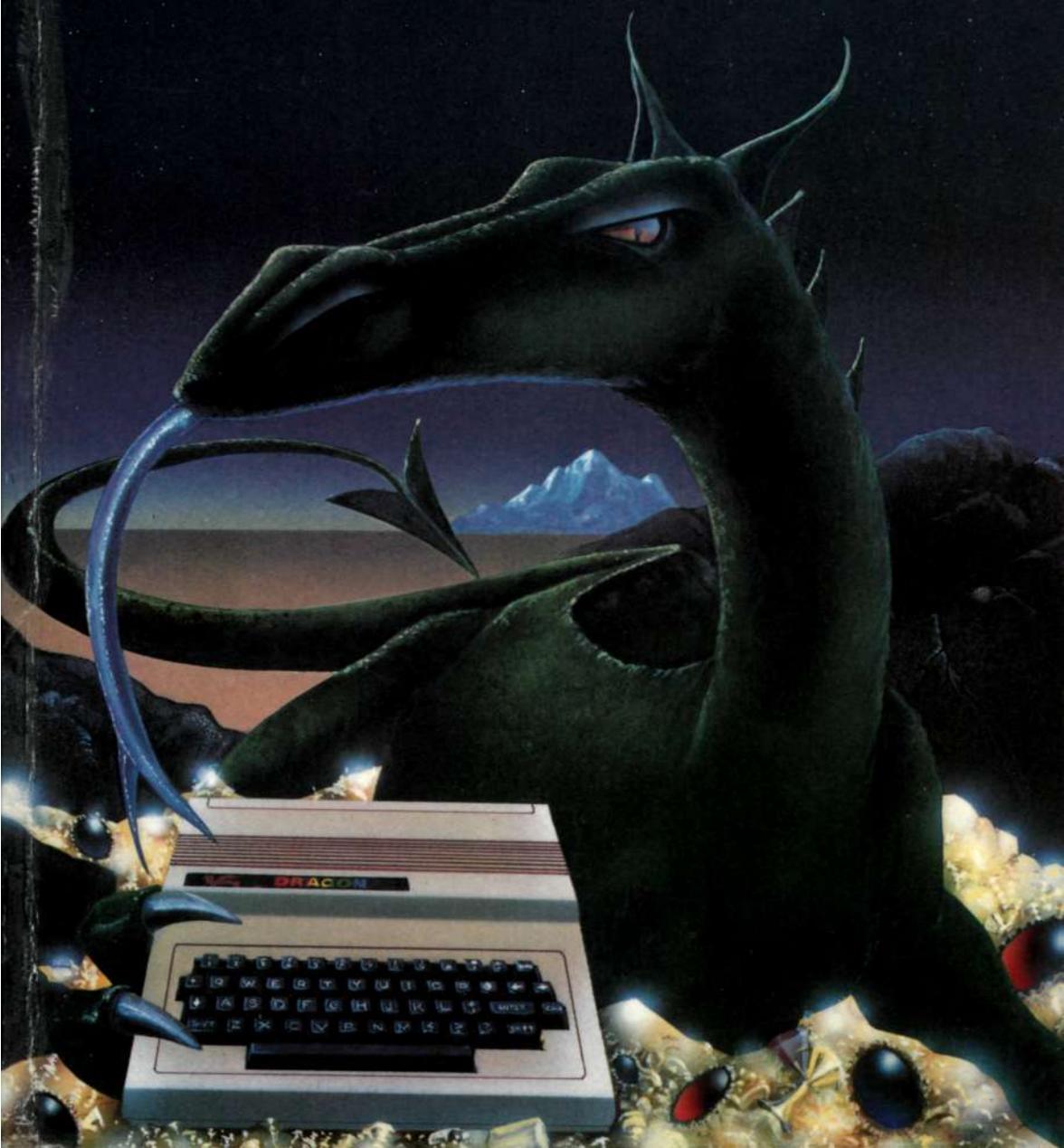


Inside the Dragon

Duncan Smeed and Ian Sommerville


DRAGON DATA
APPROVED



Inside the Dragon

Duncan Smeed
Dragon Data Ltd

Ian Sommerville
University of Strathclyde



Addison-Wesley Publishing Company

London • Reading, Massachusetts • Menlo Park, California • Amsterdam
Don Mills, Ontario • Manila • Singapore • Sydney • Tokyo

© 1983 Addison-Wesley Publishers Ltd.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior written permission of the publisher.

Set by the authors in Bookface Academic using NROFF, the UNIX text processing system, at the University of Strathclyde, Glasgow

Cover design by David John Rowe

Printed in Finland by OTAVA. Member of Finnprint.



British Library Cataloguing in Publication Data

Smeed, Duncan

Inside the Dragon.

1. Dragon 32 (Computer)

I. Title II. Sommerville, Ian

001.64'04 QA76.8.D77

ISBN 0-201-14523-5

ABCDEF 89876543

Contents

Chapter 1	Introducing the Dragon	1
1.1	Information representation	3
1.2	Processor architecture	9
1.3	The organisation of the Dragon	14
Chapter 2	The architecture of the M6809	20
2.1	The M6809 register set	21
2.2	Addressing modes on the M6809	26
2.3	Memory-mapped input/output	36
Chapter 3	The M6809 instruction set	38
3.1	Data movement instructions	41
3.2	Arithmetic instructions	46
3.3	Logic instructions	51
3.4	Test instructions	55
3.5	Branch instructions	57
3.6	Interrupt handling instructions	62
3.7	Miscellaneous instructions	63
Chapter 4	Introducing assembly language	65
4.1	The assembler program	69
4.2	Assembler directives	75
4.3	Example programs	81
Chapter 5	From BASIC to assembly code	84
5.1	Assignment statements	84
5.2	Conditional statements	89
5.3	Loop constructs	96
5.4	Goto statements	101
5.5	Input and output	101
5.6	Subroutines	105
5.7	Arrays	107
5.8	A machine code monitor	111
Chapter 6	Subroutines and strings	121
6.1	Assembly language subroutines	122
6.2	Character strings	132
6.3	String manipulation routines	137
6.4	Position-independent code	142
6.5	Combining assembly language with BASIC	148
Chapter 7	Graphics programming	151
7.1	Graphics display hardware	152

7.2	Integrating BASIC and assembly code graphics	157
7.3	Alphanumeric display modes	159
7.4	Colour graphics display modes	161
7.5	Resolution graphics display modes	164
7.6	Semigraphics display modes	166
7.7	Graphics utilities	169
7.8	Designing and implementing graphics programs	174
Chapter 8	Input/output programming	186
8.1	Interrupts	188
8.2	Input/output programming techniques	193
8.3	The peripheral interface adaptor - PIA	198
8.4	Input/output devices	201
Chapter 9	Dragon hints and tips	224
9.1	Power-up/Reset actions	224
9.2	BASIC program storage	226
9.3	BASIC'S information representation	230
9.4	Passing parameters from BASIC to machine code	235
9.5	Extending the DRAGON's capabilities	238
9.6	BASIC system variables	244
Reading list		249
Appendix 1	MC6809E data sheet	251
Appendix 2	SN74LS783 data sheet	286
Appendix 3	MC6847 data sheet	312
Appendix 4	MC6821 data sheet	322
Appendix 5	The Dragon 64	333
Appendix 6	The ASCII character set	342
Appendix 7	Dragon-specific tables	343
Appendix 8	The disk operating system	350
Index		354

Preface

The advent of the microchip has resulted in the invention of a product which, ten years ago, was completely unthinkable. This product is the personal computer and there are now millions of families who own their own computer. This book is about one such machine, the Dragon.

The Dragon is a second-generation personal computer. In contrast to early personal machines which were slow, had small memories and low-resolution monochrome displays, the Dragon offers a fairly large memory, high-resolution colour graphics, sound synthesis and a professional-quality keyboard. There are two versions of the Dragon available, the Dragon 32 and the Dragon 64, and the material in this book is relevant to both of these machines.

Personal computers are remarkable value for money. Most of them are more powerful than machines of the early 1960's which cost hundreds of thousands of pounds or dollars. Furthermore, personal machines are well-built and reliable, much more so than early large computers. However, the weakest aspect of most personal machines is the descriptive documentation provided with the machine. Whilst this is no real hardship to those who only use their machine for game playing, the hobbyist who wishes to make the most of his machine has a tough time finding out technical details of his system.

This book is intended for such readers and for those readers who have explored the BASIC programming capabilities of their machine and now want to go further. We do not assume any technical knowledge of computing apart from an ability to write and understand BASIC programs. Inevitably, this means we must include some introductory material which can be skipped by readers with experience in computing.

When this book was written, the only Dragon available was the Dragon 32. As a result, the material here was written for that machine but most of the examples are equally relevant to the Dragon 64. Time has not permitted us to include Dragon 64 details in the text, but we have provided an appendix (Appendix 5) summarising the differences between the Dragon 32 and the Dragon 64. We have also included an appendix

(Appendix 8) which covers details of the Dragon's disk operating system.

Many readers will be aware that the Dragon and the Tandy Color Computer make use of the same M6809 processor chip and the same BASIC system developed by Microsoft. As a result, much of the material here is also relevant to the Tandy machine and users of that system may be able to pick up useful hints and tips from it.

The book is about the internal workings of the Dragon rather than about programming. We describe the M6809 processor which is used in the Dragon and show how machine code programs for that processor can be written in assembly language. We also describe the graphics system and the input/output system on the Dragon and, finally, we provide bits and pieces of technical information which may be valuable to the assembly code programmer.

It is impossible for us to be comprehensive in our discussions of assembly code programming, graphics, or whatever. Rather, we provide Dragon-specific details rather than an extensive discussion of general techniques. We hope to encourage the reader to delve further into these application areas and we provide a reading list which will help you get more information about specific techniques.

Printing programs in a book like this can sometimes be very untidy. Accordingly, we have taken some liberties with program commenting and have used lower case letters for commenting in all of our programs. We may also have made some other minor changes to the program layouts so that they are easier to read but the actual program code has not been changed.

There are many people who have contributed in one way or another to the ideas and techniques presented in this book amongst them our colleagues at the Department of Computer Science, University of Strathclyde. We would also like to express our gratitude to those at Dragon Data Ltd., in particular to Tony Clarke, Richard Wadman and Derek Williams. Permission to use the Dragon logo in our examples was kindly granted by Dragon Data Ltd.

Finally, special thanks must go to our families especially our wives Pauline Smeed and Anne Sommerville for their support, encouragement and tolerance of lost evenings and weekends throughout the writing of this book.

Ian Sommerville
Duncan Smeed
August 1983

Chapter 1

Introducing the Dragon

Every computer, be it mainframe, minicomputer or microcomputer, is made up of a very large number of electronic components which can be viewed at greater or lesser levels of detail. At the highest level, the computer can be considered as an organised collection of devices namely:

- (1) A processor.
This is the device which actually carries out the computations (add, multiply, compare etc.) on elements of data.
- (2) A store.
This is the device which is used to store information so that it may be readily accessed by the processor. This information can be transferred to and from other system devices.
- (3) One or more peripheral controllers.
Every computer needs some way of getting information from and passing information to the outside world. This is accomplished through peripheral devices such as floppy disks, printers, keyboards, video displays, etc. Each of these devices needs a controller built into the computer system to ensure that information is properly transferred to and from the processor and memory.
- (4) A clock.
This is not a clock to tell the time but is really a pulse generator which ensures that the operation of all the other devices making up the system is synchronised.

There are various different ways of connecting these devices together so that they operate as a computer. One of the most common interconnection techniques, particularly in minicomputer and microcomputer systems, is to connect all the system devices to a common data highway. This connection is sometimes called a bus. A diagram of such an interconnection is shown in Figure 1.1 where P1, P2, and P3 are peripheral controllers.

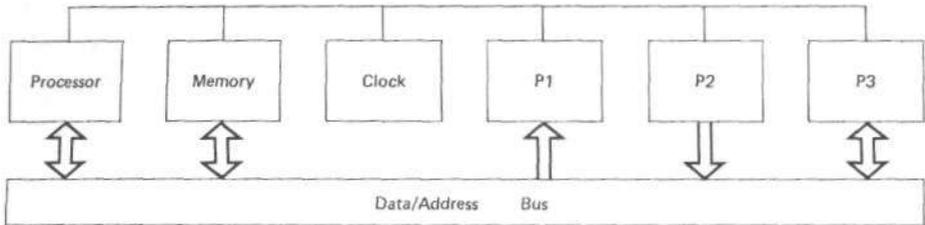


Fig. 1.1 Microcomputer organisation

Notice that the clock has a separate connection to the other system components and that some of the peripheral devices are 'one-way' devices. For example, a printer is a write-only device - you can only transfer information to it, and a keyboard is a read-only device - you can only transfer information from it.

On microcomputer systems (like the Dragon), the processor is built onto a single microchip as are each of the peripheral controllers. The memory is normally built as a number of connected microchips.

These chips are bonded into holders which have a number of pins sticking out of each edge. Some of these pins are connections to the data highway and others are connections to control lines (like the clock connection). The number of pins on a chip depends on the type of information which must be transferred and the number of control signals input and output. Normally, more complex chips, like microprocessor chips, have more pins than (relatively) simple peripheral controller chips.

The next level down from the computer organisation is sometimes called the computer architecture. In the same way as a building has an architecture which is an overall structure tailored to the building's users, so too does a computer. In the case of a computer, however, the architecture is the structure as seen by computer programs running on the machine. Just as building architecture is seen as an organisation of rooms, corridors, walls, etc. rather than an organisation of elementary components such as bricks, floorboards and pipes, computer architecture is not concerned with basic electronic logic components. Rather, it is the collection of these components into larger functional units.

The computer architect is mostly concerned with the design of the processor and how it can be set up to transfer information to and from other system components. The most important of these is the store.

Therefore, a major part of the architect's job is to design the processor so that it makes optimum use of the system's memory.

In this chapter, we introduce basic ideas of how information is represented in a computer and we describe, in general terms, the principles of computer architecture. We then go on to describe the Dragon's hardware organisation and the chapter concludes with a description of how the Dragon's memory is used.

1.1 INFORMATION REPRESENTATION

At their most fundamental level, all the components of a computer are fabricated out of electronic switches which can only be in one of two states - they can be on or off. This means that the ideal way to represent information in a computer is as a binary pattern, a pattern of 1s and 0s. These patterns can represent numbers, characters, states of a device, colours, etc. As long as the interpretation of a pattern is known in advance, any information can be encoded in binary form.

The most common use of binary patterns in a computer is to represent numbers. Binary numbers are numbers whose base is 2 and digits in a binary number represent powers of 2. For example, the binary number

10010111

can be converted to our more familiar decimal notation by considering it to be:

$$1(2^7) + 0(2^6) + 0(2^5) + 1(2^4) + 0(2^3) + 1(2^2) + 1(2^1) + 1(2^0)$$

If we carry out these multiplications and additions, we find that the above binary number represents the decimal number 151. Starting from the right, each place in a binary number represents an increasing power of 2. This is a familiar idea which is the basis of all modern number systems. Decimal numbers, numbers whose base is 10, are organised so that each place represents a power of 10. Therefore, the number 3506 can be considered as:

$$3(10^3) + 5(10^2) + 0(10^1) + 6(10^0)$$

The number of distinct numerals needed to represent any number depends on the base of that number system. In general, if the number system base is m , $m-1$ distinct numerals plus zero are needed. Therefore, for the decimal system we need the numerals 1, 2, 3, 4, 5, 6, 7, 8, 9, 0. For a hexadecimal system, whose base is 16, these must be extended with extra symbols representing 10, 11, 12, 13, 14, 15 and the numeral set is 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 0. The

binary system has a base of 2 so only a single digit, 1, plus 0 is needed in the representation of any binary number.

Normal arithmetic operations such as subtraction, addition, multiplication, and division can be carried out on binary numbers in exactly the same way as on decimal numbers. The following sums show examples of binary arithmetic.

11001101	10011001
+01101101	-00010111
 100111010	 10000010

The rules to remember are that $1 + 1$ is 0 carry 1 and that $0 - 1$ is 1 borrow 1.

The other computations ($0 + 0 = 0$, $1 + 0 = 1$, $1 - 1 = 0$, $1 - 0 = 1$) are as you would expect and have no associated carry or borrow.

Binary arithmetic is tedious and error prone for humans but, fortunately, is very straightforward for computers. It is relatively easy to build logic circuits which add binary numbers and, as we shall see later in this section, these are all that are required to implement all the arithmetic operations of add, subtract, multiply, and divide.

Normally, when we write down numbers their length is unbounded. That is, each number can have as many digits as we like. The designer of a computer memory, however, doesn't have this flexibility. Computer memory is made up of many distinct cells each of which can store a fixed number of binary digits or bits. Normally, each cell stores 8 bits (a byte) and the number of bits used to represent a number must be a multiple of 8. Combinations of 2 or more bytes used to store numbers are usually called a machine word.

The bytes in the computer's memory each have a unique address which distinguishes that byte from all others. Addresses are simply numbers which start at zero and increase by 1 for each byte. On a microcomputer like the Dragon there are 32768 bytes in user memory so addresses range from 0 to 32767. For convenience, memory bytes are divided into blocks of 1024 (called 1K) so we say that the Dragon has 32K or 64K bytes of store.

An analogy can be drawn between a computer's memory and the lockers in a sports stadium. Each locker has a number (its address) which distinguishes it from all other lockers and items can be stored in the locker. The locker number doesn't affect what's stored in it nor does the memory address in a computer. The byte with address number 23456 can have any number in it. Just as the lockers in a stadium can have names associated with them as well as numbers (John Brown's

locker, Mary Jones's locker etc.) so too can memory bytes. Names are often more convenient than numbers when referring to memory bytes and we shall see in a later chapter how this facility can be used.

On most microcomputers, the number of bits used to represent an integer (a number without a fraction) is 16, with 32 bits used to represent real numbers (numbers with fractions). This means that integers occupy 2 memory bytes and real numbers occupy 4 memory bytes. This size limitation restricts the magnitude of numbers which can be directly stored and manipulated by the computer and it is very important that the computer user bears this in mind when using his machine for numeric computations.

However, the restriction on the number of digits in a number has a hidden advantage. It allows us to represent negative numbers in such a way that the operation of subtraction can be carried out by adding the numbers concerned. This representation of negative numbers is called two's complement representation.

Complement arithmetic, which depends on numbers having a fixed, maximum number of digits, works with numbers of any base. The numbers involved, however, must have a special binary tag, called a sign bit, which indicates whether the number is positive or negative. Negative numbers have a sign bit of 1, positive numbers a sign bit of 0.

We illustrate the principles of complement arithmetic using decimal numbers rather than clumsy binary numbers but we assume that the maximum length of a number is 3 digits. That is, we place the restriction on our number system that only numbers from 0 to 999 may be represented. Say we want to carry out the subtractions $327 - 104$ and $96 - 297$. These are, of course, equivalent to the additions $327 + (-104)$ and $96 + (-297)$. The results of these additions are, in the first case, 223 and in the second -201.

Positive numbers in complement notation are represented by the number itself with an associated sign bit of 0. Therefore, 327 is 0327 and 96 is 0096. The value of negative numbers in complement notation is formed according to the following formula:

$$(\text{maximum possible number}) + 1 - (\text{absolute number value})$$

Therefore, where 999 is the maximum possible number, -104 and -297 have the following complement representations:

$$(999 + 1 - 104) = 1896$$

$$(999 + 1 - 297) = 1703$$

Notice that we have added a sign bit (=1) to the left of the number to indicate that it is a negative number.

The subtractions above can now be carried out by adding the numbers in complement form. In the first case, $0327 + 1896 = 2223$. However, because the sign bit is always binary, 2 is actually '10' so we get an answer of '10'223. Because the length of the number is restricted, we throw away the 1 in the leftmost position to get the correct answer 0223.

Similarly, $96 - 297$ is $0096 + 1703 = 1799$. This is a negative number (sign bit = 1), so we must convert it back to our more conventional representation using the same formula as was used to convert to complement form. The conversion therefore is:

$$-(999 + 1 - 799) = -201$$

This whole business might seem to be a bit of a fiddle with digits being discarded in an apparently arbitrary fashion and with binary and decimal numbers being mixed up in the sign bit and the number itself. However, it can be mathematically proven that complement arithmetic always works. The proof isn't relevant here - what is relevant is that two's complement works very well on computers and that it is very easy to form the two's complement of any binary number.

To form the two's complement of a binary number, all the 1 bits are changed to 0 and all the 0 bits to 1. This operation is called complementing. One is then added to the number to get the two's complement representation. For example, the binary numbers 01101100 and 00101101 have two's complements 10010100 and 11010011 respectively. The leftmost bit is the sign bit and operations on it fit in naturally with other binary arithmetic.

Notice, however, that the need for a sign bit reduces the maximum and minimum numbers that can be represented on a computer. On a machine which uses 16 bits to represent integers, the leftmost bit must be the sign bit so only 15 bits are used for the number representation. This means that the largest positive integer on such a machine is 32767 and the largest negative integer is -32768. It is left as an exercise for the reader to work out why there is one extra negative number.

Normally, microprocessors are only equipped with hardware units which allow them to add numbers together. Subtraction is implemented as described above and multiplication and division are implemented in software as sequences of repeated additions for multiplications and subtractions for division.

So far, we have concentrated on the representation of numbers in a computer but character processing is at least as important as numeric computation for most microcomputer users. As we said at the beginning of this section, anything can be represented as a binary

pattern as long as we know how to interpret it so characters are normally held in a memory byte as an 8-bit binary pattern.

There exist a number of different conventions governing which patterns represent which characters but the most commonly used representation on microcomputers is the ASCII (standing for American Standard Characters for Information Interchange) representation. Under this system, 7 bits are used for character representation and the 8th (leftmost) bit is always zero. As well as codes for the upper and lower case letters, 'A'-'Z', 'a'-'z', the digits, '0'-'9', and punctuation characters the ASCII system also defines special unprintable characters meaning 'end of transmission', 'ring a bell', 'please acknowledge', etc. A table of characters and their associated ASCII values is provided in Appendix 6.

1.1.1 Hexadecimal notation

The sequences of 1s and 0s which make up binary numbers are very awkward for people to use. Because the numbers are so long, it is very easy to miss out a digit or to interchange a 1 and a 0. Naturally, this changes the value of the number and this can completely change the meaning of a computation.

Ideally, it is best to work in terms of decimal numbers and names because these are the types of symbol that we learn to manipulate at an early age. However, it is, unfortunately, sometimes necessary to talk in the computer's terms, that is, in binary. A shorthand notation for binary numbers allowing us to write down binary equivalents in as few digits as possible reduces the number of errors which we make. Hexadecimal notation is one possible shorthand for binary numbers.

Hexadecimal numbers are numbers whose base is 16. This means that the rightmost hexadecimal (hex for short) digit represents 0-15, the next digit represents the number of 16s to the power 1, the next the number of 16s to the power 2 and so on. As discussed earlier, we need 15 digits plus zero for a number system whose base is 16. The hexadecimal digits are:

0 1 2 3 4 5 6 7 9 A B C D E F

The number 10 is represented by A, 11 by B, 12 by C, 13 by D, 14 by E and 15 by F. Some examples of hexadecimal numbers and their associated decimal values are:

9	9
1F	31 (16 + 15)
23	35 (2(16) + 3)
C7	199 (12(16) + 7)
FF	255 (15(16) + 15)
5BE	1470 (5(256) + 11(16) + 14)

It is very easy to convert from binary numbers to hexadecimal numbers and vice versa. Hexadecimal numbers represent values from 0 to 15 and this is exactly 2^4 to $2^4 - 1$. We need 4 binary digits to make a hexadecimal digit so converting from binary to hexadecimal involves chopping the binary number into groups of 4 bits and then writing down the associated hexadecimal digit. For example:

```
10110111010110111  16EB7
1110011011011100   E6DC
```

Conversion from hexadecimal to binary is equally easy as long as you memorise the binary patterns for the digits from 0 to F. These are:

0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

These patterns can be calculated very easily but after using binary and hexadecimal numbers for a while, you will find that you have, in fact, memorised them. Some examples of hexadecimal binary translations are:

```
A1C4      1010000111000100
4FFF      010011111111
5670      0101011001110000
```

As you read through the book, you will see lots more examples of hexadecimal numbers as we always use them in preference to binary when discussing particular representations. In particular, we always use hexadecimal numbers to refer to memory addresses so when you see an address of 433, say, this is hexadecimal 433 which is decimal 1075.

1.1.2 Decimal arithmetic

One of the problems which arise when binary arithmetic is used in a computer, where 16-bit words are used to store integer numbers, is that the maximum integer which can be represented is 32767 and the minimum integer is -32768. One way round this is to use so-called 'decimal notation' where numbers are represented as a sequence of digits rather than in absolute binary form.

From the table above, it is clear that the representation of the digits 0-9 requires that 4 bits be set aside for each digit. Therefore, each memory

cell can hold 2 digits. The table below shows examples of numbers represented in both decimal and binary form.

Number	Binary representation	Decimal representation
2	00000010	00000010
55	00110111	01010101
438	000110110110	010000111000
2583	101000010111	00100101110000011

There is a marked difference between the decimal and the binary representation of a number so special routines are required to perform decimal arithmetic. Although decimal numbers take up more space than their binary equivalents, they have the advantage that it is easier to write special routines to perform arithmetic on large decimal numbers than it is to write such routines for binary numbers whose representation requires more than 16 bits. The Dragon has an in-built instruction, called Decimal Adjust, to help the programmer in writing such routines.

Although decimal arithmetic is very important for commercial applications programs, the hobbyist and scientific computer user has no real need of it. We have introduced it here for completeness but we do not use it in this book. Rather, we assume that all numbers may be represented as integers in the range -32768 to 32767.

1.2 PROCESSOR ARCHITECTURE

The central device in a microcomputer system like the Dragon is the microprocessor chip. The processor is that device which carries out all data transformations. That is, given input information, the processor can manipulate this and transform it to the output required by the programmer. The function of a computer program, be it in BASIC or some other programming language, is to define how the processor should transform its input into the appropriate output.

The processor has an internal structure, its architecture, which consists of lower level components and their interconnections. As far as the programmer who wants to get the most out of his machine is concerned, the most important of these components are the processor registers.

A register is simply an electronic device which can be used to store information. Usually, its width (the number of bits it can hold) is equal to or some multiple of the basic memory cell size. In the Dragon's processor, register widths are either 8 or 16 bits and they can therefore hold 1 or 2 memory bytes.

There are two important distinctions between a register and an ordinary memory byte or word:

- (1) The processor can access information in a register more quickly than it can access information in a memory cell. The reason for this is partly due to the way in which registers are constructed and partly due to the fact that a bus transfer between processor and memory is not required.
- (2) Registers may be connected, via an internal processor bus, to other processor components which can transform information held in registers or which can recognise particular data patterns in the register. These patterns can be used to trigger corresponding actions by other processor components. The most important of these components, which are present in every processor, are the arithmetic and logic unit (ALU) and the control unit. These are discussed later in this chapter.

Registers in a processor may be classified as either general-purpose registers or as special-purpose registers. General-purpose registers may simply be thought of as extensions of the computer's memory. Normally, information which is accessed very frequently is held in such registers. It is up to the programmer to transfer frequently accessed information to general-purpose registers before it is accessed and to save it in memory when the register is needed for other purposes.

Special-purpose registers may also be used to store frequently accessed information. However, instead of general information, that is, anything the programmer wants, being stored in such registers particular items of information are always held there. Other types of special-purpose register are accumulator registers and condition-code registers which are used as ALU input and output registers.

The notion of an arithmetic and logic unit has already been introduced. This is a component whose function is to carry out arithmetic operations such as add, negate, etc. and logical operations such as compare, complement, etc. The particular operations available on the Dragon are described in a later chapter - you don't need to know these details to understand the general purpose of an ALU.

Accumulator registers are those registers which may act as ALU inputs and outputs. It is not usual to connect all registers to the ALU. Rather, only one or two accumulator registers are directly connected to this unit and all traffic to and from the ALU must pass through these accumulators.

When some arithmetic and logical operations take place, particular conditions resulting from these operations must be 'remembered' for subsequent

operations. For example, if two values are compared for equality, it must be remembered whether they are equal or not. Similarly, if an addition produces a carry, this must be remembered. It is the function of the condition-code register (CCR) to hold this kind of information for subsequent use by the programmer. The exact conditions noted in this register differ from machine to machine - the details of the Dragon's CCR are described in the following chapter.

Although general arithmetic operations must all take place through the accumulator registers in a processor, it is sometimes possible to perform very limited addition and subtraction operations in other special-purpose registers. These operations can take place automatically before or after the contents of a register are accessed. Typically, this auto-increment/decrement facility allows 1 or 2 to be added or subtracted from the value in the register. This is particularly useful when using so-called index addressing where a register contains the address of a memory location. Indexed addressing is fully described in the next chapter of this book.

We have already introduced the idea that a computer program specifies how program input is transformed to the appropriate output. Writing a program in BASIC, say, is a convenient way for the user to specify this transformation but, at the processor level, a BASIC program can't be directly executed.

Rather, a translation process must take place where the BASIC program is converted to a sequence of primitive machine instructions. This sequence specifies the information transfers between the computer's memory and the processor and the operations (add, compare, etc.) to be carried out on this information.

Within the processor, the machine instructions always make use of the processor's registers. Some instructions are dedicated to data movement to and from memory, some to arithmetic and logical operations, and some to controlling the order of execution of the instruction sequence.

Each instruction has a unique operation code (op-code) which distinguishes that instruction from all others. This op-code is simply a binary number which is used by the control unit in the processor to determine which operation to carry out. As binary numbers (or even hexadecimal numbers) are alien to humans, we normally refer to instructions by means of a mnemonic related to the function of the instruction. Typical instruction mnemonics are:

LD	Transfer (Load) information into a register
CLR	Set a register to zero (CLear)
INC	Add 1 (INCRement) the contents of a register.

As well as an op-code, each instruction may have one or more address fields which specify the registers and/or memory locations used by the instruction. These address fields specify where the instruction can find the data on which it operates (its operands). They can be specified in a number of different ways (addressing modes) and an understanding of these addressing modes is vital for the programmer who wishes to write his own machine language programs. Because, they vary so much from machine to machine, addressing modes are not discussed further here but those of the Dragon's processor are covered in the following chapter.

The machine instructions making up a program are themselves stored in the computer's memory and are fetched, one by one, from the memory to the processor. Each instruction may occupy one or more memory cells - in the Dragon, for example, instructions may take up 1, 2, 3, 4 or 5 bytes.

The processor control unit fetches instructions from memory, identifies each instruction and initiates those components which actually carry out the specified operations. In every processor there is a special-purpose register called the program counter (PC) which holds the memory address of the next instruction to be executed by the processor.

There is no direct way for the programmer to affect the operation of the processor's control unit in its fetching and decoding of the machine instructions. However, the address in the PC register can be changed by the programmer thus allowing him to modify the order in which instructions are executed. This facility means that it is possible to repeat groups of instructions (loops) and to skip over one or more instructions if some particular condition holds (conditions). To the BASIC programmer, the familiar forms of these are FOR statements and IF statements.

1.2.1 Stacks

The machine instructions for a particular program are normally held in a linear sequence of cells in the computer's memory. This sequence may be accessed in any order by modifying the value of PC so that the instruction to be executed is the next one fetched by the processor's control unit.

Sometimes it is also convenient to store and access data in the same way. You may normally access the data sequentially using a register to hold the address of the next data item to be selected. By modifying the value in this register, you can change this sequential data access pattern and get to any item of data which you need.

On other occasions, however, it is convenient to restrict the way in which data is accessed. Restrictions of this sort are not arbitrary but are a

safety feature which reduces the chance of the programmer making mistakes. There are various different ways in which restrictions can be applied and the particular technique chosen must depend on the application being programmed. For a full discussion of these data structures the reader must look at the specialised texts on this topic such as those suggested in the reading list. However, one of these data structures is so important that you must understand it if you are to understand the rest of the book. This structure is the stack.

Arranging data in a stack is a technique of limiting data storage and access so that the last data item placed on the stack is the only item which may be removed from the stack. Once this item has been removed, we can then get to the item below it, remove it, and so on.

This can be imagined by comparing the data on the stack to a pile of plates in a restaurant kitchen. Assume that a dishwasher is adding plates to this pile after cleaning them and that a waiter is removing plates for serving food. The plate which the waiter takes from the pile is always the last plate put on the pile by the dishwasher. Like a data stack, the pile of plates is a last-in, first-out (LIFO) structure. Items are removed in the inverse order to that in which they are placed on the stack.

Stacks are easily implemented in a computer system by reserving an area of memory for the stack and by associating a special-purpose register called a stack pointer (SP) with this memory area. The stack pointer always holds the address of the last item placed on the stack, that is, the top of the stack. When an item is added to the stack, the SP register is incremented and the item placed at this address. When an item is removed from the stack, the item pointed to by SP is first copied to a register and SP is then decremented to point at the new top stack element.

In the traditional stack model, the base of the stack is at a low memory address and the stack grows upwards so that elements placed on the stack have increasing memory addresses. However, this is an arbitrary convention and it is equally straightforward to implement a stack which grows downwards in memory. This means that push in element on the stack involves decrementing the stack pointer and popping an element from the stack involves incrementing the stack pointer.

Stacks in the Dragon are implemented in this way so that the base of the stack is at a high memory address with stack elements in successively lower addresses.

We shall see in later chapters how stacks can be extremely useful to the programmer. They are so important that many processors (including the one built into the Dragon) provide special instructions to add

information to and remove information from the stack. These instructions are:

PUSH	This instruction copies one or more registers onto the stack and moves the stack pointer 'up' by the number of register bytes copied.
PULL (or POP)	This instruction copies one or more items from the stack into registers and moves the stack pointer 'down' by the number of bytes copied.

The provision of instructions like these is one of the features of the Dragon which makes it such a powerful computer.

1.3 THE ORGANISATION OF THE DRAGON

We now move on from generalities and general principles of computer organisation to details of the organisation of the Dragon itself. Every microcomputer is inherently complex and the Dragon's hardware is made up of about 20 microchips and their interconnections plus a power supply, peripheral device connectors, etc. The usual way of describing system hardware is by means of a block diagram showing the various hardware components and their interconnections. Figure 1.2 is such a block diagram of the Dragon's hardware organisation.

As we suggested above, the hardware on a microcomputer system can be considered as being composed of three interacting subsystems. These are:

- (1) The processor
- (2) The memory
- (3) The input/output system

The processor built into the Dragon is a single microchip which is designated the M6809E or, simply, the M6809. This is an advanced 8-bit processor which means that its data highways are 8-bits wide but it also makes provision for operations on 16-bit data elements. We shall not discuss any details of this system here as both Chapter 2 and Chapter 3 are devoted to the architecture of the M6809 processor.

There is no explicit clock component shown in the block diagram although we explained in the previous section that the clock was an inherent part of every computer system. In fact, the box labelled 'Synchronous address multiplexor' is a multi-function chip which includes a clock and which acts as the interface between the processor and the random-access memory.

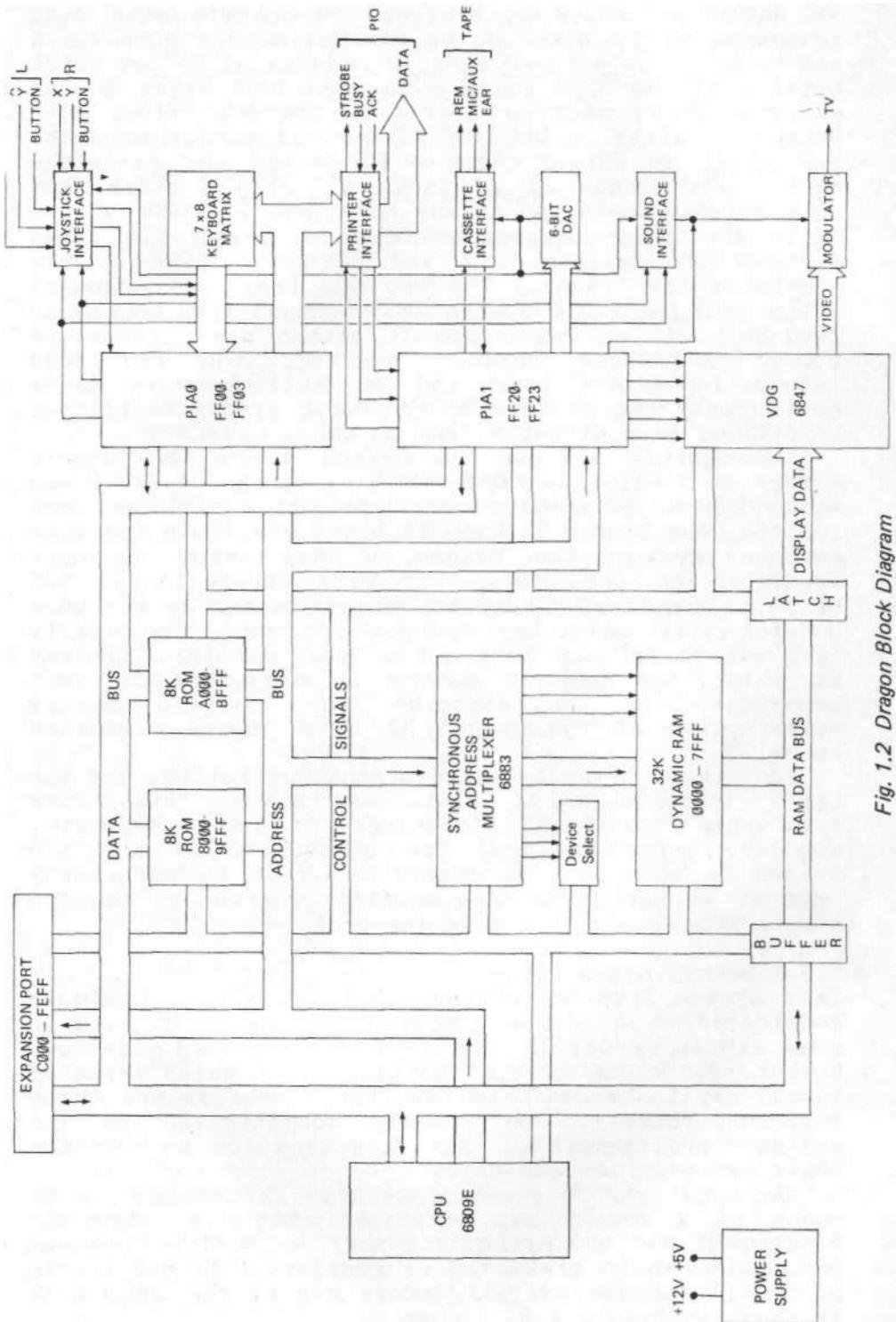


Fig. 1.2 Dragon Block Diagram

The M6809 processor is designed to operate with data addresses of 16 bits so the maximum memory size which can be built into the Dragon is made up of 2^{16} or 65536 bytes. The term 1K is used to mean 1024 bytes so the maximum memory size on the Dragon is 64K bytes. The Dragon 32 actually has 48K of inbuilt memory with the capability to expand this to 64K using the cartridge slot. The Dragon 64 has 80K of in-built memory but only 64K may be in use at any one time.

In the block diagram of the hardware, the units marked '32K Dynamic RAM' and '8K ROM' make up the memory of the Dragon. The two ROM (read-only memory) units hold the BASIC system and, because this memory is read-only, it is impossible to change any information stored in these units. However, you can read information stored there and we shall describe later how to make use of some of the BASIC system facilities by calling them directly from assembly code.

The dynamic RAM on the Dragon 32 is the user's memory area which is used for the storage of BASIC and machine code programs, user data, etc. As the name implies, the Dragon 32 has 32K bytes available for this purpose whereas the Dragon 64 has twice as much available to the user. For many applications, 32K bytes is a perfectly adequate amount of memory but when complex disk operating systems are used, you really need 64K to get the most out of your machine. The way in which the use of memory is organised is very important and we describe the logical memory organisation of the Dragon 32 in a separate section below.

The Dragon's input/output system controllers are the units labelled PIA0, PIA1, and VDG. These have associated peripheral interfaces to the keyboard, display, cassette, etc. The complexity of the I/O system is such that we cannot describe it adequately here so we have devoted a complete chapter to the I/O system (Chapter 8) later in the book.

1.3.1 Memory organisation

In a system like the Dragon, the memory is not simply considered as a single homogenous chunk to be used in some arbitrary way by the user or the BASIC system. Rather, decisions have to be made about which areas of memory are to be dedicated to which function and these decisions have to be clearly communicated to the system's programmers so that they know how to organise their own programs and data.

The usual way to communicate this information is by means of a memory map which is simply a schematic diagram of how the system's memory is used. Like any map, this can be presented at greater or lesser levels of detail and the overall memory map of the Dragon 32 is shown as Figure 1.3.

Address (Decimal)		Address (Hex)
65535	MPU VECTORS	FFFF
65522	UNUSED (RESERVED)	FFF2
65504	SAM CONTROL REGISTER BITS	FFE0
65472	UNUSED (RESERVED)	FFC0
65376	INPUT/OUTPUT DEVICES	FF60
65280		FF00
~	CARTRIDGE MEMORY	~
49152		C000
~	BASIC INTERPRETER	~
32768		8000
32767	STRING SPACE	7FFF
32566	STACK SPACE	7F36
~	PROGRAM AND VARIABLE STORAGE	~
13824		3600
12288	EXTRA PAGE 8	NORMAL
10752	GRAPHICS PAGE 7	BASIC
9216	SCREEN PAGE 6	PROGRAM/
7680	MEMORY PAGE 5	VARIABLES
6144	NORMAL PAGE 4	EXTRA
4608	GRAPHICS PAGE 3	BASIC
3072	SCREEN PAGE 2	PROGRAM/
~	MEMORY PAGE 1	VARIABLES
1536		0600
1024	NORMAL TEXT SCREEN	0400
768	EXTENDED PAGE – LINE INPUT BUFFER, ETC	0300
512	EXTENDED PAGE – CASSETTE BUFFER, ETC	0200
256	EXTENDED PAGE – SYSTEM VECTORS, ETC	0100
0	DIRECT PAGE – SYSTEM VARIABLES, ETC	0000

The 64K bytes of memory which is potentially available on the Dragon 32 can be looked at as being partitioned into eight distinct areas.

- (1) System variables
This is the area of 1K bytes in RAM from address 0000 to address 3FF. It holds various data values and I/O buffers used by the BASIC system. As these are in RAM, you may modify variables in this area but this must be done with care as incautious modification can cause the BASIC system to fail and require that the machine be reset.
- (2) Text screen
This is the 512 byte area from address 400 to address 5FF whose contents are reflected on the user's display when graphics are not being used. The use of this area is described in Chapter 7.
- (3) Graphics screens
The area of memory from address 600 to address 3600 is used by the BASIC graphics system to implement its graphics commands. Again, we describe the use of this area in Chapter 7. If graphics are not used or, if only limited graphics are used, all or part of this area may be used for the storage of the user's BASIC program and its variables.
- (4) Program and variable store
The area of memory from address 3600 to address 7F36 is used for the storage of the user's BASIC program and its variables.
- (5) BASIC string store
When character strings are used in a BASIC program, the string characters are held in a separate storage area. This area extends from address 7F36 to the top address in the dynamic RAM, 7FFF.
- (6) The BASIC interpreter
The 16K of memory required by the BASIC interpreter is provided as ROM on the Dragon 32 and is addressed from 8000 to BFFF.
- (7) Cartridge memory
Memory addresses from C000 to FEFF are allocated to the cartridge slot on the Dragon 32. When you plug in a cartridge, this contains its own read-only memory and this is addressed via these addresses.

(8) Input/output area

The Dragon's I/O system is a 'memory-mapped' system where reference to specific memory locations cause I/O operations to take place. Therefore, it is necessary to dedicate some memory locations to input/output and, in the Dragon 32, this I/O area is a 256 byte area at the very top of memory from address FF00 to address FFFF. Broadly, this area is partitioned into three separate parts. Addresses FF00 to FF5F are reserved for the use of peripheral controllers, addresses FFC0 to FFDF are used to control the synchronous address multiplexor and addresses FFF2 to FFFF are reserved for interrupt vectors. The other addresses in I/O area are unused and reserved for future system expansion. More details of the function of these I/O-dedicated addresses are provided in Chapter 8.

Chapter 2

The architecture of the M6809

The microprocessor used in the Dragon has been given the code number M6809 by its designers at Motorola Semiconductors. The M6809 processor developed from an earlier Motorola microprocessor, the M6800, and it shares some of the features of this earlier system. In fact, one of the design criteria for the M6809 was that it should be possible to run programs written for the M6800 on the M6809 processor.

The M6809 is called an 8-bit processor, indicating that its data highways are 8 bits wide. This means that a simultaneous transfer of 8 bits of information can be made from the processor to and from memory and peripheral controllers. However, the M6809 also includes a number of instructions which operate on 16 bits rather than 8 bits of data and this considerably increases the power of the processor.

Such 16-bit instructions provide extra power because 8-bit data manipulation is inadequate in many cases. For example, consider integer arithmetic. If only 8-bit representation is allowed this limits the range of integers to 0-255. This is clearly unacceptable in most cases so 16-bit arithmetic operations have to be simulated on an 8-bit machine by using combinations of 8-bit instructions. Naturally, this slows down the execution of programs.

The provision of many 16-bit operations of the M6809 means that programs can be written using fewer instructions. Therefore, these programs execute more quickly. Because of these extra instructions and because of the variety of ways in which memory can be accessed, the M6809 is sometimes called a second-generation microprocessor or, more extravagantly, the 'programmer's dream machine'.

In this chapter and in the following chapter we describe those aspects of the M6809 machine architecture which are of importance to the programmer who wishes to write machine language programs for his Dragon. This chapter covers the register organisation of the M6809, the multitude of ways in which machine memory may be addressed (addressing modes), and introduces some of the machine instructions available to the M6809 programmer. A description of all the M6809 machine instructions is provided in Chapter 3.

2.1 THE M6809 REGISTER SET

In the previous chapter we introduced the idea of a register as a fast storage element built into the processor. The M6809 has nine such registers, all of which may be considered as special-purpose registers rather than general-purpose registers. Figure 2.1 is the so-called 'programming model' of the M6809. It shows, diagrammatically, the M6809's registers and their comparative sizes.

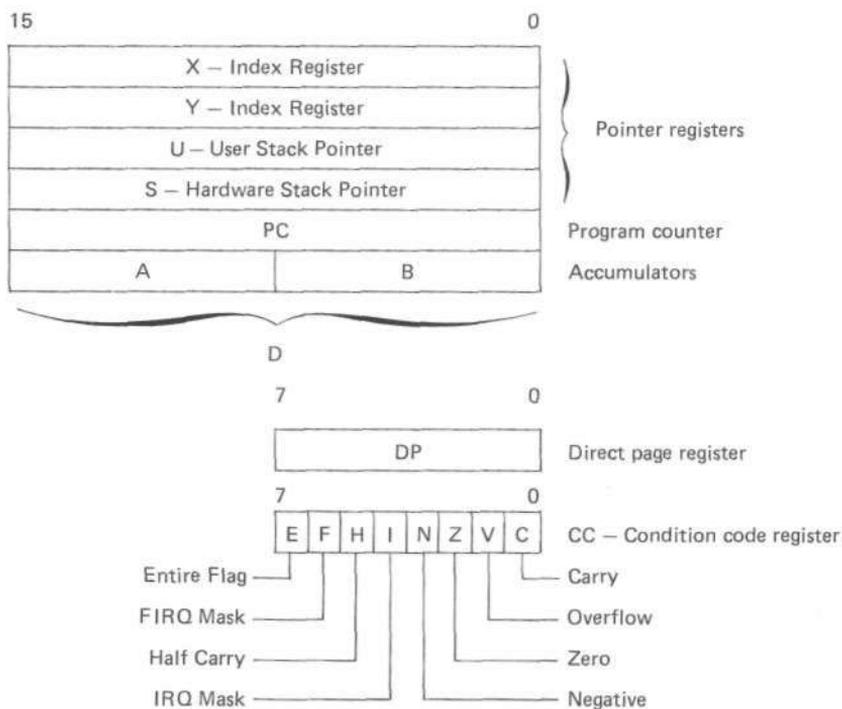


Fig. 2.1 The programming model of the M6809

The names of the M6809 registers, their width in bits, and a very brief description of their functions are listed below:

- (1) A register (8 bits) - accumulator register
- (2) B register (8 bits) - accumulator register
- (3) X register (16 bits) - index register
- (4) Y register (16 bits) - index register
- (5) U register (16 bits) - stack pointer register

- (6) S register (16 bits) - stack pointer register
- (7) DP register (8 bits) - direct page register
- (8) PC register (16 bits) - program counter register
- (9) CC register (8 bits) - condition code register

The bits in an M6809 register are numbered from right to left starting at 0. This means that bit 0 is the rightmost bit and, for 16-bit registers, bit 15 is the leftmost bit. Different machines have different conventions in this respect. Some processors number bits from left to right others, like the M6809, from right to left.

2.1.1 The A and B registers

The A and B registers are accumulator registers which are used to hold the operands and results of arithmetic operations. There are a variety of machine instructions which make use of these registers and examples of these are given below.

The instruction examples in this chapter are set out according to the following general format:

```
<machine code> (mnemonic) <operand> (comment)
```

We use diamond brackets <> to mean 'an instance of' so (mnemonic) means any instruction mnemonic may replace the character string (mnemonic). We also use the notation MEM((address)) when referring to particular addresses in memory so MEM(A0E4) means the memory location whose address, in hexadecimal, is A0E4 and MEM(FRED) means the memory location whose symbolic address is FRED. All memory addresses are given in hexadecimal or are symbolic addresses unless explicitly stated otherwise.

The machine code, in hexadecimal, is provided for each instruction example in this chapter. This is the actual code loaded into the M6809 memory whereas the instruction mnemonic and operand is a form of the instruction which is more understandable to the programmer. Most examples also have a brief descriptive comment explaining the function of that instruction.

Examples of instructions which use the A and B registers are:

```
860A    LDA #10      ; A = 10
1E89    EXG A,B     ; Tmp = A: A = B: B = Tmp
F7F1C5  STB $F1C5   ; MEM(F1C5) = B
```

```

5F      CLRB      ; B = 0
8B02    ADDA #2   ; A ← A + 2
F0F1C5  SUBB $F1C5 ; B = B - MEM(F1C5)

```

The A and B registers are both 8-bit registers which means that only a limited range of values, from 0 to 255, may be stored in them. For many arithmetic operations we need to operate on larger or smaller values than can be represented in 8 bits so the designers of the M6809 have provided instructions which allow the register pair A:B to be considered as a single register. When the registers are catenated in this way, they are called the D register.

Effectively, the A register makes up the leftmost 8 bits of the D register (bits 8-15). This is sometimes called the hi-byte. The B register forms the rightmost 8 bits of D (bits 0-7). This is called the lo-byte.

Many of the machine instructions which operate on the A and B registers have counterparts which operate on the D register. However, rather than 8-bit operations which take place automatically when A and B are used, the use of the D register or, indeed, any 16-bit register automatically results in 16-bit operations taking place. The address in the instruction refers to the leftmost (most significant) byte when 16-bit operations are specified. For example:

```

CC1000  LDD #4096 ; D = 4096
F31E62  ADDD $1E62 ; D = D + MEM(1E62)
FD0056  STD $56   ; MEM(56) = D
*                               MEM(56) = hi-byte of D
*                               MEM(57) = lo-byte of D

```

We shall look at more instructions which operate on the A, B, and D registers when we describe the M6809 instruction set in detail in Chapter 3.

2.1.2 The X and Y index registers

The X and Y registers in the M6809 may be used as general-purpose registers to store data but, more commonly, they act as special-purpose registers called index registers.

The information which is normally held in an index register is the address in memory of some other data item which may represent anything at all, even another memory address. The M6809 has several ways of accessing memory which makes use of these index registers to determine the address in memory which is being used.

Index registers are a particularly efficient way of

determining data addresses when data items stored in consecutive locations are to be accessed and processed in turn. The X and Y registers in the M6809 each have an associated auto-increment/decrement facility which means that a memory location can be accessed and, without additional instructions, the index registers can be updated to refer to the next item to be processed.

This means that the most important use of the X and Y index registers is for array processing. The index register is set up to refer to the first item of the array and the auto increment/decrement facility used to select succeeding items in turn.

The index registers may also be used as stack pointer registers if the user needs more than two stacks. The U and S registers are provided as stack pointer registers but the auto increment/decrement facilities of the X and Y registers means that they can also function efficiently in this role.

Examples of instructions which use these index registers are:

```
A684      LDA ,X      ; A = MEM(X)
A680      LDA ,X+     ; A = MEM(X): X = X + 1
A682      LDA ,-X     ; X = X - 1: A = MEM(X)
ECA012C   LDD 300,Y  ; D = MEM(300 + Y)
E7A6      STB A,Y    ; MEM(A + Y) = B
```

There are a number of other variants of index addressing available on the M6809. These will be discussed later in section 2.2.6.

2.1.3 The U and S stack pointer registers

The U and S registers are 16-bit registers which may act as index registers in exactly the same way as the X and Y registers described above. However, in many applications, these registers are best used as special-purpose stack pointer registers. Push and pull instructions are available to the programmer which assume that these registers are being used for this purpose.

