCLR, BSET and BTST (bit testing instructions)

Addressing Modes:

```
BCHG Dn,<daea>
BCHG #<imm>,<daea>
BCLR Dn,<daea>
BCLR #<imm>,<daea>
BSET Dn,<daea>
BSET #<imm>,<daea>
BTST Dn,<dea>
BTST #<imm>,<dea>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
BCHG	Bit test and change	B.L.	A
BCLR	Bit test and clear	B.L.	A
BSET	Bit test and set	B.L.	A
BTST	Bit test	B.L.	A

Description: BCHG tests a specified bit in a destination operand and sets or resets the zero flag accordingly. The bit number is specified in the source operand (modulo 32 for Dn source and modulo 8 for #<imm> source). Following this the state of the specified bit is changed (0 becomes 1 or 1 becomes 0).

BCLR works similarly except that after the test the specified bit is always left reset.

BSET works similarly except that after the test the specified bit is always left set.

BTST works similarly except that after the test the specified bit is always left unchanged.

CHK (check register against bounds)

Addressing Modes:

```
CHK <dea>,Dn
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
CHK	Check register against bounds	W.	A ? ? ?

Description: CHK is intended to allow you to check that a specified boundary allocated to a section of memory, such as an array, has not been exceeded. The source operand is the boundary value (e.g. the length of the array) as a signed integer and the destination register holds the value to be checked. If the destination value is less than zero or if it is greater than the source operand then a CHK trap exception is initiated. The N flag is set if the destination is less than zero and reset if it is greater than the source operand, otherwise it remains as it was. The Z, V & C flags may also be affected but their values have no significance.

CLR (set to zero)

Addressing Modes:

```
CLR <daea>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
CLR	Reset destination	B.W.L	0 1 0 0

Description: The destination operand is zeroed. To zero a whole register then the operation must be of long size.

CMP, CMPA, CMPI & CMPM (compare)

Addressing Modes:

```
CMP <ea>,Dn
CMPA <ea>,An
```

```
CMPI #<imm>,<daea>
CMPM (An)+,(An)+
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
CMP	Compare source and destination	B.W.L	A A A A
CMPA	Compare address	W.L.	A A A A
CMPI	Compare immediate	B.W.L.	A A A A
CMPM	Compare memory	B.W.L.	A A A A

Description: CMP Compares a source with a destination operand without altering either and alters the condition flags accordingly. The destination must be a data register.

CMPA is the same as CMP but the destination operand must be an address register.

CMPI is the same as CMP but the source operand must be an immediate value.

CMPM is the same as CMP but the source and destination operands are addressed in postincrement mode, allowing two separate sequential blocks of data in memory to be compared under the control of a program loop.

DBcc & DBRA
(decrement and branch instructions)

Addressing Modes:

```
DBcc Dn,<label>
DBRA Dn,<label>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
DBcc	Decrement and branch conditionally	W.	
DBRA	Decrement and branch unconditionally	W.	

Description: DBRA decrements the source register by 1. If the source register is then greater than –1 then execution is branched to a relative destination specified by the label, otherwise execution continues with the next instruction. The label represents a 8- or 16-bit signed displacement.

DBcc is similar to DBRA except that before the register is decremented, a specified condition is tested. If the condition is *not* true then the decrementation and register test is performed as for DBRA. If the condition is *true* then no operation is performed and execution continues with the following instruction.

The instruction DBEQ D4,LOOP for example will branch execution to a destination labelled 'LOOP' until either D4 equals –1 or the Z flag is set.

The conditions are incorporated in the instruction name: e.g. DBEQ, DBNE etc. and are as follows:

Condition	*Meaning*	*Flags*
CC	Carry clear	C=0
CS	Carry set	C=1
EQ	Equal	Z=1
F	False	0
GE	Greater or equal	(N=1 & V=1) or (N=0 & V=0)
GT	Greater than	(N=1 & V=1 & Z=0) or (N=0 & V=0 & Z=0)
HI	High	C=0 & Z=0
LE	Less or equal	Z=1 or (N=1 & V=0) or (N=0 & V=1)
LS	Low or same	C=1 or Z=1
LT	Less than	(N=1 & V=0) or (N=0 & V=1)
MI	Minus	N=1
NE	Not equal	Z=0
PI	Plus	N=0
T	True	1
VC	Overflow clear	V=0
VS	Overflow set	V=1

DIVS & DIVU (binary division)

Addressing Modes:

```
DIVS <dea>,Dn
DIVU <dea>,Dn
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
DIVS	Signed division	W.	A A A 0
DIVU	Unsigned division	W.	A A A 0

Description: DIVS divides a 32-bit destination operand (the dividend) by a 16-bit source operand (the divisor) and stores the 32-bit result in the destination. The quotient of the result is in the lower word of the destination location and the remainder is in the higher word. The sign of the remainder is the same as that of the original dividend unless the remainder is zero (Z=1).