In practical use, the S register is almost always used as a stack pointer register referring to the so-called S-stack or hardware stack. The hardware stack is used when calling subroutines and when swapping control from program to program. The state of the program which is interrupted is saved on this stack and restored when that program is restarted. This use of the hardware stack is described later in the book when interrupt-driven programming is described.

The U register may be used as a stack pointer to the so-called U-stack or user stack. However, the programmer may not need this facility in which case the U register may be used as an index register in exactly the same way as the X and Y registers.

The M6809 stack convention is that stacks grow downwards in memory. That is, when an element is pushed onto the stack, the stack pointer is decremented before the push operation so that that element has a lower memory address than the previous top stack element. The stack pointer registers S and U always point at the top byte on the stack. In this respect, the M6809 is different from some other stack-based systems where the stack pointer refers to the next available location on the stack.

Some examples of how the U and S registers may be used are:

```

3602  PSHU A           ; U = U - 1: MEM(U) = A

3436  PSHS A,B,X,Y    ; S=S-1: MEM(S)=Y: S = S-2
*      MEM(S)=X: S=S-2: MEM(S)=B
*      S=S-1: MEM(S)=A

3536  PULS A,B,X,Y    ; A=MEM(S) : S=S+1 : B=MEM(S)
*      S=S+2: X=MEM(S): S=S+2
*      Y=MEM(S): S=S+1

3704  PULU B           ; B=MEM(U): U=U+1

```

The push and pull instructions for stack manipulation are described in more detail in Chapter 3.

2.1.4 The DP register

The M6809's DP (Direct Page) register is an 8-bit register which always contains the address of the start of a 256 byte chunk (page) of memory. This register is used exclusively in the so-called direct addressing mode. In this mode, the contents of the register are added to an 8-bit value specified by the user as part of the machine instruction to form the effective memory address. For example:

```

96E9  LDA $E9        ; A = MEM(DP + E9)

D710  STB $10        ; MEM(DP+10) = B

```

2.1.5 The PC register

The PC register is the M6809's program counter. It a 16-bit register which always contains the address in memory of the next machine instruction to be executed by the M6809.

2.1.6 The CC register

The CC register is an 8-bit condition code register where individual bits mark the occurrence of particular conditions. The bits in the register have the following functions:

- Bit 0 carry bit, set in arithmetic operations
- Bit 1 two's complement overflow bit
- Bit 2 zero bit, set when result of an operation or data transfer is zero
- Bit 3 negative flag, set when result of an operation or data transfer is less than zero
- Bit 4 normal interrupt mask, used by M6809 interrupts
- Bit 5 half-carry flag, used to indicate a carry from bit 3 to 4
- Bit 6 fast interrupt mask, used by M6809 interrupts
- Bit 7 entire state saved flag, used by M6809 interrupts

The above descriptions of the flags in the CC register are very sketchy indeed but it is not appropriate to go into more detail here of what each flag means. Rather, we describe the role of individual condition code flags along with those machine instructions which set and test these flags.

2.2 ADDRESSING MODES ON THE M6809

One of the features of the M6809 architecture which distinguishes that microprocessor from earlier microprocessors is the variety of ways in which the address of a data item may be computed. In all, there are 19 distinct ways of representing an address (addressing modes) and the flexibility and power of these modes means that some applications may be coded very efficiently indeed on the M6809.

The use of the various addressing modes is illustrated in Chapters 4 and 5. In this section we confine ourselves to a description of those modes and present examples of instructions which use these various modes.

Before going on to look at addressing modes in detail, however, we must look at the structure of a machine instruction and examine how operand addresses are represented within instructions. Instructions in the M6809 may be 1, 2, 3, 4, or 5 bytes long depending on the particular instruction and on the addressing mode which is being used. Each instruction has two fields:

- (1) The op-code (1 or 2 bytes)
- (2) The operand address specifier (0, 1, 2 or 3 bytes)

Notice that, in some cases, the operand address specifier may be empty, that is, it doesn't explicitly exist. For example, the instruction CLRA clears the A register - the inherent operand address in this case is the A register and may never be anything else.

Most instructions, however, do have an address field which has the following general structure:

- (1) Postbyte (0 or 1 byte)
- (2) Value field (0, 1 or 2 bytes)

The address field, called the 'postbyte', is not needed by all the M6809 addressing modes and it will be described along with those addressing modes which make use of it. Simpler addressing modes only need the 'value' field to construct an operand address and some modes only require the postbyte field.

2.2.1 Immediate addressing

The simplest addressing mode on the M6809 is the immediate addressing mode where the instruction operand is a constant whose value is 'built in' to the machine instruction. When programming, immediate addressing is specified by preceding the constant to be included in the instruction with the symbol #. Some examples of immediate addressing are:

```
C680      LDB #128      ; B = 128 (decimal)
CC0400    LDD #1024    ; D = 1024 (decimal)
108EFF00  LDY #$FF00   ; Y = FF00 (hex)
```

Notice that a hexadecimal value is specified by preceding the immediate value with a \$ sign. The # symbol must also be included to specify immediate addressing as a \$ on its own has a completely different meaning.

Although the instruction operand in immediate addressing mode must be an absolute hexadecimal constant, this can be generated by the assembler. Most assemblers allow the association of symbolic names with constants and also allow symbolic labels representing addresses. These may be used as immediate operands.

2.2.2 Extended addressing

In the extended addressing mode, the contents of the 2 bytes following the instruction op-code are taken as the absolute address in memory of the instruction operand. Extended addressing is specified by preceding a numeric address (usually in hex) with the symbol \$ or, alternatively, by writing the symbolic address of the operand being accessed.

A symbolic address is simply a name given to a

particular address location. This idea was introduced in section 1.1.2 and it is by far the most convenient way to refer to actual addresses in the Dragon's memory. When a symbolic address is encountered in an instruction, the assembler replaces it with its actual numeric memory address. The assembler also handles the conversion of mnemonics to machine code, the conversion of decimal and hexadecimal numbers to binary, etc.

Examples of the M6809 extended addressing code are given below along with their corresponding machine code representations. Assume that the symbolic names CHAR1 and PNTR have addresses A000 and A008 respectively.

```
B7A000  STA CHAR1      ; MEM(CHAR1) = A
BEA008  LDX PNTR      ; X = MEM(PNTR)
BB03A2  ADDA $03A2    ; A = A + MEM(03A2)
```

2.2.3 Direct addressing

Recall from our description of the M6809 registers that the processor has an 8-bit register called the Direct Page or DP register which always contains the address of the start of a 256 byte chunk (page) of memory. This register is used in the direct addressing mode.

In this mode, the address of an operand is computed by taking the value contained in the instruction itself (00-FF) and using this as the lo-byte of the operand address. The hi-byte is taken as the value of the DP register. Direct addressing is used whenever the address lies in the range 00 to FF since the DP register normally contains 00. Direct addressing can be forced by preceding the address with a '<' symbol in which case it is essential that the DP register is set up with the address of the starting byte of the memory 'page' being accessed.

Registers are normally assigned values using load instructions but there is no load instruction which assigns a value to the DP register. Rather, some other 8-bit register must be assigned a value and its contents then to the DP register using a TFR instruction. For example:

```
8610  LDA #$10      ; A = 10 (hex)
1F8B  TFR A,DP     ; DP = A
```

Examples of the use of direct addressing are:

```
DD20  STD $20      ; MEM(1020) = D
9000  SUBA $00     ; A = A - MEM(1000)
```

The use of direct addressing means that instructions

are short (mostly 2 bytes) and this means that programs are efficient in both execution speed and in the storage required for the program. There are also advantages in using this mode of addressing when implementing programming languages like Pascal where global variables may be stored in a page by themselves and accessed via the DP register.

2.2.4 Register addressing

Register addressing is an addressing mode where the instruction operands are always in registers with a postbyte used to identify the registers involved. There are only two instructions which make use of this addressing mode. These are the transfer register instruction (TFR) and the exchange register instruction (EXG). The address field is simply a postbyte which is split into two parts. Bits 0-3 of the postbyte identify the destination register and bits 4-7 identify the source register. The identification value, in hexadecimal, for each register is:

0	D register	5	PC register
1	X register	8	A register
2	Y register	9	B register
3	U register	A	CC register
4	S register	B	DP register

Using the TFR and EXG instructions, it is only possible to transfer and exchange registers of like size (8 or 16 bits). Examples of instructions using the register addressing mode are:

```
1F12    TFR X,Y
```

```
1E89    EXG A,B    ; Tmp = B: B = A: A = Tmp
*                               where Tmp is some temporary register
*                               hidden from the M6809 user
```

2.2.5 Indirect addressing

Some kinds of programming are made easier if you can refer indirectly to information which you want to manipulate. That is, you don't include the address of the instruction operands in the instruction but the address reference in the instruction is to a location which holds the actual operand address.

Normally, the address part of a machine instruction directly refers to its operand. For example:

```
LDD MAXVAL
```

loads the data stored at symbolic address MAXVAL into register D. With indirect addressing, however, the address part of the machine instruction holds the address of the address of the instruction operand.

Address computation is therefore a two-stage process. First, compute the address as specified in the machine instruction. Secondly, use this to locate the operand address then use this address to fetch the operand itself.

This is illustrated in Figure 2.2.

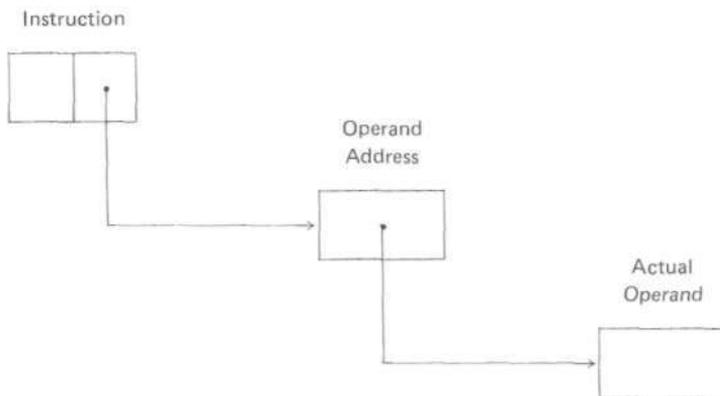


Fig. 2.2 Indirect addressing

It is important to remember that the use of indirect addressing means that the two-stage process described above always takes place. The effect of an instruction using indirect addressing is exactly the same as the same instruction using direct addressing inasmuch as the operand value, not its address, is manipulated by that instruction.

Indirect addressing can be used with a number of the M6809 addressing modes but, of the modes which we have described so far, it is only possible with extended addressing. In this case, and in all other cases where indirect addressing is allowed, indirect addressing is specified by surrounding the address part of the instruction with square brackets. For example, say a value 00E4 is stored at address 32F0. Furthermore, assume the symbolic address MAXADD has a value of 10A4 and is set up to refer to the value 00E4. The instruction

```
CC9F10A4  LDD [MAXADD]
```

specifies that the value in MAXADD is actually the address of the value to be loaded into the D register. Therefore, the effect of LDD [MAXADD] would be to copy 00E4 into register D. The actual address reference in the instruction is to address 10A4 which holds the value 32F0 - the location where 00E4 is stored.

This has illustrated how indirect addressing is used in conjunction with the extended addressing mode but it

may also be used with indexed addressing which is described below. In indexed addressing, where a postbyte is an inherent part of the address, bit 4 of the postbyte is used to indicate whether the address reference is direct or indirect. If bit 4 is set, the address is taken as in indirect reference to the instruction operand.

2.2.6 Indexed addressing

We have already described how some of the registers in the M6809 may be used as index registers where the address is computed by adding or subtracting some value from the value in the index register. There are a variety of different types of indexed addressing available to the M6809 programmer and these are all described in this section.

The format of an address in an instruction using indexed addressing is:

- (1) Postbyte (1 byte)
- (2) Offset (0, 1 or 2 bytes)

The postbyte is set up to indicate which register is the index register, whether that register is to be automatically incremented or decremented and to specify the form of the offset to be added to the value in the index register.

The forms of indexed addressing which we shall describe here are:

- (1) Zero offset indexed addressing
- (2) Constant offset indexed addressing
- (3) Accumulator offset indexed addressing
- (4) Auto increment/decrement indexed addressing

Before describing these addressing modes in detail, however, let us look at the structure of the postbyte which determines the actual addressing mode used and, in some cases, holds the offset which modifies the index register value.

Bit 7 (the leftmost bit) of the postbyte specifies whether an offset is stored as part of the postbyte. If this bit is unset, bits 0-4 are taken as a 5-bit signed offset in two's complement form. This means that values between -16 and 15 may be held as part of the postbyte and automatically added to the index register.

If bit 7 of the postbyte is set, this means that a 5-bit offset is not part of the postbyte and that bits 0-4 have a completely different meaning. In this case,

bits 0-3 are used to specify which type of indexed addressing is to be used and bit 4 is used to select direct or indirect indexed addressing. The correspondance between addressing modes and associated values of bits 0-3 is set out in the table below.

Bit 4 is the indirect select bit. If it is unset, this indicates that the computed address is the address of the instruction operand. If it is set, this means that the computed address is to be taken as the address of the address of the instruction operand.

In all types of indexed addressing, bits 5 and 6 of the postbyte are used to specify which index register is being used. Each value of this bit pair specifies a different index register as follows:

X	Bit 6 = 0 , Bit 5 = 0
Y	Bit 6 = 0 , Bit 5 = 1
U	Bit 6 = 1 , Bit 5 = 0
S	Bit 6 = 1 , Bit 5 = 1

When bit 7 is 1, bits 0-3 of the postbyte select the addressing mode to be used. The values of these bits (in hexadecimal) and their corresponding addressing modes are shown in the table below:

0	Auto increment (+1)	The index register is incremented by 1 after the address is computed.
1	Auto increment (+2)	As above, increment is 2.
2	Auto decrement (-1)	The index register is decremented by 1 before the address is computed.
3	Auto decrement (-2)	As above, decrement is 2.
4	Zero offset	The address in the index register is the operand address.
5	ACCB offset	The address is computed by adding the contents of register B to the index register contents.
6	ACCA offset	As above, but the contents of register A are added to the index register.
7	Not used	

8	8-bit signed offset	The value of the byte following the postbyte is added to the index register to compute the address.
9	16-bit signed offset	As above, but the following 2 bytes are added to the index register.
A	Not used	
B	ACCD offset	The value of accumulator D (A:B) is added to the index register.
C	PC relative,	The PC acts as an index register, with the address computed by adding an 8-bit offset to its value.
D	PC relative,	As above with 16-bit offset.
E	Not used	
F	Extended indirect	The following 2 bytes are the address of the address of the instruction operand.

We have already covered extended indirect addressing and addressing using the program counter PC will be discussed in section 2.2.7. Now let us look in more detail at the possible indexed addressing modes.

Auto increment/decrement indexed addressing

This addressing mode allows 1 or 2 to be automatically added or subtracted from the index register value. No additional add or subtract instruction is necessary to accomplish this. When using auto increment addressing, the value is added to the index register after the effective address has been computed. In auto decrement mode, the value is subtracted from the index register and the effective address then computed.

Examples of instructions using this addressing mode are:

```
A7C0   STA  ,U+      ; MEM(U) = A: U = U + 1
```

```
ECA1   LDD  ,Y++     ; D=MEM(Y): Y = Y + 2
```

```
AB82    ADDA ,X      ; X = X - 1 : A = A + MEM(X)
```

```
A3E3    SUBD ,--S   ; S = S - 2 : D = D - MEM(S)
```

Auto increment/decrement indexed addressing is particularly efficient when a number of data elements have to be processed in sequence. The index register is set up to point at the beginning or the end of the sequence in memory and, after each element is fetched, the register is incremented or decremented so that it points at the next element in the sequence.

Zero offset indexed addressing

Using this addressing mode, the value in the index register is taken to be the address of the instruction operand. Nothing is added to or subtracted from it. For example:

```
A684    LDA ,X      ; A=MEM(X)
EDF4    STD [,S]    ; MEM(MEM(D)) = D
*                               Note [] meaning indirect addressing
```

Constant offset indexed addressing

In this case, a positive or negative constant is added to the value in the specified index register to compute the address of the instruction operand. The range of possible offsets is from -32768 to 32767 (decimal) and the assembler works out whether the offset is to be stored as part of the postbyte (-16->15), as an 8-bit quantity (-128->127) or as a 16-bit quantity (-32768->32767). If the offset is not stored in the postbyte, it immediately follows the instruction postbyte in memory.

Although a constant value is added to the index register value to compute the operand address, this modified value is not stored in the index register. The addition is purely temporary and the index register value is not changed by the use of constant offset addressing. Examples of instructions using this addressing mode are:

```
EC7A    LDD -6,S    ; D = MEM(S-6).
```

```
*                               Note offset stored in postbyte
*                               in two's complement form
```

```
AB8816   ADDA 22,X   ; A = A + MEM(X + 22)
```

```
*                               Offset stored as a 1 byte value
```

```
AB89012C  ADDA 300,X ; A = A + MEM(X + 300)
```

```
*                               Offset stored as a 2 byte value
```

Accumulator offset indexed addressing

This addressing mode is similar to constant offset indexed addressing but, rather than the offset being a

constant, the value of an accumulator register is added to the index register to compute the address. The advantage of this is that the offset can be calculated and loaded into the accumulator just before it is required. The programmer need not know the offset in advance as in constant offset indexed addressing.

Examples of this addressing mode are:

```
E7A6   STB A,Y   MEM(A + Y) = B
```

```
ECB8   LDD D,X   D = MEM(D + X)
```

2.2.7 Relative addressing

Another mode of address computation in the M6809 is relative addressing where the address of an operand or of another instruction is computed by adding an offset to the program counter register. This offset may be a positive or negative, 8-bit or 16-bit value. We shall look first at how instruction operands are accessed using this addressing mode and then at the relative addressing of instructions themselves.

Relative addressing of instruction operands makes use of the postbyte in the same way as does indexed addressing. If bits 0-3 of the postbyte are C or D while bit 7 is set this specifies that the addressing is PC relative. For example:

```
AE8C08   LDX 8,PCR   ; X = MEM(PCR + 8)
```

```
DD8D0400  STD 1024,PCR ; MEM(PCR + 1024) - D
```

A very important advantage of using PC relative addressing is that it simplifies the writing of position independent code. Position independent code is code which works in exactly the same way irrespective of where that code is placed in memory. Such code must make extensive use of relative and indexed addressing because extended addressing means that the instruction operands must always be at the address 'built in' to the code.

With position dependent code, you must always load the program into exactly the same memory locations as were used previously. This is not necessarily convenient or even possible so it is good programming practice to write all programs in a position-independent way.

Relative addressing of the instructions in a program is accomplished by means of so-called 'branch instructions'. The effect of these branch instructions is to modify the program counter register. Thus the next instruction executed is not necessarily the instruction following the branch instruction but some other instruction whose address is computed by adding the specified offset to the value of PC. The relative

addressing of instructions is different from the relative addressing of operands inasmuch as the value stored in PC is modified whereas in operand addressing the value of PC is used but is unchanged by the address computation.

The computation of relative instruction offsets is a tedious and error-prone task. Usually, it is left to the assembler to work out the appropriate value to be added to PC. You may mark instructions with a name (a label) and use this name as part of the branch instruction. The assembler knows the number of bytes occupied by each instruction so it can work out the appropriate offset to allow a transfer of control to the labelled instruction.

This can be illustrated by a short assembly code sequence which is equivalent to the following BASIC statement:

```
IF VL > MAX THEN MAX = VL
```

Assume that VL and MAX are 16-bit quantities held at addresses A000 and A002 respectively. The assembly code equivalent to the above BASIC conditional is:

```
FCA000    LDD VL      ; D = MEM(VL)
10B3A002  CMPD MAX    ; Compare D with MEM(MAX)
2F03      BLE NEXT   ; If VL<=MAX goto NEXT
FDA002    STD MAX    ; MEM(MAX) = D
NEXT      ....
```

The branch instruction in the above sequence, BLE, modifies the value of PC if and only if VL is less than or equal to MAX. Notice that the value in the PC modification field is 3, the number of bytes in the STD instruction. It is not the number of bytes in the BLE instruction plus the number of bytes in the STD instruction. The reason for this is the PC always points to the next instruction in the instruction sequence rather than the instruction which is being executed.

There are many branch instructions available to the M6809 programmer. They are discussed in detail in section 3.5 of the following chapter.

2.3 MEMORY-MAPPED INPUT/OUTPUT

We have seen, in Chapter 1, that a computer organisation includes a number of units which are set up as peripheral control devices to allow information to be transferred to and from the processor and memory units. Obviously, the processor must have access to these controllers in order to initiate data transfers to and from the outside world. In this section we describe, in very general terms how this is done.

However, as it is such an important topic we devote a complete chapter to details of input and output later in the book.

Recall, from Figure 1.2, that the M6809 processor, memory and peripheral controllers all have access to a common data highway or bus. On M6809-based systems such as the Dragon, this bus is 24 bits wide. This means 24 bits of information can be simultaneously transferred from device to device. Of these 24 bits, 16 bits are reserved for the data address and 8 bits are used to transfer the data itself.

In the same way as all memory locations have a unique address, so too must input/output (I/O) devices connected to this shared bus. On some systems, the bus has an extra line indicating that the address on the bus is a peripheral rather than a memory address but this is not the case on M6809 systems. Rather, the addresses of I/O devices have exactly the same form as memory addresses with specific addresses reserved for these I/O devices. These memory addresses may not be used for straightforward data storage as they are allocated to particular I/O devices.

This is not a severe handicap as there are usually only a few I/O devices on any system. On the Dragon, there are 256 memory bytes reserved for use by the I/O system. These are at the top end of memory between FF00 and FFFF. If we access one of these addresses which is allocated to an I/O device, the effect of the access is to initiate a data transfer to or from that peripheral unit. The synchronous address multiplexor examines addresses on the bus and detects those which refer to I/O controllers. The data is then routed to these devices for input or output.

This type of I/O organisation where peripherals are associated with specific memory addresses is called, for obvious reasons, memory-mapped I/O. It is a conceptually elegant way of carrying out input and output as there is no need for specific instructions to initiate peripherals and all instructions which reference memory may be used to access the system's I/O devices. Full details of the Dragon's I/O system are provided in Chapter 8 and in the appendices.

Chapter 3

The M6809 instruction set

In Chapter 2 we described the general features of the M6809 architecture and introduced, without a great deal of explanation, some of its machine instructions. A thorough knowledge of the machine instruction set is vital for the machine code programmer so this chapter is completely dedicated to a description of the M6809 instruction set.

At this point, we must emphasise the distinction between machine instructions and assembly language mnemonics. Machine instructions are the actual binary op-codes executed by the processor as it runs a program. Assembly language instructions are the mnemonics and names used by the programmer to symbolise these machine instructions because it is much easier for us to think in symbols and names rather than numbers.

There is not necessarily a one-to-one correspondence between machine instructions and assembly language instructions. For example, on the M6809 there are over 1400 distinct machine instructions when we take into account all the different combinations of op-code and postbyte that are permitted. Fortunately, however, there are only 59 distinct instruction mnemonics which must be remembered by the assembly language programmer along with the register names and the symbolism associated with the different M6809 addressing modes. Combinations of these allow all possible machine instructions to be represented.

The reason for the enormous discrepancy between the numbers of assembly language and machine code instructions is that many assembly language instructions have variants for each of the machine registers and for each addressing mode allowed with that instruction. For example, the instruction specifying that a register is to be loaded with an immediate value has the form:

```
LD<register> <value>
```

This is all that need be remembered by the assembly language programmer. However, there are seven distinct machine language op-codes associated with this instruction, one for each register that may be directly

loaded. The assembly code mnemonics for these are LDA, LDB, LDD, LDX, LDY, LDU, and LDS. These have associated op-codes of 86, C6, CC, 8E, 108E, CE, and 10CE.

All of these load instruction mnemonics have a different op-code associated with each permitted addressing mode. For example, if immediate addressing is used with an LDA instruction the op-code is 86. If direct addressing is used, the op-code is 96, for indexed addressing the op-code is A6 and for extended addressing B6. Instructions which load the other registers also have distinct op-codes for each addressing mode so, in all, the LD instruction mnemonic has 28 distinct machine instructions which may be derived from it. If we consider postbytes to be part of the instruction, this gives many more machine language derivations from an assembly language load instruction.

It is practically impossible to program directly in machine language because of the enormous number of op-codes that must be remembered by the programmer. Normally, an assembler is used to carry out the tedious task of translating mnemonics to op-codes, working out relative offsets and constructing postbytes. At worst, if an assembler is not available, the programmer should write his program in assembly code as if an assembler is at hand and then translate manually to machine code. Attempting to program directly in machine code inevitably leads to frustration, boredom and many errors.

A complete table of assembly language mnemonics and their associated machine op-codes is provided in Appendix 1. It must be emphasised, however, that hand translation from assembly code to machine code is not recommended for anything apart from very short programs.

The instructions available to the M6809 programmer can be considered under seven distinct headings. These are:

- (1) Data movement instructions
Instructions which transfer information to and from registers and memory.
- (2) Arithmetic instructions
Instructions used to implement arithmetic operations such as add and subtract.
- (3) Logic instructions
Instructions used to execute logic operations such as or and shift.
- (4) Test instructions
Instructions which set flags in the condition code register depending on operand values.

- (5) Branch instructions
Instructions which affect the normal sequential flow of control in a program by modifying the value of PC.
- (6) Interrupt handling instructions
Instructions used to handle so-called interrupts which usually arise from peripheral devices in the system. Interrupts are described in Chapter 8.
- (7) Miscellaneous instructions
Any other instructions which don't fit under the above headings.

Many data movement, arithmetic, logic and test instructions have the effect of setting or unsetting particular bits in the condition code (CC) register. In particular, if the result of executing an instruction is zero, the zero (Z) flag in CC is always set. If the result is negative, the negative (N) flag in CC is always set.

Arithmetic, logic and test instructions may also change the value of the carry (C) flag, the half-carry (H) flag and the overflow (V) flag in the condition code register. Some of these are described later in this chapter under the appropriate headings. This description is not complete - full details of how instructions affect CC flags are provided in Appendix 1.

In the following description of the M6809 assembly code instructions, it is sometimes necessary to refer to particular CC flags. We use a dot notation, CC.<flag letter>, to make these references. Thus CC.N is the negative flag, CC.V is the overflow flag, etc. When we say a flag is set this means that its value is 1, when unset the flag value is zero.

In the remainder of this section and in subsequent chapters, we sometimes use BASIC statements to explain the meaning of assembly language instructions. We have done this informally until now but, from now on, we will use the following conventions.

- (1) Registers are indicated by BASIC variables with the same name as the register. Therefore, the names of the registers are A, B, D, X, Y, U, S, DP, CC, and PC.
- (2) The use of some other BASIC name refers to the location in memory which has that symbolic name. Therefore an assembly code instruction, LDD XVAL, might be commented with the BASIC statement, D = XVAL.

- (3) When an absolute address in memory is referenced, we consider memory as a one-dimensional array called MEM and use the absolute address as an array index. Therefore, MEM(A034) refers to the memory location whose address, in hexadecimal, is A034. We also use the same notation when referring to an indexed address. The register name plus or minus any offset is stated as an index into MEM. Thus, MEM(X + 10) means the memory location whose address is computed by adding 10 to the contents of register X. In all cases, constant values used as indices to MEM are hexadecimal constants.

Operations using a 16-bit register result in 2 bytes being loaded or stored from memory whereas 8-bit register operations result in a single byte being loaded or stored. We do not explicitly distinguish between 1 and 2 byte memory operations in the comments accompanying the assembly code examples.

The examples provided are intended to illustrate the assembly code instructions so no machine code equivalents are given in this chapter.

3.1 DATA MOVEMENT INSTRUCTIONS

The function of data movement instructions in the M6809 is to transfer information, without change, from register to register, from register to memory, and from memory to register. In all cases, apart from the EXG, register exchange instruction, and some instances of the LEA, load effective address instruction, the data movement is implemented as a copy operation. That is, immediately after the data movement instruction has been executed, the source operand and the destination operand as specified in the instruction have the same value. The value of the source operand is not destroyed by the execution of the instruction.

Data movement instructions have the following form:

```
<op-code mnemonic><register specifier> <parameter>
```

The instruction parameter may take different forms depending on the particular data movement instruction. These will be described along with the individual instructions.

There are a total of 7 types of data movement instructions:

- (1) Load instructions
Instructions which move data from memory to a register.

- (2) Store instructions
Instructions which move data from a register to memory.
- (3) Transfer instructions
Instructions which move data from one register to another.
- (4) Exchange instructions
Instructions which exchange the contents of one register with another.
- (5) Load effective address instructions
Instructions which compute an operand address and load it into an index register.
- (6) Push instructions
Instructions which push register values onto a stack.
- (7) Pull instructions
Instructions which pull values stored on a stack into registers.

3.1.1 Load instructions

Load instructions in the M6809 are used to load data values into a register from memory or as immediate operands from the instruction itself. The general form of these instructions is:

```
LD<register> <address or immediate operand>
```

Registers A, B, D, S, U, X, and Y may be used in load instructions. If the instruction specifies a 16-bit register (D, U, S, X, T), the effect of the load instruction is to move the addressed memory byte into the hi-byte of the register and to load the following memory byte (address + 1) into the lo-byte. That is, 2 memory bytes or a 16-bit immediate operand is moved into the register. If an 8-bit register is specified, the addressed byte or 8-bit immediate operand is moved into the register.

Four classes of addressing mode are allowed with load instructions. These are immediate addressing, direct addressing, indexed addressing and extended addressing. Depending on the addressing mode used and on the particular instruction op-code, load instructions are 2, 3, 4, or 5 bytes in length.

Some examples of load instructions, in assembly code, are:

```
LDA #10 ; A = 10
```

```
LDD MAXVAL    ; D = MAXVAL
LDS 10,X      ; S = MEM(X + 10)
LDB $50       ; B = MEM(DPR + 50)
```

3.1.2 Store instructions

Store instructions are the converse of load instructions. They are used to transfer information from the machine registers to memory. The general form of store instructions is:

```
ST<register> <address>
```

As with load instructions, the allowed registers are A, B, D, X, Y, U and S. The use of a 16-bit register name results in 2 bytes being moved from the register to memory, an 8-bit name results in a single byte being moved.

Allowed addressing modes are direct addressing, indexed addressing, and extended addressing. For obvious reasons, immediate addressing is not meaningful in store instructions.

Some assembly code examples of store instructions are:

```
STA I        ; MEM(I) = A
STX ,Y       ; MEM(Y) = X
STD $30      ; MEM(DP + 30) = D
```

Like load instructions, store instructions can have lengths between 2 and 5 bytes depending on the op-code and addressing mode used.

3.1.3 Transfer instructions

Transfer instructions move the contents of one register to another. Any registers may be specified as long as they are of like size, that is, both operands must be either 16-bit registers or 8-bit registers. The mnemonic for a transfer instruction is TFR and the only permitted addressing mode is register addressing. Transfer instructions are always 2 bytes in length.

Examples of transfer instructions are:

```
TFR A,DPR    ; DPR = A
TFR X,Y      ; Y = X
```

3.1.4 Exchange instructions

The exchange instruction, whose mnemonic is EXG, is similar to the transfer instruction described above. However, rather than the value of the source register

being copied to the destination register, the values of the source and destination register are swapped.

Again, register addressing is the only addressing mode which may be used with exchange instructions. For example:

```
EXG A,DPR    ; Temp = A: A = DPR: DPR = Temp
```

```
EXG S,U      ; Temp = U: U = S: S = Temp
```

3.1.5 Load effective address instructions

The purpose of the load effective address instructions is to set up one of the index registers (S, U, X, Y) to hold the absolute address of an operand in memory. Because address computations in the M6809 can be fairly complex, and hence time consuming, it is sometimes useful to carry out this computation once only and then use this computed value in subsequent instructions.

Load effective address instructions have the form:

```
LEA<index register> <address>
```

The specified address must be an indexed address. LEA instructions are either 2, 3, or 4 bytes long depending on the particular type of indexed addressing which is used. Examples of these instructions are:

```
LEAS 10,X    ; S = X + 10
```

```
LEAX D,X     ; X = D + X
```

It is clear from the BASIC representations of the instruction functions that, in many cases, the LEA operation involves an addition to an index register. This means that a subsidiary use of this operation is to allow addition and subtraction operations on the index registers without requiring that their contents be transferred to the accumulator register. For example:

```
LEAS 10,S    ; S = S + 10
```

```
LEAX -20,X   ; X = X - 20
```

The above operations can, of course, be accomplished using the accumulator registers:

```
TFR S,D     ; D = S
ADDD 10     ; D = D + 10
TFR D,S     ; S = D
```

However, the single LEA instruction executes more quickly and takes up fewer memory bytes than these instruction sequences.

3.1.6 Push instructions

The function of push instructions is to copy the contents of one or more registers onto a stack in memory whose top is addressed by the U or the S register. Push instructions have the form:

```
PSH<U or S> <register list>
```

The PSH can move the contents of up to 8 registers (CC, A, B, DPR, X, Y, S or U, PC) onto the memory stack.

Push instructions have a postbyte indicating which registers have actually to be pushed onto the stack. Individual registers are indicated by bits in the postbyte as follows:

Bit 0	CC
Bit 1	A
Bit 2	B
Bit 3	DPR
Bit 4	X
Bit 5	Y
Bit 6	S or U
Bit 7	PC

Push instructions are always 2 bytes in length. Some examples are:

```
PSHS A,B ; Push A and B onto the S-stack
```

```
* PSHU A,B,Y,X,PC,CC,DPR ; Push all registers apart
                           from U onto the user stack
```

The order in which the user specifies the registers in the push instruction is not necessarily the order in which they are pushed onto the stack. Registers are always pushed onto the stack in the following order:

```
PC, U/S, Y, X, DPR, B, A, CC
```

If all registers are pushed, CC is on top of the stack, A is the second top location, B is the third top location and so on. If only a subset of the registers are pushed onto the stack, the order above is maintained although, obviously, only the specified registers are actually stacked.

For example, after executing the instruction PSHU A,X,B, the top of the stack is a copy of register A, the second top is a copy of register B and the third top is a copy of register X although this was not the order specified in the instruction. In general, this automatic ordering of stacked registers saves the user having to care about stacking order. If, however, a particular stacking order is required this must be achieved by using separate push instructions for each register.

3.1.7 Pull instructions

Pull instructions are the converse of push instructions. They move information from stacks in memory to specified registers. The form of pull instructions is:

```
PUL<S or U> <register list>
```

Pull instructions, like push instructions, use a postbyte to specify which registers are to be pulled from the stack. Some examples of pull instructions, which are always 2 bytes long, are:

```

PULS A,B    ; Copy the top 2 locations of the
*           hardware(S) stack to A and B.
*           Adjust the stack pointer accordingly

PULU A,B,DPR,PC,X,Y,S,CC ; Copy values of all
*           registers from the
*           user stack

```

The order in which register values are pulled from the stack is again independent of the order in which they are specified in the instruction. Therefore, CC is the first register pulled, A the next register, B the third register and so on.

3.2 ARITHMETIC INSTRUCTIONS

The arithmetic instructions available on the M6809 operate on the accumulator registers and, in some cases, directly on memory locations. In all cases when an instruction operates on a register one of its operands is the value of that register and the result of the operation is placed in that register. Therefore, after an arithmetic operation on a register the previous contents of that register are destroyed.

There are twelve arithmetic operations available to the M6809 programmer which we shall consider in seven groups:

- (1) Add instructions
- (2) Subtract instructions
- (3) Clear instructions
- (4) The multiply instruction
- (5) Negate instructions
- (6) The sign extend instruction
- (7) The decimal adjust instruction

As a side effect of executing most of these arithmetic instructions, flags in the condition code register are set. Particular settings are described under the appropriate heading below.

3.2.1 Add instructions

There are four kinds of add instruction provided on the M6809. These have the forms:

ABX	$X = X + B$
ADC<A or B>	Add memory to A or B with CC.C
ADD<A, B, or D>	Add memory location to accumulator
INC<A or B>	Add 1 to register or memory location

The ABX instruction is the simplest add instruction. This instruction takes the contents of B to be an unsigned 8-bit value (0-255) and adds it to X leaving the result in X. The condition code flags are not affected. This instruction is similar in effect to the instruction LEAX B,X but there are important distinctions. Firstly, the value of B in an LEA instruction is taken as an 8-bit two's complement number so may take a value between -128 and 127. The value of B in an ABX instruction can range between 0 and 255. Secondly, ABX is a 1-byte inherent address (this means that the instruction operands are always the same) so it is shorter than the corresponding LEA instruction. The provision of this instruction allows certain kinds of indexed addressing to be implemented in a very efficient way.

The add with carry or ADC instruction operates on either accumulator A (ADCA) or accumulator B (ADCB). This instruction adds the contents of the register plus the carry bit CC.C to the specified memory location leaving the result in the register. The memory location may be addressed using direct, indexed or extended addressing or may be an immediate value.

ADC instructions are used when multiple-byte arithmetic is implemented where it is necessary to take a carry from a previous arithmetic operation into account. The ADC instruction affects the C, N, V, Z, and H bits of CC.

Examples of ADC instructions are:

```
ADCA #35 ; A = A + CC.C + 35
```

```
ADCB ,X ; B = B + CC.C + MEM(X)
```

Add instructions operate on registers A, B, and D and their function is to add an immediate operand or a memory location to one of these registers. Like ADC

instructions, the C, N, V, Z, and H bits in the condition code register are affected by an ADD instruction.

Examples of add instructions are:

```
ADDA SVAL ; A = A + MEM(SVAL)
```

```
ADDB #5 ; B = B + 5
```

```
ADDD ,--Y ; Y = Y - 2 : D = D + MEM(Y)
```

The INC instructions are special purpose add instructions which are used to add one to the single byte accumulators A and B or to a specified memory location. Although this operation can be implemented in other ways, the 'add 1 to something' operation is so common that it is worth providing it as a separate machine instruction.

The instructions INCA and INCB are 1-byte instructions with no address field whereas the memory increment instruction INC may use direct, indexed or extended addressing. For example:

```
INCA ; A = A + 1
```

```
INCB ; B = B + 1
```

```
INC FRED ; MEM(FRED) = MEM(FRED) + 1
```

The INC operation affects the N, Z and V bits of the condition code register.

3.2.2 Subtract instructions

There are three types of subtract instruction available to the M6809 programmer which are the converse of ADC, ADD and INC. These are the instructions SBC (subtract with carry), SUB (subtract), and DEC (decrement by 1).

The function of these instructions is to subtract an immediate operand or the value of a memory location from a register, leaving the result in that register. The operands for this operation must be in two's complement form.

All the subtract operations set the overflow flag CC.V if the result is too small to be held in the specified register or memory location. They also affect the N and Z flags in CC and the instructions SUB and SUBC set the carry flag in the event of a borrow occurring in the last place of a subtraction.

The SBC instructions operate on registers A and B and subtract CC.C as well as an immediate value or a memory location value from the specified register. For example:

```
SBCA J ; A = A - MEM(J) - CC.C
```

SBCB 4,Y ; B = B - MEM(4 + Y) - CC.C

The subtract instruction SUB operates on registers A, B, or D. For example:

SUBA #4 ; A = A - 4

SUBB \$30 ; B = B - MEM(30)

SUBD POINTER ; D = D - MEM(POINTER)

The decrement instruction, DEC, subtracts 1 from an 8-bit value held in either A, B or a memory location. For example:

DECA ; A = A - 1

DECB ; B = B - 1

DEC CVAL ; MEM(CVAL) = MEM(CVAL) - 1

3.2.3 Clear instructions

The function of clear instructions (CLR) is to set register A or B or a 1-byte memory location to zero, that is, to clear it of its previous value. The CLRA and the CLR B instructions are 1-byte instructions with no address field whereas the CLR instruction may use direct, indexed or extended addressing.

Examples of clear instructions are:

CLRA ; A = 0

CLRB ; B = 0

CLR A,X ; MEM(A + X) = 0

3.2.4 The multiply instruction

On most 8-bit microprocessors multiply instructions do not exist. Multiplication is implemented by a software routine which performs a sequence of repeated additions to multiply two numbers. The reason for this is that multiplication is a relatively complex operation whose result is always twice as long as its operands. To include this in an 8-bit architecture increases the complexity of that architecture as provision must be made for a 16-bit result.

The implementation of multiplication by repeated addition obviously makes it a relatively slow process compared to addition and subtraction. Furthermore, it is a fairly common operation when accessing elements of two-dimensional arrays or matrices. As the M6809 is a hybrid microprocessor whose architecture includes 8-bit and 16-bit features, the designers of that chip have included a limited form of multiply instruction. The

multiply instruction, which has the op-code MUL, is a 1-byte instruction which takes the contents of accumulators A and B as its operands and leaves the result of the multiplication in accumulator D. As D is a catenation of A and B, the original operands are destroyed.

The MUL instruction takes the values in A and B to be unsigned 8-bit values rather than two's complement numbers. The reason for this is that the use of unsigned multiplication makes it easier for the programmer to write multi-byte multiplication routines for multiplication and that the array element computation referred to above generally uses only positive array indexes.

An example of a multiply instruction is:

```
MUL    ; D = A * B
```

3.2.5 Negate instructions

Negate instructions operate on 8-bit two's complement values held in register A, register B or in memory. They are written as NEGA, NEGB, or NEG <address>. NEGA and NEGB negate the contents of registers A and B respectively whereas NEG may use direct, extended or indexed addressing.

Examples of negate instructions are:

```
NEGA    ; A = -A
```

```
NEGB    ; B = -B
```

```
NEG SVAL ; MEM(SVAL) = -MEM(SVAL)
```

3.2.6 The sign extend instruction

The sign extend instruction, SEX, is a 1-byte instruction whose function is to convert an 8-bit two's complement number held in accumulator B into a 16-bit two's complement number in accumulator D. In essence, it takes the sign bit of B and extends it so that it becomes the sign bit of D. The value of the hi-byte of D is set up to be the same as the sign bit of B. This means that if the number is positive, sign bit = 0, accumulator A is cleared. If the number in B is negative, accumulator A is filled with 1s.

3.2.7 The decimal adjust instruction

The decimal adjust instruction is used when decimal arithmetic, described in section 1.1.4, is used on the M6809. The use of decimal arithmetic entails holding two 4-bit digits in an 8-bit register rather than an 8-bit binary number.

When an add operation is performed on such a value, a binary addition takes place so that the numbers held in each of the 4-bit register fields need not

necessarily be correct. For example, say the numbers 27 and 53 are added. When represented in 4-bit decimal notation these have binary values 00100111 and 01010011 respectively. When a binary addition is performed, the result is 01111010 which cannot be represented as decimal as the first digit is 7 and the second is hexadecimal A. Clearly, the result of the addition should be 80 which in binary form is 10000000.

The decimal adjust instruction examines register *P*. and also the carry bits CC.H and CC.C. It checks to see if an incorrect decimal value is stored in that register. If so, it adjusts the decimal digits so that the correct value is restored. In the above example, it would check bits 0-3 of the number, see that they were an impossible decimal number and would convert this to the correct number by adding 6 to it. This results in a carry into bits 4-7 thus increasing the decimal value stored there to 8. The correct number is then represented in the register.

The need for the half-carry bit CC.H now becomes clear. If bits 0-3 of the decimal numbers are such that an addition generates a value which cannot be stored in 4-bits, the half-carry bit is set. The decimal adjust instruction recognises this and adjusts the decimal digits accordingly.

3.3 LOGIC INSTRUCTIONS

Like the M6809's arithmetic instructions, the logic instructions are almost exclusively concerned with operations on the A and B registers and with individual memory bytes. The two exceptions to this are instructions which operate on the condition code register and which provide a generalised mechanism for setting and unsetting individual flag bits in that register.

Logic operations manipulate the individual bits in their operands and look upon these operands as simple groups of bits (bitstrings) rather than as numeric values. For the reader who is unfamiliar with logic operations we describe the actual operation as well as the instruction format along with each class of logic instruction.

Logic instructions may be looked upon as falling into one of five classes:

- (1) And instructions
- (2) Or instructions
- (3) Complement (not) instructions
- (4) Shift instructions

(5) Rotate instructions

Individual instructions are described under the appropriate heading below.

3.3.1 And instructions

The logical and operation takes 2 bits as its operands and returns a value of 1 if, and only if, both of its operands are 1. All possible operands and results for this operation are therefore:

```
0 AND 0 -> 0
1 AND 0 -> 0
0 AND 1 -> 0
1 AND 1 -> 1
```

The M6809's and instructions operate on 8-bit data so therefore repeat the above operation for all 8-bits in the operand register. The registers A, B, and CC may take part in and operations.

The instructions ANDA and ANDB perform a logical and on the contents of the named register and a byte in memory or an immediate operand. Direct, indexed or extended addressing may be used to reference a memory byte.

The ANDCC operation, on the other hand, may only use immediate addressing. Its function is to and the CC register with the immediate byte provided leaving the result in CC.

Examples of and instructions are:

```
ANDA #$F0      ; Ands A with (hex) F0.
*              Note that the effect of this is
*              to clear bits 0-3 in A
*              and to leave bits 4-7 unchanged

ANDB MASK      ; Ands B with MEM(MASK)

ANDCC #$00     ; Ands CC with (hex) 00
*              This clears CC
```

The reader will have gathered from these examples that one of the most important functions of the and operations is to clear specific bits in a register whilst leaving the other bits unchanged. Anding a 0 with a 1 bit always clears it whereas anding a 1 with either a 1 or a 0 always leaves that value unchanged.

3.3.2 Or instructions

There are two types of or instructions provided on the M6809. These are so-called inclusive or and exclusive or which have mnemonics OR and EOR respectively.

These operations can be defined by their effect on bit values:

```

0 OR 0 -> 0      0 EOR 0 -> 0
1 OR 0 -> 1      1 EOR 0 -> 1
0 OR 1 -> 1      0 EOR 1 -> 1
1 OR 1 -> 1      1 EOR 1 -> 0

```

Like the and instructions, or instructions are provided which operate on registers A, B, and CC. However, there is no EORCC instruction - only EORA and EORB are available to the programmer.

Examples of OR and EOR instructions are:

```

ORA #$0F      ; Or (hex) 0F with register A
*
*           Note the effect of this is to
*           set bits 0-3 of A whilst leaving
*           bits 4-7 unchanged

EORB ,X      ; Exclusive or B with MEM(X)

ORCC #$03    ; Or (hex) 03 with CC thus setting
*           bits 0 and 1 in that register

```

Just as and instructions can be used to clear specific bits in a register, or instructions may be used to set specific bits. Oring with a 1 bit always sets the corresponding register bit whereas oring with a 0 always leaves that bit unchanged.

3.3.3 Complement instructions

Complement instructions simply switch the bits in a register or memory byte. That is, all 1 bits are set to 0 and all 0 bits are set to 1. For example, if B holds the bitstring 10010011, executing a COMB instruction results in the bitstring 01101100 being stored in B.

Single byte instructions are available to complement registers A and B as is a memory complement instruction which may use direct, indexed or extended addressing. An alternative name which is sometimes used for the complement operation is the 'not' operation.

Examples of complement instructions are:

```

COMA      ; Complement register A

COM B,X   ; Complement MEM(B + X)

```

The complement operation is not the same as the NEG arithmetic operation. The NEG operation forms the two's complement of a number whereas the COM operation forms the so-called one's complement value.

3.3.4 Shift instructions

The purpose of shift instructions is to move all the bits in a register along one place to the left or to the right with the leftmost or rightmost bit 'falling off the end' and being discarded. For example, if a

register holds the binary value 10110001 and is shifted left, the resultant value is 01100010. If a right shift is executed, the resultant value is 01011000. Notice that 0s are filled in on the left when a right shift is executed and on the right when a left shift takes place. The M6809's shift instructions fall into two classes:

- (1) Arithmetic shift instructions
 Arithmetic shift instructions consider bit 7 of the register being shifted to be the sign bit. This bit does not take part in arithmetic shift right instructions and its value is preserved. The bit is shifted during arithmetic shift left (ASL) instructions. For example, if a register value is 10010011 and an ASL instruction using that register is executed, the resultant value is 00100110. However, with ASR bits 0-6 are shifted with the sign bit propagated into the lower bits. The resulting value is 11001001.
- (2) Logical shift instructions
 Logical shift instructions do not recognise the sign bit and their operands are shifted to the left or to the right as described in the introduction to this section. Logical shifts have mnemonics LSL (logical shift left) and LSR (logical shift right). Notice that the LSL and the ASL instructions are equivalent.

The arithmetic and logical shift instructions operate on the A and B registers and on memory bytes accessed using direct, indexed or extended addressing. Shift instructions always affect the carry bit CC.C whose value becomes that of the bit which is shifted out of the register.

Examples of shift instructions are:

```

ASLA          ; Shift A left by 1 bit with
*             CC.C set to the value of bit 7
*             of A before the shift

ASRB          ; Shift B right by 1 bit with
*             CC.C set to the value of bit 0
*             of B before the shift

LSL SVAL      ; MEM(SVAL) is shifted left by
*             1 bit with CC.C set accordingly

LSR -16,U     ; MEM(U-16) is shifted right by 1 bit
*             with CC.C set accordingly

```

3.3.5 Rotate instructions

Rotate instructions are similar to logical shift

instructions. The only difference is that the value of the carry bit CC.C rather than a 0 is moved to the leftmost or rightmost place of the register, depending on whether a rotate right or rotate left instruction is executed.

The mnemonics for rotate right and rotate left instructions are ROR and ROL respectively and they operate on the A or B registers or on a memory byte. As usual, direct, indexed or extended addressing may be used to refer to this byte in memory.

Examples of rotate instructions are:

```

RORA      ; A is shifted right by 1 bit with
*         bit 7 becoming CC.C and CC.C taking
*         the value of bit 0 before the shift

ROL SVAL  ; MEM(SVAL) is shifted left by 1 bit
*         with bit 0 becoming CC.C and CC.C set
*         to the original value of bit 7.

```

3.4 TEST INSTRUCTIONS

The M6809's test instructions allow the programmer to determine if certain conditions are true or false. The execution of a test instruction always causes one or more bits in the CC register to be set or unset depending on the result of the test. Thus CC bit settings are the means by which test results are 'remembered' for use by following instructions.

There are three kinds of test instructions:

- (1) Bit test instructions
- (2) Byte test instructions
- (3) Compare instructions

Bit test instructions only operate on registers A and B and byte test instructions on A, B and memory bytes. Compare instructions, however, are available for all index and accumulator registers.

3.4.1 Bit test instructions

The bit test instructions BITA and BITB are used to test if particular bits (0-7) in register A or B are 1 or 0. The operand of the bit test instruction is a single byte called a mask whose value determines which bits in the specified register are to be tested.

In order to test bit *n* in the register, the mask is set up so that only its *n*th bit is 1 with all other mask bits set to 0. Therefore, to test bit 4, the mask value should be 10 (hex) and to test bit 6, it should be 40 (hex).

If the bits being tested are set, the effect of the

bit test instruction is to unset the zero flag (Z-flag) in the CC register. Recall that this flag is always set when the result of an operation is zero and unset when the result is non-zero. Bit test is implemented as an and operation but without the anded value being stored in the specified register. Therefore, if a tested bit is 1, CC.Z is 0 and if a tested bit is 0, CC.Z is 1.

Examples of bit test instructions are:

```

    BITA #80      ; Tests bit 7 of A
*           CC.Z = not A.7

    BITB MASK    ; Tests the bits of register B
*           according to MEM(MASK)

```

3.4.2 Byte test instructions

Byte test instructions are used to test if a byte in memory, register A or register B is positive, negative or zero. The mnemonic for these instructions is TST with, as usual, A or B appended to it if registers are tested. If a memory byte is being tested it may be addressed using direct, indexed or extended addressing.

Byte test instructions are implemented by subtracting 0 from the contents of the byte being tested. The result of this subtraction causes the negative flag and the zero flag in the CC register to be set or unset. We have already discussed how the Z-flag is set if the result of the previous operation is zero so, if the tested byte is zero, CC.Z is set and CC.N is unset.

If the byte tested is positive, both CC.Z and CC.N are unset, whereas if it is negative CC.Z is 0 and CC.N is 1. In all cases the byte test instruction causes the overflow bit CC.V to be unset.

Examples of byte test instructions are:

```

    TSTA          ; Test register A

    TST 16,X      ; Test MEM(16 + X)

```

3.4.3 Compare instructions

Compare instructions allow registers A, B, D, X, Y, S, and U to be compared with one or two bytes in memory or with an immediate operand. Allowed addressing modes are direct, indexed and extended addressing. The mnemonic for compare instructions is CMP followed by the name of the particular register used in the comparison.

Like byte test instructions, compare instructions are implemented as a subtraction with no permanent effect on the instruction operands. The addressed 8-bit or 16-bit quantity is subtracted from the register contents and the carry, overflow, zero and negative bits in the condition code are set accordingly.

If the value in memory is less than the register value, the result of the comparison is positive so CC.N is unset. If it is greater than the negative value, the result is negative so CC.N is set, and if the values are equal, the result of the subtraction is zero so CC.Z is set.

Examples of compare instructions are:

CMPX [MAXADD] ; Compare X with MEM(MEM(MAXADD))

CMPB #10 ; Compare B with (decimal) 10

CMPD 16,U ; Compare D with MEM(16 + U)

Compare instructions are mostly used immediately before branch instructions to implement loops, conditions, etc. The programmer need not explicitly be aware of which bits in CC are set or unset by the compare instruction when they are used in this way.

3.5 BRANCH INSTRUCTIONS

The M6809's branch instructions are provided to give the programmer control over the flow of execution of his program. They allow single bits or combinations of bits in the condition code register to be tested and, on the basis of these tests, add or subtract some value from the PC register. This PC modification results in a break in the normal sequential execution of machine instructions and transfers control to some other instruction.

Branch instructions may be considered under four headings:

- (1) Unconditional branch instructions
These always cause a transfer of control irrespective of the bit settings in the CC register.
- (2) Simple conditional branch instructions
These test a single bit in the CC register with a control transfer dependent on its value.
- (3) Signed conditional branch instructions
These are used if, in the previous test, signed register contents were compared with signed contents of memory. They test one or more bits in CC with control transfers dependent on their values.
- (4) Unsigned conditional branch instructions
These are similar to signed conditional branch instructions but are used when unsigned values were compared in a previous operation.

All branch instructions use PC relative addressing with the value to be added to PC held as an 8-bit or 16-bit instruction operand. Because the operand may be 1 or two bytes, there are 2 forms of every branch instruction, a short form and a long form. Short branch instructions have the form:

B<condition> <1 byte 2's complement displacement>

Long branch instructions have the form:

LB<condition> <two byte 2's complement displacement>

In the description and examples below, it is convenient for us to show only the short form of the branch instructions. However, the reader should bear in mind that long branch forms are also allowed. The actual machine code value for the long branch form of a branch instruction is usually made up by prefixing the corresponding short branch op-code with 10 (hexadecimal). Long branch instructions are used when the displacement in the branch instruction is less than -128 or greater than 127.

3.5.1 Unconditional branch instructions

There are three distinct unconditional branch instructions available to the M6809 programmer. These are:

BRA Branch always
BRN Branch never
BSR Branch to subroutine

The BRA instruction is equivalent to a BASIC GOTO statement and the BSR instruction to a BASIC GOSUB statement. These instructions always add their displacement to PC irrespective of the settings of CC flags. In addition, the BSR instruction, before modifying PC, stacks that register on the hardware stack referenced by the S register. This means that, on return from the subroutine, execution can be resumed at the instruction which follows the BSR instruction.

The BRN instruction is a so-called no-op instruction. In short it does nothing at all except take up 2 or 4 bytes of space. When this instruction is executed, control immediately moves on to the following instruction. This may, therefore, appear to be a useless instruction. However, it has its uses when the programmer wishes to cheat a little and hide a 1 or 2 byte instruction in the operand field of the BRN instruction. After the first execution of BRN when this instruction is ignored, it is possible to branch back to the hidden instruction and execute it. This, however, is poor programming practice and is not a recommended technique.

3.5.2 Simple conditional branch instructions

Simple conditional branch instructions examine a single bit in the M6809's condition code register. Instructions exist which branch on the setting of the carry flag, the overflow flag, the negative flag, and the zero flag. There are two instructions which test each flag. One of these instructions branches if the flag is set, the other branches if the flag is unset.

The table below lists the simple conditional branch instructions and shows their association with condition code flags.

Flag	Mnemonic	Function
C	ECS	Branch if carry bit is set
	BCC	Branch if carry bit is unset (clear)
V	BVS	Branch if overflow bit is set
	BVC	Branch if overflow bit is clear
Z	BNE	Branch if zero bit is unset that is, when comparison operands are not equal
	BEQ	Branch if zero bit is set that is, when comparison operands are equal
N	BMI	Branch if negative bit is set
	BPL	Branch if negative bit is unset

As with all other branch instructions, these may take an 8-bit or 16-bit signed two's complement offset thus allowing forward or backward branching. If a 16-bit offset is used, the mnemonics above must be prefixed with an L to indicate long branching.

3.5.3 Signed conditional branch instructions

Signed conditional branch instructions are used when a preceding operation has compared the values of signed, numeric operands. These branch instructions examine combinations of condition code flags to determine if the specified condition is true or false and if branching should occur.

The table below shows the four distinct signed conditional branch instructions available to the M6809 programmer. In addition to these, the simple conditional branch instructions BEQ and BNE may also be used as signed conditional branches, where the branch takes place if the operands in the preceding comparison were equal or not equal.

Flag combination	Mnemonic	Function
NOT(Z OR (N XOR V))	BGT	Branch if greater than
NOT(N XOR V)	BGE	Branch if greater than or equal

Z OR (N XOR V)	BLE	Branch if less than
(N XOR V)	BLT	Branch if less than or equal

Notice that the pairs of conditions above are complementary with the greater than conditions the inverse of the less than conditions. BLE is the complement of BGT and BGE is the complement of BLT. We therefore only explain the flag combinations for a single pair of instructions BLE and BLT.

The BLE instruction branches if, in the preceding comparison, the register operand was less than or equal to the memory operand. If register A was tested against MEM(VAL) say, we might write this as $A \leq \text{VAL}$. If the operands are equal, the Z-flag in CC is set. This flag is examined by BLE and branching occurs if it is set.

If A and MEM(VAL) have the same sign, the subtraction operation entailed in the comparison can never result in overflow so CC.V is always cleared. If A is indeed less than MEM(VAL), the subtraction will result in a negative value so CC.N will be set. Therefore, if CC.N is set and CC.V unset, this indicates that A is less than MEM(VAL) and branching will occur. If CC.V is unset and CC.N is unset, A is not less than MEM(VAL).

In the case where A and MEM(VAL) have different signs, the comparison may result in an overflow occurring. Thus the sign bit will have an incorrect value. If CC.V is set, indicating overflow, and CC.N is unset, indicating a non-negative value, this actually means that the result is negative. On the other hand, if both CC.N and CC.V are set, the result is positive.

Because of the meanings of these bit combinations, the exclusive or operation performed on CC.N and CC.V always gives the correct sign bit for the number. Therefore, if this operation returns 1, the result of the comparison is negative and branching should take place.

The BLT instruction can be considered as a less general form of the BLE instruction which only branches when the register operand is less than the memory operand. The above argument holds for this instruction also. The BGT and the BGE instructions are simply the complements of these so a not operation performed on the corresponding 'less than' condition bits allows these instructions to determine if branching should take place.

3.5.4 Unsigned conditional branch instructions

Unsigned conditional branch instructions are used when the preceding operation compares the values of unsigned operands. Again, these instructions test condition

code register flag combinations to determine if branching should take place.

The table below shows the four unsigned conditional branch instructions and the flags tested in the CC register. Again, the BEQ and BNE instructions may be used under this category.

Flag combination	Mnemonic	Function
C	BLO	Branch if lower
C OR Z	BLS	Branch if lower or the same
NOT(C)	BHS	Branch if higher or the same
NOT(C OR Z)	BHI	Branch if higher

Again the instruction pairs BLO/BHS and BLS/BHI are complementary so we shall only discuss the operations BLO and BLS. As these operations assume that the previous comparison tested unsigned operands, the negative flag CC.N is not tested by these instructions. As always, if the result of the comparison is zero, CC.Z is set so the BLS instruction branches if this flag is 1.

As the comparison operands are unsigned, the subtraction entailed in the comparison is essentially a subtraction of positive values. If the second operand is greater than the first, the subtraction will result in a borrow. Thus, the carry bit in CC will be set. If the second operand (the memory operand) is smaller than the first, no borrow will result so the carry bit will be unset. Therefore, the BLO and BLS instructions examine the carry bit and branch if it is set.

So far, we have not provided any explicit examples of branch instructions as, unlike other instructions considered so far, examples of these instructions are meaningless in isolation. To illustrate some of the branch instructions in use we show below the assembly code equivalent to a number of BASIC statements involving loops and conditional operations.

```

100 IF V1 > V2 THEN GOTO 500
200 IF V1 = V2 THEN GOTO 700
300 V1 = V1 + 2
400 GOTO 200
500 M = V1
600 GOTO 800
650 REM ASSUME A SUBROUTINE EXISTS AT 2000
700 GOSUB 2000
800 ...

```

Assuming V1, V2 and M are represented as 16-bit signed

quantities and that the subroutine at 2000 has the symbolic name V1EQ, an assembly code sequence which would carry out the same function is:

```

                LDD     V1      ; D = V1
CMPLAB         CMPD  V2      ; Compare this with V2
                BGT  GTLAB    ; If greater than branch
                BEQ  EQLAB    ; If equal branch.
*
*              Notice there is no need for
*              another load or comparison
                ADDD  #2      ; Add 2 to D
                STD   V1      ; and put result back into V1
                BRA  CMPLAB    ; Branch back to comparison
GTLAB         STD   M        ; V1 > V2 so M = V1
                BRA  NXTLAB    ; continue
EQLAB         BSR  V1EQ      ; Values equal, call routine
NXTLAB

```

Notice how the assembly code version of the sequence is only slightly longer than the BASIC. Whilst, in general, BASIC statements expand into multiple assembly code instructions it is often possible to eliminate much of the redundancy inherent in high level language programming and hence produce compact code.

3.6 INTERRUPT HANDLING INSTRUCTIONS

An interrupt is a means by which a program, executing on a processor, can be temporarily suspended whilst some other program executes. They are of vital importance in I/O programming where interrupts are used by peripheral devices to inform the processor that data are available. The processor must stop what it is doing, collect the data from the peripheral then restart its original activity.

The interrupt handling instructions available to the M6809 programmer are described in full in Chapter 8 which covers I/O programming. Here, we simply list the interrupt handling instructions which are available and summarise their functions.

- (1) The wait instruction
This instruction, mnemonic CWAI, takes a single byte operand which is anded with the contents of CC when the instruction is executed. The E flag in the condition code register is then set, indicating that all registers should be stacked on the hardware stack. The instruction then waits (does nothing) until a hardware interrupt occurs. Interrupt processing, as detailed in Chapter 8, then commences.
- (2) The return from interrupt instruction
The return from interrupt instruction, RTI, is

executed after interrupt processing is complete. It unstacks the register values pertaining when the interrupt occurred thus returning control to the interrupted process.

- (3) The software interrupt instruction
This instruction, which has mnemonic SWI, causes a so-called software interrupt. A software interrupt causes the processor to jump to an associated interrupt service routine which may, for example, transfer control to some other process. Thus the execution of programs in different parts of the M6809's memory may be coordinated and synchronised.
- (4) The synchronise instruction
This instruction, SYNC, is used to synchronise an executing program with some external hardware event.

Interrupt handling instructions are special purpose instructions and are unnecessary for most applications programmed in assembly code.

3.7 MISCELLANEOUS INSTRUCTIONS

In this section, we describe the remaining M6809 machine instructions which don't fit neatly into any of the above classifications. There are only four instructions in this category. These are:

- (1) The jump instruction
- (2) The jump to subroutine instruction
- (3) The return from subroutine instruction
- (4) The no operation instruction

We shall start with the 'no operation' instruction which has mnemonic NOP. Its function is very easy to describe - it does nothing. A NOP instruction is 1 byte long and all it does is take up memory space. This can be useful if it is necessary to force other instructions to occupy particular memory locations.

3.7.1 Jump instructions

The jump instructions available to the M6809 programmer are similar to the branch instructions discussed earlier in this chapter. The function of these instructions is to evaluate their operand and load its value into the program counter register. Therefore, if addresses of other instructions are saved as data, you can transfer control to these instructions using a jump instruction.

The addressing modes allowed with jump instructions are direct, indexed and extended addressing. There are two jump instructions JMP, which is an unconditional jump, and JSR, which is a jump to subroutine instruction. The only difference is that JSR stacks the program counter PC on the hardware stack before assigning its operand to the PC register.

Examples of jump instructions are:

JMP B,U ; PC = MEM(B + U)

JSR ,U ; S = S - 2: MEM(S) = PC: PC = MEM(U)

3.7.2 The return instruction

The return from subroutine instruction, whose mnemonic is RTS, is executed as the last instruction in a subroutine. It unstacks the top two bytes from the hardware stack and assigns them to PC. This effectively transfers control to the instruction following the BSR or JSR instruction which initiated the subroutine.

Chapter 4

Introducing assembly language

Assembly language programming is a form of computer programming where the programmer writes his program as a sequence of absolute directives to the processor. That is, he states exactly which machine instructions are to be used in the execution of his program.

This type of programming is sometimes called low-level programming because it is a notation which is very close indeed to machine language. By contrast, programming in a language such as BASIC is called high-level language programming. The programmer writes his program at a much higher level where the details of the machine architecture are irrelevant.

High-level programming is much easier than low-level programming because machine architectures are inherently complex. The low-level programmer must master all the details of this complexity if he is to avoid making programming errors. The high-level programmer, on the other hand, has many fewer details to remember and can concentrate on getting the logic of his program correct - a difficult enough task in itself.

The majority of computer applications can be programmed perfectly adequately in a high-level language and there is no point in programming in assembly language when BASIC will do. However, in personal computers, like the Dragon, there are some tasks which are easier to program in assembly language rather than BASIC because they require access to hardware features of the machine. Although this is possible from BASIC, it is clumsy and inconvenient as it requires the use of many POKE and PEEK instructions.

There are also some types of program which, if programmed in BASIC, are too slow. This slowness results from the way in which BASIC is implemented. Every BASIC statement must be translated to machine code just before it is executed and this takes a significant amount of time. As this translation is absolutely essential, the only way to speed these programs up is to program them or, at least those time-critical parts of them, in assembly code.

As we have already suggested, the real difference between programming in assembly language and programming in BASIC is one of detail. In BASIC,

decisions about where the program and its data are to be located in memory, how real numbers are to be provided, how character strings are manipulated, etc. are all made for the programmer by the BASIC system. As well as this, BASIC programs are expressed in such a way that they are readily understood by people whereas the notation used for assembly code bears little relation to the logical processes involved in solving the problem at hand.

However, in spite of these difficulties, there are three fundamental advantages in programming in assembly language rather than BASIC:

- (1) The programmer has complete control over the machine. If he wishes to use his own particular way of manipulating characters or to access hardware features in some non-standard way, this is possible in assembly language but impossible in BASIC.
- (2) Assembly language programs are very much faster than equivalent BASIC programs. Because the translation phase from BASIC to machine code is avoided, assembly language programs typically execute at least 100 times faster and sometimes as much as 1000 times faster than corresponding BASIC programs. This means that they are suitable for programs, like some arcade-type games, which must react very quickly to input from the user.
- (3) Assembly language programs are more compact, that is, occupy less memory, than their BASIC equivalents. This is particularly important when large programs are written which may require almost all of the memory available on the machine.

Of course, there are also disadvantages associated with programming in assembly language apart from the obvious one that the programmer must remember many low-level details of the machine. These disadvantages are:

- (1) Because the programmer has complete control over the machine, it is more difficult to detect mistakes in assembly code programs. As long as a valid instruction is written, something will happen even although the instruction does not do what the programmer really wants. Whereas the BASIC system has many built-in checks which detect errors like dividing by zero, no built-in error detection is available to the assembly language programmer.
- (2) Because of the low-level nature of assembly

language programs and because the programmer must explicitly include his own error checking facilities, assembly language programs are usually a good deal longer than their BASIC equivalents. This means that they take longer to write, are more difficult to understand, and are likely to contain more mistakes than high-level language programs.

Because of the complexity of assembly language programming, it is best to adopt a multi-stage approach when developing a program which is ultimately written in assembly code.

The first stage is to work out the solution to your problem in very general terms and to write down this solution in some stylised way. This is a very high-level expression of what your program ought to do. For example, say you are developing a game where the player must shoot down alien spacecraft. Part of the general high-level expression of this might be:

```

if firing button pressed then
    launch missile
if alien detects missile launch then
    drop anti-missile bomb
if dodge key pressed then
    move missile to avoid bomb
else
    missile destroyed

```

In fact, this approach is always how we work out the logic of programs although, sometimes, we do it in our heads rather than explicitly on paper. Writing down the solution is much better because when we hold detailed information mentally it is very easy to forget bits of the problem solution or to make mistakes when mentally translating to a programming language.

The second stage, which is particularly important for inexperienced assembly language programmers is to translate the general, abstract problem solution into a high-level programming language like BASIC. Here, you must decide how logical operations such as 'firing button pressed' are actually to be implemented. For example, in the above program, missile dodge keys might be '4' to move left and '6' to move right. We might code that part of the solution as:

```

KEY$ = INKEY$
IF KEY$ = "4" THEN MISX = MISX - 1
IF KEY$ = "6" THEN MISX = MISX + 1

```

where MISX represents the x-coordinate of the missile.

An advantage of this intermediate stage between problem solution and assembly code program is that the

BASIC program can sometimes act as a prototype for the final program. This lets you try out ideas and debug the logic of the solution before becoming involved with the details of assembly language.

The third stage in the development of an assembly language program is to take the high-level language program and to translate it, by hand, to assembly code. This is a straightforward process and the assembly language equivalents for BASIC statements are described later in this chapter.

Sometimes it isn't necessary to translate the complete program into assembly code. Typically, most programs have relatively small sections, such as a display subroutine, where they spend most of their time. It is possible to code these time-consuming subroutines in assembly language and to link them into a BASIC program. This often gives the speed-up effect desired by the programmer and the chore of translating the whole program into assembly code can be avoided. We explain how assembly code subroutines can be linked with BASIC programs in Chapter 6.

In any direct translation of a BASIC program to assembly language, there is bound to be redundancy. For example, say we have two BASIC statements:

```
M = M + 1
V = V + M
```

A direct translation of these into assembly code, assuming that both M and V can be held as 8-bit integers, is:

```
LDA M      ; A = MEM(M)
ADDA 1     ; A = A + 1
STA M      ; MEM(M) = A
LDA V      ; A = MEM(V)
ADDA M     ; A = A + MEM(M)
STA V      ; MEM(V) = A
```

However, this can be optimised by using the INC instruction to add 1 to A. An optimised version of this instruction sequence is therefore:

```
INC M      ; MEM(M) = MEM(M) + 1
LDA V      ; A = MEM(V)
ADDA M     ; A = A + MEM(M)
STA V      ; MEM(V) = A
```

The final stage in developing an assembly code program, therefore, is to take the BASIC equivalent program and to eliminate redundant steps in order to optimise the program. Some obvious elimination of redundancy can be done during stage three but program rearrangement, the use of different addressing modes, etc. should be left

until this final stage. In the examples in the following chapters we show how optimisation can make a considerable difference to the size of a program.

This chapter and the following two chapters are devoted to assembly language programming. In the remainder of this chapter, we describe a class of program called assemblers. An assembler translates instruction mnemonics, symbolic names, etc. used by the programmer to machine code. It is a vital tool for the serious assembly language programmer.

The following chapter, Chapter 5, shows how commonly used programming constructs such as assignments, loops and conditional statements may be programmed in assembly language. The approach which we use here is to take BASIC statements implementing these constructs and show how assembly language equivalents to these can be built up. We also show how these 'BASIC-equivalent' programs can usually be optimised to produce a program which has improved space and time efficiency.

Chapter 6 looks at more advanced aspects of assembly language programming. In that chapter, we describe a general-purpose technique for implementing subroutines and we show how character strings may be represented and manipulated. We also describe how to link assembly language subroutines with BASIC programs and how to write assembly code which is position independent.

It is beyond the scope of this book to discuss assembly language programming in great detail. This requires a book in itself and, to supplement the material here, the reader may find it useful to refer to some of the textbooks on M6809 assembly language programming which are listed in the reading list.

4.1 THE ASSEMBLER PROGRAM

We have already introduced, in earlier chapters, the idea of an assembler as a program which translates assembly language statements to machine code. This translation is not a difficult process as it simply requires the program to look up tables of names and associated hexadecimal values. However, for humans this is a slow, tiresome, error-prone task. In fact, it is the kind of job that computers excel at and we recommend that you should try to avoid the hand translation of assembly code.

For each machine, there are usually several different assemblers available from different suppliers. Some of these might have more sophisticated features than others but all will provide at least the following facilities.

- (1) The translation of mnemonic instructions to their equivalent hexadecimal op-codes.

- (2) The ability to associate labels with assembly language statements. Reference to these labels within the program will result in the address of the labelled statement being substituted for the label.
- (3) The association of names with specific memory locations. When these variable names are used by the programmer, the assembler substitutes the actual memory address of the variable.
- (4) The translation of decimal numbers to their hexadecimal equivalent.
- (5) The translation of symbolised address references such as [,X] for indirect indexed addressing to the appropriate postbyte, offset, etc.
- (6) Limited error checking indicating if an invalid mnemonic has been used, if a label referenced in an instruction is not declared, if a short branch is used where a long branch is required, etc.

The particular assembler whose facilities we shall describe in this chapter is the DREAM assembler, available from the manufacturers of the Dragon. This is a typical assembler which uses fairly standard Motorola M6809 notation, as set out in Appendix 1, for assembly language instructions. There may be slight differences in detail if you use a different assembler but, in general, the description of facilities below applies to all assemblers which are available for the Dragon.

The single exception to the standard notation is when indirect addressing is used. As the symbols '[' and ']' are not Dragon keyboard characters, the DREAM assembler uses round brackets '(' and ') ' to indicate indirect addressing. We shall follow this convention from now on but the reader who is using some other assembler should read (<address>) as [<address>].

As well as being an assembler, DREAM is also an editor. It provides facilities for inputting, modifying, duplicating and saving text on a cassette. This text need not be assembly code but may be anything at all. However, as the editing and assembling facilities are combined, the implementors of DREAM clearly see the creation and editing of assembly language instructions as its major task. As assembly language instructions do not have explicit line numbers, it is not possible to use the BASIC editor to create and edit assembly language programs.

The standard format for an assembler source line as input to DREAM or any other assembler based on the standard Motorola notation is:

```
<label> <mnemonic> <operand> <comments>
```

The different fields in the source line must be separated by one or more spaces. The <label> and <comments> fields are optional, the <mnemonic> field must be present as must the <operand> field except for those instructions which use inherent addressing and do not require explicit operands.

4.1.1 The label field

The label field, if present, must start in the first column of the source line. Anything that starts in column 1 is therefore taken, by the assembler, to be a label. If you make a mistake and put a mnemonic in column 1 or a label starting in some other column the assembler will get very confused indeed.

Labels must start with a letter and may only contain alphanumeric characters, that is, letters and numbers. Most assemblers impose a limit on the length of a label - the DREAM assembler, for example, insists that labels be no more than 6 characters long.

The table below shows examples of valid and invalid statement labels.

Valid Labels	Invalid Labels
A372	372A (label must start with a letter)
NEXTCH	NEXTCHAR (label too long)
OUT	IN-OUT (label may not contain '-')

There is a single exception to the rule in the DREAM assembler that labels may contain only alphanumeric characters. One label, and one label only, in the program may have a '@' as its first character. For example, @START or @BEGIN are valid labels although both may not be used in the same program. The label whose first character is '@' is one way of indicating to the assembler where to start program execution when the assembled machine code program is run on the Dragon.

Although the labels A, B, X, Y, U, S, CC, PC, and DP are not invalid, you should avoid using them because of potential confusion with the M6809 register names. Similarly, you should not use labels which are identical to assembly language mnemonics.

4.1.2 The mnemonic field

The mnemonic field of an assembler input line must contain one of the instruction mnemonics that we covered in the previous chapter. It must be separated from the label field by at least one space. If no label is present, the mnemonic field must still be preceded by one or more spaces otherwise it will be taken as a label.

4.1.3 The operand field

The operand field in an assembly language instruction must be present except for those instructions like INCA, ABX, MUL, etc. which have no operands. It must be separated from the mnemonic field by at least one space. The operand field specifies the operand address and the following conventions are used when using the DREAM assembler to indicate which addressing mode is being used.

Register addressing

The names of the source and destination registers are separated by a comma. For example:

```
TFR X,Y
EXG A,DP
```

Immediate addressing

The immediate value is preceded by a '#' symbol. By default, immediate values are decimal but hexadecimal values may be input by preceding the value with a '\$' symbol and character values by preceding them with a quote '"' symbol. It is also possible to associate symbolic names with constants and these may also be input as immediate values. For example:

```
LDA #10
LDB #$10
LDA #' +
LDA #MAXINT
LDA #LAB1
```

If the immediate operand in an instruction is a program label, the value substituted for the symbolic label is the address of the labelled statement.

Direct and extended addressing

In general, the assembler will decide for the programmer whether it is best to use direct or extended operand addressing. DREAM works out if the addressed operand is within a page of the current DP register setting and, if so, it generates a direct address. Otherwise, an extended address is generated.

A symbolic name on its own indicates either direct or extended addressing as decided by the assembler. The programmer may force extended addressing by preceding the name with a '>' character or may force direct addressing by preceding it with a '<' character. For example:

```
LDX VNAME      Direct or extended
LDX >TNM      Force extended addressing
LDX <COUNT   Force direct addressing
```

Indexed addressing

The general form of an indexed address is:

```
<offset>,<index register>
```

The offset may be a register name, a symbolic name, a constant value or may be left out altogether. The register name must be X, Y, U or S and, if no offset is present, auto increment or decrement may be specified.

The following examples all show generalised indexed addressing.

```
LDA  BASE,PCR    PC relative
STX  OFFST,Y     Symbolic constant offset
STD  -16,X       Constant offset
LDA  ,X          Zero offset
LDB  A,U         Register offset
```

Auto increment and decrement may only be used with zero offset addressing. They are indicated by prefixing the index register name with '-' or '--' or by suffixing it with '+' or '++'. For example:

```
STX  ,Y++       Auto increment by 2
STA  ,S+        Auto increment by 1
LDB  ,-Y        Auto decrement by 1
LDD  ,--S       Auto decrement by 2
```

In all cases, the assembler works out whether the specified offset should be represented as a 5-bit, an 8-bit or a 16-bit offset. The programmer may force an 8-bit offset by preceding the offset with a '<' character and may force a 16-bit offset by using a '>' symbol. It is not possible to force the assembler to generate a 5-bit offset. For example:

```
LDD  <4,X       Forces 8-bit rather than 5-bit offset.
STX  >32,Y      Forces 16-bit rather than 8-bit offset
```

Indirect addressing

Indirect addressing is indicated by surrounding the operand field with round brackets. For example:

```
LDA  (VALADD)   Indirect extended
STX  (A,Y)      Indirect indexed
```

When using constant values within the operand field, the DREAM assembler allows a limited form of arithmetic to be used. If constant expressions using '+' and '-' are specified, DREAM will carry out the necessary arithmetic as it assembles the program. For example:

```
LEAS BASE+8,U
LDD #START - 10
```

```
BRA * + 12
```

An asterisk (*) means the value of the program counter at the start of the current statement. Notice that this is not the same as the actual PC value which refers to the next instruction to be executed.

4.1.4 The comments field

The comments field is used to provide descriptive comment about the associated assembly code instruction. It must be separated by at least one space from the operand field but the convention when using the DREAM assembler is to separate the comments field from the remainder of the instruction by two or more spaces and to make the first symbol a semi-colon. For example:

```
LDA T ; put top value in A
```

Comments taking up an entire line may also be introduced by placing a '*' in column 1. Most M6809 assemblers will recognise this as a comment and ignore the remainder of the line. For example:

```
* An asterisk indicates a comment
```

In order to make assembly language statements as readable as possible, it is best to adopt a fairly rigid, fixed format layout for instructions. The following layout is suggested:

Columns 1-6	Label or blank if statement is unlabelled
Columns 8-11	Mnemonic
Columns 13-19	Operand
Columns 22-	Comment

If the operand is more than eight characters long, it will obviously overflow into the comments field. Depending on the length of the comment, you may either continue it on the same line or start a new line with '*' and include the comment field on that line. In general, when all of a comment cannot fit on the instruction line, the continuation on succeeding lines should be aligned.

Examples of this layout are:

```
BEGIN LDD MAX ; Start with max
      SUBD #1 ; Take 1 off it
      CMPD MINVAL ; Compare with min.
      BEQ VALSEQ
```

As with all layout conventions, there are many special cases which do not fit well with the convention. Slight changes may avoid taking a new line for a short

comment continuation or may make the program more readable. The programmer must use his common sense in this respect and modify the above rules accordingly.

The output from the DREAM assembler consists of a listing of the source lines with each line preceded by the address of the corresponding machine instruction and the hexadecimal representation of the instruction itself. For example, assuming the instruction address was 4E40, this might appear:

```
4E40  4C      INCA          ; increment pointer
4E41  1F8B    TFR A,DP     ; and load DP
```

Notice that the address is incremented according to the number of bytes in the instruction. We shall describe how the initial assembler address is set up in a later section (4.2.5) of this chapter.

4.1.5 Assembling without an assembler

If you don't have an assembler program but want to run machine code programs, you have to translate the assembly language statements to hexadecimal machine code by hand. This is only realistic if you have only a few statements to translate and you only do such translations fairly occasionally.

There is enough information in Chapters 2 and 3 and in the appendices to allow you to translate from assembly language to machine code. You must keep careful track of the number of bytes taken up by each instruction so that your relative addresses are correct. It is best to make a table for yourself of the symbolic names which you use and the memory addresses which you have assigned to them.

Once you have completed the translation from assembly code to machine code, you then load the hexadecimal representations of your machine instructions into memory and start executing them. This can be accomplished using another program called a loader. In the final section of this chapter, we provide a listing of a loader, written in BASIC, which POKES hexadecimal codes into memory. You may either then execute the machine code program with an EXEC command or you may include such a command in the loader so that the machine code is immediately executed.

4.2 ASSEMBLER DIRECTIVES

Assembler directives are instructions used by the programmer to give commands to the assembler. They do not cause machine instructions to be generated but they may alter internal assembler variables. Assembler directives are the means by which symbolic names are associated with addresses and they also allow the programmer to specify the initial values which memory

bytes should take before his program is executed.

The operation of assembler directives can only be understood in the context of the general memory organisation which is assumed by the assembler. Figure 4.1 shows this organisation for that part of memory used by the assembler.

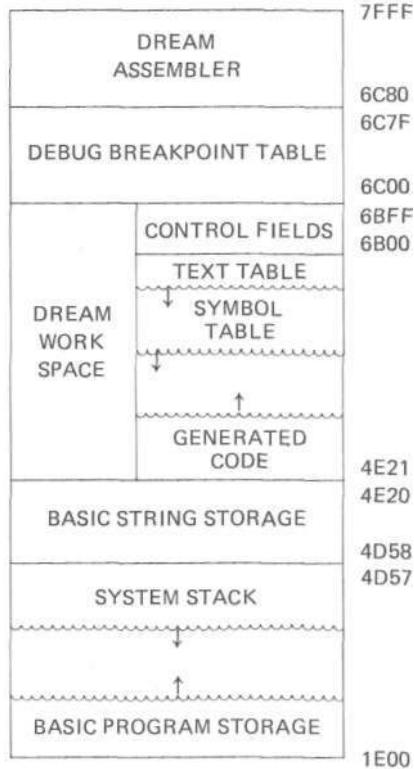


Fig. 4.1 Assembler memory map

There is a large area of RAM which is reserved by the assembler as its work space. This workspace immediately follows the machine code of the assembler program in the Dragon's memory.

At the top of this work space, the assembler creates its own internal tables which it uses in the translation of the programmer's assembly code to machine code. As the number of entries in these tables depends on the size of the program being assembled, the tables are variable in size. As new elements are added to the table, they are allocated lower memory addresses. Dynamically allocated areas of this sort are shown on memory maps as wavy lines with an arrow indicating the direction of growth.

As an illustration of how this table is set up, say the top address in the assembler's work space is 6AFF.

The first table entry, which might be 8 bytes long, is allocated address GAFF. The following table entry has address 6AF8, the one after that 6AF0, and so on. The table grows downwards in memory as each succeeding element is allocated.

As the assembly language program is translated, the generated machine code must be stored somewhere in memory. The area chosen by the assembler for the generated machine code is at the bottom of its work space and the generated machine code grows upwards in memory.

The assembler uses an internal variable called the assembler program counter (APC) to keep track of where the next generated machine instruction is to be placed in its work space. As instructions are generated, APC is incremented by the length of the instruction in bytes. Some assembler directives also affect the value of APC and their effects are discussed along with the description of the directives in question.

4.2.1 The EQU directive

The equate directive is the directive which is used to associate a symbolic name with a constant decimal or hexadecimal value. It has the general form:

```
<label>    EQU    <value>
```

It is good programming practice to make extensive use of equate directives to name constants used in your program. If you chose a name related to the constant's function, this makes the program easier to understand. Furthermore, if you need to change the value of a constant, you merely need to change the equate directive rather than search through your program changing the absolute value every time it is used in an instruction.

Examples of equates defining absolute constant values are:

```
MAXINT    EQU 32767    ; maximum allowed integer
TABSIZ    EQU 100      ; some table size
OFF       EQU $00      ; define a value meaning off
ON        EQU $FF      ; a value meaning on
```

The constant value in the equate directive may include other symbolic constants defined by an equate and may also include the symbols '+' and '-'. The assembler carries out the necessary arithmetic to compute the equated value. For example:

```
TRUE      EQU ON          ; TRUE has value $FF
FALSE     EQU OFF        ; FALSE = $00
UTABSIZ   EQU TABSIZ - 15 ; UTABSIZ = 85
```

As well as being used to associate names with program constants, the EQU directive may also be used to name locations in a memory page when direct addressing is to be used.

Recall that the direct addressing mode uses the DP register to hold the hi-byte of the memory address with the lo-byte of the address obtained from the instruction itself. Not only is this form of addressing space efficient as addresses only take up a single byte, it also means that memory locations can be reserved for variables in a position independent way.

The programmer need not decide the absolute address in memory which is to be allocated to particular variables. Rather, he may set up their addresses as a displacement from the start of a page. Where that page actually resides in memory when the program is executed is governed by the setting of the DP register which may be assigned immediately before execution. We shall say more about position independence in Chapter 6.

The equate directive is used to associate page addresses with symbolic names. For example:

```
DELAY    EQU $00    ; first byte in page
CURPOS   EQU $01    ; CURPOS takes up bytes 1 and 2
INCH     EQU $03    ; byte 3
```

The names used in an equate directive must obey the normal rules for assembler labels. That is, they must start with a letter, contain only alphanumeric characters and may be no more than six characters long.

The equate directive does not affect the assembler's program counter. Names and associated values are stored in an internal assembler table and, when the name is used in a program, its value is substituted for it.

4.2.2 The FCB/FCC directive

The FCB/FCC directive is used to format data bytes. That is, the programmer uses this directive to allocate store and to associate a particular value with each byte of that allocated memory. The general form of this directive is:

```
[<label>]    FCB <value list>
```

The label is optional and must obey the usual rules for assembler labels. If a label is used, its value is deemed to be the address of the allocated data byte. The value list is a list of one or more initial values expressed as decimal numbers, hexadecimal numbers or character constants.

In some assemblers, the directives FCB and FCC have different meanings with FCB used to format single bytes and FCC used to format ASCII character strings. In the

DREAM assembler, however, they are equivalent and are handled in exactly the same way. Therefore, the directive FCB may be replaced by FCC anywhere that it is used.

Examples of FCB directives are:

- * Set up the name of a data area for an error message
- * The first byte holds the length of the message
- * The following characters hold the ASCII characters
- * of the message itself
- *

```
ERR1  FCB 13,/NO INPUT CHAR/
```

*

- * Notice how strings are delimited by the / character

- * Set up a byte with value 1F (hex)

*

```
      FCB $1F
```

- * Set up a 5 byte memory area with bytes initialised

- * to the hex values 8E, 8F, 90,91, and 92

```
TAB1  FCB $8E,$8F,$90,$91,$92
```

The FCB/FCC directive affects the assembler program counter. If APC has the value 5000 say when the FCB labelled ERR1 above is processed, its value after processing is 5000 + 14 (decimal), that is 500D. Note that if a value greater than FF (hex) is used with an FCB directive only the lo-byte of that value is used in the initialisation.

4.2.3 The FDB directive

The FDB directive is similar to the FCB directive. However, rather than formatting single data bytes, it formats 16-bit values taking up 2 bytes (1 word). Its general form is:

```
[<label>]  FDB <value list>
```

Examples of FDB directives are:

```
DIGITS  FDB 1,2,3,4,5,6,7,8,9,0
```

```
MAXVAL  FDB 1024
```

```
INSUB   FDB GETNUM
```

The first two FDB examples above format data words to the specified values. In the third example, the constant filled in and named INSUB may be the value associated with the name GETNUM if GETNUM is defined via an EQU directive. Alternatively, if GETNUM is an instruction label, the location named INSUB is filled in with the address of the labelled instruction.

This facility allows you to create tables of

addresses and then use indirect addressing to access the instructions or data whose addresses are kept in the table. For example:

```
SUBTAB    FDB INCHAR,OUTCH,INWRD,
          OUTWRD,RESET,CLOSE
```

This directive might be used to create a table of subroutine addresses with the subroutine names given on the right hand side of the directive.

Like FCB, FDB affects the assembler program counter, incrementing it by two for every word formatted.

4.2.4 The RMB directive

The RMB directive is used to reserve one or more memory bytes. It does not set them up to any specific value, it merely increments APC by the value specified in the directive. The general form of an RMB directive is:

```
[<label>]  RMB <value>
```

The value may be either a symbolic, hexadecimal or decimal constant. For example:

```
INCHAR    RMB 1      ; reserves a single byte
OUTBUF    RMB 256    ; reserves a 256 byte buffer
```

Typically, RMB is used to reserve space which will subsequently be allocated values in I/O operations.

4.2.5 The ORG directive

The ORG directive is used to assign a value to APC and, hence, sets up the logical origin of the generated machine code which follows that directive. It is not obligatory to include an ORG directive in a program. If there is no ORG directive, the DREAM assembler sets up its program counter to have an initial value equal to the bottom of its work space.

The general form of an ORG directive is:

```
[<label>]  ORG <address>
```

Examples of this directive are:

```
          ORG $5000    ; APC = 5000 (hex)
NEWSEG   ORG * + 128  ; * means current value of APC
*        This directive is equivalent
*        to RMB 128
```

All the examples in this book have been tested with a code origin at memory address 4E21 (20001 decimal). This is set up with an ORG \$4E21 statement as shown in the example in section 4.3 below.

4.2.6 The PUT directive

The PUT directive is used to tell the assembler where, in RAM, the generated object code should be placed. It is a means of overriding the assembler's normal placing of generated code at successive addresses starting at address 4E21 which is the bottom of its work space.

The general form of a PUT directive is:

```
PUT <address>
```

Normally, a PUT directive is preceded by an ORG directive to set the APC to the address where the generated code is to be placed. This is not obligatory, however, if you are going to move the code before executing it or if the code is completely position independent.

4.2.7 The SETDP directive

The SETDP directive is used to tell the assembler the current value of the direct page register DP. Remember that the assembler decides whether to use direct or extended addressing when a symbolic name is used in the address field of an instruction. To make this decision, it must know the value of DP at that point and SETDP is used to provide that information. The general form of the directive is:

```
SETDP <hex value>
```

The operand must be a hexadecimal value in the range 00 to FF. The SETDP directive only provides information to the assembler; it does not cause instructions to be generated to assign a value to the direct page register. It is the programmer's responsibility to ensure that the actual run time value of DP is consistent with the value used in a SETDP directive.

4.3 EXAMPLE PROGRAMS

In this section we present two complete, working programs which the user may type into his machine and execute. The first of these programs is a loader program, written in BASIC, which allows the user to POKE machine code into particular locations in the Dragon's memory. This code may then be executed.

The other example program is presented in both BASIC and assembler. This is a simple program designed to illustrate just how much faster machine code programs can be. The program fills the display screen with every character, one after the other. When the BASIC version of the program executes, you will see that this operation takes about 2 seconds per screenful. The assembly language version fills the screen with each character in a fraction of a second. The machine code

for the assembly language version of the screen filler is included, in hexadecimal, as the DATA statements in the BASIC loader.

Both of these examples are commented and should need no further explanation.

```

10 ' Machine code loader
11 ' Machine codes in hex are poked into memory
12 ' locations starting at 20001 then execed
20 READ LA ' LA = load address (start of program)
30 READ EA ' EA = address of first instruction
40 PA = EA 'to be executed
50 READ HB$ ' Hex constants
60 IF HB$="END" THEN 100
70 POKE PA,VAL("&H"+HB$) ' Poke value into memory
80 PA = PA + 1 ' Increment address
90 GOTO 50
100 PRINT "MACHINE CODE LOADED"
110 PRINT "LOAD ADDRESS IS ";LA;"(DEC)";
111 PRINT HEX$(LA);"(HEX)"
120 PRINT "END ADDRESS IS "; PA-1;"(DEC)";
121 PRINT HEX$(PA-1);"(HEX)"
130 PRINT "EXEC ADDRESS IS";EA;"(DEC)";
131 PRINT HEX$(EA);"(HEX)"
140 PRINT "YOU ARE ADVISED TO SAVE LOADER "
141 PRINT "BEFORE RUNNING M/C CODE"
142 'If you want to execute the loaded code
143 'immediately, you should put an
144 'EXEC EA statement here. If you do this
145 'for this program, you lose BASIC print
146 'information
150 DATA 20001 'Load address here
160 DATA 20001 'Execute address here
165 ' You put your own machine code in hex
166 ' here to load your hand translated
167 ' programs
170 DATA 34,12 ' Machine code for the
180 DATA 86,00 ' Screen filler program
190 DATA 8E,04,00 ' given below
200 DATA A7,80
210 DATA 8C,06,00
220 DATA 25,F9
230 DATA 4C
240 DATA 81,80
250 DATA 25,F1
260 DATA 35,92
270 DATA END

```

Program 4.1 BASIC machine code loader

```

10 ' Fills screen with characters with codes
20 ' 0 to 127 in turn
30 FOR CH = 0 TO 127
35 ' Screen RAM addresses are from &H400-&H5FF

```

```

40 FOR SC = &H400 TO &H5FF
50 POKE SC,CH
60 NEXT SC
70 NEXT CH

```

Program 4.2 BASIC screen filler

* SCRFL - fill screen with characters

* Register inputs NONE

```

                ORG $4E21
SCRFL  PSHS A,X      ; Save registers
        LDA #0       ; First character
NXTSC  LDX #$400    ; Screen base address
PRCH   STA ,X+      ; Store character
        CMPX #$600   ; At end of screen?
        BLO PRCH    ; No, next character
        INCA        ; Go on to next character
        CMPA #128
        BLO NXTSC   ; Do another screenful
        PULS A,X,PC ; Restore and return

```

Program 4.3 Assembly language screen filler

Chapter 5

From BASIC to assembly code

In this chapter we describe the assembly language equivalents of the most commonly used BASIC statements. As well as the literal translations of BASIC to assembly language, we show how these constructs can often be implemented in a more efficient way by removing some of the redundancy inherent in BASIC.

The assembly language programmer must obviously know the mnemonics for the M6809, the register names and the symbolism for the M6809 addressing modes. It may seem a daunting task to memorise all this information, although it is less so than memorising about 1400 machine instructions! However, the consistent and orthogonal nature of the M6809's instruction set makes the task less difficult than might at first be supposed and, after a little practice, the programmer will easily remember all the mnemonics which he needs.

The basic building blocks of programs are assignment statements, conditional statements, loops and statements for input and output of data. We describe, in some detail, how each of these may be implemented in assembly language. We also cover the declaration and calling of BASIC-like subroutines and the representation and manipulation of arrays. The notation which we use is similar to that used in previous chapters. However, if a symbolic name is used for a memory location, we use it in comments here as if it was a BASIC name - we do not precede it with MEM.

5.1 ASSIGNMENT STATEMENTS

Assignment statements in BASIC are used to assign a constant, the result of an arithmetic expression or the value of a memory location to some other memory location. For ease of reference, we may give symbolic names to the memory locations involved although, if the memory access routines PEEK and POKE are used, we actually signify the absolute memory locations to be accessed. We describe PEEK and POKE later and concentrate here on assignments which have the general form:

```
<name> = <expression>
```

The <name> on the left side of the = sign may be either a variable name or may be a reference to an element of an array. The <expression> on the right side of the equals sign may be a constant, a variable name, an array element reference or an arithmetic expression consisting of two or more operands separated by arithmetic operators such as + and *.

Reference to array elements will be dealt with later so, in this section, we only describe assignments where numeric constants and variables are used. We shall make the further simplification that these constants and variables may only take 8-bit or 16-bit integral values represented as unsigned numbers or in two's complement notation.

This is not too great a limitation as many practical applications of computers don't need real numbers. The provision of real number arithmetic in most microcomputers is made using software routines which manipulate pairs of 16-bit quantities representing the real number. This is a fairly complex process, and if the reader is interested in how it's done he should refer to one of the computer science textbooks suggested in the reading list.

In general, assignment statements on the M6809 are implemented using the accumulator registers A, B and their catenation D when 16-bit numbers are involved. Although it is possible to make use of the index registers X, Y, S, and U, these are usually reserved for the storage of addresses.

The basic outline of an assignment statement in assembly language is:

```
Evaluate RH expression into an accumulator register.
Store accumulator in memory.
```

For example, the assembly language equivalent of the simple BASIC statement `M = 7` is:

```
LDA #7    ; A = 7
STA M     ; M = A
```

Notice how immediate addressing is used to specify that a constant value is to be loaded into a register. A very common mistake made by novice assembly language programmers is to forget the # symbol indicating immediate addressing.

```
LDA 7     ; A = PEEK(7)
STA M     ; M = A
```

The BASIC code documenting the assembly language instructions shows how this gives a completely different result. Rather than a constant value 7 being loaded into A, the contents of memory byte

7 (which may be any value between -128 and 127) are loaded into the A register.

If a constant value between -128 and 127 is being assigned, we may use either the A or the B register as the accumulator. If the value lies outside this range, we must use the D register for the assignment. For example, the BASIC statement $T = -3842$ has the assembly language equivalent:

```
LDD #-3842 ; D = -3842
STD T      ; T = D
```

As D is a 16-bit register, the STD operation results in information being stored in two consecutive memory bytes. If the address of T is 4E22, say, the assignment results in the hi-byte of D being assigned to 4E22 and the lo-byte being assigned to 4E23.

Assignments of the form $M = N$ are implemented in assembly language in a comparable way:

```
LDA N      ; A = N
STA M      ; M = A
```

If the operands in the assignment $T = R$ are 16-bit quantities, the D register must be used:

```
LDD R      ; D = R
STD T      ; T = D
```

When the right side of the assignment is an arithmetic expression consisting, in general terms, of constants, variables and arithmetic operators, the assembly language programmer must arrange the evaluation of this expression in an accumulator. The evaluated value is then stored. For example, the assignment statement $M = N + P$ has the assembly language equivalent:

```
LDA N      ; A = N
ADDA P     ; A = A + P
STA M      ; M = A
```

Notice that we are ignoring the possibility of overflow and carry here. In some arithmetic evaluations, this must be taken into account but, as we are simply illustrating concepts, we will not introduce this unnecessary complication.

If the assignment uses a mixture of 8-bit and 16-bit values, the D register must be used and, in some cases, 8-bit values will automatically be extended to 16 bits. For example, assuming T and R are 16-bit variables, the assignment $R = T - 10$ may be implemented as follows:

```
LDD T      ; D = T
SUBD 10    ; D = D - 10
STD R      ; R = D
```

A 16-bit subtraction is automatically carried out in this case. However, if mixed 8-bit and 16-bit variables rather than constants are used in arithmetic expressions, the programmer must be careful not to use a D register operation on an 8-bit variable. If such an operation is specified, the addressed variable and the following memory byte (which is not wanted) will be used in the operation.

For example, say T and R are 16-bit signed quantities and M is an 8-bit signed quantity. A careless assembly language programmer might translate the assignment $T = M + R$ as follows:

```
LDD M      ; D.hi = MEM(M): D.lo = MEM(M + 1)
ADDD R     ; D = D + R
STD T      ; T = D
```

A completely incorrect value for the addition will result because of the LDD operation which does not load the 8-bit value of M into D.

A correct assembly code sequence for this mixed-length arithmetic takes into account the fact that the lo-byte of D is the B register. The sign extend instruction is also used to make sure that the signs of the 16-bit and the 8-bit values are the same.

```
LDB M      ; B = M
SEX        ; Extend sign bit of B to A
ADDD R     ; D = D + R
STD T      ; T = D
```

This mixed-length arithmetic becomes more complex when a subtraction is involved and the order in which operands are loaded into D is significant. Assuming T, R, and M have the same values as before, the assignment $T = R - M$ cannot be implemented using the same sequence as above because the SUBD instruction has no facilities for sign extension.

There are various different ways of implementing this type of assignment in assembly language. The simplest is to convert the 8-bit value to a 16-bit value, store it in some temporary location and then perform the subtraction using 16-bit operations only. For example:

```
LDB M      ; B = M
SEX        ; D = B (propagate sign)
STD      ,--S ; Store M on hardware stack
*
*           Auto decrement S so that it points to
*           free location on stack
LDD R      ; D = R
SUBD      ,S++ ; D = R - M
*
*           Note how auto increment used to reset
*           stack pointer
STD T      ; T = D
```

There are no problems in implementing addition and subtraction operations in assembly language but generalised multiplication and division have no corresponding machine instructions. These operations must be implemented by calling machine language routines and it is beyond the scope of this section to explain how these routines may be programmed.

However, multiplication and division of 8-bit unsigned quantities by numbers which are powers of 2 may be implemented very simply by using the arithmetic shift instructions ASR and ASL. Shifting a number left n times is equivalent to multiplying it by 2^n and shifting it right n times is equivalent to dividing that number by 2^n . Naturally, the division is an integer division operation with the remainder discarded.

For example, if I and J are unsigned 8-bit integers, the assignment $I = J * 4$ might be implemented in assembly language as follows:

```
LDA J      ; A = J
ASLA      ; A = A * 2
ASLA      ; A = A * 2
STA I      ; I = A
```

Similarly, $J = I/8$ might be implemented:

```
LDA I      ; A = I
ASRA      ; A = A/2
ASRA      ; A = A/2
ASRA      ; A = A/2
STA J      ; J = A
```

Using shifts to multiply and divide signed quantities is more complex because of the need to ensure that the sign of the result is correct. We leave it as an exercise to the reader to work out how to implement signed multiplication and division by powers of 2.

The PEEK and POKE functions

The BASIC functions PEEK and POKE allow direct reference to individual memory bytes. Whereas PEEK is always used as the right hand side of a normal BASIC assignment, POKE is a specialised kind of assignment. Therefore, $T = \text{PEEK}(\&H0406)$ assigns the byte value at memory address 0406 (hex) to T and $\text{POKE ASC}("",)\&H0500$ assigns the code for ' $'$ ' to the byte in memory at address 0500.

PEEK and POKE are very easily implemented in assembly language using load and store instructions. The assembly language equivalent of the above PEEK instruction is:

```
LDA $0406 ; A = MEM(0406)
STA T     ; T = A
```

The POKE operation has the equivalent assembly code:

```
LDA #'*      ; A = '*'
STA $0500    ; T = A
```

The only difference between straightforward assignments and PEEK and POKE is that, rather than symbolic addresses, absolute memory addresses are used.

5.2 CONDITIONAL CONSTRUCTS

Conditional constructs are fundamental program building blocks which allow other statements to be selected for execution depending on the truth of some condition. In BASIC, conditional execution of statements or groups of statements is implemented using IF-THEN statements in combination with GOTO statements.

More generally, conditional constructs can be partitioned into three classes:

- (1) Single armed conditionals
These may be expressed:

```
if <condition> then <action>
```

If the specified condition is true, the <action> is executed otherwise it is skipped.

- (2) Two armed conditionals
These have the form:

```
if <condition> then <action1> else <action2>
```

If the given condition is true, <action1> is executed and <action2> is skipped. If the condition is false, <action1> is skipped and <action2> is executed.

- (3) Multi-armed conditionals
These are really conjunctions of single armed conditionals:

```
if
  <condition1> then <action1>
  <condition2> then <action2>
  <condition3> then <action3>

  <conditionN> then <actionN>
```

The conditions are evaluated in turn. If the evaluated condition is false, the associated action is skipped and the following condition is evaluated. If the condition is true, the associated action is executed and the remainder of the condition/action pairs are skipped. In BASIC,

multi-armed conditionals are usually implemented as a sequence of IF-THEN statements.

We shall consider each of these in turn and show how they may be implemented in assembly language. The approach which we use is to show first how the conditional is implemented in BASIC. We then describe how this may be literally translated into assembly language and finally optimised to remove redundancy.

5.2.1 Single armed conditionals

In BASIC, single armed conditionals are expressed as an IF-THEN statement if only a single statement is to be conditionally executed. If a number of statements are to be executed if the condition is true, a goto is used to skip over these statements if the given condition is false rather than true.