The sign of the quotient is indicated by the status of the N flag.

DIVU performs the same operation but using unsigned operands.

EOR, EORI, EORI to CCR and EORI to SR (exclusive OR operations)

Addressing Modes:

```
EOR Dn,<daea>
EORI #<imm>,<daea>
EORI #<imm>,CCR
EORI #<imm>,SR
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
EOR	Logical exclusive OR	B.W.L.	A A 0 0
EORI	EOR imm.	B.W.L.	A A 0 0
EORI to CCR	EOR imm. to CCR	B.	A A A A A
EORI to SR	EOR immediate to SR	W.	A A A A A

Description: EOR performs a logical EOR operation between a source and a destination operand with the result being stored in the destination.

EORI performs the same function but the source operand must be an immediate value.

EORI to CCR EORs an immediate operand with the low order byte of the status register.

EORI to SR is a privileged instruction which EORs an immediate operand with the entire 16 bits of the status register.

Note that with EORI to CCR and EORI to SR the flags are affected according to the bit values of the immediate value.

EXG (exchange registers)

Addressing Modes:

```
EXCH Rn,Rn
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
EXG	Exchange registers	L.	

Description: Exchanges the entire 32-bit contents of the source and destination registers.

EXT (sign extend)

Addressing Modes:

```
EXT Dn
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
EXT	Sign extend	W.L.	A A 0 0

Description: copies bit 7 of a data register into bit positions 8 to 15 or bit 15 into bit positions 16 to 31, depending on the operation size specified.

JMP & JSR (jump operations)

Addressing Modes:

```
JMP <cea>
JSR <cea>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
JMP	Jump		
JSR	Jump to subroutine		

Description: JMP transfers execution to a specified address.

JSR transfers execution to a subroutine at a specified address. The return address is automatically stacked so that on return, execution can continue with the instruction following the JSR instruction.

LEA (load effective address)

Addressing Modes:

```
LEA <cea>,An
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
LEA	Load effective address		

Description: Loads a specified address into an address register.

LINK (link and allocate)

Addressing Modes:

```
LINK An,#<imm>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
LINK	Link and allocate		

Description: LINK is used to allocate a stack frame area within the stack.

The contents of the address register specified in the instruction are pushed on to the stack. The stack pointer value is then copied into the address register and an immediate 16-bit negative displacement value is added to the stack pointer, thus creating a reserved area within the stack whose base is held in the address register.

See also 'UNLK'.

LSL & LSR (logical bit shifts)

Addressing Modes:

```
LSL Dn,Dn
LSL #<imm>,Dn
LSL <maea>
LSR Dn,Dn
LSR #<imm>,Dn
LSR <maea>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
LSL	Logical shift left	B.W.L.	A A 0 A A
LSR	Logical shift right	B.W.L.	A A 0 A A

Description: LSL shifts the bits in an operand to the left, moving the original contents of the most significant bit into the carry and extend flags and moving a zero into the least significant bit position.

LSR operates similarly except that the bits are shifted to the right, the original least significant bit being copied into the carry and extend flags and a zero being shifted in to the most significant bit position.

Multiple shifts are performed by using a count value in the source operand.

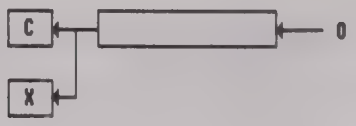


FIGURE B-3. LSL.

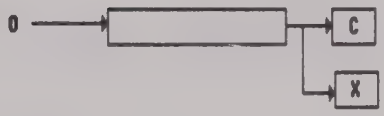


FIGURE B-4. LSR.

MOVE (move data)

Addressing Modes:

```
MOVE <ea>,<daea>
MOVEA <ea>,An
MOVEM <register list>,-(An)
MOVEM <register list>,<caea>
MOVEM (An)+,<register list>
MOVEM <cea>,<register list>
MOVEP Dn,d(An)
MOVEP d(An),Dn
MOVEQ £<imm>,Dn
MOVE <dea>,CCR
MOVE <dea>,SR
MOVE SR,<daea>
MOVE USP,An
MOVE An,USP
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
MOVE	Move data	B.W.L.	A A 0 0
MOVEA	Move address	W.L.	
MOVEM	Move multiple	W.L.	
MOVEP	Move to peripheral	W.L.	
MOVEQ	Move quick	L.	A A 0 0

MOVE to CCR	Move to CCR	W.	A A A A A
MOVE to SR	Move to SR	W.	A A A A A
MOVE from SR	Move from SR	W.	
MOVE USP	Move user stack ptr	L.	

Description: MOVE copies the contents of a source location into a destination location.

MOVEA is the same as MOVE except that the destination must be an address register. The flags are not affected.

MOVEM copies the contents of a specified list of registers on to the stack or into an area of memory. MOVEM is also used to retrieve data which has been stored previously by a MOVEM command. If the addressing mode used with MOVEM is a control mode or (An)+ then the registers are copied in the order D0 to D7 then A0 to A7. If the -(An) mode is used then the registers are loaded in the order A7 to A0 then D7 to D0.

MOVEP copies two or four bytes of data from a data register into *alternate* destination locations, or from alternate source locations into a data register. MOVEP is used with peripheral interface units to exchange data with peripheral devices. The interface units are configured in such a way that data must be passed in alternate rather than sequential byte units.

MOVEQ copies an 8-bit immediate value into a register. The higher three bytes of the register are sign extended by the operation. If the destination is an address register then the flags are not affected.

MOVE to CCR moves the low order byte of a 16-bit operand into the CCR byte of the status register.

MOVE to SR is a privileged instruction which moves a 16-bit operand into the entire status register.

MOVE from SR moves the entire contents of the status register into a destination location.

MOVE USP is a privileged instruction which copies the user stack pointer (A7) contents into an address register or vice versa.

MULS and MULU (multiply)

Addressing Modes:

```
MULS <dea>,Dn
MULU <dea>,Dn
```

Mnemonic	*Operation*	*Size*	Flags:	*N Z V C X*
MULS	Signed multiply	W.		A A 0 0
MULU	Unsigned multiply	W.		A A 0 0

Description: MULS multiplies a signed 16-bit source operand and a signed 16-bit destination operand with the 32-bit signed result being stored in the destination register.

MULU operates similarly but with unsigned operands and yields an unsigned result.

NBCD (negate decimal with extend)

Addressing Modes:

```
NBCD <daea>
```

Mnemonic	*Operation*	*Size*	Flags:	*N Z V C X*
NBCD	Negate decimal	B.		? A ? A A

Description: NBCD subtracts the binary coded decimal destination operand and the extend flag from zero and stores the result in the destination. Note that the Z flag is zeroed by a NBCD result greater than zero, otherwise it is unchanged.

NEG and NEGX (negate and negate with extend)

Addressing modes:

```
NEG <daea>
NEGX <daea>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
NEG	Negate binary	B.W.L.	A A A A A
NEGX	Negate with extend	B.W.L.	A A A A A

Description: NEG subtracts the destination operand from zero and stores the result in the destination.

NEGX subtracts the destination operand and the extend flag from zero and stores the result in the destination. NEGX is similar to NBCD except that it is a binary rather than a decimal operation. Note that the Z flag is zeroed by a NEGX result greater than zero, otherwise it is unchanged.

NOP (no operation)

Addressing Modes:

```
NOP (implicit)
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
NOP	No operation		

Description: NOP occupies two bytes of memory space in the code but has no effect, other than to advance the program counter by two.

NOT (logical not)

Addressing Modes:

```
NOT <daea>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
NOT	Logical complement	B.W.L.	A A 0 0

Description: NOT produces the 1's complement of the destination operand, storing the result in the destination.

OR, ORI, ORI to CCR and ORI to SR (logical OR operations)

Addressing Modes:

```
OR <dea>,Dn
OR Dn,<maea>
ORI #<imm>,<daea>
ORI #<imm>,CCR
ORI #<imm>,SR
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
OR	Logical OR	B.W.L.	A A 0 0
ORI	OR immediate	B.W.L.	A A 0 0
ORI to CCR	OR immediate to CCR	B.	A A A A A
ORI to SR	OR immediate to SR	W.	A A A A

Description: OR performs a logical OR operation between a source and a destination operand with the result being stored in the destination.