For example, if we want to swap the values of I and J if J is greater than I, we might write the following code:

```
100 ' Swap if J > I. So skip if J <= I
110 IF J <= I THEN 200
120 T = J
130 J = I
140 I = T
200...
```

In assembly language programming, we use exactly the same technique of reversing the sense of the comparison and skipping if this (reversed) condition is true. The outline for this is:

```
Make comparison setting CC bits
Branch if NOT desired condition to L

Code to be executed if original condition true

L  ....
```

Assuming that I and J are unsigned 8-bit values, the above BASIC sequence may be translated to assembly language as follows:

```
LDB J      ; B = J
CMPB I     ; compare B with I
BLS L200   ; if J <= I goto L200
LDB J      ; B = J
STB T      ; T = B
LDB I      ; B = I
STB J      ; J = B
LDB T      ; B = T
STB I      ; I = B
```

L200

This literal translation may be optimised by noting that the first instruction loads the value of J into B and the same instruction is repeated after the comparison. As comparison does not affect register values, the second load is unnecessary. Furthermore, we are obliged in BASIC to use an intermediate variable T in the swap sequence but in assembly code this is unnecessary. We may simply use another register. An optimised version of the swap sequence is:

```
LDA J      ; A = J
CMPA I     ; compare A with I
BLS L200   ; if A <= J then goto L200
LDB I      ; B = I
STB J      ; J = B, ie J takes original value of I
STA I      ; I = A, ie original value of J
```

Simple BASIC IF-THEN statements of the form IF P = Q THEN P = P + 1 may be directly translated to assembly language as follows:

```
LDA P
CMPA Q     ; Compare A and Q
BNE L1     ; if A <> Q then goto L1
LDA P      ; A = P
ADDA #1    ; A = A + 1
STA P      ; P = A
```

Again, this may be optimised by using the fact that P is loaded into a register to evaluate the condition and then immediately reloaded after this comparison. This second load can be eliminated. We may also use the INC instruction to add 1 to a value rather than the add instruction. The advantage of this is that INC occupies less space and executes more quickly than ADD.

An optimised form of the above sequence is:

```
LDA P      ; A = P
CMPA Q     ; Compare A and Q
BNE L1     ; if A <> Q then goto L1
INCA       ; A = A + 1
STA P      ; P = A
L1        ....
```

In fact, we can reduce the number of instructions still further by using the ability of INC to operate on a memory location:

```
LDA P
CMPA Q
BNE L1
INC P
L1        ....
```

Assuming direct addressing of both P and Q, the 4 instruction sequence takes up 8 memory bytes, the 5 instruction sequence occupies 9 memory bytes and the literal translation of the BASIC code takes up 12 memory bytes.

5.2.2 Two armed conditionals

Two armed conditionals are implemented in BASIC by using a combination of IF-THEN statements and GOTO statements. For example, the condition if <condition> then <action1> else <action2> is written:

```

100 IF <condition> THEN 200
110 <action2>
120 GOTO 300
200 <action1>
300 ...

```

Notice how we reverse the order of the actions and skip over the second action if the condition is true. Exactly the same outline structure is used when implementing two armed conditionals in assembly language.

```

Evaluate condition
Branch if true to L1
Action2
Branch unconditionally to L2
L1 Action1
L2 ....

```

For example, if we wish to assign the higher of two numbers to some other variable, we might write in BASIC:

```

100 IF P > Q THEN 200
110 ' P <= Q here
120 MAX = Q
130 GOTO 300
200 MAX = P
300 ...

```

Given that P, Q and MAX are unsigned values, direct translation of this BASIC sequence to assembly language gives:

```

LDA P ; A = P
CMPA Q ; Compare A and Q
BHI L200 ; if A > Q then goto L200
LDA Q ; A = Q
STA MAX ; MAX = A
BRA L300 ; goto L300
L200 LDA P ; A = P
STA MAX ; MAX = A
L300 _____

```

Again, optimisation of this sequence is possible. The statement labelled L200 is a redundant load as A already contains the value of P at that point. Furthermore, both actions end with an identical store operation so it may be factored out and executed after one or the other action is complete. An optimised version is:

```

        LDA P
        CMPA Q
        BHI L200      ;if P > Q goto L200
        LDA Q         ; A = Q
L200    STA MAX       ; MAX = A

```

A sequence of 8 assembly language instructions has been optimised to 5 instructions which do exactly the same thing. We must emphasise however that it is not good programming practice to try to write optimised code directly. This is an error-prone process because the programmer is liable to become caught up in optimisation details and to lose track of the correct solution. With high-level language code to serve as a master solution, the introduction of errors through optimisation is much less likely.

5.2.3 Multi-armed conditionals

Multi-armed conditionals are conditional statements where several conditions are evaluated and the action following the true condition is executed. Readers familiar with Pascal will recognise the case statement as a form of multi-armed conditional but in BASIC it must be implemented as a sequence of IF-THEN statements. For example:

```

10  IF T = 7 THEN AGE = BAND1
20  IF T = 9 THEN AGE = BAND2
30  IF T = 14 THEN AGE = BAND3
40  IF T = 15 THEN AGE = BAND4
50  ....

```

Of course, this may be translated into assembly code as a sequence of IF-THEN statements as described above. However, multi-armed conditionals often use the same value in all tests and often have similar actions with different values being assigned to the same variable in each action.

The following structure shows how multi-armed conditionals can often be implemented.

```

Load test variable
if NOT(test1) goto T2
Load value to be assigned
goto STORE
T2  if NOT(test2) then goto T3
Load T2 value

```

```
goto STORE
```

```
STORE Store value to be assigned
```

The above sequence of IF-THEN statements may be coded in assembly language:

```

        LDA T           ; Load variable to be tested
        CMPA #7         ; First test, compare A with 7
        BNE L1         ; if A <> 7 then goto L1
        LDB BAND1      ; variable to be assigned into B
        BRA L4         ; Jump to store
L1      CMPA #9         ; Second test, compare A and 9
        BNE L2         ; if not equal, go on to next test
        LDB BAND2
        BRA L4
L2      CMPA #14        ; if A <> 14 then goto L3
        BNE L3
        LDB BAND3
        BRA L4
L3      CMPA #15        ; last test
        BNE L5         ; do nothing if not equal
        LDB BAND4
L4      STB AGE         ; assign to AGE
L5      ....

```

Compound conditional expressions

So far, we have looked at conditional statements where the condition involved is a simple condition of the form `<operand> <conditional operator> <operand>`. However, compound conditional statements using ANDs and ORs to connect conditions are also frequently used. These have the general form:

```
<simple condition> <logical operator> <condition>
```

where permitted logical operators in BASIC are AND and OR.

In BASIC, therefore, the following are all valid conditional expressions:

```

P = Q AND T >= R
J > I AND J < K
J > I OR K = L
K = J AND (P > Q OR T >= R)

```

When such conditions are implemented in assembly language we may write them so that it is often only necessary to test a single condition rather than the conditions on each side of the AND or OR operator. This is possible because we know that both conditions must be true for an AND operation to be true and that both conditions must be false for an OR operation to be false.

Therefore, if we test the first condition in an AND operation and find it false there is no need to test the second condition. Similarly, if we test the first condition in an OR operation and find it true, the entire expression must be true. The second condition need not be tested. For AND operations, the outline structure of an assembly language program is:

```

Test left hand condition
  If false goto L1
Test right hand condition
  If false goto L1
Actions if condition is true
L1 ...

```

For OR conditional operators, the outline is similar:

```

Test left hand condition
  If true goto L1
Test right hand condition
  If false goto L2
L1 actions if condition is true
L2 ....

```

We illustrate this by showing how BASIC IF-statements with compound conditions may be expressed in assembly code. Again, assume that all variables are unsigned 8-bit quantities.

```
IF P = Q AND T >= R THEN M = N
```

The assembly language equivalent of this is:

```

      LDA P      ; A = P
      CMPA Q
      BNE OUT    ; if P <> Q skip second condition
      LDA T      ; A = T
      CMPA R
      BLO OUT    ; if T < R skip action
      LDA M
      STA M
OUT   ....

```

Notice how only a single test is necessary if P is not equal to Q.

```
IF (P > Q OR T >= R) AND K = J THEN M = N
```

To implement this in assembly language we re-order it to test first if K = J. If this is false, there is no need to carry out any more tests.

```

      LDA K      ; A = K
      CMPA J

```

```

        BNE OUT    ; If A <> K do no more
        LDA P
        CMPA Q
        BHI OK    ; OR condition is true, skip to action
        LDA T
        CMPA R
        BLO OUT   ; Skip over action
OK      LDA N
        STA M
OUT     _____

```

5.3 LOOP CONSTRUCTS

Loop constructs are those programming constructs which allow the programmer to specify that a group of statements is to be executed a number of times. They take three fundamental forms:

- (1) For loops
These execute the loop a specified number of times. A loop counter variable is used and the loop terminates when this variable reaches a specified value.
- (2) While loops
These execute the statements in the loop while some condition remains true. Loop execution stops as soon as this condition becomes false.
- (3) Repeat loops
Repeat loops cause the loop to be executed until some condition becomes true. The important distinction between repeat loops and while loops is that the test for loop termination comes at the end of a repeat loop whereas it comes at the beginning of a while loop. Repeat loops, therefore, always execute at least once.

BASIC provides facilities which allow each of these looping constructs to be expressed. For loops are constructed using FOR and NEXT statements and both while and repeat loops are built from combinations of IF-THEN and GOTO statements.

We shall now look at each of these loop constructs in turn and see how they may be expressed in assembly language.

5.3.1 For loops

For loops are loops which execute a given number of times. They have a controlling for-loop variable which is incremented or decremented by one or by some programmer specified value until it reaches a terminating value. For example, consider the following BASIC program which sums the integers between 1 and N, where N is some positive number.

```

100 TOT = 0
110 FOR I = 1 TO N
120 TOT = TOT + I
130 NEXT I

```

On completion of this program fragment, the value of TOT will be the desired sum. To see how this might be expressed in assembly code it is best to consider it in primitive terms using only IF-THEN and GOTO statements to implement looping.

```

100 TOT = 0
110 I = 1
120 IF I > N THEN GOTO 160
130 TOT = TOT + I
140 I = I+1
150 GOTO 120
160 _____

```

Now we have reduced the loop to conditionals and gotos which we know how to express in assembly language:

```

        CLR TOT      ; TOT = 0
        LDA #1       ; A = 1
        STA I
LOOP    LDA I        ; A = I
        CMPA N
        BHI OUTLP   ; IF I > N stop looping
        LDA TOT
        ADDA I
        STA TOT
        INC I       ; Notice use of INC rather than ADD
        BRA LOOP
OUTLP   ....

```

In this example we have implemented the statement $I=I+1$ as `INC I` which appears to be a sensible optimisation. However, if we look at the body of the loop we see that `I` is not actually modified in the loop body so we can keep the loop counter in a register for the duration of the loop.

```

        CLR TOT      ; TOT = 0
        LDB #1       ; B = 1, loop counter
LOOP    CMPB N
        BHI OUTLP   ; if B > N then skip
        TFR B,A     ; A = B
        ADDA TOT    ; A = A + TOT
        STA TOT     ; TOT = A
        INCB       ; B = B + 1
        BRA LOOP
OUTLP   ....

```

The above code shows how the for loop may be implemented when TOT is an 8-bit value. If TOT is a

16-bit value, an alternative strategy may be adopted. Because of the existence of the register add instruction ABX, the B register may be used to hold the loop counter and the X register the sub-total as the loop is executed.

```

                LDX #0          ; X = 0, initial total
                LDB #1          ; Loop counter
LOOP           CMPB N
                BHI OUTLP      ; If B > N then goto OUTLP
                ABX             ; X = X + B
                INCB           ; B = B + 1
                BRA LOOP       ; goto LOOP
OUTLP          STX TOT         ; TOT = X

```

You can see from these examples that there is no single 'best' way of implementing for loops in assembly language. Rather, if optimal code is required, the programmer must look at the statements within the loop and code his loop with how they interact with the loop counter.

As a final example in this section, we show how a FOR-NEXT loop using a negative step might be implemented in assembly code. This example is also our first introduction to arrays. The program fragment assigns those numbers between 100 and 50 which are divisible by 8 to adjacent array elements. Therefore, the first element holds 96, the second 88, the third 80 and so on. In BASIC, this may be written as follows:

```

100  I=0
110  FOR J = 100 TO 50 STEP -2
115  RM = J - (INT(J/8) * 8)
120  IF RM <> 0 THEN 150
130  ARR(I) = J
140  I = I + 1
150  NEXT J

```

A completely literal translation of this program is not possible because there is no direct equivalent in assembly language to the divide operator. However, the calculation of the remainder may be simulated by using the fact that a binary number which is divisible by 8 always has its 3 least significant bits (bits 0-2) equal to 000. If the bit pattern 00000111 is anded with a number and the result is zero then bits 0-2 of that number must be 000 and the number is divisible by 8.

In the assembly language example below, the array ARR is accessed by placing the address of its first element in register X. Indexed addressing is then used to access this and succeeding elements.

```

                CLR I          ; I = 0, not 1 as assembly language
*              array indexes always start at 0

```

```

        LDA #100
        STA J          ; J = 100
LOOP    LDA J
        CMPA #50
        BLO OUTLP     ; if J < 50 then goto OUTLP
        ANDA #$F8     ; AND with bit pattern 11111000
        CMPA J        ; Compare anded value with original
        BNE L150      ; if not divisible by 8 goto L150
        LDB I
        STA B,X       ; Register B holds array index
        INCB
        STB I         ; I = I + 1
L150    LDA J         ; next J
        SUBA 2
        STA J         ; J = J - 2
        BRA LOOP     ; Back to LOOP

OUTLP   _____

```

This code may be optimised by making use of registers to hold the value of the loop counter J and the array index I. We leave this optimisation as an exercise for the reader.

5.3.2 While loops

While loops are loops which execute while some condition is true. When this condition becomes false, execution of the loop terminates. In BASIC, while loops are implemented using IF-THEN and GOTO statements.

For example, consider the following while loop:

```

count = 0
while m > n do
    m = m - n
    count = count + 1
end while

```

In BASIC, this loop might be written:

```

100 COUNT = 0
110 IF M <= N THEN 150
120 M = M - N
130 COUNT = COUNT + 1
140 GOTO 110
150 _____

```

It is a straightforward task to translate this to assembly language using the techniques which we have already described for converting IF-THEN and GOTO statements to assembly code:

```

        CLR COUNT    ; COUNT = 0
WLOOP   LDA M
        CMPA N
        BLS OUTLP    ; If M <= N goto OUTLP

```

```

LDA M
SUBA N
STA M      ; M = M - N
INC COUNT  ; COUNT = COUNT + 1
BRA WLOOP

```

OUTLP _____

As usual, the direct translation of BASIC to assembly code may be optimised by removing redundancies and making more effective use of the processor registers.

```

                CLRB      ; Use B to hold COUNT
                LDA M      ; A = M
WLOOP          CMPA N
                BLS OUTLP  ; If M <= N goto OUTLP
                SUBA N     ; M = M - N.
*              Don't store back into M
*              as value is needed
                INCB      ; COUNT = COUNT + 1
                BRA WLOOP
OUTLP          STB COUNT   ; COUNT = B
                STA M     ; M = A
                * * *

```

Although there are exactly the same number of instructions in this optimised sequence, the number of instructions executed within the loop has been reduced from 8 to 5. As these are the instructions which are each executed several times (once for each loop execution), this reduction means that the optimised program will run more quickly than its unoptimised equivalent.

5.3.3 Repeat loops

Repeat loops and while loops are similar. The most important difference is that the test for loop termination in a repeat loop comes at the end of the loop whereas in a while loop the termination test is placed at the start of the loop. The result of this is that repeat loops always execute at least once whereas, if the while test is initially false, the while loop will not execute at all. Again, the BASIC programmer uses IF-THEN and GOTO statements to implement repeat loops.

For example, consider the following repeat loop:

```

repeat
    m = m + t
    p = p + m
until p >= n

```

In BASIC, this might be written:

```

100 M = M + T
110 P = P + M
120 IF P < N THEN 100

```

Translating this BASIC program to assembly language results in the following program fragment:

```

RLOOP   LDA M
        ADDA T
        STA M           ; M = M + T
        LDA P
        ADDA M
        STA P           ; P = P + M
        LDA P
        CMPA N
        BLO RLOOP      ; If P < N goto RLOOP

```

We leave the optimisation of this assembly code sequence as an exercise for the reader.

5.4 GOTO STATEMENTS

Although you may never have considered them as such, the only function of BASIC GOTO statements is to provide a means for the programmer to implement conditional statements and loop statements. You will have surmised by now that the equivalent, in assembly code, to BASIC'S GOTO statement is the unconditional branch instruction BRA <label>.

There is also an alternative form of the BASIC GOTO in assembly language and that is the unconditional jump instruction JMP. Executing a JMP instruction causes the program counter to be set to the value of JMP's operand. Unlike the BRA instruction where the operand is added to or subtracted from PC, JMP's operand is not a relative but is an absolute value.

In general, you will probably find that you use BRA more often than JMP as it is part of the fundamental mechanism involved in the implementation of loops and conditional statements.

5.5 INPUT AND OUTPUT

One of the most significant advantages of programming in a language like BASIC, rather than in assembly language, is the fact that BASIC provides easy-to-use statements for the input and output of program data. Generalised input/output programming is very complex; indeed, we devote the whole of Chapter 8 to this topic, and the BASIC system hides much of this complexity from the programmer.

In BASIC, we may say INPUT N in order to read a number from the keyboard into variable N. Similarly,

PRINT N prints the contents of the variable N on the display screen. When you think of it however, you don't really type the binary form of a number on the keyboard nor do you get the binary pattern representing the number printed on the screen. Rather, you type characters, which happen to be the digits making up the number required, and you read characters on the screen.

The BASIC system contains routines which convert character sequences, say '5' and '8', to the binary representation of 58. Similarly, when printing a number say -326, the PRINT routine converts the binary pattern representing -326 to the characters '-', '3', '2', '6'.

The assembly language programmer does not have ready access to these BASIC conversion routines so must always deal with input and output in terms of characters rather than numbers. If conversion to and from numbers is required, you must write your own conversion routines for this task. Some of these routines are provided as part of the machine code monitor program given in the final section of this chapter.

As I/O programming is described in general in Chapter 8, we only describe very basic facilities here which allow you to input characters from the keyboard and output characters to the screen. These operations are carried out by calling subroutines which are an inherent part of the Dragon's input/output system.

We call the routine which is used to input characters from the keyboard INCH. The details of how this routine works are not important, all the user must know is how to call this routine and the results of the routine call. When INCH is called, it interacts with the keyboard controller and returns an 8-bit value in the A accumulator. This value is either zero, which means that no key has been pressed, or is a code representing the input character.

The key code returned by INCH is, in most cases, the ASCII value of the character typed by the user. The exceptions to this, when another value is returned in A, are shown in the table below.

Character	Hex Code
Up arrow	5E
Shift up arrow	5F
Down arrow	0A
Shift down arrow	5B
Shift @	13
BREAK	03
Shift BREAK	03
Left arrow	08
Shift left arrow	15
Right arrow	09

Shift right		5D
ENTER	arrow	0D
Shift ENTER		0D
CLEAR		0C
Shift CLEAR		5C

A jump to the starting address of the INCH routine is always stored in memory at address 8006. The INCH subroutine can therefore be called directly either by using this address as the instruction operand or by equating the name INCH with the address and using INCH in the operand field of the instruction.

We call the routine using the jump subroutine instruction JSR which pushes the value of PC onto the S-stack and jumps to the called routine. On termination, the called subroutine restores the value of PC. Therefore, a character may be input as follows:

```
JSR INCH
```

However, when you actually look for a character using INCH there is no guarantee that a key has been pressed. INCH returns 0 in A if no key is pressed and also sets up the condition code register flags. Remember, the Z bit in CC indicates whether the result of the previous operation was zero or not so, if CC.Z is set, this means that A = 0. The following short loop continually calls INCH until a character is actually input.

```
GETCH JSR INCH ; Look for a character
      BEQ GETCH ; if none input, keep looking
```

The routine INCH does not destroy any register contents apart, obviously, from A and CC. If the value of CC is precious and must be preserved, it must be saved before calling INCH and restored after the return from the subroutine. For example:

```
      PSHS CC ; Save CC on S-stack
GETCH JSR INCH ; get a character
      BEQ GETCH
      PULS CC ; Restore CC
```

Normally, it is not necessary to save and restore CC as it should not be used to hold permanent information.

INCH's complement, a character output routine, is accessed via address 800C and the name OUTCH may be equated with this address. As well as actually printing the character on the screen, OUTCH also moves the cursor one space when a character is printed and handles the control characters 'Backspace', 'Return', etc.

To output a character, that character should be placed in the A register and OUTCH called. The value

of the condition code register is lost when OUTCH is called but the values of all other registers, including the A register, are not affected.

The use of OUTCH is illustrated by the following example which outputs a '*' at the current cursor position.

```
LDA #'*      ; A = ASC("*")
JSR OUTCH   ; Output character
```

Using these simple character input and output routines, we may now write an assembly language program which reads characters from the keyboard and prints them on the display. Assume that the read/print sequence halts when the BREAK key is pressed.

```
        LDA #$03
        STA BREAK      ; Set location BREAK to
*       BREAK key input code
GETCH   JSR INCH
        BEQ GETCH      ; Get a character
        CMPA BREAK     ; Is it BREAK
        BEQ DONE      ; If so, finish with no print
        JSR OUTCH     ; Print the character
        BRA GETCH     ; Get next character
DONE    ....
```

The final example in this introduction to assembly language input and output reads 10 characters into a memory area then prints them in reverse order. Notice how auto increment and decrement of the X register is used in this sequence.

```
        CLRB          ; B is counter register
        LDX #CHARS    ; Set up address of memory area
GETCH   JSR INCH
        BEQ GETCH     ; Get a character
        STA ,X+       ; Store it and increment X
        INCB          ; Add 1 to counter
        CMPB #10     ; If counter <= 10 then
        BLS GETCH    ; get next character
```

```
*   Now all characters are input and the address in X is
*   one greater than the address of the last character
*   in the sequence
*   Count downwards to output them in reverse order
```

```
        DECB          ; Reset B to correct number
COUT    LDA , -X      ; Decrement X
*       and fetch character to A
        JSR OUTCH     ; Print it
        DECB          ; One off counter
        BNE COUT     ; If counter <> 0 goto OUTCH
```

5.6 SUBROUTINES

A subroutine is a self-contained section of code which, usually, is set up to implement a particular function. Subroutines may be called from within a program. They carry out their specified function and control then return to the statement following their call.

Subroutines are a very important programming construct and the assembly language programmer has great flexibility in how he defines and uses subroutines. In fact, much of the next chapter is dedicated to this topic and we confine our description here to an explanation of how BASIC'S GOSUB command may be implemented.

In BASIC, when we set up or declare a subroutine, we assign it a line number which is out of sequence with the numbers in the rest of our program. To call the subroutine, we set up the values which it needs in program variables and then execute a GOSUB <line number> instruction. This transfers control to the subroutine until a RETURN statement is executed when control returns to the calling program.

For example, the following BASIC sequence calls a subroutine to check if a number is an odd number less than 20. If so, the subroutine converts it to another number by adding 20 to it. Otherwise, it returns the number unchanged. The subroutine expects its input to be stored in the variable INN and returns its output in the variable OUTN.

```

100 INPUT INN
110 GOSUB 1000
120 PRINT OUTN
130 ....
1000 RM = INN - (INT(INN/2)*2)
1010 IF INN < 20 AND RM = 0 THEN 1040
1020 OUTN = INN
1030 GOTO 1050
1040 OUTN = INN + 20
1050 RETURN

```

When using subroutines in assembly code, we may either use the BSR instruction or the JSR instruction. The BSR instruction is like the unconditional branch instruction BRA, but as well as branching it saves the value of PC on the S-stack. The JSR instruction is used when we have subroutines set up at known addresses or when it is necessary to use indirect addressing to call the subroutine.

Consider how the above BASIC code might be translated to assembly language. As we haven't yet covered the input and output of numbers, let us assume that there exists a subroutine GETNUM which inputs a number to the A register and a corresponding routine

PUTNUM which outputs the A register, as a number, to the screen.

```

        JSR GETNUM      ; INPUT A
        STA INN         ; INN = A
        BSR CONVON     ; To convert number
        LDA OUTN
        JSR PUTNUM     ; PRINT A

```

* CONVON - Add 20 to odd numbers < 20

```

CONVON  LDA INN
        CMPA #20
        BHI EXIT      ; If INN > 20 then goto EXIT
        BITA #$01     ; Test bottom bit of A
*
        BEQ EXIT      ; If it is 0, number is even
        ADDA #20      ; Add 20 to number
EXIT    STA OUTN      ; and store in OUTN
        RTS           ; return to calling code

```

The RTS instruction is used to return control to the instruction which immediately follows the subroutine call. As CONVON is called above, the first load instruction LDA INN is redundant as INN is already held in register A. However, we don't optimise this by removing the load instruction as the subroutine specification does not require the programmer to store INN in register A before the subroutine call.

Notice also that the subroutine alters the value of register A. In general, subroutines should leave the states of registers exactly as they were when the subroutine was called. Therefore, all subroutines ought to have the following structure.

```

    Save registers used by subroutine on stack
    Subroutine code
    Restore register values from stack
    Return

```

The subroutine CONVON may be adapted to reflect this structure:

```

CONVON  PSHS A,CC      ; Save A and CC on stack.
        LDA INN
        CMPA #20
        BHI EXIT
        BITA #$01
        BEQ EXIT
        ADDA #20
EXIT    STA OUTN
        PULS A,CC,PC  ; Restore and return

```

All the RTS instruction does is to pull PC from the S-

stack so it can be left out if PC is pulled explicitly when the saved registers are restored from the stack.

This mechanism of passing parameters to and from a subroutine in fixed memory locations is not ideal. We shall describe its deficiencies and introduce better parameter passing conventions in the following chapter.

5.7 ARRAYS

Arrays are one of the most commonly used data structures where a sequence of storage elements is given a name and particular elements in that sequence are accessed by number. In this section, we show how arrays of numbers may be stored and accessed using assembly language.

In BASIC, the programmer may use one-dimensional arrays which are made up of a linear sequence of numbers or two-dimensional arrays which, conceptually, may be considered as a table or matrix of numbers. In fact, two-dimensional arrays are also stored in the computer's memory as a linear sequence and the BASIC system provides routines to map a row/column pair (l,m), say, to the appropriate address n in the linear sequence. Two possible mappings which may be used by the assembly language programmer are described later in this section.

When using one-dimensional arrays in assembly language, you must know the address of the first element in the array. You get this by associating a label with a 'reserve store' directive as described in section 4.3. This label identifies the so-called 'base address' of the array. We assume, in the remainder of this section, that NARR is the base address of a one-dimensional array of 8-bit numbers and that MATRIX is the base address of a two-dimensional numeric array.

These may be set up using assembler directives as follows:

```
NARR      RMB 15
MATRIX    RMB 100
```

The index registers X and Y are the mechanism through which consecutive array elements may be accessed. The base address of the array is loaded into one of these index registers and the auto increment/decrement facilities used to sequence through the array. For example, say NARR is made up of 15 8-bit values and you want to set all elements to 0. In BASIC, you would write:

```
100 FOR I = 1 TO 15 DO
110 NARR(I) = 0
120 NEXT I
```

Exactly the same assignments may be specified in assembly language but there is no need for an explicit counter variable.

```

        LDH #NARR          ; Put array base address in X
SET0    CLR  ,X+          ; NARR(X) = 0 : X = X + 1
        CMPX #NARR+15     ; Compare X to base address+15.
*       ; see if all elements cleared
        BLS   SET0        ; If not, goto SET0
*       ; to clear next element

```

The availability of index registers makes array element access a very efficient operation. Even when auto increment or decrement cannot be used to update the index register, because the step is not one or two, the LEA instruction may be used to perform arithmetic on the index register.

For example, say the following BASIC code is to be implemented in assembly language:

```

100 FOR I=3 STEP 3 TO 15
110 NARR(I) = NARR(I - 1) + 1
120 NEXT I

```

Using assembly language, there is again no need for an explicit array index variable:

```

        LDH #NARR + 2     ; X = base of NARR + 2
*       ; As NARR+0 is first element
*       ; this refers to 3rd element
SETVAL  LDA  ,-X         ; A = Previous element
        INCA           ; A = A + 1
        LEAX 1,X        ; X = X + 1 to get back to
*       ; address to be assigned
        STA  ,X         ; NARR(X) = NARR(X-1) + 1
        LEAX 3,X        ; X = X + 3
        CMPX #NARR + 15 ; Are we finished?
        BLS  SETVAL     ; If not, back to SETVAL

```

The use of index registers to hold the address of the array element to be accessed is easy to implement for one-dimensional arrays. However, when two-dimensional arrays are used, the programmer must devise a way of storing the array as a linear sequence and must invent a mapping to convert a row/column address to an address in that sequence. There are two techniques which are commonly used for this conversion.

The first of these techniques stores the entire array, row by row, in contiguous memory locations. So, if an array is declared in BASIC as MATRIX(10,10) this takes up 100 memory elements. The first 10 elements are row 1, written as MATRIX(1,*), the next 10 are row 10, MATRIX(2,*), etc. The position of an element in row m say is found by finding where row m starts then

adding the column displacement to it.

The starting position of a particular row, the row base, is computed by multiplying the row number by the length of the row. As the row base of the very first row is the same as the array base address, we count row numbers from 0. Therefore, to find the row base of the sixth row, we actually multiply the row length by five. For example, $MATRIX(6,*)$ would have a row base address of $MATRIX + 50$ ($5 * \text{row length}$) and the element $MATRIX(6,8)$ has the address $MATRIX + 50 + 8$.

An alternative storage technique for two-dimensional arrays does not require array rows to be stored consecutively nor does it require a multiplication to compute the row base address. Rather, the row base addresses are all stored separately in another array called an Iliffe vector, named after J. Iliffe, the inventor of the mapping technique. This is best illustrated diagrammatically as shown in Figure 5.1.

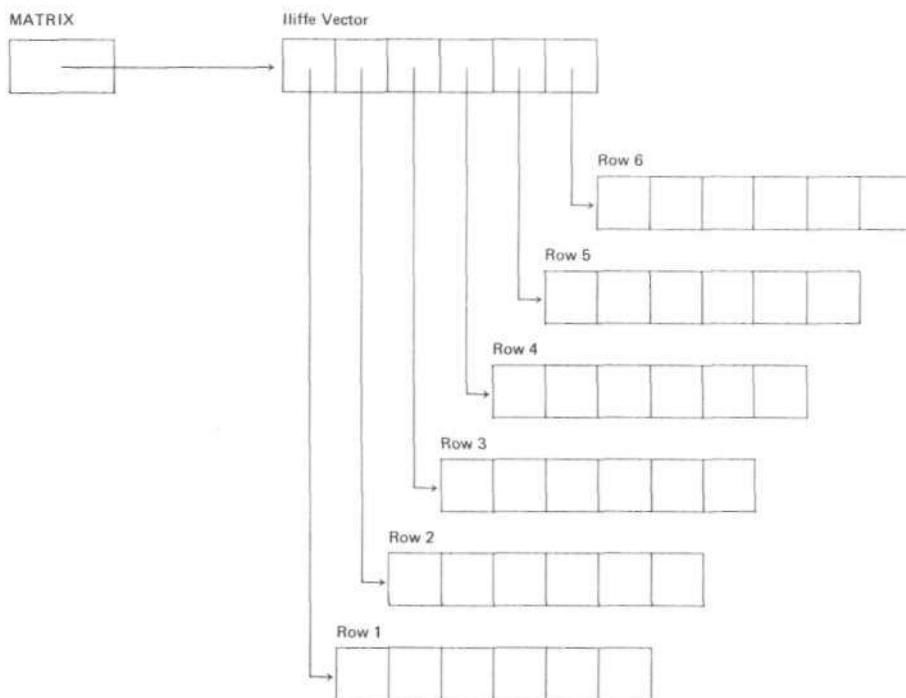


Fig. 5.1 Using Iliffe Vectors to implement 2-D arrays

To find out the row base address, the row number is used as an index into this Iliffe vector and the

Notice, from this example, how the powerful indexed addressing features of the M6809 makes the computation of the row base very efficient indeed. In fact, both techniques of two-dimensional array implementation are efficient on the M6809.

5.8 A MACHINE CODE MONITOR

Rather than present a number of small examples of working assembly code programs, we have chosen to illustrate the principles described in this chapter with a single, substantial assembly code program. However, we have written this program in a structured way so that it is made up of a number of easily understood routines.

The reason for adopting this approach is that we want to present a program which is of use to the novice assembly code programmer and which can help him debug his own programs. The program below is a so-called 'monitor' which provides facilities for the user to examine the contents of specified memory addresses and to change them by typing the revised value.

The monitor issues a prompt to the user and responds to two commands:

- (1) J - this means jump to the start of the user program.
- (2) M <address> - this displays the contents of the specified address.

Once an M command has been issued, the user may examine subsequent addresses by typing any letter and may examine the previous address by typing an 'up arrow' character. If the user types a value made up of three decimal digits or two hexadecimal digits preceded by a '\$' sign, this value is filled in to the current address.

To return to the program which called the monitor, you must type a 'BREAK' character. A number is normally terminated with an 'ENTER' character but can be terminated early with any other character in which case, the change is ignored.

The sequence below is an example of a possible dialogue with the monitor. User input is underlined.

```
*M $1000
$1000 000 $00 255
$1001 001 $01 $FF
$1002 128 $80 2[ENTER]
$1003 016 $10 [up arrow]
$1002 002 $02 $A+
$1003 016 $10 [up arrow]
$1002 002 $02 [BREAK]
```

The monitor program itself now follows. Do not worry if you cannot understand it completely on your first reading. You may find it helpful to read Chapter 6 and then come back to this program for further study.

* MONITOR - memory examine and change system

*

* This program is intended to help with the
 * development and debugging of assembly language
 * programs. It provides facilities for the
 * user to input a memory address and display its
 * contents. These contents may then be modified
 * by the user.

* Unless otherwise specified, all routines preserve
 * all register values except CC and any registers used
 * for returning results.

```

                ORG 20001
                LBRA DRAMON ; Entry point of the monitor
INTRO          FCC "DRAGON MONITOR 1.0"
                FCB 0      ; Terminator for string
CR            EQU $0D
QMARK        EQU $3F
UPAROW       EQU $5E
BREAK        EQU $03
DOLLCH       EQU $24
STAR         EQU $2A
CBLINK       EQU $8009    ; Cursor blink routine
INCH         EQU $8006    ; Keyboard input routine
OUTCH        EQU $800C    ; Output character routine

```

* INECHO - read a character and echo it to screen

* Register inputs NONE

* Register outputs A - contains character input

```

INECHO        PSHS X,B    ; Save registers affected
INLOOP       JSR CBLINK   ; Blink the cursor
                JSR INCH   ; Scan the keyboard
                BEQ INLOOP  ; and wait for a character
                JSR OUTCH  ; Echo the character
                PULS X,B,PC ; Restore registers and return

```

* OUTSTR - print string of characters

*

* Register inputs X - pointer to beginning of string

* Registers destroyed X,A

*

* String must be terminated with a null byte

*

```

OUTSTR        LDA 0,X+    ; Get character from string
                BEQ ENDSTR ; Terminated by a zero byte
                JSR OUTCH  ; Output the character
                BRA OUTSTR ; and deal with the next one
ENDSTR        RTS

```

```

* OUTCR - output a carriage return

* Register inputs NONE

OUTCR    PSHS A           ; Preserve A
         LDA #CR         ; Load Carriage Return code
         JSR OUTCH       ; and send it
         PULS A,PC       ; Restore and return

* OUTSP - output a space

* Register inputs NONE
*
OUTSP    PSHS A
         LDA #$20        ; Code for space
         JSR OUTCH       ; and output it
         PULS A,PC

* READY - Prompt user for new command
*
* Register inputs NONE
*
READY    PSHS A
         BSR OUTCR       ; Take a new line
         LDA #STAR       ; before outputting
         JSR OUTCH       ; prompt character
         PULS A,PC

*
* DOLLAR - prompt for hexadecimal value
*
* Register inputs NONE
*
DOLLAR   PSHS A
         BSR OUTSP
         LDA #DOLLCH     ; Hexadecimal prompt
         JSR OUTCH
         PULS A,PC

*
* INHEXD - input a hexadecimal value
*
* Register inputs NONE
* Register outputs A - if valid hex char then hex
*                      value else character
*                      CC.V = 0 if valid hex character
*                      = 1 if non-hex character
*
INHEXD   BSR INECHO      ; Read a character
         CMPA #'0        ; and check the range
         BLO INHERR     ; for "0" to "9"
         CMPA #'9
         BLS CHOSUB     ; and convert if so
         CMPA #'A        ; Could be "A" to "F"
         BLO INHERR

```

```

        CMPA #'F
        BHI INHERR
        SUBA #7          ; Make "A" to "F" follow "9"
CHOSUB  SUBA #'0        ; Convert to numeric value
        ANDCC #$FD      ; Valid return
        BRA INHXIT
INHERR  ORCC #2         ; Error return, V bit set
INHXIT  RTS

```

* OUTHXD - Output hex digit as character

* Register inputs A - hexadecimal value

```

OUTHXD  ANDA #$F        ; Mask off MS 4 bits
        CMPA #9         ; Check for decimal digit
        BLS ADDCHO
        ADDA #1         ; A to F offset
ADDCHO  ADDA #'0        ; Convert to character
        JSR OUTCH       ; and output it
        RTS

```

* INDECD - input decimal digit and convert to value

*

Register inputs NONE

* Register outputs A - decimal value if in range 0-9

* - character if non-decimal

* CC.V - 0 if valid input

* =1 otherwise

*

```

INDECD  BSR INECHO
        CMPA #'0
        BLO INDERR
        CMPA #'9
        BHI INDERR
        SUBA #10        ; Converts to numeric value
        ANDCC #$FD
        BRA INDXIT
INDERR  ORCC #2
INDXIT  RTS

```

* OUTDCD - output decimal digit as character

*

* Register inputs A - decimal value

```

OUTDCD  ANDA #$F
        ADDA #'0
        JSR OUTCH
        RTS

```

* HCNVAB - combine hex digits into single byte

*

* Register inputs A - new hex digit

* B - existing hex digit

* Register outputs B - new hex value = B*16+A

```

*
HCNVAB   STA 0,-S           ; Save for later
          ASLB              ; Move LS 4 bits
          ASLB              ; of B
          ASLB              ; to the
          ASLB              ; MS 4 bits
          ADDB 0,S+         ; Add in new hex digit
          RTS

*
* INHEXB - input a hexadecimal byte
*
* Register inputs NONE
* Register outputs A - hex byte value
*           CC.C = 0 means value is OK
*           CC.V = 0 means that B contains last
*                 hex value input
*           CC.V = 1 means hex byte terminated
*                 prematurely and B holds
*                 character read in.
*
INHEXB   CLRB              ; Initialise to 0
          BSR INHEXD       ; Read a hex digit?
          BVS NONHEX
          BSR HCNVAB       ; yes, so add to byte
          BSR INHEXD       ; Second hex digit?
          BVS NONHEX
          BSR HCNVAB       ; yes, so add that also
NONHEX   ANDCC #$FE       ; Indicate OK
          EXG A,B          ; Return with A and B set up
          RTS

* OUTHXB - output a hex byte as characters
*
* Register inputs A - contains byte value
*
OUTHXB   PSHS A
          LSRA              ; Shift MS 4 bits
          LSRA              ; to LS 4 bits
          LSRA
          LSRA
          BSR OUTHXD       ; and output the hex digit
          LDA 0,S          ; Get original again
          BSR OUTHXD       ; MS 4 bits masked off by OUTHXD
          PULS A,PC        ; Return intact
*
* MULB10 - multiply by 10
*
* Register inputs B - value to be multiplied
* Register outputs B = B*10
*           CC.C = 0 means result between 0-255
*           =1 result out of range
*
MULB10   CLR 0,-S         ; Create temp on stack
          ASLB              ; Evaluate 2*B

```

```

        BCS MULXIT      ; Too big?
        STB 0,S        ; Save as temp result
        ASLB          ; Evaluate 4*B
        BCS MULXIT      ; Too big?
        ASLB          ; Evaluate 8*B
        BCS MULXIT      ; Too big?
        ADDB 0,S       ; Evaluate (2*B)+(8*B)
* If this is too big a result C will be set
MULXIT  LEAS 1,S      ; Release temp.
        RTS
*
* DCNVAB - combine decimal values
*
* Register inputs A - new decimal digit
*                B - old decimal value
* Register outputs B - result = B*10 + A
*                CC.C = 0 - result in range 0-255
*                = 1 - result out of range
*
DCNVAB  PSHS A        ; Save register
        BSR MULB10    ; B:=B*10
        BCS DCNXIT    ; Too big?
        ADDB 0,S      ; B:=(B*10)+A
DCNXIT  PULS A,PC     ; Restore and RTS
*
* INDECB - Input decimal byte value
*
* Register inputs NONE
* Register outputs A - input value if valid
*                CC.C = 0 value in range 0-255
*                =1 value out of range
* If CC.V = 1 then number terminated early so must
* be in range 0-255. B holds last converted digit
* of all 3 typed otherwise set to terminator.
*
INDECB  CLRB          ; Initialise byte
        BSR INDECD    ; Valid digit?
        BVS NONDEC    ; Valid digit?
        BSR DCNVAB    ; yes, add to byte
        BCS IDBXIT    ; Too big?
        BSR INDECD    ; Valid digit?
        BVS NONDEC    ; Valid digit?
        BSR DCNVAB    ; yes, add to byte
        BCS IDBXIT    ; Too big?
        BSR INDECD    ; Valid digit?
        BVS NONDEC    ; Valid digit?
        BSR DCNVAB    ; yes, add to byte
        BCS IDBXIT    ; Too big, so leave C set
NONDEC  ANDCC #$FE    ; Result is in range 0 - 255
IDBXIT  EXG A,B      ; Return registers
        RTS
* OUTDCB - output byte as 3 digit decimal value

```

* Register inputs A - contains byte to be output

*

```

OUTDCB   PSHS D           ; Both A and B used
         TFR A,B         ; A is to be used in sub.
         CLRA           ; Clear the 100s digit
NXTHUN   CMPB #100       ; Any 100s?
         BLO TRYTEN
         INCA           ; yes, so update digit
         SUBB #100      ; Subtract a 100
         BRA NXTHUN    ; and try again
TRYTEN   BSR OUTDCB     ; Output the 100s digit
         CLRA           ; Set up for 10s
NXTTEN   CMPB #10       ; Any 10s
         BLO TRYONE
         INCA           ; yes, so update 10 digit
         SUBB #10      ; Subtract 10
         BRA NXTTEN    ; and try again
TRYONE   LBSR OUTDCB    ; Output 10s digit
         CLRA           ; Now count the 1's
NXTONE   CMPB #1        ; Any 1's
         BLO OUTONE
         INCA           ; yes, update 1's digit
         DECB          ; Subtract 1
         BRA NXTONE   ; and try once more
OUTONE   LBSR OUTDCB    ; Final digit
         PULS D,PC     ; Restore and return

```

*

* HCONVX - Add hexadecimal digit to X

*

* Register inputs A - new hex digit

* X - old hex value

* Register outputs X = X*16 + A

*

```

HCONVX   STA 0,-S       ; Save away for later use
         EXG X,D        ; So we can do arithmetic
         ASLB          ; This performs an ASL
         ROLA          ; on the D register
         ASLB
         ROLA
         ASLB
         ROLA
         ASLB
         ROLA          ; Have now space in LS 4 bits
         ADDB 0,S+     ; To add in new hex digit
         EXG D,X       ; Restore D and return X
         RTS

```

*

* INHEXW - input hex word (address)

* May be up to 4 hex digits

*

* Register inputs NONE

* Register outputs X - hex address value

* CC.V = 1 address terminated

```

*           CC.V = 0 full 4-digit address
*           read in
*           CC.C = 0 value in range 0-FFFF
*

```

```

INHEXW    PSHS D           ; Save from harm
          LDX #0           ; Initialise address
          LBSR INHEXD      ; Read hex digit?
          BVS NONHW
          BSR HCONVX       ; yes, add to X
          LBSR INHEXD      ; Read hex digit?
          BVS NONHW
          BSR HCONVX       ; yes, add to X
          LBSR INHEXD      ; Read hex digit?
          BVS NONHW
          BSR HCONVX       ; yes, add to X
          LBSR INHEXD      ; Read hex digit?
          BVS NONHW
          BSR HCONVX       ; yes, add to X
NONHW     ANDCC #$FE      ; Result valid
          PULS D,PC        ; Restore and return

```

```

¥

```

```

* OUTHXW - output hex word as 4 hex digits

```

```

* Register inputs X - value to be output

```

```

*

```

```

OUTHXW    PSHS D
          TFR X,D           ; D := hex word
          LBSR OUTHXB      ; Output MS byte first (A)
          TFR B,A
          LBSR OUTHXB      ; followed by LS byte (B)
          PULS D,PC

```

```

* MCOMND - memory examine and change

```

```

*

```

```

* Register inputs NONE

```

```

* Registers destroyed X, A, CC

```

```

*

```

```

* Interprets user commands as defined in introduction

```

```

MCOMND    LBSR DOLLAR      ; Prompt for hexadecimal
          BSR INHEXW       ; Expecting an address (hex)
EXAMIN    LBSR OUTCR       ; Prefix the address
          LBSR DOLLAR      ; with a "$"
          BSR OUTHXW       ; followed by the address
          LBSR OUTSP       ; separate by a space
          LDA 0,X          ; Get contents of that address
          LBSR OUTDCB      ; Shown as decimal value
          LBSR DOLLAR      ; and followed by the
          LBSR OUTHXB      ; hexadecimal value
          LBSR OUTSP       ; Then a space
          LBSR INDECB      ; Assume decimal change
          BCS QUERY        ; Too big a number?
          BVC CHANGE       ; If OK just change the byte

```

```

* A non-digit has been typed, check for hex prefix

```

```

        CMPB #DOLLCH      ; Hex number?
        BNE CHKCR
        LBSR INHEXB      ; yes, so get the rest
        BVC CHANGE      ; If OK just change the byte
* At this point an early end to the number has
* been typed. Only CR (ENTER) will be allowed.
* Note: If only a CR is typed then the byte is
* cleared to zero!. Be careful!
CHKCR   CMPB #CR         ; CR (ENTER)?
        BEQ CHANGE      ; yes, then change the byte
* Check for the "up arrow" key since this
* returns to the previous location.
        CMPB #UPAROW    ; "up arrow"?
        BEQ LSTLOC      ; yes, move back to last
* Now check for the BREAK key since this exits
* the Monitor
        CMPB #BREAK     ; BREAK in?
        BEQ MCDXIT      ; yes, then exit
NXTLOC  LEAX 1,X         ; Move location address on
        BRA EXAMIN      ; and repeat
LSTLOC  LEAX -1,X       ; Back up location address
        BRA EXAMIN      ; and repeat
CHANGE  STA 0,X         ; Make the change
        CMPA 0,X        ; and check afterwards
        BEQ NXTLOC      ; OK?, move on if so
QUERY  LDA #QMARK      ; Made a mistake.
        JSR OUTCH       ; so report it.
        BRA EXAMIN      ; Don't do anything untoward
MCDXIT  RTS             ; Return
*
* JCMND - jump to start of program
*
* Register inputs NONE
*
JCOMND  LBSR DOLLAR     ; Put out $ prompt
        LBSR INHEXW     ; Get hex address
        BVS JERR       ; MUST be all 4 hex digits
        JMP 0,X
JERR    RTS             ; Only get here on error
*
* DRAMON - main driving routine
*
* Register inputs NONE
* Registers destroyed X, A, B, CC
DRAMON  LBSR OUTCR      ; Prompt on a new line
        LEAX INTRO,PCR  ; Output intro.
        LBSR OUTSTR
NXTCMD  LBSR READY      ; Prompt the user
        LBSR INECHO     ; Read the command
        CMPA #'M        ; Memory examine and change?
        BNE TRYJ
        BSR MCOMND     ; yes, then obey it
        BRA NXTCMD     ; and repeat

```

```
TRYJ      CMPA #'J           ; Is it the Jump command?
          BNE TRYBRK        ; no, then check for BREAK
          BSR JCOMND        ; yes, so obey it
          BRA NXXTCMD       ; but don't expect to get here
TRYBRK    CMPA #BREAK       ; Is it the BREAK key?
          BNE NXXTCMD       ; no, then prompt again
          RTS               ; yes, return to caller
```

Chapter 6

Subroutines and strings

When we try to solve a problem, we do not go directly from the general statement of the problem to a detailed solution unless the problem is very trivial indeed. Rather, we split the problem into a sequence of sub-problems and work out the individual solutions to these smaller problems. The sub-problem solutions are then integrated and coordinated to form the general problem solution.

When a problem is intended for computer solution, we can use exactly the same approach. The overall problem solution is a computer program but, rather than generate this as a monolithic code sequence, it can be made up of calls to subroutines. Each subroutine is the solution to a particular sub-problem. By adopting this approach, we reduce the overall complexity of the program because we never have to understand or think about any more than one subroutine at any one time.

The idea of a subroutine as a self-contained section of code which can be initiated from elsewhere in the program was one of the earliest advances in computer programming. Subroutines are an essential tool for the programmer as they allow him to create 'black boxes' implementing particular functions. Once these have been written and tested, the programmer need not be bothered how they work as long as he knows their function and how to use them.

To make the most effective use of this problem-solving method, the programming language which we use must allow us to create subroutines which are independent of their environment. Unfortunately, BASIC subroutines are very primitive indeed and are not truly self-contained. Their disadvantages can be summarised as follows:

- (1) BASIC subroutines cannot be made independent of their environment because the only way of passing information to and returning information from a subroutine is through its environment. That is, program variables must be used to pass information to and from the subroutine. This means that BASIC subroutine libraries cannot be created because both the subroutine and the program must 'agree' on what variables should be used for passing input and output parameters.

- (2) There is no way, in BASIC, for a subroutine to have a completely private data area which no other subroutine may tamper with. A private or local variable area is essential if the subroutine is to be self-contained and if the programmer is to be sure that a call of the routine always does exactly what's expected of it.
- (3) The BASIC programmer cannot give his subroutine a name which reflects its function. Rather, he must refer to it by a meaningless line number. When a program has many subroutines, it is difficult to discern what operations are implemented by a sequence of subroutine calls, especially if the program is not properly commented.

The subroutine facilities available to the assembly language programmer are actually slightly less primitive than BASIC'S subroutine mechanism. At least in assembly language, a mnemonic name rather than a number can be given to a subroutine. As in BASIC, there are no built-in mechanisms for passing information to and from a subroutine or for establishing local data space.

However, the flexibility of assembly language programming is such that the programmer may establish a set of conventions which allow local data areas to be created and which allow parameters to be passed to and from a subroutine without using global variables. These conventions provide a more powerful, effective and safer mechanism for using subroutines than that available to the BASIC programmer.

In this chapter we show how the M6809's architecture is well suited to the implementation of self-contained subroutines and we describe a very general way of declaring and calling subroutines. We also describe a subroutine calling technique which can be used when execution speed is the paramount consideration and we explain how to construct subroutines which are position independent. The final sections of the chapter discuss techniques for representing and manipulating character strings and we show how assembly language subroutines may be integrated with BASIC programs.

6.1 ASSEMBLY LANGUAGE SUBROUTINES

We have already shown in section 5.6 how the BASIC GOSUB and RETURN statements can be implemented in assembly language using the BSR, JSR, and RTS instructions. In that section, we showed how parameters could be passed to and from subroutines using shared global variables but this is not a recommended technique. Furthermore, if it is important

to produce very efficient code, using shared variables for parameter passing has the additional disadvantage that it takes time to set up and access these shared variables.

In many cases, there is no need for separate variables to be used for parameter passing. Rather, if one or two parameters only are to be passed to and from the subroutine, it is often possible to pass their values or addresses in registers. This saves both the calling program storing register values and the subroutine reloading these values into registers.

The use of registers for parameter passing also has the advantage that the parameters do not take up memory space and that the impermanent nature of register values emphasises that subroutine parameters are distinct from other permanent program variables.

Program 6.1 shows how the A and X registers can be used to pass parameters to and from a subroutine.

```
* SQUARE - compute square of input parameter
*
* Register input A - positive number to be squared
* Register output X - square of input
* Method used is to add n to itself n times

SQUARE   PSHS B           ; Save B register
          TFR A,B         ; B = A
          LDX #0          ; Clear X
SLOOP    ABX             ; X = X + B
          DECA            ; Use A as counter of the
*                   number of adds
          BNE SLOOP
          TFR B,A         ; Restore value of A
          PULS B,PC       ; Restore B and return
```

Program 6.1 SQUARE - compute square of input

Notice that a return from subroutine instruction, RTS, is not required as the program counter is explicitly restored using a PULS instruction.

To call this subroutine, the input parameter must be set up in register A. A possible calling sequence might be:

```
LDA #28           ; Compute 28 squared
PSHS X           ; Save value of X as it is
*               destroyed by SQUARE
BSR SQUARE       ; Call routine
STX RESULT      ; Store result of call
PULS X           ; Restore X
```

Notice how the S-stack is used to save register values which are subsequently restored. Of course, the value of X before the call of SQUARE is not necessarily

precious. If this is the case there is no need to save it before the call and restore it after the subroutine has been executed.

Any of the registers A, B, X, Y, or U may be used to pass parameters to and from subroutines. However, the S register is never used for this purpose because of its role as a system stack pointer. As the return address, the value of PC when the routine is called, is stacked, it is important that the value in S is not corrupted otherwise a proper return from the subroutine is impossible.

In some subroutines it is useful to return an error indicator specifying whether or not the subroutine has succeeded in its task and the best way to do this is to make use of the CC register. The programmer may use CC.V or CC.C as error indicators or, alternatively, the settings of CC.Z and CC.N may indicate that an event has or has not occurred.

We have already seen an example of how this latter method can be used to determine if an input routine has returned a character. If a character has been input, CC.Z is unset otherwise CC.Z is set. Therefore, the following code loops until a character is input:

```
GETCH   JSR INCH   ; Call input routine
        BEQ GETCH
```

When using the CC register to return results from a subroutine, the ANDCC and ORCC instructions may be used to set and unset particular bits in that register.

Using registers for subroutine input and output parameters is an efficient parameter passing technique which should be used when subroutine calls must be executed as quickly as possible. However, this technique requires that the programmer knows exactly what registers must be set up when the subroutine is called and what registers are used by the subroutine to return results. Typically, different subroutines have different conventions in this respect depending on the number and type of input parameters and on whether they return one or more results. The programmer must know, in detail, the conventions for each subroutine before he can make use of it.

If there are only a few subroutines used in a program, it may be fairly easy to memorise such details, but in a large program, where there might be tens or even hundreds of subroutines, this is not possible. Furthermore, the programmer may wish to build up a library of useful subroutines to be included in his programs as they are required. It is obviously a good idea to have all the subroutines in the library used in a consistent way so passing parameters in registers is not really suitable.

There are two different general mechanisms which can

be devised to support subroutine parameter passing. The first technique, which we do not describe in detail, is to allocate a specified parameter area for each subroutine and store the addresses of the parameters in that area. When calling the subroutine, this area is set up immediately prior to the subroutine call. The address of the parameter area is assigned to an agreed register such as the Y register, and indirect indexed addressing is used to access the subroutine parameters.

This technique works well in most cases but cannot support so-called recursive subroutines. Recursive subroutines are subroutines which contain an embedded call to themselves. Although this may seem an unusual idea to the programmer who has only ever programmed in BASIC, recursion is very useful in many situations as it allows you to write compact programs which, with practice, are easy to understand. Readers who wish to experiment with recursive programming should consult textbooks which describe data structures such as lists and trees to see how recursion is used.

The second generalised technique of subroutine parameter passing can handle recursive routines. It makes use of a stack to pass parameters to and return results from subroutines. This technique can be implemented very efficiently on the M6809 because of its built-in stack manipulation instructions. It is described in detail below.

6.1.1 Parameter passing using a stack

The M6809 processor is designed so that two stacks may be used, at the same time, by the assembly language programmer. One of these stacks, the hardware or system stack, is referenced via the S register and is always in existence as it is used to hold the program counter when a subroutine is called. The user stack, or U-stack, is referenced via the U register and may or may not be used depending on the application being programmed.

A parameter passing mechanism can be devised which uses the S-stack to hold information such as the subroutine return address and which uses the U-stack to hold subroutine parameters. This works perfectly well and is often used. It does, however, require considerable housekeeping by the calling and called routine to make sure that the stacks are always consistent.

The technique which we describe below uses only a single stack, the S-stack, but uses two stack pointer registers, S and U. As well as being useful to the assembly language programmer, this technique of subroutine parameter passing is that used by structured high-level languages such as Pascal.

To understand this parameter passing method, we must

introduce the idea of a stack frame. A stack frame is a data area which is set up on the stack when a subroutine is called. The exact number of bytes making up a stack frame depends on the number of subroutine parameters, the registers saved by the subroutine and the local data space required by the subroutine. Figure 6.1 is a diagram of a stack frame in its most general form:

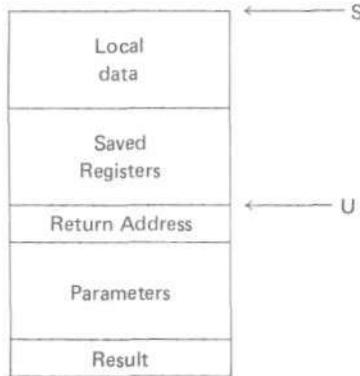


Fig. 6.1 Stack frame organisation

The saved registers are those registers which are modified by the subroutine. The return address is the value of PC stacked by the BSR or JSR instruction.

The subroutine parameter area is set up by the calling program with the values of the subroutine input parameters and, if a result is returned by the subroutine, the calling program reserves a location on the stack for it.

To illustrate how stack frames are used, consider the SQUARE subroutine described earlier in this chapter. This is a subroutine which we might wish to implement as a function which returns the square of its parameter. Assuming that we use the stack for parameter passing, we would call the function SQUARE using the following instruction sequence:

```

LEAS -2,S      ; Decrement S by 2.
*
*             As stacks in the M6809 grow downwards
*             this leaves a 2-byte 'hole' in the
*             stack for the result
LDA N         ; A = number to be squared
PSHS A       ; Put parameter onto the stack
BSR SQUARE   ; call SQUARE

```

The call of SQUARE pushes PC onto the S-stack. The subroutine SQUARE first pushes the registers which it uses onto the stack and then sets up a register so that

indexed addressing may be used to access the subroutine parameters and result. As the X and Y registers are often used for array accessing, it is best to use the U register as the stack index register set to the base of SQUARE'S stack frame.

Given that the subroutine SQUARE saves the registers A, B, X, U, and CC, the stack structure after subroutine entry is shown in Figure 6.2.

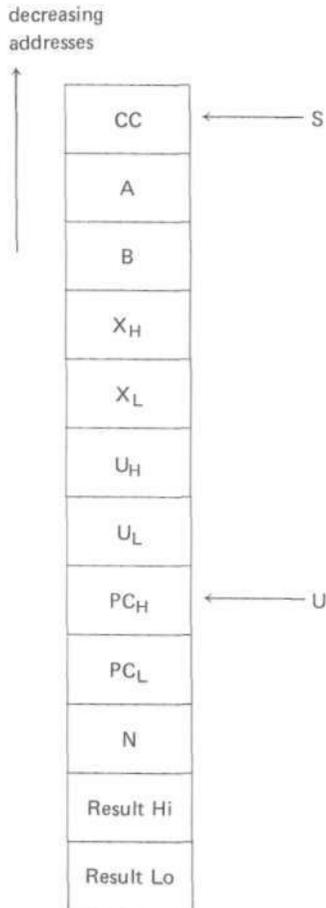


Fig. 6.2 Stack structure after entry to SQUARE

In general, a called routine should save the value of the U register then reset it so that it points to the current stack frame. It is important to ensure that the U register is set to the same relative position in the stack frame for every subroutine but the particular location chosen does not matter a great

deal. In our examples, the U register is set so that it refers to the hi-byte of the subroutine return address on the stack.

The U register is assigned after the registers have been saved using an LEAU instruction. As S points at the top location on the stack, the stack address assigned to U is computed using the S register value and the number of register bytes stacked.

The code implementing the subroutine SQUARE is:

```
* SQUARE - returns the square of its input
*
SQUARE   PSHS A,B,U,X,CC
          LEAU 7,S           ; set U register
          LDA 2,U           ; Get parameter from stack
*                               it is immediately below
*                               the return address
          TFR A,B           ; Use repeated addition to
          LDX #0            ; square N
SQLOOP   ABX               ; X = X + B
          DECA              ; A counts adds
          BNE SQLOOP
          STX 3,U           ; Store result
*                               Result is always immediately
*                               below parameter on the stack
          PULS A,B,U,X,CC,PC ; Restore and return
```

Program 6.2 SQUARE with stack parameter passing

On return from the subroutine, the S register is set so that it points to the subroutine parameter on the stack. As this is no longer required, the calling program must increment S to discard the parameter or parameters. After this modification of S, S then refers to the result returned by the subroutine.

The complete call/return sequence for SQUARE is therefore:

```
LEAS -2,S   ; Space for result
LDA N
PSHS A      ; parameter onto stack
BSR SQUARE ; call routine
LEAS 1,S   ; discard parameter
PULS D     ; result in D register
*          for processing, store, etc
```

Obviously, if the subroutine does not return a result, there is no need to reserve space on the stack for the result. It is, therefore, very important that the programmer ensures that each subroutine has an associated comment at its head which states the size, in bytes, and the type of any result. This is essential so that the correct call/return sequence may be used for that routine.

In general, the call sequence for a subroutine where the stack is used for parameter passing is as follows:

```
Reserve space, if necessary, for subroutine result
Evaluate parameters and store on S-stack
Call subroutine
Discard parameters
Retrieve subroutine result from the stack
```

The called routine must have an entry and exit sequence as follows:

```
Save U register and other registers as necessary
Set up U as stack frame register
<Body of subroutine>
Restore registers including PC
```

An important advantage of using this technique of parameter passing is that the stack may also be used as a local variable area for the called subroutine. These local variables are accessed using indexed addressing via the U or S registers.

Rather than allocate specific memory locations as private working store for the subroutine, it is possible to use stack locations for this purpose. This store is allocated dynamically on entry to the subroutine and de-allocated on exit from the routine. Thus store is only allocated when it is required and need not be set aside permanently for subroutine local variables.

Program 6.3 takes an array base address and an array length as parameters on the stack and returns the maximum and minimum values of that array as results. It uses local variables to hold the maximum and minimum values which have been determined so far.

* MAXMIN - determines MAX and MIN array values

*

* Results are left on the stack in space left

* by calling routine.

*

```
MAXMIN   PSHS U,A,B
          LEAU 4,S      ; U points at return address
          LDA 2,U       ; Array length in A
          LDX 3,U       ; address in X
          LDB ,X+       ; 1st element in B
          PSHS B
          PSHS B        ; Push locals onto stack
*
*           Both MAX and MIN initially set up
*           to be the value of 1st element
*           MAX=stack(S), MIN=stack(S+1)
          DECA
          BEQ DONE     ; If only one element, all done
MMLOOP   LDB ,X+       ; Array element in B
```

```

        CMPB ,S          ; Compare with MAX
        BGT NEWMAX      ; If greater, re-assign MAX
        CMPB 1,S        ; Compare with MIN
        BGT ELOOP      ; Value greater, go on to next
        STB 1,S         ; Otherwise re-assign MIN
        BRA ELOOP
NEWMAX  STB ,S          ; Re-assign MAX
ELOOP   DECA
        BNE MMLLOOP
DONE    LDA ,S+         ; Maximum value
        STA 5,U         ; into result space
        LDA ,S+         ; Minimum value
        STA 6,U         ; into result space
        PULS U,A,B,PC   ; Restore and return

```

Program 6.3 MAXMIN - find maximum and minimum of
array

This technique of local variable allocation allows recursive subroutines, subroutines which call themselves, to be implemented. When a subroutine calls itself, a completely new local variable area is set up on the stack and the data area of the calling routine is not destroyed.

We illustrate this using a recursive routine which, given an input parameter N, returns the Nth Fibonacci number. Fibonacci numbers are numbers in a sequence where the value of a given number is computed by adding the previous two numbers in the list. The first values in the sequence are 0 and 1 so the first 10 Fibonacci numbers are:

0 1 1 2 3 5 8 13 21 34

Fibonacci numbers are not just mathematical oddities but have practical uses in sorting large data files held on magnetic tape. Readers interested in how they are used should consult a textbook on sorting techniques.

A general formula for computing the Nth Fibonacci number is recursive:

```

if N = 1 then
    FIB(N) = 0
else
    if N = 2 then
        FIB(N) = 1
    else
        FIB(N) = FIB(N-1) + FIB(N-2)

```

So, if the 5th Fibonacci number is required, this formula would be evaluated as follows:

$$\begin{aligned}
 \text{FIB}(5) &= \text{FIB}(4) + \text{FIB}(3) \\
 &= \text{FIB}(3) + \text{FIB}(2) + \text{FIB}(2) + \text{FIB}(1)
 \end{aligned}$$

```

= FIB(2) + FIB(1) + 1 + 1 + 0
= 1 + 0 + 1 + 1 + 0
= 3

```

The assembly code routine below takes an 8-bit input parameter N and returns a 16-bit result which is the Nth Fibonacci number. As BASIC does not support recursion, we cannot first translate our logical solution above into BASIC but must go straight to assembly code.

```

* FIB - Computes Nth Fibonacci number
*
* Result left on stack in location P+1 where P
* is parameter address
* Set up equates to refer to stack locations
*
FRES1 EQU 3           ; Result
FPAR1 EQU 2           ; Parameter
FIBL1 EQU -5          ; Local variable
FIBL2 EQU -7          ; Local variable

FIB   PSHS A,U,CC     ; Save registers
      LEAU 4,S        ; Set stack frame register
      LEAS -4,S       ; Space for local variables
*
      LDA FPAR1,U     ; Get input parameter
      BLE ERR1        ; If it is not positive, error
      CMPA #1         ; is it 1st Fibonacci number?
      BNE FIB2        ; If not, try the second
      LDD #0          ; D = FIB(1)
      BRA EXIT        ; Get out of routine
FIB2  CMPA #2         ; Is FIB(2) required
      BNE FIBN        ; No, compute FIB(n)
      LDD #1          ; D = FIB(2)
      BRA EXIT        ; Get out
FIBN  LEAS -2,S       ; Get stack space for result
      DECA            ; FIB(N-1) is being computed
      PSHS A          ; Parameter for recursive call
*
      BSR FIB         ; Call FIB
      LEAS 1,S        ; Discard parameter
*
      PULS D          ; Pull result into D
      STD FIBL1,U     ; Store D into local variable

* Now call Fib again to compute FIB(N-2)
*
      DECA            ; A = N - 2
      LEAS -2,S       ; space for result
      PSHS A          ; stack parameter
      BSR FIB         ; and call FIB recursively
      LEAS 1,S        ; discard parameter
      PULS A,B        ; D = FIB(N-2)
      STD FIBL2,U     ; Assign to local

```

* Now add locals to get Fibonacci number
*

```

LDD FIBL2,U
ADDD FIBL1,U ; D = FIB(N-1)+FIB(N-2)
BRA EXIT ; get out
ERR1 LDD #-1 ; D = -1 if error
EXIT STD FRES1,U ; Store D in result space
PULS A,U,CC,PC ; Restore and return

```

Program 6.4 FIB - compute nth Fibonacci number

This routine can be optimised by using the space on the stack reserved for the result of FIB as local working store and by removing some redundant load instructions. We leave this optimisation as an exercise for the reader.

You will probably have to think quite hard to understand exactly what the FIB program is doing. You may find it helpful to draw a diagram of the stack structure and see how it expands and contracts as the routine is called recursively. Whilst this example demonstrates the power of assembly language, it also shows that, if you try to do complex things, the code to implement them can be difficult to understand!

The generalised parameter passing and local variable allocation techniques which we have described are useful when you are writing large programs with many subroutines or when you are building a subroutine library. For fairly small assembly language programs their generality can be confusing and it is better to adopt a simpler parameter passing technique.

However, we do recommend that you should avoid the allocation of fixed local variable space for subroutines. In many cases, you can use registers as local work areas and this is often the most efficient approach. In other cases, where this is impossible, you should use the stack as a local work area. You may either set up the U register as a pointer to this area or may use S register relative addressing to access local subroutine variables. These techniques are illustrated in some of the character string manipulation routines which are described later in this chapter.

6.2 CHARACTER STRINGS

We have described how BASIC arrays can be set up using the FCB, FDB, and RMB directives. These arrays can be accessed using index registers with the array length held in an accumulator register. Naturally, these arrays can be arrays of characters and this is one way of carrying out character manipulation in assembly language.

However, the use of fixed-length arrays to hold

character strings means that the decision as to the number of characters in a string must be made when the array holding the string is declared. In this respect, character arrays are not like BASIC'S character strings where the number of characters in a string may vary from 0 to 255. When this flexibility is required, it is not usual to implement character strings as fixed-length arrays.

In this section we describe how the assembly language programmer may set up variable-length character strings and we explain how various string manipulation operations can be implemented. In section 6.3 we provide listings of a package of subroutines which implement character string operations.

In order to implement variable-length strings the programmer must set aside a large data area for string storage where the actual characters making up the string are kept. The string name is associated with a 2-byte area which holds the address of the string characters within the string storage area.

The fundamental operations which are normally allowed on character strings are as follows:

- (1) Comparison
Character strings are compared for equality
- (2) Assignment
One character string is assigned to another
- (3) Catenation
Two character strings are put together (catenated) to form a longer string
- (4) Substring selection
Part of a character string (a substring) is selected
- (5) Length computation
The number of characters making up a string is computed

There are also other operations which may be carried out with character string operands such as determining the ASCII value of a particular character and converting numeric strings to integers and vice-versa.

Given that all character strings are to be stored in a common string storage area, the first decision that the programmer must make is how to represent strings so that the length of the string can be determined. All of the string operations listed above need to know the string length in order to operate correctly.

Probably the simplest variable-length string representation technique is to associate an explicit 'end-of-string' character with each string. This

character is catenated with the characters making up the string so that the storage space required for the string is the length of the string plus one byte. Usually the null byte, hexadecimal 00, is used as the string terminator. Therefore the string 'HI THERE' would be stored as 'HI THERE<NULL>' .

There are two advantages of using this technique of variable-length string representation.

- (1) There is no limit to the length of the strings which may be represented.
- (2) Strings whose length cannot be predicted can be stored in this way as the string modification (adding the null byte) is carried out after the entire string is known. This means that the technique is very useful for representing strings which are input from the keyboard or some other device. Obviously, the length of such strings is not known in advance.

The disadvantage of this representation technique is that string length determination requires a program to explicitly count the string characters until a null byte is detected. This takes time and when a program does a lot of character manipulation, this time penalty may be unacceptable.

An alternative technique for string representation is to hold the length of the string as the very first byte of the string. For example, the string 'HI THERE' would be stored as <8>HI THERE. This means length computation is very fast but has the disadvantages that the maximum string length is 255 characters and that the length of the string must be known in advance before it can be entered in the string store.

As character strings are represented as a 2-byte reference to the string store, the assignment of one character string to another is a very efficient operation. There is no copying of the string characters themselves. Assignment simply involves assigning one string reference to another. However, this can result in much wasted store. The reason for this is best illustrated by an example.

Assume that the variables STR1, STR2, and STR3 have been set up using an FDB directive and have been initialised to refer to strings as follows:

```
STR1 -> 'HI THERE'
STR2 -> 'WELCOME'
STR3 -> 'HELLO'
```

If STR1 is assigned to STR2, this means that STR2 now points to the string 'HI THERE' and the string 'WELCOME' is no longer referenced by anything. However,

the space occupied in the character store by this string cannot magically disappear so, if many string assignments are executed, the string store soon fills up with such inaccessible 'garbage'.

This is a general problem which is inherent in all systems where variable-length strings are allowed. The BASIC programmer has the advantage that the BASIC system has an in-built 'garbage collection' routine which finds all unreferenced strings in its string store and marks the store which they occupy as reusable. Garbage collection is a fairly complex operation and the interested reader should refer to a computer science textbook which covers data structures for a description of various garbage collection algorithms.

Rather than discuss garbage collection, we describe how routines can be written to allocate and deallocate space in the string storage area so that the amount of garbage is minimised. The first routine described below is called GETSP. This takes one parameter, say *n*, and returns an address in the string storage area of *n* consecutive unused bytes. The second routine below is FREESP, which is called after string assignment, to mark a group of bytes as being available for re-allocation.

Let us assume that the string storage area is called HEAP and is set up using the following directive:

```
HEAP   RMB 4096   ; String storage area
```

Furthermore, let us assume that we use an explicit length byte at the start of each string. If this byte has a value between 0 and 254, this is taken as the string length. If the length byte is 255, the following two bytes hold a number which is the number of unused bytes in that area and therefore available for string allocation.

Figure 6.3 shows part of HEAP with intermingled character strings and free space. Initially, HEAP is set up so that the very first byte (byte 0) is 255 and bytes 1 and 2 hold the 16-bit integer 4096 indicating that the entire storage area is available for allocation. The routine GETSP starts at the beginning of HEAP searching for a byte whose value is 255. When such a byte is found, GETSP checks if the number of free bytes available is enough to satisfy its request.

If so, GETSP claims what it needs from this free space and marks the remainder as free. If the free space is not sufficient, GETSP goes on to find the next byte whose value is 255. If no free space is found before the end of the string storage area, GETSP returns an error indicator showing that it is unable to satisfy the request for space.

5	H	E	L	L	0	19	W	E	L	C	0	M	E	
T	0		B	R	I	G	H	T	0	N	255	0	12	
								4	M	A	R	Y	255	0
3	13	P	I	Z	Z	A		P	A	R	L	0	U	R
255	0	10								0	255	0	19	

Fig. 6.3 String storage area organisation

The code for the routines GETSP and FREESP is provided in section 6.3. For the moment, let us assume that they are available and have the following specifications:

```
* GETSP - gets space on heap
*
* Register input B - number of bytes required
* Register outputs Y - pointer to space requested
*                   CC.V = 0 if no space available
*                   CC.V = 1 if request satisfied
*
* FREESP - returns free space to heap
*
* Register input X - address of space to be freed
* Register output CC.V = 0 if invalid address
*                   CC.V = 1 if space freed
```

Given these routines, the initialisation of strings can be implemented as shown below. Assume that a string, terminated by a null byte, has been read into an input buffer area called INBUF. The routine STINIT takes the address of INBUF as its parameter in register X and returns in register Y the address of the initialised string on the heap. The assembly code for this routine is:

```
* STINIT - Initialise a string
*
* Register input X - input buffer address
* Register outputs Y - string address in heap
*                   CC.V = 0 if error
*                   CC.V = 1 if no errors
```

```

STINIT   PSHS X,A,B,PC      ; Restore and return
          CLRB              ; B is counter
*
*          holding length of string to be
          initialised
          TFR X,Y           ; Save value of X
STCNT    LDA ,Y+            ; Get string byte
          BEQ FSPCE        ; If null byte, stop count
          INCB             ; Otherwise, count it
          BRA STCNT
FSPCE    INCB              ; To account for length byte
          BSR GETSP        ; get space
          BVC XIT         ; No space found - error
          DECB            ; No of characters in string
          STB ,Y+         ; Store length
          BSR CPSTR       ; String copy -see examples
          ORCC #2         ; Set success flag
          LEAY -1,Y       ; To point at length byte
XIT      PULS X,A,B        ; Restore and return

```

Program 6.5 STINIT - string initialisation

Further examples illustrating string manipulation techniques are provided in the following section.

6.3 STRING MANIPULATION ROUTINES

This section is entirely taken up with listings of routines which carry out string manipulation. All the examples here are written in a position-independent way and may readily be incorporated with your own programs.

```

* CHKHP - check string validity
*
* Register input X - string address
* Register output CC.V = 1 if string in heap
*                   CC.V = 0 if not in heap
*
CHKHP    PSHS X            ; Save register
          LEAX HEAP,PCR    ; Heap start
          CMPX ,S          ; comparison
          BHI HPERR       ; Input address <• heap start
          LEAX HEAPEND,X  ; Heap end
          CMPX ,S          ; comparison
          BLO HPERR       ; Input address > heap end
          ORCC #2         ; Set CC.V
          BRA XIT1
HPERR    ANDCC #$FD       ; CC.V = 0
XIT1     PULS X,PC        ; Restore and return

```

Program 6.6 CHKHP - check string address validity

```

* CPSTR - copy string characters
* Register inputs X - source string address

```

* Y - destination string address
 * B - string length
 *

```
CPSTR   PSHS X,Y,A,B      ; Save registers
        TSTB             ; Check for zero length
CPLOOP  BEQ XIT2          ; Check if finished
        LDA ,X+          ; Get character
        STA ,Y+          ; and copy it
        DECB            ; B is counter
        BRA CPLOOP
XIT2    PULS X,Y,A,B,PC   ; Restore and return
```

Program 6.7 CPSTR - copy characters

* GETSP - get space for string
 *
 * Register input B - number of bytes required
 * Register output Y - string address
 * CC.V = 0 if request fails
 * CC.V = 1 if request satisfied
 * Uses first-fit algorithm, ie, returns first area
 * large enough to satisfy request. Returns excess
 * space as free if space found > space requested

```
GETSP   PSHS A,B,X,U      ; Save registers
        TFR S,U          ; U is pointer to locals
        LEAX HEAPEND,PCR ; 1st local = U-2
        CLRA            ; U-4 is 16-bit length
        PSHS X,A,B       ; Locals onto stack
        LEAY HEAP,PCR    ; Initialise to heap start
FFREE   CMPY -2,U        ; At heapend?
        BHS NTFND       ; Yes, no space available
        LDA ,Y          ; Check if free area
        CMPA #255       ; by comparing with 255
        BEQ SPFND       ; If so, space found
        LEAY 1,Y        ; Otherwise increment Y
        BRA FFREE       ; and keep looking
SPFND   LDD 1,Y          ; Pick up free area length
        CMPD -4,U       ; Compare with length needed
        BHS LENOK       ; We have enough
        LEAY 3,Y        ; No, look for next free
        BRA FFREE       ; area on heap
```

* Now check if too much space. Don't return
 * an extra 1 or 2 bytes as they are unusable

```
LENOK   LDD -4,U        ; Space requested
        ADDD #2         ; If D + 2 >= that available
        CMPD 1,Y       ; don't return space
        BHS EXITOK     ; and exit
        LDD 1,Y        ; get space available
        SUBD -4,U      ; subtract space requested
        PSHS A,B       ; and save on stack
        LDB -3,U       ; B = 8 bit length
        LEAX B,Y       ; start of free string
        LDA #255       ; Free indicator
```

```

        STA ,X+           ; and mark byte free
        PULS A,B         ; get free string length
        STD ,X          ; and store it
        BRA EXITOK      ; and exit
NTFND   ANDCC #$FD      ; Error indicator
        BRA XIT3
EXITOK  ORCC #2         ; No errors
XIT3    LEAS -4,S       ; Discard local space
        PULS A,B,X,U,PC ; Restore and return

```

Program 6.8 GETSP - get string space

* FREESP - free space on heap

* Register input X - address of space to be freed

* Register output CC.V = 1 if space freed

* CC.V = 0 if invalid input

```

*
FREESP  PSHS A,B,X,Y    ; Save registers
        BSR CHKHP      ; Is input valid?
        BVC EEXIT      ; No, error return
        LDB ,X         ; String length
        INCB          ; To get actual no of bytes
        LDA #255       ; Free space indicator
        STA ,X         ; Mark string free
        CLRA          ; and store 16-bit
        STD 1,X        ; free string length
        LDA #255       ; See if following string
        CMPA B,X       ; is free
        BNE LKLAST    ; No, try preceding string
        LEAY B,X       ; yes, so join strings
        BSR JOIN
LKLAST  TFR X,Y         ; Find preceding free string
FLOOP   CMPA , -X      ; Is byte free
        BEQ CHKJN     ; Yes, can it be joined
        BSR CHKHP     ; At heap start?
        BVC XIT7      ; No preceding free string
        BRA FLOOP
CHKJN   LDD 1,X        ; Length of free string
        STD , --S     ; Stack it
        TFR X,D
        ADDD ,S++     ; D = address+length
        PSHS Y
        CMPD ,S++     ; Are strings adjacent
        BNE XIT7      ; No, return
        BSR JOIN      ; Yes, join them
XIT7    ORCC #2        ; Set CC.V
        BRA END7
EEXIT   ANDCC #$FD     ; Clear CC.V
END7    PULS A,B,X,Y,PC
*

```

* JOIN - join adjacent free segments

* Register inputs X,Y - addresses of areas to be

* freed

```

JOIN      PSHS A,B
          LDD 1,X           ; Length of 1st area
          ADDD 1,Y          ; Length of 2nd area
          STD 1,X           ; Store total
          CLR ,Y            ; Get rid of free indicators
          CLR 1,Y
          CLR 2,Y
          PULS A,B,PC

```

Program 6.9 FREESP - free string space

```

* CMPSTR - compare strings for equality
* Register input X - string 1
*           Y - string 2
* Register output CC.Z = 1 if strings equal
*           CC.Z = 0 if not equal
*
CMPSTR    PSHS A,B,X,Y      ; Save registers
          LDA ,X+           ; Length of string 1
          CMPA ,Y+          ; must be same as length 2
          BNE CMPXIT        ; If not, exit, CC.Z=0
          TSTA              ; Check for 0 length
CMPPLP    BEQ CMPXIT        ; A = 0, so all done, CC.Z=1
          LDB ,X+           ; Get character
          CMPB ,Y+          ; and compare
          BNE CMPXIT        ; Not the same, CC.Z=0
          DECA              ; Yes, decrement length
          BRA CMPPLP        ; and continue comparisons
CMPXIT    PULS X,Y,A,B,PC  ; Restore and return

```

Program 6.10 CMPSTR - compare strings

```

* STRCAT - concatenate strings
*
* Register inputs X - string 1
*           Y - string 2
* Register outputs Y - new string
*           CC.V = 1 - no errors
*           CC.V = 0 - error
*
STRCAT    PSHS X,A,B
          BSR CHKHP         ; Check 1st string
          BVC XIT8         ; Invalid, abort
          EXG X,Y
          BSR CHKHP         ; Check 2nd string
          BVC XIT8         ; Invalid, abort
          LDB ,Y            ; work out length
          ADDB ,X           ; of new string
          BVS EEXIT         ; Too long(overflow), abort
          CMPB #255         ; 255 also too long
          BEQ EEXIT
          STX ,--S          ; Stack string addresses
          STY ,--S
          INCB              ; Total space needed incl.

```

```

BSR GETSP          ; length byte. Get space
BVC XIT8           ; No space, abort
DECB              ; New string length
STB ,Y+           ; stored as 1st byte
LDX ,S++          ; Get source address
LDB ,X+           ; and its length
BSR CPSTR         ; and copy characters
LEAX -1,X         ; Then free space
BSR FREESP
LEAY B,Y          ; Update destination
LDX ,S++          ; Get source address
LDB ,X+           ; and its length
BSR CPSTR         ; and copy characters
LEAX -1,X         ; Then free space
BSR FREESP
ORCC #2           ; Set CC.V
BRA XIT8
EEXIT            ANDCC #$FD      ; Error indicator
XIT8            PULS A,B,X,PC

```

Program 6.11 STRCAT - catenate strings

```

* SUBSTR - select substring
*
* Register inputs X - source string address
*                   A - substring length
*                   B - offset from string start
* Register outputs Y - new address or error number
*                   CC.V = 1 - no errors
*                   CC.V = 0 - error
*
SUBSTR  PSHS X,A,B      ; Save registers
        BSR CHKHP      ; Is string valid?
        BVS STROK      ; yes, next check
        LDY #0         ; error type indicator
        BRA EEXIT1     ; error exit
STROK   INCB           ; To get offset from 1st char.
        LEAY B,X       ; Substring address
        PSHS Y         ; Stack it
        LDB ,X        ; Total string length
        INCB          ; To account for length byte
        LEAY B,X       ; End of string address
        CMPY ,S        ; With substring address
        BLS INDXOK     ; If invalid index
        LDY #1        ; Index error = 1
        BRA EEXIT1
INDXOK  LDU ,S         ; Substring address
        LEAU A,U       ; Add length
        PSHS U         ; and stack it
        CMPY ,S++     ; Compare with end of string
        BLS LENOK     ; is index + length valid?
        LDY #2        ; No, length too long
        BRA EEXIT1
LENOK   INCA          ; To get number of bytes for

```

```

        TFR A,B           ; getspace parameter
        BSR GETSP        ; Get the space
        BVS GSPOK
        LDY #3           ; No space available error
        BRA EEXIT1
GSPOK  PULS X           ; Source address
        DECB            ; For string length
        STB ,Y+         ; New string length
        BSR CPSTR       ; Now copy characters
        ORCC #2         ; No errors
        BRA XIT5
EEXIT1 ANDCC #$FD       ; Indicate error
XIT5   PULS X,A,B,PC   ; Restore and return

```

Program 6.12 SUBSTR - select substring

6.4 POSITION-INDEPENDENT CODE

One of the problems which can arise when you try to use machine code routines which have been written by other people is that these routines make assumptions about the contents of particular memory locations which you have used for other things. What has happened is that the operation of the routines depends on particular instructions and/or data residing at fixed addresses and, if these instructions/data are not at these addresses, the routines will not work.

Routines like this are called 'position dependent' and often cause many problems for the assembly language programmer. However, it is possible to write 'position independent' code which executes correctly irrespective of where it is loaded into the machine memory. If you are building a library of subroutines or writing a program which may run on other machines, you should always write position-independent code.

Position-independent code (PIC) is code that executes in the same way regardless of where it resides in memory. In other words, if it is located at a different address from that which it was originally assembled, it will still execute correctly. To produce position-independent code for the Dragon, you must adhere to a single fundamental rule:

All addresses which you use in your program should be relative rather than absolute addresses.

In general, it is best to write your routines so that addresses are all relative to PC but it is also possible to use the direct addressing mode of the M6809 in the production of PIC. For the meantime, however, we shall concentrate on how to produce PIC by using PC-relative addressing.

We have already seen examples of PC-relative addresses as all the M6809 branch instructions refer to

the destination address as an offset from the current value of the program counter. Therefore, even if the code is moved (relocated) to some other address, the relative distance between the branch instruction and its destination remains the same. However, you cannot cheat by adding or removing machine code instructions without re-assembly. If you do so, the program will not work as the relative distance specified in the branch instruction will be incorrect.

In early microprocessors, the production of PIC was often difficult because relative branch instructions only allowed an 8-bit offset thus restricting the relative branch to the range -128 -> 127. However, no such problem exists in the M68C9 as long branch instructions allowing offsets from 32767 to -32768 may be used. In fact, if you wish to use some of the examples discussed in earlier chapters in combination with the examples in this chapter, you may have to change some of the BSR instructions to LBSR instructions as the subroutine code may be located more than 127 bytes away from the subroutine call.

As well as addressing instructions in a position-independent way, it is also essential that data are also addressed using the PC-relative addressing mode. Although we introduced this addressing mode in Chapter 2, our examples so far have mostly used direct, extended or indexed addressing. The reason for this is that we felt that the introduction of PC-relative addressing was peripheral to the concepts illustrated in the examples.

Recall that the M6809's PC-relative addressing mode uses the program counter as an index register and adds either an 8-bit or a 16-bit offset to it. The table below shows examples of how data can be addressed in a position-independent way using PC-relative addressing. Assume that TABLE, WORD, and DATA are storage locations set up using an FCB or RMB assembler directive.

Non-PIC	PIC
LDX STABLE	LEAX TABLE,PCR
LDX WORD	LDX WORD,PCR
STA DATA	STA DATA,PCR

Notice how easy it is to write code in a position-independent way. Instead of referring to the absolute symbolic address, all you have to do is to tell the assembler that PC-relative addressing is to be used. The assembler works out the correct displacement from the instruction position and generates the appropriate postbyte and offset.

The only instructions which cause any real difficulty are those which use 16-bit immediate

addressing where the 16-bit value in the instruction refers to an absolute address. To load such addresses in a position-independent way, the LEA instruction rather than the LD instruction is used. Therefore, rather than saying LDX STABLE to load the address of TABLE into register X, this should be written LEAX TABLE,PCR.

However, other instructions such as CMP which might also use immediate values which are addresses do not have position-independent forms. This means that when a 16-bit register is to be compared with an immediate value representing an address, we have to make use of a temporary location on the stack.

For example, consider the following fragment of non-PIC code which is often found in programs which look up tables of values.

```

        LDX #TABLE      ; Set up base address of table
LOOP    ....
        Code to look
        up table
        CMPX #TABEND   ; is table completely scanned
        BNE LOOP

TABLE   FCB <<table data values)
TABEND  EQU *          ; table end

```

In this example, TABLE and TABEND represent absolute addresses and, if relocated without reassembly, this code would not execute properly. In order to make this code position independent, we must ensure that all absolute addresses are eliminated. We do this by using the LEA instruction to compute an address and we then store this address where it may be accessed and compared. We need a temporary location for the absolute address and, as always, the best place to allocate temporary store is on the stack.

We might, therefore, write the above example in a position independent way as follows.

```

        LEAX TABEND,PCR
        PSHS X          ; Stacks address of TABEND
        LEAX TABLE,PCR
LOOP    ....

        CMPX ,S        ; Compare X with top stack
        BNE LOOP
        LEAS 2,S       ; Discard top stack element

```

In general, when you are writing your own routines you should always try and use PC-relative addressing so that PIC is generated by the assembler. However, if you are making use of routines built into the BASIC system, such as the input and output routines INCH and

OUTCH described in Chapter 5, PC-relative addressing should not be used.

The reason for this is that these routines always reside at fixed locations and if you relocate your own program, the system routines do not move with your program. Therefore, you should always use jump rather than branch instructions to reference these system routines.

For example, to reference the input routine at address 8006, you might write the following code:

```
INPUT    EQU $8006
        . . .
        JSR INPUT
```

It would be quite incorrect to say LBSR INPUT as relocating your code would cause the displacement built into the branch instruction to be incorrect. Naturally, the same applies to memory areas which have a dedicated function, such as the BASIC screen area. This starts at absolute address 400, so LD rather than LEA instructions are used to pick up that address.

6.4.1 Jump tables

The only real problem associated with PIC arises when some other program is assembled and uses PIC routines. Naturally, the addresses of these routines are assembled into the program and, if the routines are relocated, these addresses will be wrong. After relocation, it is necessary to modify the program to reflect the new, relocated addresses and this seems to negate some of the advantages of producing PIC.

In order to avoid a great deal of tedious address modification, an addressing technique can be used which isolates the necessary changes so that only a single table need be changed. This technique is based around the idea of so-called 'jump tables' or 'vector locations'.

A jump table contains, at known positions, a link to the actual addresses of routines and data used by a program. If these addresses change, only the jump table need be modified to reflect the new addresses. There is no need to change the program which refers to these addresses through the jump table.

Where routines are addressed, the jump table is usually made up of jump or branch instructions (hence the name) which immediately jump to the addressed routines. We shall see shortly how such a table, which is called a direct jump table, may be set up.

When data are referenced via a jump table, the table locations do not contain instructions but merely hold the address of the referenced data. The data item can be accessed using indirect addressing. Hence, this type of jump table is often termed an indirect jump

table. Of course, there is no reason why the data in such a table should not be subroutine addresses. The actual routines would then be called using a JSR instruction with the indirect addressing mode.

Jump tables are the mechanism which provides access to the BASIC I/O routines. In fact, there are two jump tables referencing these routines - a direct jump table starting at address 8000 and an indirect jump table starting at address A000.

As an example of how these tables can be used, consider the character input routine discussed in Chapter 5. In the direct jump table, address 8006 holds a jump to this routine whereas the first location in the indirect jump table (A000) is set up with the address of the input routine.

If we wish to use the direct jump table, the following instruction is used to call this input routine:

```
JSR $8006
```

On the other hand, if the indirect jump table is used, indirect addressing must be used to reference the input routine:

```
JSR ($A000)
```

The jump tables for these BASIC I/O routines are set up at known locations but if you envisage that other programs will use your routines, it is a straightforward matter to set up your own jump tables.

The skeleton example below shows how direct and indirect jump tables may be defined by the assembly code programmer.

```
SUB1
    <code for subroutine 1>
SUB2
    <code for subroutine 2>
SUB3
    <code for subroutine 3>
*
*   Now set up an origin for the jump table
*
                ORG $1000
SUB1V          JMP SUB1
SUB2V          JMP SUB2
SUB3V          JMP SUB3
*
*   If an indirect jump table is required it
*   might be set up as follows:
*
SUB1V          FDB SUB1
SUB2V          FDB SUB2
```

SUB3V FDB SUB3

This is a simple way to set up jump tables but the disadvantage with this technique is that the addresses filled in the jump table are those known when the program is assembled. They are called 'static addresses'. If the program is relocated, these addresses remain as they were and are therefore incorrect. What is needed is a technique which allocates addresses to a jump table immediately before the program runs. That is, the jump table must be set up dynamically each time the program is executed.

To calculate the addresses at run-time requires the use of initialisation code which fills in the jump table addresses. The following initialisation code shows how this can be achieved.

```
INIT        LEAX SUB1,PCR
            STX SUB1V+1            ; SUB1V+1 because the
*                                  JMP opcode is at SUB1V

            LEAX SUB2,PCR
            STX SUB2V+1
            LEAX SUB3,PCR
            STX SUB3V+1

            ORG $1000            ; Jump table address
SUB1V       JMP $0000
SUB2V       JMP $0000
SUB3V       JMP $0000
```

We leave it as an exercise for the reader to work out how to initialise an indirect jump table dynamically.

Normally, the INIT routine is the very first routine in a program as it is essential that its address is known in order that it may be called to set up the jump table. Placing INIT at this position also means that the program can be initiated from BASIC once CLOADMed by using the EXEC command. There is no need to specify an address for EXEC.

The use of an initialisation routine opens up the possibility of using an alternative technique of producing position-independent code. This technique relies on all addresses being direct addresses with the actual address computed by adding the contents of DP to the address specified in the instruction. In other words, the instruction address is actually a DP-relative address.

In order to produce PIC code using direct addressing, DP must be set up dynamically at the start of program execution. The INIT routine must search for an available page in memory and assign its address to the direct page register. You might wish to explore the possibilities of this technique but be warned that the BASIC system keeps many pages for its own use and

assumes that they will not be used by the programmer. You have to be very careful about saving and restoring the value of the DP register and it is our opinion that the use of PC-relative addressing is a better way of producing position-independent code.

6.5 COMBINING ASSEMBLY LANGUAGE WITH BASIC

A disadvantage of assembly language programming is that it is difficult to write and test low-level language programs even when strict rules of programming are adhered to. This is in contrast to BASIC programs which, because of the way in which BASIC is implemented, are easy to test. It is simple to print out the values of variables as the program executes or to break in and inspect variable values that you think might be wrong. Ideally, we would like this flexibility but with the speed and power of assembly language.

There is no such ideal system but, in many cases, it is possible to call assembly code routines from BASIC programs thus using high and low level programming in the most productive way. It is a fact that most programs spend most of their time executing a relatively small proportion of the total program code. The speed of BASIC programs can be significantly increased by identifying execution-intensive sections and replacing these by machine code equivalents. In this way, the majority of the program made up of user prompts, print statements, etc. can remain in BASIC with only time critical sections programmed in assembly language.

The easiest way to incorporate machine code routines in a BASIC program is to use BASIC'S EXEC statement. The EXEC statement takes an address as a parameter and transfers control to the code residing at that address. It is used as follows:

```
EXEC <address>
```

In actual fact, the address operand, which must lie in the range 0000 to FFFF, in the EXEC statement is optional. If it is present, the machine code routine at that address is executed with control returned to BASIC after a RTS or PULS PC instruction is executed. If the address is omitted, EXEC consults a jump table (the EXEC vector) to find the address of the code to be executed.

The EXEC vector is located at address 9D and is made up of a single word only. Therefore, the memory locations 9D and 9E should contain the address of the code to be EXECed. Initially, the EXEC vector is set up to contain the address of an error routine which explains why the message '?FC ERROR' is output when an EXEC without a parameter is used as the first EXEC in a

program. If an address is specified in an EXEC call, that address is filled into the EXEC vector with the result that subsequent EXECs without an address parameter call the machine code at that address.

An alternative way to set up the EXEC vector is via the CLOADM command.

```
CLOADM "Name"
```

```
EXEC
```

This instruction sequence sets up the EXEC vector to refer to the execution address of the machine code program called "Name" which has just been loaded. The EXEC instruction then transfers control to this code.

The main advantage of EXEC is its simplicity and the fact that it can be used to invoke any number of machine code routines. The main disadvantage with EXEC is that any routine parameters must be passed in memory locations and the programmer must POKE these parameters into known locations before the EXEC call. Similarly, the results of executing the machine code routine must be in known locations and can only be retrieved using PEEK.

An alternative way to invoke machine code routines, which permits parameter passing, is to make use of the USR call. The number of USR calls available to the BASIC/machine code programmer is restricted to ten and these are named USR0 to USR9. USR calls do not take an explicit address but transfer control to the address which the programmer has previously associated with that USR call.

The addresses to which particular USR calls should transfer control are set up using a DEF USR statement. This has the general form:

```
DEF USRn = address
```

The number n must be a single digit in the range 0 to 9 and the address must lie in the range 0 to FFFF. The general form of the USR call itself is:

```
USRn(<argument>)
```

Executing a call of USRn causes control to be transferred to the address specified in the corresponding DEF USRn statement. Although the definition of the USR call function states that the name USR should be followed by a single digit from 0 to 9, readers who try to call USR in this way will find that all USR calls actually result in a call to USR0. This is due to an error in the BASIC system which, fortunately, can be circumvented very easily.

The bug in the BASIC system causes the interpreter to skip the digit so that USR0 is taken to be the same

as USR1, USR2, etc. As BASIC takes a USR call without a parameter to be equivalent to USR0, the effect of the bug is to make all USR calls default to USR0.

Rather than call a USR call as USR1, USR6, USR9, etc., the digit indicating which USR call is to be used should be padded with an extra zero. Therefore, to call USR1, you must actually write USR01, to call USR6, you must write USR06, etc. Obviously, this is not necessary for USR0 but for reasons of consistency it is probably better to call this as USR00.

A USR call from BASIC is treated like a BASIC function so that it is used as part of an expression and should return a value to the BASIC program. Examples of USR calls are:

```
10 DEF USR0 = &H1000 : DEF USR1 = &H2000
20 A = USR0(A) : ' Transfers control to &H1000
30 IF USR01(0) = 0 THEN B = B + 1
```

If a USR call is used without first defining the address it refer to, the USR call will cause a message '?FC ERROR' to be printed. Like the EXEC statement, each USR call has an associated vector which contains the address of the entry point of the machine code routine to be executed. The USR vector is initially set up to refer to the error routine which prints the '?FC ERROR' message. When a DEF USR statement is used, this fills in the address in the appropriate vector.

The table below lists the vector addresses associated with each USR call.

USR Call	USR Vector
USR0	134:135
USR1	136:137
USR2	138:139
USR3	13A:13B
USR4	13C:13D
USR5	13E:13F
USR6	140:141
USR7	142:143
USR8	144:145
USR9	146:147

If you are trying to link machine code and BASIC for the first time, we recommend that you experiment with the technique by using EXEC rather than USR calls. Unfortunately, to set up USR call parameters requires knowledge of how BASIC represents numbers and strings. We therefore return to the use of USR calls in Chapter 9 after BASIC'S data representation has been described.

Chapter 7

Graphics programming

One of the greatest advantages of assembly code programming, its total flexibility, is also one of its most serious drawbacks as the programmer has to concern himself with every detail of the problem. One area in particular where this lack of support is very evident is in graphics and animation.

The problem becomes very obvious if the would-be animator has relied on the graphics facilities provided in Extended Color BASIC and has come to expect such facilities when designing and writing graphics programs. However, the major disadvantage of BASIC programming is its inherent slowness and it is in graphics applications that this is most evident. Only the simplest of games, for example, with minimal movement can be programmed in BASIC if they are to present a challenge to the player.

A very large part of the Dragon's BASIC system is dedicated to providing graphics facilities and it is not an easy task to duplicate those features as assembly code routines. Nevertheless, if speed is required, some graphics programming must be carried out in assembly code but the programmer should, as far as possible, make use of BASIC for those parts of his program which are not time critical.

In general, a good graphics programming strategy is to develop the complete program using BASIC'S facilities and to iron out program bugs at this stage. This will probably result in a system which is far too slow but you may then replace BASIC routines with assembly code routines to speed up your system.

It is seldom necessary to duplicate the BASIC routines exactly unless they are components of other routines. Rather, it is usually possible to make all sorts of simplifications and later in the chapter we look at how to design, code and animate screen patterns. The chapter also discusses, in some detail, the Dragon's graphics hardware and describes the different graphics modes available to the programmer.

Firstly however, we describe in general terms, how the Dragon's display system is organised. As in most personal computer systems, the display system on the Dragon is memory-mapped. This means that an area of memory is scanned 50 or 60 times per second, depending

on the local mains frequency, and the contents of that area are translated by special hardware to a standard TV signal which may be displayed on a domestic television set.

The machine allocates a 512 byte area of memory for an alphanumeric display and it is this area which is used to display BASIC program text as it is input, and program results as they are output. The Dragon's alphanumeric display is organised as 16 lines with 32 characters per line. We show later that this display area can also be used as a low-resolution graphics area. This text segment is always allocated at address 400 in memory so locations 400-5FF are dedicated to the alphanumeric display.

The memory dedicated to graphics, that is, the display of pictures rather than text, is organised into graphics segments of 512 bytes each. In full graphics mode, a minimum of 2 segments must be allocated but there is no inherent maximum number of graphics segments. Obviously, however, the maximum number of such segments is limited by the amount of free memory available to the graphics programmer. These graphics segments are usually allocated from address 600 onwards, that is, immediately after the BASIC text segment. The BASIC system organises these graphics segments into 'pages' of 1536 bytes and a maximum of 8 pages is available to the BASIC programmer.

In order to display characters, the display screen is considered as a two-dimensional array of 'picture elements' or pixels. The more pixels on the screen, the finer detail which can be resolved and the Dragon compares favourably with other personal computers in this respect. The Dragon's display is made up of 256 horizontal pixels by 192 vertical pixels. The Dragon's graphics hardware provides various graphics modes where the screen is considered as a matrix of elements. Each element is made up of a single pixel at the highest resolution or consists of an array of pixels. Depending on the resolution chosen, this array can vary from 2 by 1 pixels to 12 by 8 pixels.

In a memory-mapped graphics system, all information about a particular screen element must be encoded in memory. This means that the pixel settings and colours must be held in memory locations so there is a trade-off between display resolution and the number of colours available to the programmer. High-resolution, multi-colour displays require a great deal of memory to encode the screen information so the Dragon's graphics system limits the number of colours available when resolution graphics are used.

7.1 GRAPHICS DISPLAY HARDWARE

The Dragon's graphics display hardware is made up of 3

microchips. Working in combination, these chips extract information from the system's memory and display that information on a standard television screen. The so-called 'Video RAM' is the memory area which is devoted to the display and the contents of this memory area determine what is actually displayed on the user's screen.

The chips making up the graphics system are the Video Display generator (VDG - 6847), the Synchronous Address Multiplexor (SAM - 6883), and a Peripheral Interface Adapter (PIA - 6821). The interconnections of these chips is shown in the system block diagram in Figure 1.2. In spite of the fact that the names of these chips sounds daunting, it is fairly easy for the assembly language programmer to control these devices. Each of them, and the Video RAM, is described below.

7.1.1 The VDG chip

The video display generator (VDG) chip is the main component of the Dragon's graphics system. As the name suggests, it generates the video signals that are input to the user's television set to provide the screen display. For those readers with experience in electronics, a complete description of this chip is provided in Appendix 3.

However, you do not need experience in electronics to understand how to control this chip. All you must understand is that the chip has a set of control lines which may be in one of two states representing the binary values 1 and 0. When a line represents a 1, we say that it is HI, when it represents a binary zero, we say that the line is LO. Control signals can be generated by writing information to specific memory addresses.

The VDG chip determines the graphics capabilities of the Dragon and it does so by providing a selection of modes of operation. These modes dictate the resolution of the display, the number of display colours, the actual colours displayed, etc. In all, there are a total of 14 different display modes:

- (1) Four alphanumeric modes
- (2) Two Semigraphics modes
- (3) Four colour graphics modes
- (4) Four resolution graphics modes

The PMODE statement in BASIC allows some of these modes to be provided but not all of them are available to the BASIC programmer. However, the assembly language programmer may use all of the display modes by directly configuring the VDG chip. Each of these modes is

described in a separate section later in this chapter.

The VDG chip has eight control lines which are used to select the mode of the display. The table below shows the function and the names of each of these control lines.

Control line	Function
A/G	Set LO to indicate Alphanumeric HI to indicate Graphic mode
A/S	LO to indicate Alphanumeric HI to indicate Semigraphic mode
INT/EXT	Selects between internal (LO) and external (HI) character generator ROM.
GM0,GM1,GM2	Selects the graphics mode
CSS	Selects between the two colour sets
INV	Selects between inverse and normal video

The mode control lines, A/G, INT/EXT, GM0, GM1, GM2, and CSS, are connected to the PIA chip as described in the following section. The desired mode may be set up by setting the appropriate bit pattern in the PIA's data register. This causes the appropriate control signals for the VDG chip to be generated.

Although six of the VDG control lines are set up via the PIA, there are only five output lines from the PIA to the VDG chip. There is no need for six lines as the INT/EXT and GM0 input lines share a single PIA output line. When GM0 is needed in graphics mode, the value of INT/EXT is irrelevant and when the value of INT/EXT is actually needed in alphanumeric/Semigraphics mode, the value of GM0 is not used.

The remaining VDG control lines A/S and INV are connected to two of the RAM data lines, D6 and D7. These lines can therefore be set on a character by character basis in the alphanumeric/semigraphic modes.