ORI performs the same function but the source operand must be an immediate value.

ORI to CCR ORs an immediate operand with the low order byte of the status register.

ORI to SR is a privileged instruction which ORs an immediate operand with the entire 16 bits of the status register.

Note that with ORI to CCR and ORI to SR the flags are affected according to the bit values of the immediate value. For every set bit in the immediate value the corresponding flag is set, otherwise it is unchanged.

PEA (push effective address)

Addressing Modes:

```
PEA <cea>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
PEA	Push effective addr.	L.	

Description: PEA calculates the effective address of the operand and pushes it, as a long word, on to the stack.

RESET (reset external devices)

Addressing Modes:

```
RESET (implicit)
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
RESET	Reset		

Description: RESET is a privileged instruction which resets the reset lines, resetting all external devices.

ROL, ROXL, ROR and ROXR (bit rotation instructions)

Addressing Modes:

```
ROL Dn,Dn
ROL #<imm>,Dn
ROL <maea>
ROXL Dn,Dn
ROXL #<imm>,Dn
ROXL <maea>
ROR Dn,Dn
ROR #<imm>,Dn
ROR <maea>
ROXR Dn,Dn
ROXR #<imm>,Dn
ROXR <maea>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
ROL	Rotate left	B.W.L.	A A 0 A
ROXL	Rotate left with extend	B.W.L.	A A 0 A A
ROR	Rotate right	B.W.L.	A A 0 A
ROXR	Rotate right with extend	B.W.L.	A A 0 A A

Description: ROL rotates the bits in the destination operand to the left, copying the original high order bit into the carry flag and also copying it into the least significant bit position. The source operand specifies the number of times the destination operand is rotated; if the source is immediate it must be in the range 1 to 8. A memory operand (maea) can only be rotated once and the operand must be word sized.

ROXL works the same way except that the high order bit is copied into the extend flag as well as the carry flag. The previous value of the extend flag is copied into the low order bit position.

ROR rotates an operand to the right, copying the low order bit of the operand into the carry flag and also copying it into the high order bit position.

ROXR is similar to ROR except that the low order bit is copied into both the carry and extend flags and the previous extend flag value is copied into the high order bit.

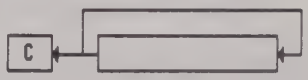


FIGURE. B-5. ROL.

FIGURE B-6. ROXL.

FIGURE B-7. ROR.

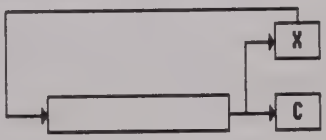


FIGURE B-8. ROXR.

RTE, RTR and RTS (return instructions)

Addressing Modes:

RTE (implicit)
RTR (implicit)
RTS (implicit)

Mnemonic	*Operation*	Flags: *N Z V C X*
RTE	Return from exception	A A A A A
RTR	Return & restore CCR flags	A A A A A
RTS	Return from subroutine	

Description: RTE is a privileged instruction which is used to return from exception subroutines to the program which was being executed before the exception was initiated.

RTR is used to return from a subroutine to the program which was being executed before the subroutine was called. The instruction pops the return address from the stack and copies it into the PC register and also pops the previous value of the CCR register from the stack and replaces it in the CCR. (The previous contents of the CCR must first have been saved at the beginning of the subroutine).

RTS is used as a straightforward return from a subroutine to the program which was being executed before the subroutine was called. It pops the return address from the stack and copies it into the PC register.

SBCD (subtract decimal with extend)

Addressing Modes:

```
SBCD Dn,Dn
SBCD -(An),-(An)
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
SBCD	Sub dec with extend	B.	? A ? A A

Description: SBCD subtracts a source operand, together with the value of the extend flag, from a destination operand and stores the result in the destination. The operation is performed using binary coded decimal arithmetic.

Note that the zero flag is zeroed by a non-zero result, otherwise it is unchanged.