The VDG chip has the capability of generating eight colours but, when colour graphics modes are used, memory restrictions limit the number of colours which may be displayed to four. The eight colours are therefore separated into two colour sets and the CSS control line on the VDG indicates which colour set is in use.

The colours in each colour set are:

Colour set 1	Colour set 2
Green	Buff
Yellow	Cyan
Blue	Magenta
Red	Orange

When the memory bits defining an element are set, this means that the element is 'on' and it is displayed in

colour. When the associated bits are unset, the element is off and is displayed as black.

7.1.2 The peripheral interface adaptor

The PIA is an example of a general-purpose programmable interface device which is used to interface the M6809 processor to other devices. We describe the operation of the PIA in Chapter 8 as it plays a very important role in input/output programming.

The block diagram of the Dragon in Figure 1.2 shows that the system contains two PIA chips. The PIA used to control the VDG chip is PIA1 and, by setting the appropriate bits in the PIA's B-side peripheral data register, control signals for the VDG chip can be generated.

As the M6809 uses memory-mapped addressing, this data register is set by writing bit patterns to the appropriate memory address. PIA1 is addressed via memory locations FF20 through to FF23 with the B-side peripheral data register located at location FF22. We might therefore set up the VDG inputs as follows:

```
LDA <VDG input state>
STA $FF22
```

In fact, only bits 3 to 7 of this register are used to set the VDG control lines with bits 0 to 2 used for other purposes by the Dragon. The values of these bits are irrelevant for graphics programming. The table below shows the association of bits in the PIA register and VDG control lines.

Bit 3	CSS
Bit 4	GM0
Bit 5	GM1
Bit 6	GM2
Bit 7	A/G

7.1.3 The video RAM

Whilst it is the VDG chip which determines how data is displayed on the user's screen, it is the contents of the video RAM which specifies what is displayed. Remember that the display is made up of 256 by 192 pixels and the contents of the video RAM determine which pixels should be displayed and the colour of displayed pixels.

The VDG continually scans the video RAM and uses the data there to build up an image on the screen. Therefore, by changing a data byte in the video RAM, the programmer can change the pixels in the corresponding screen position. The resolution of the display is determined by the number of pixels affected when a single data byte of video RAM is modified.

7.1.4 The synchronous address multiplexor - SAM

The SAM chip has been specifically designed to provide the necessary control and timing signals for the M6809, the VDG chip, and the video RAM. Much of this information is of no relevance to the programmer but some aspects of the operation of the SAM chip are important. We concentrate on these aspects in this section rather than describe the SAM chip in detail.

Three bits in the SAM control register are used to set the appropriate display mode. These bits should be set to the same value as bits 3-5 in the VDG control register. The SAM control register is memory mapped at address FFC0 and occupies the address range FFC0 to FFDF. As the control register is 16 bits wide, why are 32 bytes allocated in memory to that register?

The 16-bit control register maps onto the 32-bit range FFC0 to FFDF so that each register bit is represented by two memory bytes which have adjacent even and odd values. Therefore bit 0 in the control register is represented by FFC0/1, bit 1 by FFC2/3, bit 2 by FFC4/5 etc. In order to clear a particular bit, a write operation to the even address is carried out, and to set a SAM control register bit you must write to the associated odd address. This technique of setting and unsetting the control register bits is the reason why 32 bytes are allocated to a 16-bit register.

The SAM control register bits which indicate the current graphics mode are the bottom three register bits termed V0, V1, and V2. These have associated addresses FFC0/1, FFC2/3 and FFC4/5. To set up the graphics mode required, you must carry out the requisite write operations to these addresses.

As well as these mode control bits, there are seven other SAM control register bits (bits F0-F6) which are used to indicate the base address in memory of the graphics segments used for the video RAM. The table below shows the association between these SAM control register bits and memory bytes:

F0	FFC6/7
F1	FFC8/9
F2	FFCA/B
F3	FFCC/D
F4	FFCE/F
F5	FFD0/1
F6	FFD2/3

The 7-bit value in the SAM control register is multiplied by 512 to compute the base address of the graphics segments used. This is the reason why graphics segments always have a base address which is a multiple of 512 and why they are always 512 bytes long.

Because the VDG and SAM chips must operate in tandem, they are normally set up in the same mode so

that signal timings, etc. are compatible. If set up in different modes, the system will produce garbage except when the VDG chip is in alphanumeric mode and the SAM chip is in one of the colour graphics modes. In this case, extra Semigraphics modes are available and these are described in section 7.6.

7.2 INTEGRATING BASIC AND ASSEMBLY CODE GRAPHICS

One of the strengths of the BASIC system on the Dragon is the graphics facilities provided by Microsoft's Extended Color BASIC. These facilities allow complex graphics programs to be written with relative ease but, as with all BASIC programs, they are relatively slow. Using assembly code speeds up the system's graphics very considerably but is much less convenient for the programmer. The ideal solution is to use the convenience of the BASIC facilities when execution speed is not important and to program time-critical sections of the program in assembly language.

Typically, those parts of a graphics program which are not time critical are the parts involved with initialisation and hardware setup. In this section we look at some useful BASIC graphics commands and describe a BASIC subroutine which will set up the graphics system then call an assembly language program which actually creates the display.

Of the many BASIC commands used for high resolution graphics, three are of particular importance to the assembly language programmer.

(1) SCREEN type,colourset

This command is used to specify whether full graphics or alphanumeric/Semigraphics mode is to be used. For a full graphics mode, type is 1, otherwise 0. The colourset parameter is either 1 or 0 and selects the colour set as defined in section 7.1.1.

(2) PMODE mode,startpage

This statement selects one of the five graphics modes available with Extended Color BASIC and is only meaningful if a SCREEN 1, colourset command has been issued. The modes available are summarised below:

Mode	Resolution	RAM bytes	Graphics type
0	128 by 96	1536	Resolution
1	128 by 96	3072	Colour
2	128 by 192	3072	Resolution
3	128 by 192	6144	Colour
4	256 by 192	6144	Resolution

The startpage value is used to select the base

address of the graphics display. In Extended Color BASIC, the display area is made up of one to four pages of 1536 bytes each with up to eight pages used for the display. Therefore, the starting page value must lie in the range 1 to 8 with page 1 starting at address 600, immediately after the text page. The table below shows the relationship of pages to RAM addresses.

Page	RAM address range
1	600-BFF
2	C00-11FF
3	1200-17FF
4	1800-1DFF
5	1E00-23FF
6	2400-29FF
7	2A00-2FFF
8	3000-35FF

(3) PCLS c

This command is used to clear the high-resolution display screen to the background colour *c*, provided that *c* is in the available colour set for the current mode. If this is not the case, or *c* is omitted, the default background colour is used. This is green if colour set 1 is selected and buff if colour set 2 is used.

SCREEN, PMODE and PCLS are useful to the assembly language programmer since they can be used to set up a graphics display prior to its use in an assembly language program. In other words, the use of these commands avoids the need to write machine code routines to perform similar functions.

We show how these can be used in the BASIC subroutine below. This routine initialises the graphics system using SCREEN, PMODE, and PCLS commands then calls an assembly language routine at address 4E21.

```

1000 SCREEN 1,0 'Select graphics screen
1010 PMODE 0,1 'Select graphics mode
1020 PCLS 'Clear graphics display
1030 EXEC &H4E21 'Call machine code
1035 ' Don't return immediately to BASIC
1036 ' as this means switch to text screen
1040 IF INKEY$="" THEN 1040 ' and display is lost
1048 'Switch colour sets and watch
1049 'screen colours change
1050 SCREEN 1,1
1060 IF INKEY$="" THEN 1060
1070 SCREEN 0,0 'Now revert to text mode
1080 RETURN

```

7.3 ALPHANUMERIC DISPLAY MODES

The alphanumeric mode is the mode adopted by the Dragon when it is switched on or reset. In this mode, the display is made up of a 32 by 16 matrix of display elements with 512 bytes of video RAM dedicated to this display. The BASIC system allocates this display area at address 400 so that the SAM control register bits F0-F6 are set to 02.

Although the VDG chip supports four different alphanumeric modes, the Dragon hardware is only designed to make use of one of these. The other modes require special read-only memories to be installed and attempting to use them will result in unpredictable access to RAM. Nevertheless, it is possible to use these modes if you are prepared to spend some time in experimentation to determine where the VDG accesses RAM. The information in the VDG data sheet should be sufficient to get you started with these experiments.

Each character on the display is represented by 8 by 12 pixels although only 5 by 7 pixels are used to form the actual character. The remaining pixels define the space between the characters. The shape of the characters in alphanumeric mode is determined by a read-only memory (ROM) which is build into the VDG chip. Unfortunately, this ROM has space for only 64 characters so this means that the full ASCII character set is not available. In particular, lower case characters have been excluded and this limits the display capabilities of the Dragon.

As the maximum number of characters which may be held in the VDG's ROM is 64, this means that 6 bits of an 8-bit byte are required to represent the character value. The remaining 2 bits represent the INV and A/S control inputs to the VDG chip. One bit specifies whether the display mode is alphanumeric or Semigraphic and the other specifies whether the character is to be displayed in reverse or normal video. The table below shows the usage of the bits in an 8-bit data byte:

Bits 0-5	Character code
Bit 6	INV control bit
Bit 7	A/S control bit

One of the problems which arises with this display mode is that there is not a one-to-one correspondence between the character code in the video RAM byte (which is an ASCII character) and the character which is actually displayed. To illustrate this, you might like to run the following BASIC program:

```

5 CLS
10 FOR K = 0 TO 127
20 K$ = CHR$(K)
30 POKE &H400 + K,K

```

```

40 PRINT @256 + K,K$;
50 NEXT K

```

Program 7.2 Screen character mapping

The statements at lines 30 and 40 should be equivalent in that they should both place the ASCII value of a character in the video RAM. However, some characters will be displayed differently.

To further illustrate this point, type in the following amendments to the program:

```

10 K$=INKEY$: IF K$="" THEN 10
20 K =ASC(K$)
50 P1 = PEEK(&H400 +K)
60 P2 = PEEK(&H400 + 256 + K)
70 PRINT @480,HEX$(K),HEX$(P1),HEX$(P2);
80 GOTO 10

```

When the program is run, you will see that the actual ASCII codes K and P1 remain the same but that P2, the result of printing a character, is different. This means that the BASIC print routine is altering the character code before placing it in the video RAM.

This conversion is carried out by the standard BASIC character printing routine OUTCH. We have already mentioned this routine in Chapter 5 and, because it takes care of the necessary character conversions for the VDG chip, we recommend that it always be used for character output.

OUTCH places the character to be output at the current cursor position on the screen. The cursor position is held in a system variable called CURADR and the contents of that variable determines where, on the screen, the cursor is displayed. Cursor blinking is under the control of a system routine called CBLINK and the blinking effect is the result of inverting and re-inverting the cursor position character.

As well as performing code conversions, the routine OUTCH also carries out other screen 'housekeeping' duties. It handles screen scrolling when the end of a line is reached, deletes characters from the screen when the delete key is pressed, and updates the cursor position so that the next character input is at that position.

Normally, the Dragon display consists of dark characters on a light background. In fact, the 'normal' character set of the VDG chip consists of light characters on a dark background so the Dragon's display is actually the inverse character set. This means that the INV control bit (bit 6) of each data byte must be set to indicate dark-on-light display. To illustrate this, the following program manipulates the INV bit of every character in the display:

```

        LDX #$400      ; Display start address
NEXTCH  LDA  ,X       ; Character into A
* INV bit manipulation here - see below
        STA  ,X+     ; Put character back
        CMPX #$5FF   ; Reached end of screen?
        BLS NEXTCH  ; No, repeat
        RTS

```

Program 7.3 INV bit manipulation

The INV bit can be manipulated in the following ways:

Instruction	Effect
ORA #\$40	Sets INV to 1 so 'normalising' the display
ANDA #\$BF	Clears INV so inverting the display
EORA #\$40	If INV is set, it is unset and vice versa. The effect of this is to reverse the display

7.4 COLOUR GRAPHICS DISPLAY MODES

The VDG provides eight full graphics modes although only five of these are directly supported by Extended Color BASIC. The modes range from a four-colour 64 by 64 element display requiring 1024 bytes of video RAM to a two-colour 256 by 192 display requiring 6144 bytes of video RAM. Four of these modes are termed colour graphics modes and these are described in this section. Each of these modes is numbered 1, 2, 3 or 6 depending on the number of graphics pages required and colour graphics modes are indicated by using this number and suffixing it with C.

In any colour graphics mode, the setting of each element in the display is controlled by two bits in the video RAM byte so that the element may be one of four colours. The general format of a video RAM byte for colour graphics is shown in Figure 7.1.

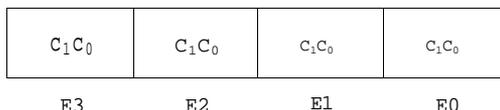


Fig. 7.1 Colour graphics byte format

Because the VDG is capable of generating eight colours, two colour sets each of four colours are available. Which colour set is in use is determined by the CSS input line to the VDG. The table below shows the available colours and their associated coding in the video RAM byte.

7.4.1 The colour graphics 1 mode

This mode provides a 64 element wide by 64 element high four colour graphics display and is referred to as the 1C mode. As the display screen is 256 by 192 pixels, this means that each element is 4 pixels by 3 pixels in size. Given that the code for 4 screen elements can be held in each byte, the display memory requirement for this mode is therefore 1024 bytes.

The pattern generator program is set up initially in this mode. However, as the BASIC PMODE command does not recognise this particular mode, the SAM and VDG chips have to be set up directly in the BASIC test rig by poking values into their control registers. It is still possible to use the SCREEN command to select the graphics screen since this is independent of the mode. It is also possible to use one of the colour graphics PMODEs (1 or 3) to set up the start page and PCLS to clear the screen graphics display since the byte format is the same. This does mean that 3072 (3C) or 6144 (6C) bytes will be cleared when only 1024 bytes need be but this is not usually a problem.

The following amendments to Program 7.1 configure the graphics hardware for the 1C mode.

```

1010  PMODE 1,1
1022  POKE &HFFC1,1 'Set V0 in SAM
1024  POKE &HFFC2,0 'Clear V1 in SAM
1026  POKE &HFFC4,0 'Clear V2 in SAM
1028  POKE &HFF22,&H80 'Configure VDG

```

The lines 1022-1028 are used to configure the VDG and SAM directly and therefore override the PMODE 1 command.

7.4.2 The colour graphics 2 mode

The display generated by this mode is in four colours on a 128 by 64 grid. Elements are made up of 2 by 3 pixels and a total of 2048 bytes of video RAM is required to support this mode. To convert the checkerboard generator to this mode, the following equates must be made:

```

DSIZE EQU 2048
DWIDTH EQU 32

```

Again, the programmer must configure the SAM and VDG chips by the use of POKES to set their control registers. The amendments to the BASIC test rig below set these devices for this mode.

```

1010  PMODE 1,1
1022  POKE &HFFC0,0
1024  POKE &HFFC3,1
1026  POKE &HFFC4,0
1028  POKE $HFF22,&HA0 'Configure VDG

```

7.4.3 The colour graphics 3 mode

This mode considers the screen to be made up of 128 by 96 elements and, like all the colour graphics modes, can display up to four colours. The total video RAM requirement for this mode is 3072 bytes or two high-resolution graphics pages.

To reconfigure the checkerboard program for this mode requires the following redefinitions

```
DWIDTH EQU 32
DSIZE  EQU 3072
```

BASIC recognises this mode so the hardware can be set up using a PMODE 1 command.

7.4.4 The colour graphics 6 mode

This is the highest resolution colour graphics mode. The screen is made up of 128 by 192 elements and there are four possible colours. Elements are each 2 by 1 pixels in size. The memory requirements for this mode are 6144 bytes which needs four high-resolution graphics pages.

The following alterations to the pattern generator program are needed:

```
DSIZE  EQU 6144
DWIDTH EQU 32
```

Again, this mode is recognised by BASIC and can be set up by using a PMODE 3,1 command.

7.5 RESOLUTION GRAPHICS DISPLAY MODES

Resolution graphics, as the name implies, are more concerned with screen resolution rather than colour so, in these graphics modes, the colours are limited. The display is black on a background colour or a foreground colour on black.

The background or foreground colours are green and buff as shown in the table below.

CSS	Colour	RAM bit	value
0	Black		0
0	Green		1
1	Black		0
1	Buff		1

In resolution graphics, each element in the display is controlled by a single bit which means that an element can be one of two colours.

The bit pattern used to define the checkerboard consists of bits with alternating values, that is, 01010101, so for all resolution graphics modes the DBITS constant in Program 7.4 is set to \$55.

The general format of a video RAM byte for resolution graphics is shown in Figure 7.2.

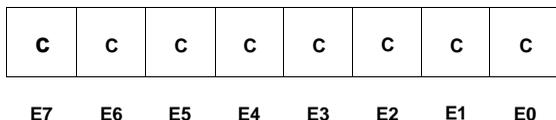


Fig. 7.2 Resolution graphics byte format

There are four resolution graphics modes which are given the names 1R, 2R, 3R and 6R. These are the resolution graphics equivalents of modes 1C, 2C, 3C, and 6C and each is described below.

7.5.1 The resolution graphics 1 mode

This mode generates a 128 element wide by 64 element high two-colour graphics display. Each element is controlled by a single bit in the video RAM byte and is 2 pixels by 3 pixels in size. The total memory requirements for this mode are 1024 bytes. Like the 1C mode, this mode is not supported directly by BASIC.

The pattern generator can be altered for this mode by redefining some of the constants as follows:

```
DSIZE EQU 1024
```

The BASIC test rig must be modified to set up the VDG and SAM chips but a PMODE 0 command followed by a PCLS will clear enough screen bytes for this mode. The following amendments to the BASIC test rig will configure the VDG and SAM chips for the 1R mode.

```
1022 POKE &HFFC1,1 'Set V0 in SAM
1024 POKE &HFFC2,0 'Clear V1 in SAM
1026 POKE &HFFC4,0 'Clear V2 in SAM
1028 POKE &HFF22,&H90 'Configure VDG
```

7.5.2 The resolution graphics 2 mode

This resolution graphics mode generates a display of 128 elements wide by 96 elements high. This means that each element is 2 pixels by 2 pixels in shape. Its memory requirements are 1536 bytes or 1 high-resolution graphics page.

The checkerboard program may be modified for this mode by redefining the equates as follows:

```
DSIZE EQU 1536
```

The 2R mode is supported by BASIC and can be invoked by issuing a PMODE 0 command.

7.5.3 The resolution graphics 3 mode

This mode generates a 128 by 192 element display in two colours. Each element is 2 pixels by one pixel and the total memory requirement is 3072 bytes.

To reconfigure the pattern generator for this mode only requires DSIZE to be equated to 3072. The mode is supported by BASIC as PMODE 2.

7.5.4 The resolution graphics 6 mode

This is the highest resolution mode possible since each pixel is controlled by a single bit in the video RAM. The display is arranged as a 256 by 192 pixel grid and therefore the video RAM size required for this is 6144 bytes. To set up the checkerboard routine for this mode requires DSIZE to be equated to 6144 and the BASIC test rig must be modified so that a PMODE 4 command is issued.

7.6 SEMIGRAPHICS DISPLAY MODES

As well as graphics and alphanumeric modes, the VDG chip has two Semigraphics modes where special-purpose characters representing graphics symbols can be built up and displayed on the screen. As the fundamental display element is the character, it is possible to mix these graphics characters with normal alphanumerics thus allowing text and graphics to appear together on the Dragon's display. Furthermore, the use of a Semigraphics mode allows the use of eight-colour rather than four-colour graphics, thus opening up more creative possibilities for the graphics programmer.

The in-built Semigraphics modes are termed Semigraphics 4 and Semigraphics 6 modes with the associated number referring to the number of elements making up a graphics character. As well as these in-built modes, it is also possible to set up three additional Semigraphics modes (8, 12, 24) by setting the VDG chip in alphanumeric mode and the SAM chip in 2C, 4C, or 6C colour graphics mode. Details of these additional modes are briefly described below and fully described in Appendix 2.

When in Semigraphics mode, each character is made up of a number of elements. The character organisation for Semigraphics 4 mode is shown as Figure 7.3. The other modes have a similar pixel organisation although, obviously, they offer higher resolution graphics as each character is made up of more elements. In all cases, the horizontal width of an element is 4 pixels but the vertical width varies from 1 to 6 pixels. Apart from the Semigraphics 6 mode, all of the Semigraphics modes allow eight-colour graphics and use three bits in each byte to represent the colour of the character elements represented in that byte. Bits 4-6 in the byte hold the colour information and the table below

defines the colours associated with each three-bit colour value.

Colour	Bit pattern
Green	000
Yellow	001
Blue	010
Red	011
Buff	100
Cyan	101
Magenta	110
Orange	111

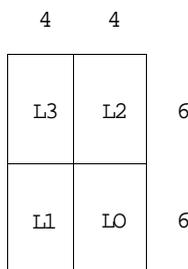


Fig. 7.3 Semigraphics 4 character organisation

A Semigraphics byte is arranged so that bits 0-3 hold the settings of character elements, bits 4-6 hold the colour and bit 7 is the mode bit. In Semigraphics 4 and 6 modes, bit 7 is 1, in Semigraphics 8, 12, or 24 modes, bit 7 is 0. All elements that are 'on' are displayed in the colour specified in bits 4-6 and elements which are 'off' are displayed as black. There is no way that elements represented in the same byte can take different colours.

7.6.1 The Semigraphics 4 mode

In Semigraphics 4 mode, each character is split into 4 elements of size 4 by 6 pixels. A single video RAM byte is therefore needed to hold each character where bits 0-3 are named L0-L3.

To experiment with this mode, you might like to modify Program 7.2 which manipulates the INV bit in the video RAM bytes. Rather than manipulate bit 6, you manipulate bit 7 using AND, OR and EOR instructions. These will turn the Semigraphic mode on and off.

7.6.2 Semigraphics 6 mode

The Semigraphics 6 mode splits each character into 6 elements of size 4 by 4 pixels giving a display resolution of 64 horizontal by 48 vertical elements. Each element is controlled by a bit in the video RAM

byte so, as a single byte is used for each character position in this mode, six bits of that byte are required to encode element settings. This leaves only two bits (bits 6 and 7) for colour information so only four colours may be represented. As in the colour graphics modes, the setting of CSS determines which colour set is used.

In fact, the number of colours available in this mode is even more restricted as bit 7 has a double function as a colour coding bit and as a mode setting bit. In order to remain in Semigraphics mode, bit 7 must always be set to 1 so this means that only blue and red from colour set 0 and magenta and orange from colour set 1 may be used.

7.6.3 The Semigraphics 8 mode

The Semigraphics 8 mode is the first of the extra Semigraphics modes which can be used by setting up the VDG chip to alphanumeric mode and the SAM chip to one of the colour graphics modes. In this mode, a standard 8 by 12 pixel character is split into eight elements of 4 by 3 pixels.

In order to set up the Semigraphics 8 mode from BASIC you must issue a SCREEN 0,0 command to put the VDG chip into alphanumeric mode then poke the bit value 01 into the SAM control bytes as shown in the graphics examples above.

In this mode, 4 bytes of video RAM are required to represent each character position and only the bottom two bits (L0 and L1) are used to hold element settings. As before, bits 4-6 hold the colour value and bit 7 should be set to indicate Semigraphic mode. Bits 2 and 3 are not used but should be set so that bit 2 has the same value as bit 0 and bit 3 is the same as bit 1.

Each character is built up as 4 rows of 4 by 3 pixel elements. However, the bytes representing these rows are not contiguous but are actually spaced 32 bytes apart. The reason for this is that the SAM chip is configured to a colour graphics mode where the image is built up row by row, with each complete row taking up 32 bytes. As Semigraphic elements consist of a number of rows, this means that the bytes specifying the element must be set up at this spacing.

As four bytes are used, it is possible to mix element colours when using this mode as, obviously, each pair of elements in a byte has its own colour information. Furthermore, it also allows character rows from different characters to be incorporated into new characters and symbols. This means you can provide facilities such as character underlining by switching to Semigraphics mode at the appropriate time.

However, using this facility requires great care as you must build up each character individually with each row of elements defined in a separate byte. You also

have the problem of spacing character definition bytes 32 bytes apart as explained above so we recommend that you write a program to help you organise byte layout if you wish to use this facility.

7.6.4 The Semigraphics 12 mode

In this mode, the VDG chip is set to alphanumeric mode and the SAM chip to colour graphics 4C mode. Each character element is represented by twelve 4 by 2 pixel elements held in six bytes. As in the Semigraphics 4 mode, only the bottom two bits per byte are used for element settings and different bytes may be set to different colours.

To set up this mode, you must issue a SCREEN 0,0 command from BASIC then poke the value 001 into the SAM control bytes.

7.6.5 The Semigraphics 24 mode

In this mode, the SAM chip is set up to 6C mode and each character element is made up of twenty four 4 by 1 pixel elements thus giving a screen resolution of 64 by 192 elements. A total of 12 bytes is required to hold these element settings and, again, the colour of the two elements represented in each byte may be set up independently.

To set up this mode, you must issue a SCREEN 0,0 command from BASIC then poke the value 011 into the SAM control bytes.

7.7 GRAPHICS UTILITIES

So far we have shown how the various display modes can be set up from BASIC and we have assumed that this is carried out before an assembly code graphics routine is called. Sometimes, setting up the display hardware from BASIC is neither possible nor desirable so in this section we describe how BASIC commands such as SCREEN, PMODE, PCLS, etc. may be implemented in assembly language.

We have described, in section 7.1, the various hardware control bits and have explained that they are set up via memory-mapped I/O addresses. Remembering which bit means what is difficult, so it is good practice to set up mnemonic names for the various control bit settings. A table of equates defining these names, which we use throughout the remainder of this chapter, is shown below.

* VDG/PIA and SAM addresses

*

```
VDGP1A    EQU $FF22    ; Port B of PEA - VDG control
SAMVOC    EQU $FFC0    ; Used to clear V0
SAMV0S    EQU $FFC1    ; Used to set V0
SAMV1C    EQU $FFC2    ; Used to clear V1
```

```

SAMV1S EQU $FFC3 ; Used to set V1
SAMV2C EQU $FFC4 ; Used to clear V2
SAMV2S EQU $FFC5 ; Used to set V2
SAMF0C EQU $FFC6 ; Base address of F0-F6
*
```

```
* VDG/PIA bit patterns - assumes CSS=0
```

```
*
```

```

ALPHAI EQU $00 ; Internal alphanumeric
ALPHAE EQU $10 ; External alphanumeric
MODES4 EQU ALPHAI ; Semigraphics 4
MODES6 EQU ALPHAE ; Semigraphics 6
MODES8 EQU MODES4 ; Semigraphics 8
MODS12 EQU MODES4 ; Semigraphics 12
MODS24 EQU MODES4 ; Semigraphics 24
*
```

```
* Full graphics modes
```

```
*
```

```

MODE1C EQU $80 ; Graphics 1C
MODE1R EQU $90 ; Graphics 1R
MODE2C EQU $A0 ; Graphics 2C
MODE2R EQU $B0 ; Graphics 2R
MODE3C EQU $C0 ; Graphics 3C
MODE3R EQU $D0 ; Graphics 3R
MODE6C EQU $E0 ; Graphics 6C
MODE6R EQU $F0 ; Graphics 6R
```

Normally, the modes of the VDG and the SAM chip are the same but for some of the extra Semigraphics modes they must be set up differently. Therefore, rather than use a single routine with complex parameters to set up these devices, it is better to use two separate routines. The routine to configure the VDG chip is called VDGMOD and the routine to configure the SAM chip is SAMMOD. They are shown below as Program 7.5.

```
* VDGMOD - sets up VDG chip
```

```
* Sets control lines A/G, GM0-2, and CSS
```

```
*
```

```
* Register input A - configuration bit pattern
```

```
* to be written to PIA
```

```
* Note only bits 3-7 of PIA are set so bits 0-2 must
```

```
* be preserved
```

```
*
```

```

VDGMOD PSHS A ; Preserve setup pattern
LDA VDGPIA ; Preserve bottom bits
ANDA #7 ; of PIA register
ORA ,S ; Or in setup pattern
STA VDGPIA ; Setup VDG
PULS A,PC ; Restore and return
*
```

```
* SAMMOD - Setup SAM chip
```

```
*
```

```
* Register input A - bit pattern used to set up VDG
```

- * In general, V0,V1,V2 in SAM are set up as GM0, 1, 2
- * in VDG but there are three special cases:
- * If A/G = 0 then V0V1V2 = 000
- * If A/G = 1 and GM0GM1GM2 = 000 then V0V1V2 = 100
- * If A/G = 1 and GM0GM1GM2 = 111 then V0V1V2 = 011

```

SAMMOD   PSHS   A           ; Preserve VDG pattern
         STA   SAMVOC      ; Clear V0
         STA   SAMV1C     ; Clear V1
         STA   SAMV2C     ; Clear V2
         ANDA  #$F0       ; Clear bottom 4 bits of A
         BPL   NOTGM0     ; Text mode (B7=0)
         CMPA  #MODE1C    ; no, is it 1C?
         BNE  NOT1C
         ORA  #$10       ; yes, special case->1R
NOT1C    CMPA  #MODE6R    ; Is it 6R
         BNE  NOT6R
         ANDA  #$E0       ; yes, special case->6C
NOT6R    ROLA  A           ; Get rid of A/G bit
         BPL  NOTGM2     ; GM2 set?
         STA  SAMV2S     ; yes, set V2
NOTGM2   ROLA  A           ; get rid of GM2 bit
         BPL  NOTGM1     ; GM1 set
         STA  SAMV1S     ; yes, set V1
NOTGM1   ROLA  A           ; get rid of GM1 bit
         BPL  NOTGM0     ; GM0 set?
         STA  SAMV0S     ; yes, set V0
NOTGM0   PULS  A,PC      ; Restore and return

```

Program 7.5 VDG and SAM setup routines

These routines set up the SAM and VDG chips. Normally, these devices are configured in the same mode so the bit pattern defining the VDG's control bits is set up in register A and each routine is called in turn.

* GMODE - sets up graphics hardware

* Register input A - VDG's control bit settings

*

```

GMODE   BSR  VDGMOD
         BSR  SAMMOD
         RTS

```

You can use this routine in conjunction with the equate table defined above to set up any of the graphics modes. You simply have to load the A register with the mode required then call GMODE to configure the VDG and SAM chips. The exceptions to this are when Semigraphics 8, 12, or 24 modes are to be set up when VDGMOD and SAMMOD must be called individually to configure the VDG and SAM chips to different modes.

For example:

```

* SEMI8 - selects Semigraphics 8 mode
*
SEMI8    LDA #ALPHAI        ; Alphanumeric mode
        BSR VDGMOD         ; for VDG
        LDA #MODE2C        ; and 2C mode
        BSR SAMMOD         ; for SAM
        RTS

*
* FULL6R - Selects resolution graphics 6 mode
*
FULL6R   LDA #MODE6R
        BSR GMODE
        RTS

```

These mode setup routines combine some of the features of BASIC'S PMODE and SCREEN commands which, together, set up the mode required, define the colour set and determine which graphics pages are to be used. In fact, it is better programming practice for a routine to do one thing and one thing only so we have defined separate assembly code routines to select the colour set and to define the starting page. These are shown as Program 7.6.

```

* CSS - Select colour set

* Input register A = 0 -> colour set 0
*                   = 1 -> colour set 1
*
CSS       PSHS A            ; Preserve A
        LDA VDGPIA        ; Read current state of VDG
        TST ,S            ; Check set selection
        BEQ CSS0
        ORA #8             ; Set the CSS line
        BRA XIT

CSS0      ANDA #$F7        ; Clear the CSS line
XIT       STA VDGPIA      ; update the VDG
        PULS A,PC        ; Restore and return

* To set up the SAM control bits, we must manipulate
* 7 bits. We use the utility routine
* from page 16 of Appendix 2 to carry out
* this bit manipulation
*
* SAMSET - Configures SAM control bits
*
* Register inputs X - address in SAM control register
*                   A - SAM configuration bit pattern
*                   B - number of bits to be copied from
*                   A to SAMCR
* NB. This routine does not preserve registers
*
SAMSET    LSRA             ; Shift bit 0 to carry
        BCC NOTSET       ; Set corresponding CR bit?

```

```

LEAX 1,X           ; Yes, odd address
STA 0,X+          ; set bit and adjust X for
BRA CHKCNT        ; next address
NOTSET STA 0,X++   ; Clear bit and continue
CHKCNT DECB        ; All done?
BNE SAMSET
RTS                ; Yes, return

```

*

* SAMSET is used by PAGEX to set the page number
 * in the control register. This is passed to PAGEX
 * as a 16-bit address and PAGEX selects the top 7 bits
 * to get the graphics segment base address. This means
 * that you can pass an address within a segment
 * not just the segment base address

*

* PAGEX - Set up graphics segment base address

*

* Register inputs X - Address of or within display area

```

PAGEX PSHS X,D      ; Save registers
      TFR X,D        ; A = X(HI), B = X(LO)
      LSRA           ; We only need top 7 bits
      LDB #7         ; as specified in B
      LDX #SAMFOC    ; Copy to F0-F6
      BSR SAMSET
      PULS X,D,PC    ; Restore and return

```

Program 7.6 Colour set and graphics page setup

We have now defined those utility routines which allow the assembler programmer to dispense with BASIC'S SCREEN and PMODE commands and also with any POKEs that are needed to set up extra graphics modes from BASIC. We leave it as an exercise for the reader to convert the BASIC test rig, defined as Program 7.1 to assembly code.

Now let us look at other useful graphics utility routines which may be used by the assembly language programmer. The first of these is a routine to clear the screen to a given colour.

* CLS - Clear screen to specified colour

*

* Register input B - colour specification

* 0 = Black, 1 = Green, 2 = Yellow,
 * 3 = Blue, 4 = Red, 5 = Buff,
 * 6 = Cyan, 7 = Magenta, 8 = Orange

*

```

CLS PSHS X,B        ; Save registers
    TSTB           ; B = 0 is special case
    BEQ SEMION     ; as all elements turned off
    DECB           ; BASIC -> VDG colour code
    ASLB           ; Move colour code bits
    ASLB           ; into bits D4-D6

```

```

ASLB
ORB   #$0F      ; Turn on all elements
SEMION ORB   #$80      ; Set up Semigraphics bit
LDX   #$400     ; Set default screen address
NXTBYT STB   ,X+      ; Colour element
CMPX  #$5FF     ; on the screen
BLS   NXTBYT    ; If not finished, repeat
PULS  X,B,PC    ; Restore and return

```

Program 7.7 Clear screen

When considering the graphics screen, the programmer thinks in terms of row and column numbers but, in memory, the screen is simply a one-dimensional contiguous area. There are 32 character bytes per row so it is useful to have a routine which, given a row and column number, translates this to the appropriate address. This routine, R32COL, is also useful for full graphics modes which have 32 bytes of colour/resolution information per row. Other graphics modes need a variation of this which is easily derived by replacing the LDB #32 instruction with an LDB #16 instruction. The code for this routine is shown below.

```

* R32COL - Calculate screen position
* Register inputs B - column number
*           A - row number
* Register output D - screen position

```

```

* Calculates position as A * 32 + B
*

```

```

R32COL STB   ,-S      ; Save column on stack
LDB   #32      ; Set up multiplier
MUL           ; D = A * 32
ADDB  ,S+     ; D = D + B
ADCA  #0      ; Propagate carry
RTS

```

7.8 DESIGNING AND IMPLEMENTING GRAPHICS PROGRAMS

An essential first step in the design of a graphics program is to sketch out the graphics symbols which you would like to use in your program. Examples of such symbols are space-ships, laser bursts, bombs, explosions, etc. if you are writing games programs, and character fonts, pie charts, maps, etc. if you are concerned with more serious applications.

A well-designed graphics program is built up incrementally with later developments built up on earlier design stages. It is therefore very important that assembly language routines should be as flexible and as general-purpose as possible. In this section we discuss various tools and techniques used in the design

that the Dragon logo should be contained in a 16 by 16 element grid.

The next stage is to work out how to set up the picture elements in this grid so that the shape of a Dragon is produced. The easiest way to do this is to use a graphics worksheet, which is simply lined graph paper with the same height to width ratio as the chosen resolution. If this is not available, squared paper can be used but you must take into account any differences in the height to width ratio. The grid representation of the Dragon logo is shown as Figure 7.4.

Once the symbol has been mapped out on the graphics worksheet, the binary patterns for each row must be encoded and included as data for the assembly language program. The simplest way to do this is to make up a data array of constant byte values using the assembler directives `FCB` and `FDB`. This process is straightforward when resolution graphics is used but requires rather more care when colour is required as there is not a one-to-one relationship between symbols on the worksheet and bits in the data byte.

Let us take the easiest case first and look at how the graphic grid can be converted to resolution graphics data bytes. Since there is a one-to-one relationship between the screen elements and the grid elements, row 0, byte 0 is encoded as `$00` (all elements off) and row 0, byte 1 is encoded as `$3F` (00111111). Row 1, byte 0 is encoded as `$20` (00100000), row 1, byte 1 as `$3F`, and so on. Therefore, the first few bytes of the assembly language data table might be written as follows:

```
FCB $00
FCB $3F
FCB $20
FCB $3F
```

However, as we are dealing with a 16-bit entity, it is better practice to encode the information as a single 16-bit value using `FDB` directives. For example:

```
FDB $003F
```

After you have worked out the appropriate byte values, you should then label the table with a symbolic name such as `DRAGON`. The complete table for the Dragon logo is shown below:

```
DRAGON  FDB $003F    ;Row 0 bit pattern
        FDB $203F    ;Row 1 bit pattern
        FDB $F860
        FDB $18DE
        FDB $E9BC
        FDB $2B7C
```

```

FDB $52E0
FDB $B6C0
FDB $BFFC
FDB $BFFE
FDB $C007
FDB $7FEB
FDB $3FFD
FDB $140A
FDB $2814
FDB $0000      ;Row 15 bit pattern

```

After encoding the graphics symbol, the next step is to design a routine which takes the encoded symbol and displays it on the screen. Because the display is memory mapped, all that you need to do is to copy the data from the graphics symbol table into the appropriate locations in the video RAM. The copy routine must map the row/column representation of the screen onto the video RAM which is organised as a linear sequence. As the Dragon logo only takes up part of the screen, succeeding row addresses are actually located 16 bytes from each other in this resolution graphics mode.

This is an example of a situation where you should write a general-purpose routine which can carry out the mapping of data tables to screen locations for any graphics mode. This routine has to satisfy the following design criteria:

- (1) It must be able to move data from any source address to any destination address.
- (2) It must be able to cope with any row separation.
- (3) It must be able to cope with any number of rows.
- (4) It must not interact, in any way, with its calling program.

A general-purpose copy routine which meets these criteria is shown as Program 7.8.

```

* COPY2B - Copies 2 byte chunks to video RAM
*
* Register inputs X - destination address (in RAM)
*                  Y - source address
*                  B - row width (number of bytes
*                      between video RAM addresses)
*                  A - number of rows (number of words
*                      to copy)
*
COPY2B  PSHS X,Y,U,A,B      ; Save registers
NEXT2B  LDU  ,Y++          ; Pick up source word
        STU  ,X            ; and store in row

```

```

        ABX                ; Add row width
        DECA               ; Repeat until all
        BNE NEXT2B        ; rows dealt with
        PULS X,Y,U,A,B,PC ; Restore and return

```

Program 7.8 COPY2B - update video RAM

An example of how this routine might be used is shown in the program fragment below which displays the Dragon logo in the top left hand corner of the screen.

```

DSTART EQU $0600      ; Display start
DWIDTH EQU 16         ; Display width

LOGOTL  LDX #DSTART   ; Destination address
        LEAY DRAGON,PCR ; Source address
        LDB #DWIDTH   ; Display width
        LDA #16       ; Number of rows
        BSR COPY2B    ; Transfer logo to screen
        RTS

```

Now that we have managed to draw a dragon in the corner of the screen, we can now go on to repeat the pattern over and over again.

* FILLER - fills screen with dragons

*

* Register inputs NONE

* Registers destroyed X,Y,A,B

```

FILLER  LDX #DSTART   ; Destination address
        LDB #DWIDTH   ; Colour display row width
        LDA #16       ; Number of display rows
NXTPOS  LEAY DRAGON,PCR ; Address of source
        BSR COPY2B    ; Now start copying
        LEAX DWIDTH+2,X ; dragons diagonally
        CMPX #DEND-512 ; until no room left
        BLS NXTPOS    ; for another one
        RTS

```

Program 7.9 FILLER - fills screen with dragons

Now let us look at how the Dragon logo can be displayed using the colour graphics 6 mode which allows a four-colour display at the same resolution. The first stage in this process is to convert the grid pattern to the appropriate four-colour data bytes.

The best way to tackle this is to encode the diagram as if it was to be displayed in resolution graphics then convert every element that is off (0) to the chosen background colour (green in this case) and convert elements which are on to the chosen foreground colour (red, naturally). This means that the colour data table is twice the size of the resolution graphics

data table because two bits rather than a single bit are used to represent each screen element. Examples of the conversion of resolution data to colour data are shown below.

Resolution data	Colour data
\$00	\$0000
\$3F	\$0FFF
\$20	\$0C00

The conversion from the resolution grid data to colour data is a tedious, mechanical, error-prone process which means that it is ideal for automation. Program 7.10 below is a BASIC program which does the conversion for you.

```

100 PRINT"RESOLUTION TO COLOUR GRAPHICS"
110 PRINT"DATA CONVERSION PROGRAM"
120 PRINT"COLOURS AVAILABLE ARE:"
130 PRINT"COLOUR SET 0"
140 PRINT"      GREEN    1, YELLOW 2"
150 PRINT"      BLUE     3, RED   4"
160 PRINT"COLOUR SET 1"
170 PRINT"      BUFF     5, CYAN   6"
180 PRINT"      MAGENTA  7, ORANGE  8"
200 INPUT"BACKGROUND COLOUR (1-8)";BG
210 IF (BG<1) OR (BG>8) THEN 120
220 INPUT"FOREGROUND COLOUR (1-8)";FG
230 IF (FG<<1) OR (FG>8) THEN 120
240 IF (BG<5) AND (FG<5) THEN 300
250 IF (BG>4) AND (FG>4) THEN 290
260 PRINT"FOREGROUND/BACKGROUND NEED TO"
270 PRINT"BE IN THE SAME COLOUR SET"
280 GOTO 120
290 BG=BG-4: FG=FG-4 'Convert to CS 0
300 BG=BG-1: FG=FG-1 'Convert to VDG code
310 PRINT"ENTER RESOLUTION GRAPHICS CODE"
320 INPUT"BYTE (IN HEX)";RB$
330 RB=VAL("&H"+RB$) 'Convert to numeric value
340 BM=1 'Bit Mask (first power of 2)
350 BP=1 'Bit Pair (first power of 4)
360 CW=0 'Colour Word value (16 bits)
370 BC=0 'Bit Colour (FG or BG)
380 'Now convert the single resolution
390 'bits into pairs of colour bits
400 FOR BIT = 0 TO 7
410 IF (RB AND BM) THEN BC=FG ELSE BC=BG
420 CW=CW+BC*BP 'Add colour pair to CW
430 BM=BM*2 'Next power of 2
440 BP=BP*4 'Next power of 4
450 NEXT BIT
460 PRINT"COLOUR CODE WORD (HEX) IS ";HEX$(CW)
470 PRINT:GOTO 310

```

The complete data table for the Dragon logo as a red dragon against a green background is:

```

DRAGON   FDB $0000,$0FFF
          FDB $0C00,$0FFF
          FDB $FFC0,$3C00
          FDB $03C0,$F3FC
          FDB $FCC3,$CFF0
          FDB $0CCF,$3FF0
          FDB $330C,$FC00
          FDB $CF3C,$F000
          FDB $CFFF,$FFF0
          FDB $CFFF,$FFFC
          FDB $F000,$003F
          FDB $3FFF,$FCCF
          FDB $0FFF,$FFF3
          FDB $0330,$00CC
          FDB $0CC0,$00CC
          FDB $0000,$0000

```

With this colour data table, the COPY2B routine can now be used to update the 128 by 192 colour graphics display. However, the updating process is slightly more complex since 16 bits of data are needed to represent 8 elements. Therefore, it is best to modify this routine so that 4-byte chunks may be copied. This modification simply involves adding statements to copy another word before the ABX instruction. The code for this routine is provided as part of Program 7.12.

7.8.2 Animating a graphics display

Until now, all the graphics displays which we have discussed have been static in nature. In this section we describe how to create simple animation sequences. The technique used in animating graphics are very similar to those used in cartoon filming because the animation relies on rapidly flicking through different versions of a basic graphics symbol. If the flicking is sufficiently rapid, the movement will be smooth but if it is too slow, as is often the case with BASIC, the movement of the symbol is very jerky.

To illustrate this technique we shall breath some life into the Dragon logo introduced in the last section. To be more exact, we shall breath some fire into the beast as it is well-known that no self-respecting dragon is without this mythical power. What we shall do is to add flames which will flare out from the dragon's mouth, flicker and dance and then die down.

At first sight this might seem to be a difficult and complex task but, in fact, if tackled one stage at a time, it is relatively easy. The secret is to build up the flames one at a time, flicker the flames with slightly differing sequences and then extinguish the

fire in the opposite order to that used to build it up. This is best illustrated by means of an example.

Say an 8 by 8 element grid is used to contain the flame pattern and the flame colour chosen is yellow. The data tables for building up the final flame sequence labelled FLAME0 to FLAMES are shown below. It is left as an exercise for the reader to reconstruct the actual flame patterns which have been used to derive this data.

```
FLAME0    FDB  $0,$0,$0,$0,$0,$0,$0,$0
FLAME1    FDB  $0,$0,$0,$5,$0,$0,$0,$0
FLAME2    FDB  $0,$0,$50,$5,$0,$0,$0,$0
FLAME3    FDB  $0,$0,$50,$5,$150,$0,$0,$0
FLAME4    FDB  $0,$500,$50,$505,$150,$1400,$0,$0
FLAME5    FDB  $5000,$500,$1050,$505,$4150,$1400,$5000,$0
```

The first flame sequence (FLAME0) is completely blank as this is used to extinguish the flames completely. The next flame sequence (FLAME1) is slightly more built up than FLAME0, the sequence FLAME2 is more developed than FLAME1 and so on.

Since 2 bytes of colour data has to be copied to the video RAM we can make use of the COPY2B routine to set up the display area in memory. However, if we had used resolution graphics instead of colour graphics, then we would only have needed to copy 1 byte of resolution data to the video RAM. A routine called COPY1B that performs this function is provided in this section for the sake of completeness.

```
* COPY1B - copy data in 1 byte chunks
*
* Register inputs X,Y,B as COPY2B
*           A - number of bytes to copy
*
```

```
COPY1B    PSHS X,Y,A,B          ; Save registers
          STA  ,-S              ; Save count on stack
NEXT1B    LDA  ,Y+              ; Pick up row byte pattern
          STA  ,X+              ; and store it
          ABX                    ; Add row width indicator
          DEC  ,S                ; Decrement count
          BNE  NEXT1B           ; All rows done?
          LEAS 1,S              ; Discard local
          PULS X,Y,A,B,PC       ; Restore and return
```

Program 7.11 COPY1B - video RAM update

The next stage in the development of the animated sequence is to provide the means to display the individual flame sequences. This is also the time to consider the delay between sequence updates since too short a delay will result in the animation sequence

lasting only a fraction of a second whilst too long a delay will result in faltering effect. The following routine is used to implement a short delay and is based on a software delay loop.

* DELAY - hold up activity for a while

```
DELAY   PSHS X           ; Save register
        LDX #$4000      ; This value may be changed
LOOP    LEAX -1,X       ; to adjust the timing
        BNE LOOP        ; delay
        PULS X,PC       ; Restore and return
```

The next routine is typical of routines needed to display the individual sequences:

```
FIRE0   LEAY FLAME0,PCR ; Set up flame sequence address
FIRE    BSR COPY2B      ; Update display with sequence
        BSR DELAY       ; Stop for a while
        RTS
```

The two subroutine calls to COPY2B and to DELAY are common to all the updating routines so we can enter FIRE0 at label FIRE to allow them to be called as a subroutine. This is a fairly common and harmless trick of assembly language programming which serves to reduce the size of the finished code.

The next routine, to continue the animation, makes use of this technique:

```
FIRE1   LEAY FLAME1,PCR
        BSR FIRE
        RTS
```

This process of routine development continues until you end up with a complete animation program. Program 7.12 is an animated fire-breathing dragon which is an appropriate conclusion to this chapter. For brevity, we have not duplicated the code of routines which have been described earlier in this chapter. Rather, we have indicated by comments where these routines should be included.

*

* ANIMATED DRAGON PROGRAM

```
ORG 20001
```

```
LBRA ACTION ; This preserves the entry point
```

* DRAGON - Data for the Dragon logo

* on a 16x16 colour grid

* INCLUDE DRAGON DATA TABLE HERE

* Flame data follows. This is on an 8x8 colour grid

* INCLUDE FLAME TABLE HERE

* Constants used in the program follow

*

```
DWIDTH EQU 32 ; Due to 128x196 colour mode
ROWS16 EQU 16 ; For a 16 row grid
ROWS8 EQU 8 ; For an 8 row grid
DSTART EQU $0600 ; May be changed
```

*

* INCLUDE VDG/PIA and SAM ADDRESS EQUATES HERE

* INCLUDE VDGMOD, GMODE and SAMMOD ROUTINES HERE

* FULL6C - Selects colour graphics 6 mode

*

```
FULL6C LDA #MODE6C
      BSR GMODE
      RTS
```

*

* INCLUDE CSS ROUTINE HERE

*

* INCLUDE SAMSET AND PAGEEX ROUTINES HERE

*

* INCLUDE COPY2B ROUTINE HERE

*

* COPY4B - Copies 4 byte chunks to video RAM

*

* Register inputs X - destination address (in RAM)

*

Y - source address

*

B - row width (number of bytes

*

between video RAM addresses)

*

A - number of words to copy

*

```
COPY4B PSHS X,Y,U,A,B ; Save registers
NEXT4B LDU ,Y++ ; Pick up source word
      STU ,X ; and store in row
      LDU ,Y++ ; Pick up next word
      STU 2,X ; and store after first
      ABX ; Add row width
      DECA ; Repeat until all
      BNE NEXT4B ; rows dealt with
      PULS X,Y,U,A,B,PC ; Restore and return
```

* DELAY - hold up activity for a while

```
DELAY PSHS X ; Save register
      LDX #$4000 ; This value may be changed
LOOP LEAX -1,X ; to adjust the timing
      BNE LOOP ; delay
      PULS X,PC ; Restore and return
```

* The following routines play with fire!

* FIRE0 - Deals with the first flame sequence

```
FIRE0  LEAY  FLAME0,PCR  ; Set up address first pattern
```

* FIRE - Used throughout the fire routines for display

```
FIRE   BSR   COPY2B      ; Flames 8x8 in colour
```

```
      BSR  DELAY          ; Wait briefly
```

```
      RTS                ; before returning
```

* FIRE1 - Deals with the second flame sequence

```
FIRE1  LEAY  FLAME1,PCR
```

```
      BSR  FIRE
```

```
      RTS
```

* FIRE2 - Third flame sequence

```
FIRE2  LEAY  FLAME2,PCR
```

```
      BSR  FIRE
```

```
      RTS
```

*

* FIRE3 - Fourth flame sequence

*

```
FIRE3  LEAY  FLAME3,PCR
```

```
      BSR  FIRE
```

```
      RTS
```

*

* FIRE4 - Fifth flame sequence

```
FIRE4  LEAY  FLAME4,PCR
```

```
      BSR  FIRE
```

```
      RTS
```

*

* FIRE5 - Sixth flame sequence

```
FIRE5  LEAY  FLAME5,PCR
```

```
      BSR  FIRE
```

```
      RTS
```

*

* KINDLE - Ignite (start the flame sequence) and

* gradually build the fire up.

*

```
KINDLE BSR  FIRE0
```

```
      BSR  FIRE1
```

```
      BSR  FIRE2
```

```
      BSR  FIRE3
```

```
      BSR  FIRE4
```

```
      BSR  FIRE5
```

```
      RTS
```

*

*

* FLARE - Plays flames by varying flame sequences.

*

```
FLARE  BSR  FIRE4
```

```
      BSR  FIRE5
```

```
      BSR  FIRE3
```

```

        BSR FIRE1
        BSR FIRE4
        BSR FIRE3
        BSR FIRE5
        RTS
*
* DOUSE - Gradually extinguish the flames
* by reversing the sequence.
*
DOUSE    BSR FIRE4           ; Was at full flame
        BSR FIRE3
        BSR FIRE2
        BSR FIRE1
        BSR      FIRE0       ; Fire now out.
        RTS
*
* FLAMES - Animates the fire-breathing dragon.
*
FLAMES   BSR KINDLE         ; Kindle the fire.
        BSR FLARE           ; Play the flames.
        BSR DOUSE          ; Now douse the fire.
        RTS
*
* ACTION - Set up the display and start the animation.
*
ACTION   LBSR FULL6C        ; Select the graphics mode
        LDA #0              ; Choose colour set 0
        LBSR CSS
        LDX #DSTART        ; Set up the display start
        LBSR PAGEX         ; and select the page
        LEAX 16,X          ; Get breathing space
        LEAY DRAGON,PCR    ; Set up dragon data
        LDB #DWIDTH        ; and display width
        LDA #ROWS16        ; and number of rows
        LBSR C0PY4B        ; for a 16x16 colour grid
        LEAX -2,X          ; Flames come out of mouth!
        LDA #ROWS8         ; Now displaying flames
RETAKE   BSR FLAMES         ; on an 8x8 colour grid
        BRA RETAKE         ; Repeat the animation

```

Program 7.12 A fire-breathing Dragon

Chapter 8

Input/output programming

The topic of input/output programming is one that is often neglected in books like this. The reason for this is that the subject is so detailed and complex that it is very difficult to present a coherent overview of it in a single chapter. However, we have tried to do so and, in this chapter, we explain some of the general principles of I/O programming and describe the specifics of the Dragon's I/O system.

As we explained earlier in the book, the Dragon's I/O system is memory mapped which means that I/O devices (or more accurately controllers) are accessed by reference to specific memory locations. Reading or writing to these locations results in an I/O transfer from or to the I/O device.

The Dragon's offers a variety of I/O interfaces such as a cassette interface, two joystick interfaces, a printer interface, etc. These are shown in Figure 8.1 which is a block diagram of the Dragon's I/O system. Some of the terms in this diagram will be unfamiliar to the reader who is new to I/O programming but they will be explained as the chapter progresses.

Because the I/O system is so complex, it is impossible for us to provide a description here which contains enough detail for the electronics enthusiast to connect his own devices to the system. Rather, we have concentrated here on information for the I/O programmer and have avoided going into specific details of the system electronics. Readers who want to interface hardware to the Dragon must use the data sheets presented in the appendices for complete hardware details of the I/O system devices. These data sheets have been provided by the chip manufacturer and contain complete details of the chip functions and signals.

There are four major sections in this chapter. The first two are concerned with the generalities of I/O programming and cover the concept of interrupts and I/O programming techniques. The final two sections cover details of the Dragon's I/O system, with section 3 concentrating on the PIA chip, which is the principal interface controller on the system, and section 4 describing the various I/O ports built into the system.

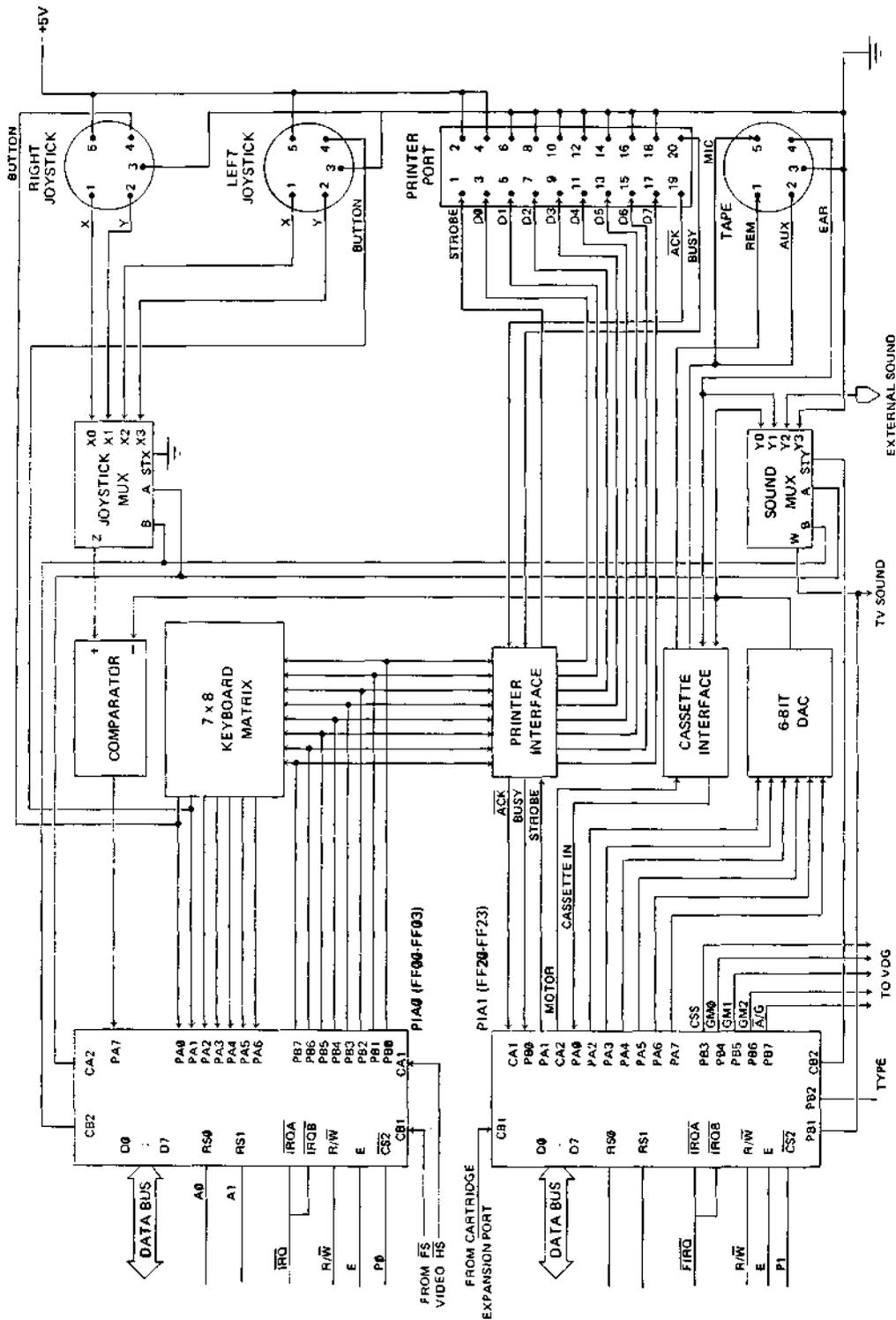


Fig. 8.1 The Dragon's I/O hardware

8.1 INTERRUPTS

The central concept on which much I/O programming is based is the notion of an interrupt. An interrupt is a signal to the processor to temporarily stop what it's doing and carry out some other task. We shall explain the steps involved in this by means of an analogy.

Say you are busy programming your Dragon when your doorbell rings. You stop what you are doing to answer the door and you find the occupier of the apartment below who tells you that water is dripping through his ceiling. You immediately rush to the bathroom where you find that you have left the water running and the bath has overflowed. You turn off the water, mop up the mess then go back to your programming. Whilst mopping up the bathroom you ignore other interruptions unless they are very urgent such as flames shooting from the cooker.

In this scenario, events can be identified which correspond to the events which occur in a computer system when an interrupt is received and processed.

(1) The interrupt

This is the doorbell ringing to tell you to stop what you are doing as some other task requires your attention. A computer system has one or more interrupt request control lines. A signal on one of these lines is an interrupt which causes the currently executing task to be suspended and the interrupt processed.

(2) The interrupt vector

This is the front door. The interrupt (doorbell) tells you to go to a known place in order to start interrupt processing. In a computer there are usually several pre-determined memory locations, called interrupt vectors. The program counter is automatically loaded with the contents of one of these locations when the interrupt is detected and this causes a transfer of control to an interrupt service routine.

(3) The interrupt service routine

This is the mopping up of the bathroom or, in other words, what you must do to clear the condition causing the interrupt. In a computer the address of this routine is held in the interrupt vector and control is transferred automatically to it.

(4) The interrupt mask

This is, effectively, what you do when you ignore interruptions in order to get the water off your bathroom floor. In a computer it is usually pos-

sible to set up a so-called 'interrupt mask' which indicates that an interrupt is being processed and that no more interrupts should be accepted until that processing is complete.

- (5) The priority interrupt
This is the cooker catching fire. Some events are so urgent that they must be handled in spite of the fact that another interrupt-handling process is already underway. In a computer, there is often a non-maskable interrupt control line. A signal on this line means 'something urgent has happened' and must be processed immediately.
- (6) Process resumption
After handling the interrupt, you can breath a sigh of relief and go back to programming. In a computer, the interrupted process is restarted and execution proceeds as if the interrupt had not occurred.

Interrupts are very important in I/O programming because they are one way that a peripheral controller can tell the processor that data are ready for input or that the peripheral is ready to accept more data for output. If interrupts are not used the processor has to examine all the peripheral devices at periodic intervals to see if they have completed their input or output operations.

The M6809 processor has a total of four interrupt request control lines, a number of interrupt handling instructions and uses three of the bits in the condition code register in its interrupt processing. Before going on to describe these in detail, however, we describe a typical sequence of actions which take place automatically when a 'normal' interrupt occurs.

The 'normal' interrupt control line on the M6809 is called IRQ and a signal on this line causes the processor to suspend the currently executing process after it has completed execution of its current instruction. The processor then sets the Entire flag in the condition code register (CC.E) and pushes all the processor registers, except S, onto the S-stack.

The processor then sets the IRQ Mask flag in the condition code register (CC.I) to indicate that an interrupt is being processed and that no more interrupts on IRQ should be accepted. PC is then loaded with the contents of the IRQ interrupt vector (memory locations FFF8:FFF9) which causes a transfer of control to the interrupt service routine whose address is held in the interrupt vector.

The interrupt service routine services whatever condition caused the interrupt then returns to the interrupted process by executing a Return from

Interrupt instruction. This instruction restores the register contents thus transferring control back to the interrupted process when PC is restored and clearing the interrupt mask bit CC.I when CC is restored.

As well as the standard interrupt request line IRQ, the M6809 also has three other interrupt request lines called NMI (non-maskable interrupt), FIRQ (fast interrupt request) and RESET.

The RESET line is used when the system is switched on. An interrupt on this line is not handled in the same way as other interrupts as, obviously, there is no executing process to be suspended. When the machine is switched on, a RESET signal causes transfer of control to a system initialisation routine in ROM which sets up the S-stack and causes some other process, which is usually the BASIC interpreter, to be initiated.

The FIRQ interrupt line signals that fast interrupt processing is to take place. This is similar to the processing of an IRQ interrupt but instead of all registers being stacked and unstacked, only PC and CC are stacked before control is transferred to the interrupt service routine. The E flag in CC is unset to indicate that only CC and PC have been stacked.

When an RTI instruction restores CC, the top stack location, it examines the CC.E flag to see if it must restore all other registers or if it is only necessary to restore PC. An FIRQ request causes the FIRQ mask in the condition code register (CC.F) to be set thus locking out other interrupts on FIRQ.

The NMI interrupt line is used to signal an urgent interrupt which should not be ignored. If CC.I or CC.F is set, the processor ignores interrupt requests on IRQ and FIRQ but NMI interrupts are always processed irrespective of the settings of these flags. The processing sequence is the same as that for an IRQ request although, obviously, a different interrupt vector is used.

As well as these hardware interrupts, the M6809 can also process so-called 'software interrupts'. Software interrupts occur when an SWI instruction is executed and, like hardware interrupts, they have an associated interrupt vector and service routine. Software interrupts are handled in the same way as IRQ interrupts and will be discussed in more detail in the following section where the SWI instruction is described.

In all, there are seven 'levels' of interrupt which can be processed by the M6809. The table below shows the locations of the interrupt vectors associated with each of these.

Vector location	Associated interrupt
FFF2:3	SWI3
FFF4:5	SWI2

FFF6:7	FIRQ
FFF8:9	IRQ
FFFA:B	SWI
FFFC:D	NMI
FFFE:F	RESET

The interrupt priorities are as follows:

RESET > NMI > SWI > FIRQ > IRQ > SWI2 > SWI3

Notice that this order is not the same as the order of the interrupt vectors.

8.1.1 Interrupt handling instructions

The sequence of actions described above which initiates an interrupt device routine takes place automatically whenever a hardware or software interrupt is detected. No explicit instruction is needed to start interrupt processing but a return from interrupt instruction is necessary to restart the interrupted process.

We have already introduced, in Chapter 3, the four M6809 instructions which are used in interrupt processing. Now, we describe each of these instructions, SWI, CWAI, SYNC, and RTI, in more detail.

RTI - Return from Interrupt

This instruction is always executed as the last instruction in an interrupt service routine. The instruction unstacks the CC register and examines CC.E. If it is unset, RTI then unstacks the next two stack bytes to PC thus returning control to the interrupted process.

If CC.E is set, this indicates that all the registers were stacked before entry to the interrupt service routine so RTI restores all register values from the stack. As PC is the last register restored, control is thus returned to the interrupted process.

SWI - Software Interrupt

There are three levels of software interrupt which may be used by the M6809 programmer. There are identified by the instructions SWI, SWI2, and SWI3. Each of these has a different priority in the order SWI > SWI2 > SWI3. The SWI instruction also has a higher priority than FIRQ and IRQ hardware interrupts and sets the interrupt mask bits CC.F and CC.I.

The sequence of operations executed when an SWI instruction is executed is the same as that which takes place automatically when an IRQ interrupt is detected. The register values of the interrupted process are stacked and control is transferred, via the appropriate interrupt vector, to the interrupt service routine.

The execution of an SWI instruction is not unlike calling a subroutine. The advantage to the programmer,

however, is that SWI instructions can be used to call system routines without the programmer having to know the routine address at either load or run time.

CWAI - Wait

The wait instruction is used to suspend the execution of a process until a hardware interrupt occurs. The instruction takes a single-byte operand which is anded with CC as the first step in the execution of the instruction. This means that the programmer can set up CC prior to its stacking and can guarantee the value of CC when the interrupted process is resumed.

After the ANDCC operation, CWAI sets CC.E and stacks all the processor registers. It then does nothing (waits) until a hardware interrupt occurs. If this is an NMI or RESET interrupt, it is immediately processed but if it is an IRQ or FIRQ interrupt and the corresponding mask bit is set in CC, the instruction continues to wait until a higher priority interrupt occurs or until the interrupt mask is cleared.

SYNC - Synchronise

The synchronise instruction, SYNC, is used to synchronise the operation of the M6809 processor and some other external device. A situation where this might be necessary is when data are being transferred from some fast I/O device, like a disk, to memory.

When a SYNC instruction is encountered, the processor enters a wait-for-interrupt loop which is called the 'syncing state'. If a hardware interrupt occurs and is not masked, the appropriate interrupt service routine is activated but the processor registers are not stacked.

If the interrupt is masked and the processor is in the syncing state, the effect of the interrupt is to cause the wait-for-interrupt loop to terminate. However, rather than cause a transfer of control to an interrupt service routine, execution continues with the instruction following SYNC. Thus processor and peripheral operation are synchronised by the interrupt.

Note that the programmer must explicitly set and clear the interrupt mask bits CC.I and CC.F by using ANDCC and ORCC instructions.

8.1.2 Dragon-specific interrupts

In the above section we explained that the M6809 system assumed interrupt vectors lying between addresses FFF2 and FFFF. In actual fact, the Dragon's address decoder (the SAM chip) maps these addresses onto alternative interrupt vectors in locations BFF2 to BFFF. This means that the interrupt vectors are in the BASIC read-only memory area and that their contents cannot be altered by the user.

As a result, these locations do not contain the

address of the interrupt service routine but contain the address of a secondary interrupt vector in RAM which should be set up with a jump to the address of the interrupt service routine. There is one exception to this. The RESET interrupt which is issued when the machine is switched on must, obviously, have a service routine in ROM. Its address is stored in the RESET interrupt vector.

The table below shows the how the interrupt vectors are assigned.

Vector	Contents	Use
BFF2:3	\$0100	SWI3 secondary vector
BFF4:5	\$0103	SWI2 secondary vector
BFF6:7	\$010F	FIRQ secondary vector
BFF8:9	\$010C	IRQ secondary vector
BFFA:B	\$0106	SWI secondary vector
BFFC:D	\$0109	NMI secondary vector
BFFE:F	\$B3B4	RESET service routine

The Dragon does not use all the M6809's interrupts but only makes use of the IRQ and FIRQ interrupts. Therefore, their secondary interrupt vectors are set up with a jump to their service routines, namely JMP \$9D93 and JMP \$B469 respectively. All the other secondary interrupt vectors are set to 0 and must be initialised by the user if required.

8.2 INPUT/OUTPUT PROGRAMMING TECHNIQUES

In memory-mapped input/output, described in Chapter 2, all I/O devices appear to the programmer as memory locations called I/O ports. Although memory-mapped I/O devices must communicate over the same bus structure as memory devices, it is not normally possible or desirable to connect a physical device directly to the computer's bus structure. Instead, a device interface is used to control the peripheral device according to commands from the CPU and to isolate that device from the bus structure. The interface may also convert data into whatever format is required by the physical device and vice versa.

However, unlike normal RAM memory where a read returns the last value written to a location, read and write operations to a particular I/O port address may be completely independent. In other words, an input port can occupy the same address as an output port and which one is selected is dependent on whether the operation is a read or a write.

Most device interfaces that have been designed for microprocessor use are 'programmable' thereby allowing them to be used in a variety of applications. A typical example of such an interface is the PIA (Peripheral Interface Adaptor) which is explained in detail in the

following section. This device has a number of built-in registers corresponding to addressed locations which can be accessed by the programmer. By writing the appropriate bit patterns into these registers, the programmer can configure the device in a variety of ways. In addition to these control registers, such an interface contains a status register which can be read by the I/O program to determine the state of the device; for example, 'is the data ready', 'can I send data now', etc.

Given the programming model of a peripheral device's interface, the programmer is faced with the problem of what technique to use to transfer data across that interface under program control. In this section we describe three I/O programming techniques unconditional I/O transfer, polled or conditional I/O transfer and interrupt-driven I/O transfer.

8.2.1 Unconditional I/O transfer

This is the simplest method of I/O transfer where the programmer always works on the assumption that the peripheral controller handling the I/O is always ready for output and always has up-to-date input information available. Data to be input or output can therefore be read from or written to the device at any time.

This, however, can lead to problems unless the exact timing of the I/O process is known. If a request is made for input before the peripheral controller has received that input from the peripheral device, the information returned will probably be that input in the previous operation. Similarly, if output is sent before the device has completed its previous operation, information may be lost.

As a result of these timing problems, this I/O programming technique is not advised unless the programmer is forced into it by primitive I/O devices which have no means of communicating their status to the processor.

8.2.2 Polled I/O transfer

This is a widely used I/O programming technique which involves a program polling the status of a device periodically. When the controller status indicates that input is available or that the device is ready for output, the I/O transfer takes place. Because the transfer is conditional on the controller status bits, this method of I/O programming is sometimes called conditional transfer.

In some cases the status bits of the peripheral controller are continually examined and the program waits for the I/O device to become available. In other cases, the status bits are examined at periodic intervals of, say, a millisecond. The program must ensure that the intervals between checks are not so

long that information is lost in the intervening period.

The following fragment of BASIC program illustrates this I/O programming technique by polling the button on a joystick to see whether or not it has been pressed.

```
100 BS = PEEK(&HFF00) ' BS = button state
110 IF (BS AND 1) = 0 THEN 100 'Wait for press
120 'Button pushed so deal with it here
```

The drawback of simply looping indefinitely until the button is pushed lies in the fact that it is necessary to 'busy wait' for the peripheral to become available. This time is completely wasted as no useful work can be done until the I/O transfer is complete. In our example, the program will 'hang' until the right joystick button is pressed.

In some situations this may be acceptable but it can cause problems in situations where several I/O devices have to be serviced. For example, say a two-player game involves movement and firing, both controlled by joystick. It would be impossible for one player to move until the other fired thus making evasion very difficult indeed!

However, the program can be modified to cope with this situation. Rather than looping indefinitely until an event occurs, the program can check the status bits of each device in turn. This is illustrated in the program below.

```
100 BS = PEEK(&HFF00)
110 IF(BS AND 1) = 0 THEN 140
120 ' Deal with right button
130 GOTO 100 ' Look at buttons again
140 IF(BS AND 2) = 0 THEN 100
150 ' Deal with left button
160 GOTO 100
```

However, the player with the left joystick still has a problem as the right joystick button always takes priority. It is polled first and, if it is held down, the left button is ignored. The solution to this particular problem is to poll the devices in a round-robin manner where the program establishes a polling order. The program polls each device in that order then cycles back to the beginning of the order. Our previous example can be converted to this form by removing line 130.

8.2.3 Interrupt-driven I/O transfer

We have already introduced the idea of an interrupt and suggested that they are very useful in I/O programming. In fact, the use of interrupt allows a programming technique to be devised which eliminates the need for a

'busy wait' or, indeed, any kind of polling system.

The disadvantage of the conditional transfer method of I/O programming is the fact that the processor cannot do useful work whilst it is waiting for a device to become ready for I/O. A more satisfactory technique is to have the device inform the processor when it is ready thus eliminating the need for the processor to check the device's status at periodic intervals. The sequence of events that occurs upon an interrupt has already been explained in the previous section and we shall not repeat them here.

Interrupt-driven transfers can be used as an I/O programming technique if the peripheral control register has the following facilities:

- (1) An interrupt enable/disable bit which allows the programmer to switch interrupts off and on.
- (2) An interrupt status bit which allows the service routine to find out which device has initiated the interrupt.

As it is common practice to connect a number of peripheral devices to the same interrupt line, the interrupt service routine must poll each of these devices to see which one caused the interrupt. Once the interrupt service routine has identified which device/condition is responsible, the condition is dealt with by normal instructions on an I/O port.

The service routine must also make sure that dealing with the condition includes removing the cause of the interrupt request otherwise, as soon as interrupts are enabled again, that same condition will cause another interrupt. The service routine will still, presumably, fail to remove the condition causing the interrupt and thus the system will hang indefinitely.

If you want to incorporate interrupts into a program you must carry out the following steps.

- (1) Write a suitable interrupt service routine.
- (2) Set up the appropriate interrupt vector with a reference to that routine.
- (3) Configure the device/interface for interrupts.
- (4) Enable (switch on) processor interrupts.

It is very important the steps 1 and 2 above be carried out before steps 3 and 4 and equally important that interrupts should be disabled whilst any changes to the interrupt system are being made. If this is not done and an interrupt occurs while the service routine is being changed unpredictable consequences can ensue when

the interrupt is processed.

To illustrate interrupt processing, we describe how to make amendments to the existing Dragon interrupt system.

The normal interrupt (IRQ) is derived from the video circuitry which provides an interrupt request every 20 milliseconds, that is, in correspondence with every cycle of the mains frequency. This will be slightly different in countries where the mains frequency is 60Hz rather than 50Hz. The role played by this interrupt is to update the system clock which is used by the BASIC function TIMER as well as the functions SOUND and PLAY. As we saw earlier, the IRQ vectors through a secondary vector at addresses \$010C, \$010D, and \$010E which normally contain a jump to the clock update service routine at address \$9D3D. Our example replaces the existing interrupt service routine with one that manipulates the text character in the top left hand corner of the screen. This manipulation is carried out as follows:

```
ADDCH  LDA $400    ; Pick up character at top left
        INCA      ; alter it by adding one
        STA $400    ; and return it to its place
```

To use this as an interrupt service routine, we must provide an IRQ vector set up routine. An example of this is::

```
IRQSET  ORCC #$10   ; Disable IRQ
        LDX #ADDCH  ; Set up entry address
        STX $10D    ; and store into IRQSV
        ANDCC #$EF  ; Enable IRQ
        RTS
```

The new service routine also has to be augmented by the code that removes the interrupt request. In this case, the interrupt request may be removed with a read operation to PIAO's peripheral data register which is mapped through address \$FF00. We also have to add an RTI instruction so that the interrupted process may be restarted after the character has been altered.

We have excluded a check to see what device is causing the interrupt since there is normally only one IRQing device on the Dragon. We have also omitted the code that configures/enables the device interface as this required detailed knowledge of the PIA which will not be described until later in this chapter. However, it is configured by the system as part of the initialisation sequence when the machine is switched on.

The final version of the new service routine is therefore:

```

* IRQSET - set up interrupt vector
*
* Must be called to install new service routine
*
IRQSET  ORCC #$10      ; Disable IRQ and put entry
        LDX #ADDCH    ; address of new service
*                               routine
        STX $010D     ; in secondary IRQ vector
        ANDCC #$EF    ; Enable IRQ
        RTS
* Interrupt service routine
ADDCH  LDA $FF00      ; Clear interrupt condition
        LDA $400      ; Pick up LH corner
        INCA          ; Alter it
        STA $400      ; And put it back
        RTI           ; Return from interrupt

```

Once the interrupt vector has been set up and IRQ interrupts enabled you should see the top left hand character cycle through the 256 character set of the Dragon at the rate of 50 characters/second.

You will now find that the TIMER, PLAY, and SOUND functions will not work properly. We leave it as an exercise to the reader to add the above code to the existing service routine so that the clock is updated and a character is updated on the screen.

8.3 THE PERIPHERAL INTERFACE ADAPTOR - PIA

The peripheral interface adaptors or PIAs which act as the Dragon's I/O interfaces are multi-purpose peripheral controllers. We saw in the previous chapter how one of these PIAs played an essential role in the graphics display hardware where it is used to set up the video display generator. This section is an overview of the functions of a PIA but, for those readers who require further information, a full technical description of the PIA chip is provided in Appendix 4.

The function of a PIA is to interface a member of the M6800 processor family (in this case, the M6809) to various peripherals and, to do so, it provides various peripheral data input/output lines as well as peripheral control/interrupt lines. Each PIA has two 8-bit peripheral data buses and four control lines thus giving a total of 20 lines which can be used to control and interface peripheral devices.

The flexibility of the PIA is derived from the fact that it is a 'programmable' device. This does not mean that it can execute machine code instructions but rather that the PIA is not dedicated to a specific peripheral type. It contains various internal registers which can be manipulated by the assembly language programmer to configure the PIA to a particular mode of

operation. Each of the peripheral data lines can be configured to act as an input or as an output and each of the control/interrupt lines may be set up in one of several control modes.

A PIA is functionally split into two, independent sides called the A-side and the B-side. Each side is configured and controlled by three internal 8-bit registers. These are:

- (1) The control register (CR)
- (2) The data direction register (DDR)
- (3) The peripheral data register (PDR)

Effectively, therefore, we have four programmable input/output interfaces which may be used separately or together. Figure 8.2 is a schematic representation of this system.

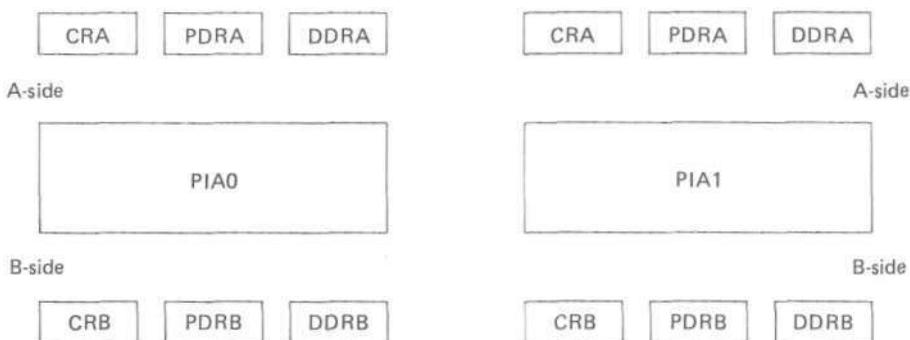


Fig. 8.2 PIA organisation

The control register is used to configure/control the four peripheral control lines which are named CA1, CA2, CB1, and CB2. It also allows the assembly language programmer to enable and disable the interrupt lines IRQA and IRQB and to monitor the status of the interrupt flags IRQA1, IRQA2, IRQB1, and IRQB2. A full description of the functions of this register is provided in the PIA Data Sheet (Appendix 4).

There is one bit in the control register which is not used in the configuration of the peripheral control/interrupt lines. This is the data direction access bit which is used to select between the data direction register and the peripheral data register. The reason for this is that the PIA only responds to four unique addresses and, as the programmer always needs access to the control registers, the other two

addresses must be shared by the data direction register and the peripheral data register.

The data direction register (DDR) is used to control the direction of data through each corresponding peripheral data line of the peripheral data register. If a DDR bit is 0, this means that the corresponding peripheral data line is an input whereas if the DDR bit is 1, the associated data line is an output line.

Therefore, each peripheral data register may have any combination of input and output lines thus providing a good deal of flexibility when interfacing the PIA to external devices. However, it is the responsibility of the programmer to keep track of which lines are inputs and which lines are outputs and to make sure that when outputs are updated, other, independent outputs are not affected.

The data direction register and the peripheral data register share an address and the particular register addressed depends on the setting of the data direction access bit in the control register. This sharing does not usually cause problems as typical usage involves setting up the data direction register and then accessing the peripheral data register without further changes to the DDR.

The peripheral data register is used to transfer data to and from peripheral devices. Each of the peripheral data lines can be configured as an input or as an output as described above. When a line is configured as an output, it will go HI when the corresponding bit in the PDR is set to 1 and will go LO when the corresponding bit is cleared in the PDR. In general, after an output has been written it can be read back although this is dependent on the loading of the line (see Appendix 4).

When a line is configured as an input, the data on the peripheral data line appears directly on the corresponding M6809 data line during a read operation. As output line states are also read back, the programmer has to explicitly mask them out when the PDR is accessed.

8.3.1 The Dragon's PIAs

The Dragon's I/O subsystem makes use of two PIAs named PIA0 and PIA1. This means that there are 40 peripheral data/control lines but, as some devices share lines, more than 40 lines can actually be supported. Each PIA responds to four unique addresses with PIA0 associated with addresses FF00 to FF03 and PIA1 associated with addresses FF20 to FF23.

These address ranges can be further subdivided into the individual addresses of the registers within the PIA. In the examples in this chapter, we shall make use of these addresses and shall refer to them using the symbolic names defined in the table below.

* Equates for PIA 0

* A- side registers

```
PODDRA EQU $FF00 ; Data direction register
POPDRB EQU P0DDRA ; Peripheral data register
POCRA EQU $FF01 ; Control register
```

* B-side registers

```
PODDRB EQU $FF02 ; Data direction register
POPDRB EQU P0DDRB ; Peripheral data register
POCRB EQU $FF03 ; Control register
```

* Equates for PIA1

* A-side registers

```
P1DDRA EQU $FF20 ; Data direction register
P1PDRA EQU P1DDRA ; Peripheral data register
P1CRA EQU $FF21 ; Control register
```

* B-side registers

```
P1DDRB EQU $FF22 ; Data direction register
P1PDRB EQU P1DDRB ; Peripheral data register
P1CRB EQU $FF23 ; Control register
```

The connections of the Dragon's PIA registers to specific devices is summarised in Appendix 7.

8.4 INPUT/OUTPUT DEVICES

Apart from the display, which we discussed in the previous chapter, the Dragon is equipped with a variety of input/output ports to which peripheral devices may be connected. Some of these ports, such as the keyboard port, are connected to I/O devices which are an inherent part of the system. Others, such as the printer port and cassette port, are available for the user to connect his own peripherals.

Each I/O port in the system is connected to either PIA1 or PIA0 so that each PIA is an I/O controller for a number of devices. In particular, PIA0 is responsible for the keyboard, the joystick control, sound source selection, the printer port and video synchronisation. PIA1 is responsible for printer handshake control, sound generation, the cassette port, VDG mode selection, D-to-A voltage control, RAM type detection and ROM cartridge detection.

In this section we shall look at how the user may access these I/O ports and how to make use of peripheral devices connected to these I/O ports.

8.4.1 Keyboard control

The Dragon's keyboard is of extremely simple design. It consists of a number of keyswitches arranged in a 7 by 8 matrix with the columns of the matrix connected to the peripheral data lines of the B-side of PIA0 and the matrix rows connected to the peripheral data lines on

the A-side of the same PIA. When a key is pressed, this causes the row/column position of that key to be short-circuited and this change of state can be detected by the keyboard control routine.

The keyboard is connected so that the column lines (PB0-PB7) are configured as outputs and the row lines (PA0-PA6) are configured as inputs. With this setup, a key depression can be detected by outputting a 0 to a column line then reading the input lines PA0-PA6. If no key in that column has been pressed, the result read back will be 1111111 whereas a key depression causes a 0 to be read in one of PA0-PA6.

The keyboard scanning routine is activated approximately every ten milliseconds and it outputs a zero to each of the columns in turn. It immediately looks at the inputs PA0-PA6 and, if the result is not 1111111, it knows that a key has been pressed. The keyboard scanner keeps track of which column has an associated row input containing a zero and from the position of that 0 in PA0-PA6, it can work out which key has actually been pressed.

The actual arrangement of the keys in the matrix is shown in the table below.

	PB0	PB1	PB2	PB3	PB4	PB5	PB6	PB7
PA0	0	1	2	3	4	5	6	7
PA1	8	9	*	;	,	-	•	/
PA2	@	A	B	C	D	E	F	G
PA3	H	I	J	K	L	M	N	0
PA4	P	Q	R	S	T	U	V	W
PA5	X	Y	Z	Up	Down	Left	Right	Space
PA6	ENT	CLR	BRK	N/C	N/C	N/C	N/C	SHFT

To illustrate how a key depression can be detected, say the user presses the 'U' key. The keyboard scanning routine outputs a zero on lines PB0-PB7 and, when a zero is output on PB5, an input pattern containing a zero will be detected. This zero will be in position PA4 thus indicating that 'U' (coordinate PA4/PB5) has been pressed.

In order to keep track of which keys are pressed so that it can ignore the same key closure on the next scan (remember it scans about 100 times per second) and perform key rollover, the keyboard scan routine maintains a record of key closures in nine bytes of RAM at addresses 151-159 inclusive.

The first byte (151) records the seven row states, that is, it records whether any of the keys in a particular row are pressed. When a key in a row is pressed, the corresponding bit in location 151 is set to 0. Thus, if the 'D' key is pressed, bit B2 in 151 is cleared to indicate that a key in row 2 has been depressed. For other rows, where no key has been pressed, the bits in byte 151 are set.

The individual bits in 151 are used by the keyboard scan routine to determine whether there has been a change of state of any of the rows. The value read in PA0-PA6 is compared with the value in location 151 and, if these values are the same, the keyboard scanner assumes that the same key is still held down by the user. This is fairly sensible as, when you press a key, you are likely to hold it down for more than a hundredth of a second. If there has been a change to any of the rows, because the user has taken his finger off a key perhaps, then the value of location 151 is modified to reflect this and a full keyboard scan takes place to find out which key has been depressed or released.

This two-stage scanning technique is used to speed up the scanning routine although it does mean that key rollover does not occur for keys on the same row. In other words, holding the 'A' key down and then pressing a key on the same row, say 'C', does not register the new character but pressing a key on a different row, say 'H', does register.

The remaining 8 bytes 152-159 are used to record the state of all the keys in the matrix as each byte records the state of the rows for its corresponding column. Column 0 state is held in 152, column 1 in 153, etc. When a key is pressed, then the corresponding bit in the column byte is cleared. For example, if address 152 contains FE, this indicates that the '0' key has been pressed.

One drawback of this technique is that it prevents the same key from being recognised again unless it is released and re-pressed. Furthermore, if a key is held down, it prevents other keys in the same matrix row from being recognised. This is illustrated by the BASIC program below which, at first sight, seems to be able to recognise all the arrow keys.

```

10 R5 = &H20 'Row 5's position in column byte
20 C4 = &H155 'Column 4's byte
30 C5 = &H156 'Column 5's byte
40 C6 = &H157 'Column 6's byte
50 C7 = &H158 'Column 7's byte
60 IF (PEEK(C4) AND R5) = 0 THEN PRINT "UP"
70 IF (PEEK(C5) AND R5) = 0 THEN PRINT "DOWN"
80 IF (PEEK(C6) AND R5) = 0 THEN PRINT "LEFT"
90 IF (PEEK(C7) AND R5) = 0 THEN PRINT "RIGHT"
100 GOTO 60

```

In fact, this program does not recognise more than one arrow key being pressed at a time. As the arrow keys are on the same row, the keyboard scan routine does not do a full scan because the row state byte indicates that the row state has not changed. This is perfectly reasonable for most normal typing but can be limiting

for the game programmer who wishes to provide keys which allow simultaneous movement and firing.

However, it is possible to program around this limitation and to force a complete scan of the keyboard by setting all bits in byte 151 before each IF statement. This can be implemented by including the statement POKE &H151,&HFF as statements 55, 65, 75, and 85 in the above program and changing line 100 to goto 55.

This has the effect of fooling the keyboard scanner into thinking that all the keys on the row have been released and so the next scan will register any keys that are down as new closures. Notice that it is necessary to poke the value FF to the column state byte between every BASIC statement as the keyboard scan routine is executed after every BASIC statement and sets byte 151 to its old value. Although it works, this technique is clumsy and we shall describe a better, more elegant technique in Chapter 9 which does not involve such program modifications.

A keyboard auto-repeat facility, where holding a key down causes that character to be continuously input, is very useful in applications such as games, where keys may indicate movement, text preparation, where you might want to input strings of the same character and screen layout design.

This facility is not provided on the Dragon because a complete keyboard scan is not carried out when a key is held down. A complete scan can be forced, however, by using a technique comparable to that above and poking a value to the individual column bytes. By setting a particular column byte, we can force an already pressed key to be registered as a new key closure.

The following BASIC program illustrates this facility for the INKEY\$ function

```

10 RB = &H151      ' Row state byte
20 CB = &H152      ' Column state byte
30 POKE RB,&HFF : POKE CB,&HFF
40 A$ = INKEY$ : IF A$ = "" THEN 30
50 PRINT A$;
60 GOTO 30

```

It is left as an exercise to the reader to predict which keys will auto-repeat given the above program and to modify the program so that auto-repeat will work with all keys.

It may seem that you could write your own simplified keyboard controller in BASIC which directly manipulates the PIAs using POKE and PEEK statements. Unfortunately, this is not possible because there is no way of disabling the keyboard polling routine. If you try to control the keyboard yourself from within a BASIC

program, you are liable to get interactions between your controller and the standard BASIC polling routine. Naturally, you could write your own routine in assembly code.

8.4.2 Printer control

The Dragon's printer interface is provided so that the user may connect his own printer to the system. The interface is configured as a so-called Centronics interface which means any one of a number of printers designed to use this interface type may be connected to the Dragon. The user is not restricted to a single specialised printer as is the case with some personal computer systems.

Character data is output to the printer over eight interface data lines D0-D7 which are connected to P0PDRB's peripheral data lines. These PIA peripheral data lines are also used by the keyboard column lines but, in practice, this does not cause problems. When characters are being printed, the I/O system knows that the keyboard should not be scanned and vice-versa.

There are also a number of control (handshake) lines which coordinate data transfer to the printer. These are called 'printer strobe', 'printer busy', and 'printer acknowledge'. Although the Dragon hardware supports all three of these control lines only two of them (strobe and busy) are used by the standard Dragon software.

The printer interface control lines are connected as follows:

- (1) The 'printer busy' line is connected to bit 0 of PIA1's B-side peripheral data register (P1PDRB/PB0). This line is set up in the associated DDR as an input.
- (2) The 'printer strobe' line is connected to bit 1 of PIA1's A-side peripheral data register (P1PDRA/PA1). This line is configured as an output.
- (3) The 'acknowledge' line is connected to PIA1's CAI interrupt input but this line is not actually used by the Dragon's I/O system. Naturally, however, you can make use of it if you wish to write your own printer control programs.

The printer interface connections are shown in the block diagram of the I/O system (Figure 8.1) which shows the interface port pins and their associated control/data lines.

It is fairly straightforward to send characters to the printer but appropriate control signals must be organised so that a character is not sent before the

printer is ready for it. The printer's state of readiness is indicated using the 'printer busy' line which is HI when the printer cannot accept a character for printing and goes LO when the printer is not busy. To tell the printer that a character is to be printed, that character must be set up on the data lines and the 'printer strobe' line must be taken from HI to LO and then back to HI again.

To send a character to the printer, the sequence of events is therefore:

```

10 IF 'printer busy' THEN GOTO 10
20 Printer data lines = Character to be printed
30 Printer strobe = LO : Printer strobe = HI

```

The strobe line is actually buffered through an inverting buffer so that a 0 set up in PA1 results in a 1 on the actual strobe pin and vice-versa. Therefore to set the strobe line LO and then HI, you must send signals which first set it HI and then LO.

If you wish to use your own printing routine, you must manipulate the PIA using assembler rather than BASIC POKE and PEEK statements. The reason for this is that the printer and the keyboard share PIA data lines and it is not possible to disable BASIC'S keyboard polling routine. Therefore, if you try to poke to the PIA locations associated with the printer, to get direct output say, the keyboard scanner resets the PIA and your pokes will have no effect.

The standard Dragon printer output routine is called LPOUT and it can be addressed through location 800F. This routine expects register A to contain the character code, in ASCII, of the character to be printed.

Because a variety of printers can be attached to the Dragon, some of the actions performed by this routine are determined by a set of parameters stored in RAM locations. The initial values of these parameters are set up for the most commonly connected printers but they can be modified to configure the routine for other printers. Modification involves poking the values for your printer to the appropriate location.

The table below gives the addresses of the RAM locations used by the printer routine and briefly describes the functions of the information stored at these addresses.

Address	Initial value	Function
99	\$10 (16)	Line printer comma field width. This is used by BASIC to determine which columns to print items separated by a comma in a PRINT statement.

9B	\$84 (132)	Line printer width, that is, the length of the printed line.
9C	\$00 (0)	The line printer character position. LPOUT keeps track of the position of the next character to be printed by updating this byte.
148	\$FF (255)	Automatic line feed on buffer full. If the line printer does not automatically flush its buffer when it is full then setting this byte to 0 will cause the routine to send an end-of-line sequence to the printer which forces the buffer to be flushed.
14A	\$01 (1)	The number of characters in an end-of-line sequence. The end-of-line characters follow in succeeding locations from 14B to 150. Thus there may be a maximum of 6 characters in an end-of-line sequence.
14B	\$0D (13)	Carriage return (CR) code (normal EOL sequence)
14C	\$0A (10)	Line feed (LF) code. If the printer needs a CRLF end-of-line sequence, 14A should be set to 2.

As an example of how the line printer routine may be reconfigured, the following BASIC direct statements set up the printer width to be the same as the Dragon's display width.

```
POKE $H148,0
POKE &H9B,32
LLIST
```

The routine LPOUT carries out a number of 'housekeeping' duties such as forcing end-of-line sequences, outputting extra spaces to cause line feeds, etc. Sometimes, you don't really want this to happen but all you really want is a routine which simply pumps characters to the printer. If this is what you need, there is a no-frills printer output routine called TXLPCH with entry point at location BCF5. Again, this routine expects register A to contain the character to be printed.

8.4.3 Sound control

In common with many other modern personal computers, the Dragon is equipped with facilities for sound

generation. In this respect, personal computers differ from larger, professional machines and, unlike graphics say, the creative use of sound in human/computer interaction is a largely unexplored area. At the moment, the sound generation system is mostly used to provide sound effects for games but it seems likely that applications for sound generation will be discovered in many other areas of computer usage.

There are four possible sound sources in the Dragon. These are:

- (1) From the 6-bit digital to analogue converter.
- (2) From the cassette unit.
- (3) From the cartridge port.
- (4) From PBI of P1PDRB.

The first three of these sound sources above are analogue sources which means that they generate a varying voltage level which is converted to sound. The PIA bit, on the other hand, is a binary sound source which generates either a HI or a LO pulse. Sound output is channelled to the television's speaker by modulating the sound signal with the video signal that is fed to the aerial (antenna) input of the user's television set.

The analogue sound sources are connected to the inputs of a device called an 'analogue multiplexor'. This is a device which can accept a number of inputs and, according to control line settings, switch any one of these inputs to its output line. The Dragon's analogue multiplexor has four input lines, a single output line and three control lines. One of the input lines is not used so it is permanently LO. Two of the control lines are used to select which sound source is to be routed to the output line with the third control line used to enable and disable the multiplexor output. This output must be disabled if a binary sound pulse generated from the PIA is to be used as the input to the sound generator.

The multiplexor control lines are connected to PIA control lines. The lines used are PIA1-CB2 which is connected to the output enable/disable multiplexor control line and PIA0-CB2 and PIA0-CA2 which are used for sound source selection. The table below shows the values which these control lines may take and the associated sound selections.

Sound enable	Sound select		Sound source
PIA1-CB2	PIA0-CB2	PIA0-CA2	
1	0	0	6-bit DAC
1	0	1	Cassette

1	1	0	Cartridge
1	1	1	Not used
0	X	X	Single bit sound

To select a sound source, you must output an appropriate value to the PIA control lines which are connected to the analogue multiplexor. When you do so, it is absolutely vital that you preserve the value of the other bits in the PIA's control register otherwise you are liable to cause all sorts of havoc. An example BASIC subroutine which selects the 6-bit DAC sound source is shown below.

```

2000 ' Select and enable 6-bit DAC sound source
2010 ' POCRA = &HFF01: POCRB = &HFF03 : P1CRB = &HFF23
2019 ' Now set PIA0-CA2 LO
2020 POKE &HFF01,(PEEK(&HFF01) AND &HF7)
2029 ' Set PIA0-CB2 LO
2030 POKE &HFF03,(PEEK(&HFF03) AND &HF7)
2040 POKE &HFF23,(PEEK(&HFF23) OR 8) ' Enable sound
2050 RETURN

```

Rather than write your own routine for sound source selection, you can make use of an inbuilt system routine which we shall call SNDSEL. This can be called via address BD41. Its specification is:

```

* SNDSEL - Selects 1 of 4 input lines for analogue MUX
*
* Register inputs B - input line number
*                   0 = DAC, 1 = Cassette
*                   2 = Cartridge
* Registers destroyed U, A,B,CC

```

This routine does not enable the output of the multiplexor so you must do this as a separate step. The routine below will switch it on and select the source required by calling SNDSEL.

```

* SNDON - Switch on sound output from MUX

```

```

* Register inputs - as SNDSEL
*

```

```

SNDON  PSHS U,A,B          ; Save registers
       JSR SNDSEL         ; SNDSEL equated elsewhere
       LDA P1CRB          ; Now enable the
       ORA #8             ; source by setting
       STA P1CRB          ; PIA1-CB2 HI
       PULS U,A,B,PC      ; Restore and return

```

To switch off the sound source, a similar routine is required although, obviously, SNDSEL is not called and, rather than or 8 into P1CRB, you must and \$F7 into that register.

The 6-bit digital to analogue converter (DAC) is a device which takes a 6-bit binary value and converts it to an analogue voltage. Such a device is very useful as many external devices rely on analogue rather than binary signals whereas the M6809, obviously, deals only with binary information. As well as being an integral part of the sound generation system, the DAC is also used in joystick manipulation and in the recording of data on cassette. We shall look at these applications in later sections of this chapter.

You don't have to understand how the DAC works to make use of it. Basically, it converts a 6-bit value between 0 and 63 to an equivalent voltage in the range 0.25V to 4.75V. The approximate output voltage corresponding to an input signal may be computed according to the following formula.

$$\text{Output voltage} = (\text{Input value} * 0.0715) + 0.25\text{V}$$

The DAC's six input lines are connected to P1PDRA's peripheral data lines PA2-PA6. Because the top 6 bits of P1PDRA are used to control these input lines, the 6-bit value to be input to the DAC must be offset by 2 bit positions before loading it into the PIA's register. Furthermore, the bottom 2 PIA bits are used for other purposes so their values must be preserved before the PIA is set up for DAC input.

The following assembly code routine accomplishes this by shifting the input value then oring it into P1PDRA.

* DACOUT - Output a 6-bit value to DAC

* Register inputs A - 6 bit value

*

```
DACOUT  PSHS A           ; Save register
        ASLA            ; Move bottom 6 bits
        ASLA            ; into top 6 bits
        STA , -S        ; and save until later
        LDA P1PDRA     ; Preserve the original
        ANDA #$03      ; bottom 2 bits
        ORA , S+       ; and or in new value
        STA P1PDRA     ; Output to DAC
        PULS A, PC     ; Restore and return
```

The equivalent BASIC routine is:

```
1000 ' BASIC DAC output routine
1010 ' Input parameter N, 6-bit value
1020 POKE &HFF20, ((PEEK(&HFF20) AND 3) OR (N * 4))
1030 RETURN
```

The DAC is used to support Extended Color BASIC'S SOUND and PLAY commands. To create a sound waveform with this

device, suitable values must be sent to it at appropriate intervals. The volume of the sound output is determined by the magnitude of the values output to the DAC and the pitch depends on how often the DAC is updated with new waveform values.

We illustrate this with Program 8.1 which generates a very crude approximation of a sine wave. This program requires that a DAC sound selector routine be available at line 2000 and a routine to send a value N to the DAC be available at line 1000.

```

10 GOSUB 2000 'Select DAC sound source
20 FOR V = 1 to 15 ' V = volume
30 V2 = V*2: V4 = V*4 'V2=mid-volume,V4=max-volume
40 PRINT "MAX VOLUME = ";V4
50 TIMER = 0 ' Reset time period
60 FOR C = 1 to 50 'No. of cycles
70 N=0: GOSUB 1000 ' Vary the waveform
80 N=V: GOSUB 1000 ' from min volume
90 N=V2: GOSUB 1000 ' through mid volume
100 N=V4: GOSUB 1000 ' to max volume
110 N=V2: GOSUB 1000 ' and back down
120 N=V: GOSUB 1000 ' to min volume
130 NEXT C ' Start next cycle
140 CP = (TIMER/50)/50 'Cycle period
150 PRINT "CYCLE PERIOD = ";CP;" OF A SECOND"
160 NEXT V ' Repeat for next volume setting
170 END

```

Program 8.1 Sound output generation via the DAC

If you run this program, you should hear a steady but ragged note with increasing volume. The print statements giving details of the volume and the cycle period give an indication of how long it takes to update the DAC when using BASIC and the slowness of BASIC does limit the upper frequency range. The note sounds ragged because the DAC is not producing a continuously varying voltage which is necessary to produce a pure sine wave. Rather, the voltage steps from one value to another and to produce a smoother note you would have to include more approximations to a sine waveform by reducing the differences in step levels. Since this would slow the system down, this would also reduce the upper frequency range of the sound.

Because of the restrictions on the upper frequency range when programming in BASIC, it is better to use an assembly code subroutine for DAC sound generation.

The input signal from the cassette recorder can also be used as a sound source. All that happens in this case is that the multiplexor routes the cassette input signal directly to the TV loudspeaker so you can play back music, commentary or anything else you have

recorded on the cassette under the control of a program.

This facility is directly supported in BASIC by the AUDIO ON and AUDIO OFF commands which actually switch the multiplexor to cassette input. You can use the SNDON routine above to perform the equivalent action if you are programming in assembly code.

Similarly, the cartridge ROM port has one of its input lines connected to the sound multiplexor and an input signal from the cartridge can be switched to the loudspeaker. This is a rarely-used facility and if you are using the cartridge input port to connect some other device to your Dragon (we discuss this later), we do not recommend that you use this sound input because the voltage levels for it are unspecified.

Apart from the DAC, the alternative method of sound generation on the Dragon is to use the single bit sound source. In this case the sound multiplexor is bypassed and must be disabled. The signal from the PIA, which is the single bit sound signal, is fed directly onto the output line of the multiplexor. Naturally, as this is a binary value, the sound generated consists of a train of pulses corresponding to the binary input to PB1 of the PIA.

To make use of the single bit sound source, the programmer must take the following steps.

- (1) Disable the sound multiplexor output
- (2) Select P1DDRB by clearing bit 2 of P1CRB
- (3) Configure PB1 as an output by setting bit 1 of P1DDRB
- (4) Select P1PDRB by setting bit 2 of P1CRB

We demonstrate this in the following BASIC program which shows how single bit sound may be used.

```

10 GOSUB 3000 ' Disable MUX output
20 GOSUB 4000 ' Select single bit sound
30 TIMER = 0 ' Reset timer
40 OV = PEEK(&HFF22) ' Save value of P1PDRB
50 HI = OV OR 2 ' To set PB1 HI
60 LO = OV AND &HFD ' To set PB1 LO
70 FOR C = 1 TO 50
80 POKE &HFF22,HI ' Set PB1 HI
90 POKE &HFF22,LO ' and LO
100 NEXT C
110 CP = (TIMER/50)/50 ' Cycle period
120 PRINT "CYCLE PERIOD WAS ";CP;" OF A SECOND"
130 GOSUB 5000 'Switch off single bit source
140 END

```

We leave it as an exercise for the reader to write the subroutines at lines 3000, 4000 and 5000 which disable the multiplexor output and select the single bit sound source. It is important to switch off the single bit sound as it may interfere with the output when the multiplexor is in use. To switch off this sound source, P1PDRB-PB1 should be configured as an input rather than an output.

8.4.4 Cassette control

One of the great advantages of personal computer systems is that they can make use of commercially available tape cassette recorders and standard cassette tapes for input and output. These recorders are designed for recording and playing back music or speech so are analogue devices. Therefore, to use them for data input and output, there must be a way of converting a binary output to an analogue signal recorded on the tape and vice-versa for inputs from the tape.