Scc (set from condition)

Addressing Modes:

```
Scc <daea>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
Bcc	Set from condition	B.	

Description: Scc tests a destination byte operand for a specified condition. If the condition is true then the destination byte is set to the value 255, otherwise it is set to zero.

The conditions are incorporated in the instruction name: e.g. SEQ, SNE etc. and are as follows:

Condition	*Meaning*	*Flags*
CC	Carry clear	C=0
CS	Carry set	C=1
EQ	Equal	Z=1
F	False	0
GE	Greater or equal	(N=1 & V=1) or (N=0 & V=0)
GT	Greater than	(N=1 & V=1 & Z=0) or (N=0 & V=0 & Z=0)

HI	High	C=0 & Z=0
LE	Less or equal	Z=1 or (N=1 & V=0) or (N=0 & V=1)
LS	Low or same	C=1 or Z=1
LT	Less than	(N=1 & V=0) or (N=0 & V=1)
MI	Minus	N=1
NE	Not equal	Z=0
PI	Plus	N=0
T	True	1
VC	Overflow clear	V=0
VS	Overflow set	V=1

STOP (load status register and stop)

Addressing Modes:

```
STOP #<imm>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
STOP	Stop execution		A A A A A

Description: STOP is a privileged instruction which loads an immediate value into the status register increments PC and then stops all execution until a trace or external reset exception is initiated or until an external interrupt of sufficient priority occurs.

SUB, SUBA, SUBI, SUBQ and SUBX (binary subtraction)

Addressing Modes:

```
SUB <ea>,Dn
SUB Dn,<maea>
SUBA <ea>,An
SUBI #<imm>,<daea>
SUBQ #<imm>,<aea>
SUBX Dn,Dn
SUBX -(An),-(An)
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
SUB	Subtract binary	B.W.L.	A A A A A
SUBA	Subtract address	W.L.	
SUBI	Subtract immediate	B.W.L.	A A A A A
SUBQ	Subtract quick	B.W.L.	A A A A A
SUBX	Subtract with extend	B.W.L.	A A A A A

Description: SUB subtracts a source from a destination operand and stores the result in the destination.

The SUBA form of SUB specifies that the destination operand must be an address register and that the data size must be either word or long. Word sized results are sign extended. The flags are not affected.

The SUBI form of SUB specifies that the source operand must be an immediate value.

The SUBQ form of SUB specifies that the source operand must be an immediate value in the range 1 to 8. The flags are not affected if the destination is an address register.

The SUBX form of SUB specifies that the extend bit is added to the source operand before it is subtracted from the destination. Note that with SUBX the zero flag is set if the result is zero, otherwise it is unchanged.

SWAP (swap register words)

Addressing Modes:

SWAP Dn

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
SWAP	Swap register words	W.	A A 0 0

Description: SWAP exchanges the values of the hi and lo words of the specified data register.

TAS (test and set)

Addressing Modes:

```
TAS <daea>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
TAS	Test bit and set	B.	A A 0 0

Description: TAS tests the byte contained in the effective address specified in the instruction and sets or resets the N and Z flags according to its value. The high order bit of the operand is then set. No other processor may access the operand while the instruction is being executed.

TRAP & TRAPV (trap exceptions)

Addressing Modes:

```
TRAP £<imm>
TRAPV (implied)
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
TRAP	Trap exception		
TRAPV	Trap if overflow		

Description: A TRAP instruction forces a trap exception, diverting execution to one of 16 trap handling subroutines as specified by the immediate operand in the instruction (in the range 0 to 15).

TRAPV forces a TRAPV exception if the overflow flag is set at the time the instruction is executed.

TST (test)

Addressing Modes:

```
TST <daea>
```

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
TST	Test	B.W.L.	A A 0 0

Description: TST compares the specified operand with zero, altering the condition flags according to the result.

UNLK (unlink)

Addressing Modes:

UNLK An

Mnemonic	*Operation*	*Size*	Flags: *N Z V C X*
UNLK	Unlink		

Description: UNLK reverses the operation of the LINK instruction, relinquishing a stack frame. The contents of the specified address register are loaded into A7 and the long word on top of the stack is then loaded into the address register. See also 'LINK'.

Appendix C

Conversion Table

		WORD				
	x16	BYTE 2		BYTE 1		
HEX	DIGIT 5	DIGIT 4	DIGIT 3	DIGIT 2	DIGIT 1	BINARY
0	0	0	0	0	0	0000
1	65536	4096	256	16	1	0001
2	131072	8192	512	32	2	0010
3	196608	12288	768	48	3	0011
4	262144	16384	1024	64	4	0100
5	327680	20480	1280	80	5	0101
6	393216	24576	1536	96	6	0110
7	458752	28672	1792	112	7	0111
8	524288	32768	2048	128	8	1000
9	589824	36864	2304	144	9	1001
A	655360	40960	2560	160	10	1010
B	720896	45056	2816	176	11	1011
C	786432	49152	3072	192	12	1100
D	851968	53248	3328	208	13	1101
E	917504	57344	3584	224	14	1110
F	983040	61440	3840	240	15	1111

FIGURE C-1. Conversion Table.

The above table is designed for rapid conversion between decimal, hexadecimal and binary numbers. The columns headed Digit 1, HEX and BINARY contain the decimal, hexadecimal and binary values 0 to 15. Columns 'Digit 1' and 'Digit 2' together represent the lo and hi order nibbles of a single byte. Columns 'Digit 3' and 'Digit 4' together represent the hi byte of a 16-bit number. Column 5 represents the decimal value of the high order nibble of a 20-bit number.

Converting from hexadecimal to decimal

Look up the least significant hexadecimal digit in the HEX column and read across to the corresponding decimal value in Column 'Digit 1'. Look up the second least significant hexadecimal digit in the HEX

column and read across to the corresponding decimal value in Column 'Digit 2'. Repeat this process for each hex digit and then add the decimal values obtained. For example, to convert hex A24B3:

Hex		*Decimal*
3	=	3
B	=	176
4	=	1024
2	=	8192
A	=	655360
	=	664755

Converting from decimal to hex

Locate the nearest number in the table which is less than or equal to the decimal number and read off the corresponding hex digit in the HEX column. Subtract the decimal number in the table and repeat the above procedure until the result equals zero. For example, to convert decimal 754368 to hex:

	Decimal		*Table*	*Hex*
	754368	–	720896	B
=	33499	–	32768	8
=	731	–	512	2
=	219	–	208	D
=	11	–	11	B
=	0			

754368 = B82DB

Converting from decimal to binary

Follow the same procedure as for decimal to hex but substitute the numbers from the BINARY column for those in the HEX column. For example, to convert decimal 75436 to binary:

	Decimal		*Table*	*Binary*
	754368	—	720896	1011
=	33499	—	32768	1000
=	731	—	512	0010
=	219	—	208	1101
=	11	—	11	1011
=	0			

754368 = 1011 1000 0010 1101 1011

Converting from binary to decimal

Divide the binary numbers into 4-bit sections and add the corresponding decimal values, starting with the low order nibble. For example, to convert 1001 1101 1010 0011 to decimal:

Nibble	*Col No.*	*Decimal*
0011	1	3
1010	2	160
1101	3	3328
1001	4	36864
	=	40355

Converting from hex to binary and binary to hex

Each hex digit corresponds to a 4-bit binary value. Use the HEX and BINARY columns for direct conversion.

For example, hex 2AD46 is

```
0010 1010 1101 0100 0110.
---- ---- ---- ---- ----
  2    A    D    4    6
```

Index

First Steps in 68000 Assembly Language

Owners of Motorola 68000-based micros, such as the Apple Macintosh, Commodore Amiga and Atari 520 & 1040 STs, enjoy some of the best user interfaces available. All of these computers have been designed to shield the user from the technical side of machine operation. However, some working knowledge of assembly language is needed in order to fully understand the way data is processed inside the machine. These computers, which utilise an extremely sophisticated machine architecture, can appear to be difficult to master. **First Steps in 68000 Assembly Language**, however, describes the ins and outs of assembly language and shows that it can be easily learned.

This book clearly explains the meaning of all the basic assembly language details, including data storage and data addressing; the use of registers, flags and stacks; conditional branching and referencing indexed tables of data. These concepts are presented in a clear and concise manner with many illustrations to guide the reader.

Topics covered include:

- Mnemonics and machine code – converting instructions to numbers
- Memory, Registers and Stacks – storing data for all occasions
- Flags – making conditional decisions
- Branching – redirecting program execution
- Indexing – addressing data

Robert Erskine, the founder of the software house Microgame Simulations, is the author of a number of books on computer programming and two best-selling computer games. He has broadcast regularly about computing and written many articles for the computer press.

£12.50

ISBN 1-85181-081-1

9 781851 810819

GLENTOP

Glentop Press Ltd
Standfast House
Bath Place
Barnet
Herts EN5 5XE