The technique used in the Dragon to record programs and data on a cassette tape is known as Frequency Shift Keying (FSK). This technique involves representing ones and zeros as different frequencies as they are recorded. A LO signal (binary 0) is recorded as a single cycle of frequency 1200Hz and a HI signal (binary 1) is recorded as a single cycle of frequency 2400Hz. This means that the effective data transfer rate from Dragon to cassette recorder is 1800 bits/second as, on average, there will be an equal number of ones and zeros recorded for a program or data file.

The choice of recording frequencies is not entirely arbitrary as the 6-bit DAC, described in section 8.4.3, is used to generate the appropriate sine waves of 1200Hz and 2400Hz, with the signal attenuated to about 1V before output to the recorder. The choice of frequencies is the result of a trade-off between the number of approximations required to produce these sine waves and the execution speed of the instructions needed to update the DAC.

When a tape is used to store programs or data, it is not sufficient simply to dump these on the tape and hope that you will be able to read them back at some later date. Rather, the information written to the tape must be preceded by header information which gives a name to the information stored, specifies its type, and, perhaps, contains information which allows hardware synchronisation. As well as this header, a trailer or end-of-file block must also be written marking the end of the data or program on the tape.

All the information on the tape, including the initial information and end-of-file information, is written out in a sequence of blocks which may store

from 0 to 255 information bytes. Each block also contains header and trailer information as well as actual data.

The overall tape format assumed by the Dragon splits the tape into six logical sections:

- (1) A leader consisting of 128 bytes where each byte has the hex value 55.
- (2) A namefile block.
- (3) A blank section of tape to allow BASIC time to evaluate the namefile block. Around 0.5 seconds are needed for this.
- (4) Another leader of 128 bytes where each byte has the hex value 55.
- (5) One or more data blocks.
- (6) An end-of-file block.

Namefile, data blocks and end-of-file blocks all share the same format with an identification byte used to mark the block type. This format is:

- (1) A leader byte - 55 (hex).
- (2) A synchronisation byte - 3C (hex).
- (3) A block type byte where 01 means data block, FF means end-of-file block and 00 means namefile block.
- (4) A block length byte - 00-FF (hex).
- (5) 0-255 bytes of data.
- (6) A checksum byte which is the sum of all the data + block type + block length.
- (7) A trailer byte - 55 (hex).

An end-of-file block has no associated data bytes and a namefile block has 15 data bytes giving the name and type of the file. These 15 bytes are organised as follows:

- (1) An 8-byte program name.
- (2) A 1-byte file type where 00 indicates a BASIC file, 01 indicates a data file and 02 a machine language file.

- (3) An ASCII flag byte which indicates if the file is recorded as ASCII characters or as binary digits.
- (4) A gap flag byte set to 01 when the tape is written as a contiguous stream of blocks and to FF when there are gaps between blocks. When writing data to the tape, the associated processing time usually means that output is not continuous and that the tape is switched off and on between blocks. The gaps are the result of this switching.
- (5) Two bytes for the start address of a machine language program.
- (6) Two bytes for the load address of a machine language program.

The length of the leader on the tape is held as a BASIC system variable in locations 90:91 and it is possible for the user to modify this length to increase the length of the tape leader. By poking a value of 1 to address 90, the leader is made 3 times longer and poking a value of 2 gives a leader 5 times longer than normal. The advantage of this is that it gives the cassette recorder's automatic volume control more time to stabilise thus reducing the probability of cassette I/O errors.

After a program or data file has been recorded, you can verify the tape by using the SKIPF command. This reads the tape in exactly the same way as CLOAD/CLOADM and reports any errors. It does not, of course, load the contents of the tape.

We do not recommend that you write your own routines for cassette input and output. Rather, it is much better to use the built-in system routines which have been implemented as part of the Extended Color BASIC system. This provides routines to turn the cassette on and off, to read and write blocks of data to the cassette, to read and write single bytes to the cassette and to read a single bit from the cassette.

A specification for each of these routines in the form of a header comment is provided below.

```
* CASON - turn on cassette motor
*
* Register inputs - NONE
* Registers destroyed - X, A, CC
* Turns on motor and delays until motor comes up to
* speed. Delay value is a 16-bit value at location 95
* with initial value = $DA5C representing 0.5 seconds
*
* WRTLDR - Turn on tape for writing
*
```

```
* Register inputs - NONE
* Registers destroyed X, A, CC
* This routine disables IRQ and FIRQ interrupt lines
* to avoid interruption of recording then calls CASON.
* It then outputs a bit sync leader. The number of
* bytes in the leader is a 16-bit value at address 90.
* It has default value $0080.
*
* CSRDON - turn on tape for reading
*
* Register inputs NONE
* Registers destroyed ALL
* This routine disables FIRQ
* and IRQ, calls CASON and uses the bit sync
* information to synchronise the tape input.
*
* CASOFF - turn off cassette
*
* Register inputs - NONE
* Registers destroyed A,CC
* Re-enables IRQ and FIRQ and turns off cassette motor
*
* CBOUT - write byte to cassette
* Register inputs A - byte to be written
* Registers destroyed Y, B, CC
*
* CBIN - read byte from tape
*
* Register inputs - NONE
* Register output A - byte read from tape
* Registers destroyed A,B,CC
*
* BITIN - read a bit from cassette
*
* Register inputs NONE
* Register outputs - bit read is carry bit, CC.C.
* Registers destroyed B,CC
*
* BLKOUT - Output data block
*
* Register inputs NONE
* Registers destroyed ALL
* WRTLDR must be called before this routine to get
* the tape up to speed and to write out tape leader.
* Input parameters are passed in RAM locations
* $7C - block type
* $7D - number of data bytes
* $7E:7F - address in memory of start of block
* Interrupts remain disabled on exit from this routine
*
* BLKIN - read a block from tape
*
* Register inputs NONE
```

```

* Register outputs CC.Z = 0 if I/O error
*                   CC.Z = 1 if no I/O error
* Registers destroyed ALL except U and Y
* CSRDON must be called before this routine to get
* the tape up to speed and turn on bit sync.
* Input and output parameters are in RAM locations
* $7E:7F - 16-bit start address of block to be read
* $7C - block type read from tape
* $7D - Number of bytes read from tape
* $81 - Error indicator. No errors=0, Checksum error=1
* Memory error = 2
* On exit, interrupts remain disabled

```

The entry points to these routines are held in a direct jump table starting at address 8000 and/or in an indirect jump table located at address A000. The table below shows which routine is located at which address.

Routine	Direct	jump address	Indirect	jump address
CASON		8015	-	
CASOFF		8018	-	
WRTLDR		801B	A00C	
CBOUT		801E	-	
CSRDON		8021	A004	
CBIN		8024	-	
BITIN		8027	-	
BLKIN		-	A006	
BLKOUT		-	A008	

We can illustrate the use of the output routines with the following program which writes a data block to the cassette.

```

CASOFF    EQU $8018
WRTLDR    EQU $801B
DBADR     EQU $7E
BLKTYP    EQU $7C
DBLEN     EQU $7D
BLKOUT    EQU $A008
          LDX #BLOCK      ; Set up address of
          STX DBADR       ; data block
          LDA #$01        ; File type = DATA
          STA BLKTYP
          LDA #255        ; Number of bytes
          STA DBLEN       ; is set up
          JSR WRTLDR      ; Prior to writing tape
          JSR (BLKOUT)    ; Write block
          JSR CASOFF      ; Switch off tape

```

An input program which reads blocks from the tape and displays their contents on the screen is shown below. Assume that the equates in the above program have been made.

```

CSRDON   EQU  $8021
BLKERR   EQU  $81
BLKIN    EQU  $A006
          LDX  #$400      ; Screen RAM
          STX  DBADR      ; as block copy area
NXTFIL   JSR  CRDON      ; Turn on tape for reading
NXTBLK   JSR  (BLKIN)    ; Read a block
          BNE  BINERR     ; Abort on error
          LDA  #$FF       ; Check for end of file
          CMPA BLKTYP
          BNE  NXTBLK     ; Not eof, get next block
          BRA  NXTFIL     ; Otherwise, re-sync and get
*
BINERR   JSR  CASOFF
          RTS

```

These routines have shown how blocks of data can be read from and written to a cassette and the CASON and CASOFF routines are generally used in this context. However, there may be circumstances where you wish to avoid using these routines as they both manipulate the interrupt mask bits in the condition code register.

The relay used to control the cassette motor is connected to the CA2 control line output of PIA1 and you can switch the cassette motor on and off by setting and clearing this bit. To switch on, the bit should be set HI; to switch off, the bit should be cleared to LO. This can be accomplished with the following BASIC statement.

```

POKE &HFF21,(PEEK(&HFF21) OR 8) ' Switch on
POKE &HFF21,(PEEK(&HFF21) AND &HF7) 'Switch off

```

8.4.5 Joystick control

For game playing, where continuous movement is required, typing different keys is not really the most convenient way of specifying that movement. The Dragon's designers have provided an input port which can be used to connect joysticks to the system. These joysticks can be moved in the X-Y plane to control movement and have a button input for firing. Each joystick has two potentiometers associated with it whose output voltage is related to the X and Y coordinates of the stick. These voltages can be detected and their variations related to the movement of a symbol on the screen.

The easiest way to read joystick values is to make use of Extended Color BASIC'S JOYSTK command. This command is a BASIC function which takes as a parameter the axis number of a joystick and returns a numeric value in the range 0-63 which represents the position of that axis. The actual axes specified by JOYSTK are:

- (1) JOYSTK(0) reads right joystick, X-axis (horizontal)

- (2) JOYSTK(1) reads right joystick, Y-axis (vertical)
- (3) JOYSTK(2) reads left joystick, X-axis (horizontal)
- (4) JOYSTK(3) reads left joystick, Y-axis (vertical)

The arrangement of the axes potentiometers is such that they return values relating to the screen graphics coordinate convention of the top left corner being 0,0. Therefore, reading the right joystick at the extreme top left of its travel will result in JOYSTK(0) = 0 and JOYSTK(1) = 0 and, at the extreme bottom right, the readings will each be 63. In fact, these values may vary slightly because of slight differences in the potentiometers built into the joysticks.

A feature of the JOYSTK function is that new joystick readings are only taken when JOYSTK(0) is used. This means that JOYSTK(0) must be used before other JOYSTK commands even if JOYSTK(2) and JOYSTK(3) are the only values used in the program.

The reason for this becomes clear when we consider how the JOYSTK function is implemented. It actually calls a system routine called JOYIN which reads all the joystick values and this routine is only activated when JOYSTK(0) is used. When other JOYSTK parameters are used, the value returned is simply that which was read by the previous activation of JOYIN.

JOYIN can be used by the assembly language programmer. It may be accessed through the direct I/O jump table at location 8012 or through the indirect jump table via location A00A. Its specification for the assembly language programmer is as follows:

- * JOYIN - read joystick values
- *
- * Register inputs NONE
- * Registers destroyed ALL
- * JOYIN returns a value in the range 0-63 for each
- * of the joystick potentiometers.
- * These values are returned in RAM
- * locations as follows
- * \$15A - X-coordinate of right joystick
- * \$15B - Y-coordinate of right joystick
- * \$15C - X-coordinate of left joystick
- * \$15D - Y-coordinate of left joystick

The buttons on each joystick are arranged so that, when they are pressed, they ground an input line. These button outputs are connected to PA0 (right button) and PA1 (left button) of POPDRA. The following BASIC code demonstrates how button pushes may be detected.

```
10 F0 = PEEK(&HFF00) AND 1 ' Read PA0 button state
20 F1 = PEEK(&HFF00) AND 2 ' Read PA1 button state
```

```
30 IF F0 = 0 THEN PRINT "FIRE1"  
40 IF F1 = 0 THEN PRINT "FIRE2"  
50 GOTO 10
```

As these button outputs are shared with the first two rows of the keyboard matrix, the standard keyboard scanning routine must make sure that spurious characters are not generated due to the misinterpretation of button closures as key closures. Unfortunately, the only reliable way to do this is to disable the keyboard completely so, even if you write your own keyboard scanner, you should not try to use the keyboard and joysticks at the same time.

If you use joysticks, we advise you to use the JOYIN routine but for those readers who wish to write their own joystick routines, we now describe, very briefly, how an analogue voltage input from the joystick is converted to a value between 0 and 63.

Each of the four joystick potentiometers produces a voltage between 0V and 5V depending on the position of the joystick. The technique used to convert this to a digital value involves inputting a known value to the DAC (described above) and then comparing the DAC output with the potentiometer value. If they are approximately the same, the DAC input is taken as the digital representation of the potentiometer output voltage. If the values are not the same, the DAC input value is varied until the values match.

This simple analogue to digital conversion system actually has three components. These are the DAC, an analogue multiplexor which can select one of the potentiometer, and a voltage comparator to compare the DAC and multiplexor outputs. The comparator output is connected to PA7 of P0PDRA. If the output from the multiplexor exceeds the DAC output, the comparator output is HI and if the DAC output is greater, the comparator output is LO.

In performing the conversion, a value at the halfway point in the DAC range, namely 32, is written to the DAC and the resulting output voltage compared with the multiplexor output. If the DAC voltage is higher than the multiplexor output, a new DAC input value which is half the old value is tried. If the DAC output is lower than the multiplexor output, a new DAC value which is half as much again as the current value is attempted. This process continues until a DAC output which is approximately equal to the multiplexor output results.

This technique is called binary search and is often used when values have to be compared. It is an efficient searching technique which has applications in many different kinds of program. The table below summarises the approximations resulting from this binary search.

Range	DAC Value	DAC Voltage	Joystick Voltage	Comparator P0PDRA-PA7	New range
0-63	32	2.53	3.5	1	32-63
32-63	48	3.69	3.5	0	32-48
32-48	40	3.11	3.5	1	40-48
40-48	44	3.40	3.5	1	44-48
44-48	46	3.54	3.5	0	44-46
44-46	45	3.47	3.5	1	45-45

The four joystick input lines are connected to the analogue multiplexor in exactly the same way as the multiplexor connection used for the sound source. The same PIA lines are used for this connection so the sound selection routine described above may be used for joystick selection. The association of PIA bit values and joystick source is as follows:

Joystick select		
PIA0-CB2	PIA0-CA2	
0	0	Right - X-axis
0	1	Right - Y-axis
1	0	Left - X-axis
1	1	Left - Y-axis

8.4.6 The cartridge expansion port

The final section in this chapter is devoted to a very brief description of the cartridge/expansion port fitted to the Dragon. This port is intended for a plug-in cartridge containing ROM-based software but the 40-pin connector does provide access to most of the M6809's bus signals. This means that other devices can be interfaced to the Dragon via this expansion port. It is beyond the scope of this section to discuss such interfacing and the interested reader is referred to one of the books on this topic listed in the reading list.

The minimum information required to make interfacing possible is a description of the signals available at the expansion port connector. These are summarised in the table below. Some of the signal names are suffixed with an asterisk, indicating that these are 'active low' signals, meaning that they must go LO (0 volts) for action.

Pin No.	Use	Description
1	+12V	+12V Power supply connection
2	+12V	As above
3	HALT*	Connected to HALT input of M6809
4	NMI*	Non-maskable interrupt input to M6809
5	RESET*	Main reset/power-up signal
6	E	Main M6809 clock
7	Q	Quadrature clock, leads E by 90 degrees

8	CART	Interrupt input for cartridge detection
9	+5V	+5V power supply connection
10-17	D0-D7	Pins 10-17 are connected to the M6809's data lines
18	R/W*	M6809 Read/Write signal
19-31	A0-A12	Pins 19-31 and 37-39 are connected to the M6809's address lines
32	CTS*	Cartridge select signal
33	GND	Signal ground
34	GND	Signal ground
35	SND	Sound input
36	P2*	PIA2 address select
37-39	A13-A15	Top three address lines
40	DSD*	Device selection disable interrupt

The signals supplied by the expansion port fall into one of four categories:

- (1) Power supply
- (2) M6809 bus signals
- (3) Device select signals
- (4) Cartridge I/O signals

The Dragon power supply provides +5V and +12V and these can be used to power the cartridge components. However, you must be careful when using these as there is not a great deal of spare power available.

The M6809 bus signals (D0-D7, A0-A15, R/W*, RESET*, NMI*, HALT*, E and Q) are fully described in Appendix 1 and the device select signals are described in Appendix 2. The function of the other signals is summarised below.

- (1) CTS*
The expansion port occupies the address space between C000 and FFFF inclusive. This line is pulled LO by the SAM address decode logic whenever an address in this range is specified.
- (2) P2*
There is provision in the address decode logic for a third device select signal similar to the ones that select PIA0 and PIA1. P2* goes LO whenever an address is specified in the range FF40-FF5F.
- (3) DSD*
When this signal is pulled LO by suitable cartridge circuitry, it disables the Dragon's internal device select logic. This has the effect of

switching off those devices which are normally selected and, as a result, their address space may be used by the cartridge hardware. This avoids contention problems which can arise when two or more devices are activated and try to place data on the data bus at the same time.

(4) CART

This signal is tied to the CBI interrupt input pin of PIA1. This can be used for an auto-start facility where a clock line (normally Q) is connected to this line to produce the appropriate edge for PIA1's interrupt logic. This is the normal arrangement for games cartridges.

(5) SND

This signal is connected to the sound multiplexor and allows the cartridge to provide the sound source.

Chapter 9

Dragon hints and tips

In a complex system such as the Dragon there are, inevitably, many details which cannot be neatly packaged under a particular heading. These details are often of importance to the programmer who wishes to exploit the capabilities of his machine so, in this chapter, we present a pot-pourri of Dragon information which we hope may be useful to you. We cannot explain everything in great detail as this would require a book in itself but we do provide enough information to get you started with your own experiments.

In this chapter we describe what happens when you switch on your machine, how BASIC programs and data are stored, how to pass parameters to assembly code programs and how to add new commands to BASIC. We also provide tables of BASIC system variables, some of which may be altered to tailor the system to your own specification.

9.1 POWER-UP/RESET ACTIONS

When you switch on your machine a RESET interrupt occurs and this causes a transfer to an initialisation routine in ROM at address B3B4. The first thing that this routine does is to configure the SAM chip as, until this is done, RAM cannot be used. Then the PIAs are configured to their default settings which, in turn, sets up the VDG and I/O devices. The hardware initialisation routine may also be accessed via the direct jump table and a jump to this routine is stored at location 8000. The routine expects the return address to be in register Y as RAM is not available.

Once the hardware is initialised, a software initialisation stage is entered which first sets up a temporary stack for subroutine calling. The next step is to find out whether this is a 'cold-start' or a 'warm-start' reset and this is determined by the contents of the reset flag at address 71 and the secondary reset vector at addresses 72:73. If the reset flag is \$55 and the secondary reset vector points to a NOP instruction, a warm-start sequence is initiated otherwise a cold-start takes place.

The software initialisation routine may be accessed via the jump table (8003) and has neither input nor output parameters.

9.1.1 Cold-start initialisation

When first switched on, the machine's random access memory contains random bit patterns as the contents of RAM are lost when the power is switched off. Therefore, the first action of the cold-start routine is to clear RAM to zeros so that system variables, etc. have a default value of zero.

The next step is to clear the text screen and this is followed by a RAM-sizing operation which detects the top of the useable RAM on the machine. The size of RAM can be determined by altering consecutive memory locations in turn until an unalterable location is detected. This is the start of ROM so the last RAM address is the address which immediately precedes this.

The final step in the software initialisation is to set up BASIC'S system variables and these are initialised by copying their values from tables in ROM.

When software initialisation is complete, the reset service routine looks for the occurrence of a disk controller cartridge by checking if the characters 'DK' occupy addresses C000:C001. If a controller is present, the disk controller is initialised by jumping to an initialisation routine at address C002.

If there is no disk controller, IRQ and FIRQ interrupts are enabled by clearing the appropriate bits in the condition code register. This allows an auto-starting ROM cartridge to interrupt on FIRQ thereby transferring control to the cartridge software. If no ROM cartridge is present, the secondary reset vector is set to point at the warm-start routine and the reset flag set to \$55.

Finally, the BASIC system is initiated and the system is ready for use.

9.1.2 Warm-start initialisation

The warm-start initialisation routine is invoked on a manual reset after the initial powering up of the machine. The code in this routine is used to re-initialise those variables needed to restart an existing program, to clear the text screen and to re-enable the PIA and processor interrupts. The user's BASIC program is left intact. However, if a BASIC program or an injudicious POKE has corrupted some vital system variables, then the only course of action left to the user is to initiate a cold-start initialisation by switching the machine off and on.

The following BASIC statements show how to perform a warm-start from BASIC.

```
10 EXEC &HB3B4 'Execute the reset routine
20 PRINT "WE WON'T GET HERE AS STACK IS RESET"
```

When run, the above program will exit and display the 'OK' prompt but the program will remain intact and can be listed, run, etc. However, if the statement POKE

&H71,0 is added as line 5, this will force a cold-start and the full sign-on message will appear.

9.2 BASIC PROGRAM STORAGE

When you type in a BASIC program, the system saves it in memory and operates on this saved program when you type a RUN command. The format of a saved BASIC statement is:

```
<link><line number><statement>EOL
```

The <link> field is a 16-bit field and holds the address of the following line. The <line number> field is also 16 bits wide and holds the statement's number as an unsigned 16-bit integer. This is followed by the text of the line and the end of the line is marked by EOL which is a null byte. The end of a program is indicated as two zero bytes.

The BASIC program below is a dump program which scans a stored BASIC program and prints the stored form of that program. In its present form, it actually prints the stored form of itself but it may readily be modified to print out the internal form of another BASIC program.

```
10 'DUMP PROGRAM
15 'Peeks a word (16 bits)
20 DEF FNW(A)=PEEK(A)*256+PEEK(A+1)
30 DIM ZZ(2),XX(2,1),ZZ$(2),XX$(2,1)
40 DIM XY(1,2,3)
50 'DEFINING VARIABLES BEFORE USE
60 FOR I=0TO2
70 ZZ(I)=I:ZZ$(I)=STR$(I)
80 FOR J=0TO1
90 XX(I,J)=C:XX$(I,J)=STR$(C)
100 XY(J,I,0)=C
110 C=C+1
120 NEXTJ,I
130 PS=0 'Program Start
140 PE=0 'Program End
150 VS=0 'Variables Start
160 VE=0 'Variables End
170 AS=0 'Array Start
180 AE=0 'Array End
190 DA=0 'Dump Address
200 NA=0 'Next Address
210 DV=0 'Device number
220 DBYTE=0:AN$="":SBYTE=0
230 R=0:H3=0:H2=0:H1=0:H0=0
240 PA=&H19
250 PS=FNW(PA) '19:1A contain PS address
260 VS=FNW(PA+2) '1B:1C contain VS address
270 AS=FNW(PA+4) '1D:1E contain AS address
```

```
280 PE=VS-1 'Variables are after program
290 VE=AS-1 'Arrays are after variables
300 AE=FNW(PA+6)-1 '1F:20 holds free space address
310 CLS
320 INPUT"OUTPUT TO PRINTER";AN$
330 IF AN$ = "Y" THEN DV=-2 ELSE DV=0
340 PRINT#DV,"PROGRAM START ADDRESS = ";:D BYTE=PS
350 GOSUB 1000:PRINT#DV
360 PRINT#DV,"PROGRAM END ADDRESS = ";:D BYTE=PE
370 GOSUB 1000:PRINT#DV
380 PRINT#DV,"VARIABLE START ADDRESS = ";:D BYTE=VS
390 GOSUB 1000:PRINT#DV
400 PRINT#DV,"VARIABLE END ADDRESS = ";:D BYTE=VE
410 GOSUB 1000:PRINT#DV
420 PRINT#DV,"ARRAY START ADDRESS = ";:D BYTE=AS
430 GOSUB 1000:PRINT#DV
440 PRINT#DV,"ARRAY END ADDRESS = ";:D BYTE=AE
450 GOSUB 1000:PRINT#DV
460 INPUT "DO YOU WISH A PROGRAM DUMP";AN$
470 IF AN$<>"Y" THEN GOTO 630
480 PRINT#DV,"PROGRAM DUMP"
490 DA=PS
500 NA=FNW(DA)
510 IF (NA=0) OR (DA=PE) THEN GOTO 620
520 PRINT#DV:PRINT#DV
530 D BYTE=DA:GOSUB 1000 'PRINT CURRENT ADDRESS
540 PRINT#DV
550 D BYTE=NA: GOSUB 1000 'PRINT NEXT ADDRESS
560 DA=DA+2
570 D BYTE=FNW(DA) : GOSUB 1000 'PRINT LINE NUMBER
580 DA=DA+2
590 S BYTE=PEEK(DA): GOSUB 2000 'PRINT LINE CONTENTS
600 DA=DA+1
610 IF DA<>NA THEN GOTO 590 ELSE GOTO 500
620 PRINT#DV
630 INPUT "DO YOU WANT A VARIABLE DUMP";AN$
640 IF AN$ <> "Y" THEN GOTO 690
650 PRINT#DV:PRINT#DV:PRINT#DV,"VARIABLE DUMP"
660 FOR DA = VS TO VE
670 S BYTE=PEEK(DA): GOSUB 2000
680 NEXT DA
690 PRINT#DV
700 INPUT "DO YOU WANT AN ARRAY DUMP";AN$
710 IF AN$<'>"Y" THEN GOTO 760
720 PRINT#DV:PRINT#DV:PRINT#DV,"ARRAY DUMP"
730 FOR DA = AS TO AE
740 S BYTE=PEEK(DA): GOSUB 2000
750 NEXT DA
760 PRINT#DV:PRINT#DV,"DUMP FINISHED"
770 STOP
1000 'PRINT 2 BYTES AS 4 HEX CHARS
1010 R=D BYTE
1020 H3=INT(R/4096): R=D BYTE-H3*4096
1030 H2=INT(R/256): R=R-H2*256
```

```

1040 H1=INT(R/16): H0=R-H1*16
1045 HW$=HEX$(H3)+HEX$(H2)+HEX$(H1)+HEX$(H0)
1050 PRINT#DV, USING "% %"; HW$
1060 RETURN
2000 'PRINT 1 BYTE AS 2 HEX CHARS
2010 H1=INT(SBYTE/16): H0=SBYTE-H1*16
2020 PRINT#DV, USING"% %";HEX$(H1)+HEX$(H0);
2030 RETURN

```

If you run this program, you will see that the BASIC program is actually stored in a semi-compressed form with reserved words (DIM, INPUT, PRINT, etc.) represented as a single byte or token. This token always has its top bit set to distinguish it from normal ASCII characters which all lie in the range 0-127. You can see this if you modify the above program by printing each byte as an ASCII character rather than as a pair of hexadecimal digits. The following subroutine may be used to do this:

```

2000 ' Print 1 byte as a character
2010 PRINT CHR$(SBYTE);
2020 RETURN

```

If you now run the dump program with this amendment, you will see most of the text as it was originally typed into the machine. However, all reserved words will have disappeared and will have been replaced by Semigraphic characters.

There are two reasons for storing the program in this semi-compressed way:

- (1) Space is saved. Rather than reserved words like INPUT taking up 5 bytes, only a single byte is required to represent that command.
- (2) Program execution speed is increased. The reason for this is that the token representing the reserved word is used as an index into a jump table of routines to carry out the action specified in the command. Therefore, the token representing INPUT is a form of indirect address to the INPUT routine.

All tokens representing reserved words have values between 128 and 255 because the top bit of the token is always set. Unfortunately, Extended Color BASIC has more than 127 reserved words, by the time you take the names of all system routines into account, so an alternative technique is used to encode function names.

This involves use a 2-byte code for the function name where the first byte is always FF. A byte whose value is FF means that the following byte is the token for a routine and, obviously, the value FF is not

itself used as a reserved word token. The table below shows some examples of reserved word tokens as single and as double bytes.

Reserved word	Token
DIM	\$8C
FOR	\$80
=	\$CB
STR\$	\$FF8E
PEEK	\$FF8C

The addresses of the routines, called action routines, which are used to execute the reserved words are held in a table called the dispatch table. This is indexed by the reserved word token. A complete list of reserved words, associated tokens and dispatch addresses is provided in Appendix 7. Obviously, there must be separate dispatch tables for the normal reserved words and the function reserved words so that indices do not clash with each other.

Reserved words are converted to tokens by the BASIC system's input routine which tries to match each input character sequence against a reserved word list which is a table of reserved words. The token associated with each reserved word is taken to be its position in the reserved word table plus 80 (hex) to set the top bit. Thus the 17th reserved word has token value 91 (hex), the 37th has value A5 (hex), etc. If a match is found, the word is replaced by its token. The same table is used by the LIST routine which converts tokens into reserved words so that the program may be listed.

The BASIC program statement storage area is immediately followed in memory by storage areas devoted to simple variables and array variables. Simple variables are either numeric or string variables and only the first two characters of the variable name are used to identify the variable. All others are ignored. The variable area is made up of the variable name as a pair of ASCII characters followed by a 5-byte representation of the number or string. These representations are described in section 9.3.

All names are represented as two characters so single character names are made up of the single character plus a zero byte. Notice that this does not mean that A may be confused with A0 as the ASCII character '0' does not have byte value 0. String type variables are distinguished from numeric variables by setting the top bit of the last character in the variable name.

Array variables are held in a separate storage area and array names are distinguished from simple variable names by suffixing a '(' to the name. In the array storage area, each array is stored as a structure:

<array name><array descriptor><array values>

The array descriptor provides information about the size and number of dimensions of the array. It is structured as follows:

- (1) Length of array in RAM - 2 bytes.
- (2) The number of array dimensions - 1 byte.
- (3) For each dimension, a list (2 bytes each) of the number of elements in that dimension.

The array values are held as normal string type or numeric type values.

The BASIC system uses a number of variables to keep track of the program, variable and array storage areas. These are:

Address	Use
\$19:1A	Holds address of start of BASIC program
\$1B:1C	Holds address of start of simple variables
\$1D:1E	Holds address of start of array variables
\$1F:20	Holds address of start of unused memory

9.3 BASIC'S INFORMATION REPRESENTATION

When devising any high-level programming language system, the system designer must make some very fundamental decisions regarding the way in which numeric and character variables are represented in his system. BASIC is very flexible in this respect, allowing variable length character strings and allowing numbers to be either real numbers (numbers with a fractional part) or integers.

If you want to interface assembly language with BASIC it is useful, although not essential, to have some idea of how the BASIC system represents information. Such knowledge also helps you to understand some of the limitations of BASIC and why some applications are particularly slow to execute. In this section, therefore, we describe how numbers and string variables are represented in BASIC.

9.3.1 Number representation

The original designers of BASIC made a very important (and, in our view, a correct) decision that no distinction should be made between real numbers and integers. A BASIC numeric variable may be either a real number or an integer and there is no need for the programmer to indicate, in advance, the type of the variables which he uses.

In this respect, BASIC differs from almost all other programming languages which treat integers and real

numbers differently. The reason for this is that operations on integers are easier to implement and much more efficient than operations on real numbers. If no distinction is made between them, the system must assume that any number may have a fractional part. As a result, integers in BASIC are represented as real numbers with a fractional part equal to zero.

This explains why applications which involve a lot of integer arithmetic are relatively slow in BASIC. The integer arithmetic involved is actually carried out using real number operations and it is not possible to take advantage of the M6809's fast integer arithmetic facilities. This is one reason why assembly language programs which only use integers are so much faster than corresponding BASIC programs.

The representation of real numbers in a computer is made up of two parts. These parts are called the mantissa, which is the number to be represented held as an integer representing a fraction between 0 and 1, and the exponent, which is also an integer, representing the power to which that fraction is raised in order to form the correct real number.

This is really quite a familiar system. If the exponent represents the numbers of powers of 10 to which a fraction is to be raised, the table below gives some examples of how real numbers might be represented using exponent/mantissa notation.

Real Number	Exponent	Mantissa
5.55555	1	.555555
.0032	-2	.32
3456789	7	.3456789
234.456	3	.234456

We have explained this representation in terms of powers of 10 because that is the number base with which we are most familiar. In the BASIC system, however, the exponent represents the power of 2 to which the exponent is raised. Therefore, in BASIC, the value of a number is computed by multiplying the fraction by 2 raised to the power specified in the exponent.

Real number in BASIC are represented using 5 bytes. These bytes are used as follows:

Byte 0 (leftmost byte)	Exponent
Bytes 1-4	Mantissa

The leftmost bit of the mantissa, that is bit 7 of byte 1, represents the sign of the mantissa. If it is 0 this means a positive mantissa, if it is 1 this means that the mantissa is negative. The mantissa itself is not held as a two's complement number but is represented as a positive integer. A mantissa of zero means that the number represented is zero. In this

case, the exponent part is ignored by the BASIC system.

The exponent is held as a positive number between 0 and 255 which seems to imply that negative exponents cannot be represented. However, to get the actual value represented by the exponent, 128 must be subtracted from it. Therefore, an exponent of zero means that the number represented is raised to the power -128, a exponent of 128 means that the number is raised to the power 0 and an exponent of 255 means that the number is raised to the power 127.

Given this representational system, the smallest number which the BASIC programmer may use is 10.14 to the power -38 and the largest number is 10.14 to the power 38. In many cases, however, the actual number represented is an integer between -32768 and 32767. If this is the case, the system function INTCNV can be called by the assembly language programmer. This function returns the integer value of the BASIC number in accumulator D.

INTCNV is designed to convert a real number to an integer. If the number to be converted lies outside the range of 16-bit integers, INTCNV causes an overflow error and control automatically reverts to the BASIC system. Similarly, if a string rather than a number is passed as a parameter to INTCNV, a type mismatch error is signalled and control reverts to BASIC.

The routine INTCNV uses a so-called floating-point accumulator in memory locations 4F-54. The floating-point accumulator (FAC) gets its name because real numbers represented in exponent/mantissa notation are sometimes called floating-point numbers. It is a 6-byte area organised as follows:

Byte 0	Exponent of real number
Bytes 1-4	Mantissa value
Byte 5	Mantissa sign

The sign of the mantissa is factored out of the real number representation and held as a separate byte on its own. The reason for this is that it helps speed up internal floating-point computations. As you would expect, a byte value of zero means that the mantissa is positive and a byte value of FF means that the mantissa sign is negative.

There is no need for the programmer to write code which explicitly converts a BASIC numeric variable to FAC representation. Rather, there is an in-built routine called MOVFM which carries out such a conversion. MOVFM uses the system FAC in locations 4F onwards and expects the address of a numeric variable in the BASIC variable table or anywhere else in memory to be in register X.

A complementary function called GIVABF is used to convert an integer held in accumulator D to its

floating-point representation in a 6-byte floating point accumulator. Again, the system's FAC is assumed by this routine. The following example shows an assembly code routine which uses INTCNV and GIVABF to add 1 to a BASIC numeric variable.

```
*
*
* Input register X - Address of FAC
*
* The value in FAC is incremented by 1 by this routine
*
* Registers destroyed D, CC
*
ADD1   JSR INTCNV           ; Convert to integer
        ADDD #1             ; Add 1 to D
        JSR GIVABF
        RTS
```

9.3.2 String representation

The BASIC system maintains a string storage area where the actual characters making up a string are stored and holds string variables as references into this area. However, rather than store the string length along with the string characters, BASIC holds that length along with the address, in the string storage area, of the string characters. In fact, each string is represented by a 5-byte object called a string descriptor. This is structured as follows:

Byte 0 (leftmost byte)	String length
Byte 1	Housekeeping byte
Bytes 2-3	String address
Byte 4	Housekeeping byte

The parts which we are interested in are bytes 0, 2 and 3. Bytes 1 and 4 are used by the BASIC system to hold information which allows it to perform garbage collection in the string storage area as discussed in Chapter 6. The techniques used for garbage collection are not important here and bytes 1 and 4 should not be changed by the assembly language programmer.

The programmer may write assembly code routines to manipulate string descriptors and call these routines with a USR call. However, the modification of descriptors is a dangerous business as it is possible to corrupt other system information which could result in inexplicable system failure. As the descriptor holds the string length, the only really safe operation is to shorten the string by modifying the length byte. On no account should you try to lengthen the string or change the actual string characters addressed by bytes 2 and 3 of the descriptor. The only safe operation on the address field of the string descriptor is to change it

to point at some character rather than the first string character. You must, naturally, also change the string length byte if you modify the address field.

9.3.3 BASIC variables

The reader will have noticed that both numbers and strings in BASIC are represented as 5-byte objects. This means that the construction of a variable table for the BASIC interpreter to use is straightforward. This table has two components:

- (1) The variable name
- (2) The variable 'value'

We assume here that the value of a string is actually the value of its descriptor.

The built-in BASIC function VARPTR returns a numeric value which is an index to the system's variable table. An example, in BASIC, of this is:

```
10 A$ = "HERE IS A STRING"
20 PRINT VARPTR(A$)
```

This code would cause the location in the variable table of the string A\$ to be printed. Note that the value returned by VARPTR is a 5-byte BASIC numeric variable which always represents an integer. The assembly language programmer must therefore convert this number using INTCNV before he can use it in his program.

We conclude this section with a table listing the conversion routines, their addresses and their functions.

Name	Address	Function
VALTYP	\$06	This variable holds the type of a parameter to a routine initiated by a USR call and also the type of the result returned to BASIC
INTCNV	\$8B2D	This routine converts a BASIC numeric variable to an integer. It expects the X register to point at a floating-point accumulator holding the number to be converted and returns its result in register D.
GIVABF	\$8C37	This routine complements INTCNV in that it converts an integer in accumulator D to a BASIC numeric value. GIVABF also sets VALTYP to zero.

MOVFM \$D3BF This routine moves a 5-byte BASIC floating-point number into the floating-point accumulator. The number to be converted is addressed in register X.

9.4 PASSING PARAMETERS FROM BASIC TO MACHINE CODE

We have already shown, in Chapter 6, how the EXEC and USR calls may be used to interface machine code routines with BASIC programs. The parameter passing technique described there involved poking parameter values into known memory addresses and peeking the results from other addresses. In this section we describe how knowledge of BASIC'S internal data representation allows parameters to be passed to machine code routines via USR calls.

To help with its 'housekeeping', the BASIC system keeps track of the type of parameter being passed to a USR function in a memory location called VALTYP. If the argument is a number, VALTYP is set to zero. A non-zero value assigned to VALTYP means that the argument to the USR function is a string. Type mismatch errors, resulting in the message '?TM ERROR', are caused by the contents of VALTYP and the actual operand being incompatible. For example, if VALTYP is zero and a string is used as an operand, a type mismatch occurs.

On entry to a USR function (remember this is defined by the assembly code programmer), the A accumulator register reflects the contents of VALTYP. If A is zero, the parameter is numeric, if A is non-zero, the parameter is a string.

According to the BASIC manual, the parameters to a USR call may be either a number or a string. However, experimentation with this will show that the use of a USR call with a string argument results in a type mismatch error message 'TM ERROR' being printed. We describe below how to pass numeric parameters to a USR call and also how to program around the system bug which causes an error when string parameters are used.

9.4.1 Numeric parameters

As a USR call is a BASIC function it may only take a single parameter. When this parameter is a number, the USR call sets up the X register to point at the FAC holding the number and sets A to zero, indicating that the type of the parameter is numeric. Usually, the first thing that the assembly language routine does is convert this number to an integer using INTCNV. Computation may then proceed.

Because the USR call is considered to be a BASIC function, it must return a result to the calling program. If the USR call is to initiate a subroutine

rather than a function, where the value returned is irrelevant, the convention adopted is for the assembly language routine initiated by the USR call to return its input parameter as a result. This is easily accomplished by returning to BASIC using a RTS or equivalent instruction with the X register pointing at the FAC containing the result. As long as the subroutine does not change the X register, the result will be the same as the input parameter.

However, if the USR call initiates a proper function, it is possible to return a result which is not the same as the routine's input. The result is computed as an integer and the routine GIVABF is called to convert the contents of accumulator D and load the floating-point accumulator. GIVABF also sets the type indicator VALTYP to zero to reflect the fact that a numeric variable is being returned to the BASIC system.

Any integer values may be returned in this way even when a string is the input parameter to the USR call. For example, consider the routine below which accepts a single character string as an input parameter and returns, as its result, the ASCII code of that input character. The X register, on input, points to the string descriptor so indirect addressing is used to fetch the actual string character.

```

USRASC    LDB (2,X)      ; Fetch first character of string
          CLRA          ; Zero most significant byte of D
          JSR GIVABF    ; Return ASCII code in FAC
          RTS

```

9.4.2 String parameters

Although the BASIC system signals a type mismatch error when a string is passed to a USR call, it is perfectly legal to pass a string as a parameter. The error occurs when the result is returned rather than when the parameter is passed via the USR call.

When a string is used as a parameter to a USR call, the X register is set up to point at the 5-byte descriptor for the string and VALTYP is set non-zero. The string descriptor format is as described in section 9.1.2 and the assembly language routine may manipulate the string as required.

The problems arise when an attempt is made to return to the BASIC system. BASIC expects a numeric result from the USR call so, if a string result is returned, a type mismatch error is signalled. The assembly language routine must somehow fool the BASIC system into thinking that a numeric rather than a string result has been returned if the error is to be circumvented.

There are two ways of doing this. The BASIC system carries out the type checking by calling a routine to check that VALTYP is zero. If the assembly language

programmer explicitly clears VALTYP before returning to the BASIC system, the return will execute normally because the checking routine will see that VALTYP is zero. However, this does mean that the result will be treated as numeric and cannot, therefore, be assigned to a string variable.

To return a string variable properly, the user must cut out the call to the type checking routine by discarding the normal return address on the stack. This can be done by executing a LEAS 2,S instruction to pop it off the stack before a normal RTS instruction. By discarding the return address, you avoid a return to the point where the checking routine is called. The return which is actually executed is a return to the statement immediately after the call to the this routine.

The way to return strings from a USR function is best illustrated by example. The example chosen is a modification of the USRASC routine presented in the previous section. The routine shown below accepts a string as its parameter and returns the same string with the first character incremented.

```
USRINC   LDB (2,X)   ; Fetch first character of string
          INCB      ; increment it
          STB (2,X)  ; and store it back
          LEAS 2,S   ; Discard return address
          RTS
```

If this routine is assembled using the standard DREAM settings, then the following BASIC program can be used to test it.

```
10 DEF USR0 = 20001 'Default code address
20 A$ = "123"
30 PRINT USR0(A$)
40 STOP
```

When this program is run "223" will be printed by line 30. If the program is then listed, statement 20 will be converted to A\$ = "223" showing that the string descriptor associated with A\$ points to the BASIC program text area. Because string descriptors may point into the text area, you must be very careful when modifying strings as such changes can corrupt the surrounding BASIC text.

To avoid string descriptors pointing into the BASIC text area, it is possible to force space for the string to be allocated in the string storage area. One technique makes use of the fact that string catenation results in the string descriptor referring to the string storage area so catenating the empty string to another string ensures that the string descriptor does not point to the text area. In the above example, this

involves changing statement 20 to A\$ - "123" + "".

An alternative technique may be used when it is not important what characters are passed as an input string as the string will be built up by the USR function and then returned. The most convenient method of creating a suitable string descriptor is to use BASIC'S STRING\$ function. For example:

```
A$ = USR00(STRING$(" ",255))
```

This creates a descriptor to a string made up of 255 blanks. If you know the length of the string to be returned, you can obviously set it up as required. If, however, the length is unpredictable, you should create a descriptor for the longest possible string (255 characters) then modify the length byte to reflect the actual string length. This way, you can be sure that sufficient space is always available for the actual string characters.

A useful side-effect of using a string as a parameter in a function call is that the Y register is set up to point at the first character of the string. There is therefore no need to extract the address from the string descriptor. As a result, the indirect reference, (2,X), to the first character of the string in USRASC and USRINC can be replaced by a normal indexed reference using Y.

This side-effect also applies to VARPTRed strings so there is no need to convert the numeric value to an integer with INTCNV to get the string descriptor. Therefore, the code sequence:

```
JSR INTCNV    ; Convert to 16-bit value
TFR D,X      ; and transfer to index register
LDB (2,X)    ; and reference character indirectly
```

may be replaced with the single statement LDB ,Y.

9.5 EXTENDING THE DRAGON'S CAPABILITIES

One way of extending the Dragon's capabilities is, of course, to call your own assembly language subroutines from BASIC and we have described how to do this in the above section. However, it is also possible to augment the BASIC system with new commands which carry out functions which are not provided in Extended Color BASIC.

To add new commands to BASIC, you must add new reserved words to the language and this involves extending the reserved word table described in section 9.2. Information concerning the reserved word tables is contained in an area of RAM called a command interpretation vector or 'stub'. This information is structured as follows:

Byte	Use
0	Number of normal reserved words
1:2	Address of normal reserved word list
3:4	Address of normal reserved word dispatch table
5	Number of function reserved words
6:7	Address of function reserved word list
8:9	Address of function reserved word dispatch table

A number of such command interpretation vector tables may be used provided that they are contiguous in RAM and that the last used table is followed by a zero byte. Extended Color BASIC uses two such tables although the second table is simply a dummy stub with a zero in its first byte indicating that it is a terminator. The first stub occupies RAM space from addresses 120-129 inclusive with each entry set up as shown in the table below:

RAM byte	Contents
120	4E
121:122	8033
123:124	8154
125	22
126:127	81CA
128:129	8250

To add new commands, the user must define a new stub following the normal BASIC command interpretation vector table and this must be followed by a terminator. The format of user-supplied stubs differs slightly from the standard BASIC stub in that bytes 3 and 4 and bytes 8 and 9 should contain the addresses of new dispatch routines for the added commands rather than the addresses of dispatch tables.

We illustrate the process of extending BASIC by showing how two new commands may be added. These commands are a HELP command which prints some user-supplied 'help' information and an OUTPUT command which is exactly the same as PRINT. To add these new commands requires that the following steps should be carried out.

- (1) Set up a new reserved word table.
- (2) Set up a new reserved word dispatch routine.
- (3) Define a new stub with references to this new table and associated routine.

The first step, setting up the reserved word table, is straightforward. This table is made up of the characters in the word with the last character having its top bit set to indicate 'end-of-word'.

```

NEWGRDS   FCC /HEL/
          FCB $D0       ; 'P' with top bit set
          FCC /OUTPU/
          FCB $D4       ; 'T' with top bit set

```

The new reserved word dispatch routine performs similar tasks to BASIC'S reserved word dispatch routine. These tasks include checking the validity of the new token value, calculating the appropriate index into the new dispatch table and setting up the base address of the new dispatch table. Once this has been done, the new dispatch routine can re-enter BASIC ROM at the appropriate point to deal with the new command.

The tokens associated with each reserved word are computed by the system by counting the number of reserved words scanned, including new reserved words if present. As the last normal BASIC reserved word has a token value of CD, the values for HELP and OUTPUT are CE and CF respectively. We use equates to define the first new token value and number of tokens and set up a table of dispatch addresses for the new commands.

```

NEWTOK    EQU $CE       ; First new token value
TOKENS    EQU 2         ; Number of new tokens
NEWTBL    FDB HELP      ; Address of HELP command
          FDB $903D     ; OUTPUT = PRINT

```

The new reserved word dispatch routine makes use of this information when determining which action routine to call. A text input routine called CHRGET is used by the system to scan the BASIC text and passes the token value to this routine in register A. A suitable dispatch routine for these new commands is:

* NEWDSP - New dispatch routine

* Register input A - token value

*

* This routine checks token validity and invokes the
 * appropriate action routines

*

```

NEWDSP    CMPA          #NEWTOK       ; Check that
          BLO NEWERR      ; token given
          CMPA #NEWTOK+TOKENS ; is within range
          BHS NEWERR

```

*

* One of the new commands at this point

*

```

          SUBA #NEWTOK       ; Convert to table index
          LEAX NEWTBL,PCR    ; and set up table base
          JMP ROMCMD        ; before jumping to BASIC
NEWERR    JMP SYNERR        ; error jump into BASIC

```

The HELP routine is very simple and also makes use of a

ROM routine to perform some of its duties:

```
HELP    LEAX HELPME-1,PCR    ; Point to byte before
        JMP OUTSTR          ; string for ROM's OUTSTR
```

Notice that the above routines have no explicit RTS instructions as they terminate by jumping to ROMCMD, OUTSTR, etc. Returns from these routines therefore return to the program which called the new dispatch routine. These routines also use a number of equates which are defined as follows:

```
OUTSTR  EQU $90E5    ; String output routine
SYNERR  EQU $89B4    ; BASIC syntax error routine
ROMCMD  EQU $84ED    ; BASIC dispatch point
```

The HELP message is held in an area of store named HELPME and is output by a standard output routine called OUTSTR. This routine takes as its parameter the address of the byte before the string and expects the string to be terminated with a zero byte.

```
HELPME  FCB $0D
        FCC /DON'T ASK ME I'M ONLY A MACHINE/
        FCB 0
```

After defining the dispatch routine, a new stub must be set up. Unfortunately, the address of the zero byte required to mark the end of stubs clashes with the first byte of USR vectors (134). However, this has been allowed for in Extended Color BASIC as the USR vector is referenced indirectly through the direct page location B0:B1 and the USR vector area may be moved elsewhere and these locations filled in with its new address.

The following routine sets up a new stub and relocates the USR vectors.

* NEWSET - set up new stub for reserved words

* Register input NONE
 * Registers destroyed A,X,Y,CC
 *

```
NEWSET  LDX #STUB1      ; First of all copy
        LDY #STUB2      ; old second stub
NXTBYT  LDA ,X+         ; bytes into
        STA ,Y+         ; third stub
        CMPX #STUB2
        BLO NXTBYT
        LDA #TOKENS     ; Number of reserved words
        STA STUB1       ; set up new stub
        LEAX NEWRDS,PCR ; New reserved word list
        STX STUB1+1     ; set up
        LEAX NEWDSP,PCR ; New dispatch routine
        STX STUB1+3     ; set up
```

* No new functions so second part of stub unchanged

*

```

                LEAX NEWUSR,PCR    ; Relocate the USR
                STX USRPTR        ; vectors
                LDY #FCERR        ; and initialise
                LDA #10
NXTVEC         STY ,X++          ; them to FC ERROR
                DECA              ; continue until all
                BNE NXTVEC        ; done
                RTS

```

This routine assumes that the following equates and declarations have been made:

```

STUB1         EQU $12A          ; Address of second stub
USRPTR        EQU $B0           ; USR vectors
FCERR         EQU $8B8D        ; FCERR entry point
NEWUSR        RMB 20           ; relocated USR vectors. Must be
*                               set up before defining any USR
*                               addresses

```

9.5.1 RAM hooks

The BASIC system designers allow new commands to be added to BASIC so that extra commands needed to support a disk version of BASIC may be included. However, some of the existing commands such as OPEN and CLOSE also need to be enhanced for disk BASIC and so addresses have been set up in RAM which allow extra facilities to be added to action routines. These addresses normally contain an RTS instruction so that a reference to them from within an action routine does nothing. The RAM addresses are called 'hooks' and the RTS instruction may be replaced by a jump to some other routine which enhances the capabilities of the action routine.

There are a total of 25 RAM hooks available at locations 15E to 1A8 inclusive. A brief description of each is given in the following table.

Address	Called from	Potential use
15E	B829	Open device or file
161	B7EC	Check I/O device number
164	B596	Return device parameters
167	B54B	Character output
16A	B50B	Character input
16D	B624	Check device is open for input
170	B63D	Check device is open for output
173	B65D	Close all devices and files
176	B664	Close a single device or file
179	84DE	About to deal with first character of new statement
17C	8792	Disk file item scanner
17F	B77C	Poll for BREAK and special keys
182	B5C7	Read a line of input
185	B6FE	Finish loading ASCII program

188	B801	End of file (EOF) function
18B	8954	Evaluate an expression
18E	8344	User error trap
191	8347	System error routine trap
194	85A5	RUN statement
197*	8C80	String copy check
197*	8424	CLEAR statement
19A	849F	Fetch next statement
19D	86D7	LET string copy check
1A0*	BA60	CLS statement
1A0*	9EEB	RENUM statement
1A0*	AAF7	PUT/GET statement
1A0*	850F	Function assignment
1A3	8F67	Compress BASIC line for storage
1A6	8F08	Expand BASIC line for listing

Starred addresses in the above table mean that several hooks share the same address. The only way to determine which is used is to check the return address!

In order to use some of these RAM hooks, you need an in-depth knowledge of the BASIC interpreter and, therefore, these hooks are not useful to the ordinary programmer. However, some of the hooks are very useful indeed and can be used to enhance the standard system facilities. We shall illustrate this by showing how the character output hook (\$167) can be used to copy all character output to the printer and how the new statement hook (\$19A) can be used to force a complete keyboard scan.

Our first example involves setting up the character output hook with the address of the printer output routine.

```
LPTOUT EQU $800F      ; Printer output address
HKCHR0 EQU $167      ; Character output hook
HKUPC0 LDX #LPTOUT   ; Set up printer output address
        STX HKCHR0+1 ; hook up to character output
        LDA #$7E     ; JMP opcode value
        STA HKCHR0   ; into hook
        RTS
```

This example may be set up using BASIC pokes as it simply involves replacing the three bytes of character output hook with a JMP \$800F. However, you must be very careful when setting up hooks from BASIC as the hook may be called between each POKE statement. This means that you must first set up the address in bytes 1 and 2 of the hook and, as the last step, replace the RTS instruction with a JMP instruction. The following BASIC statements set up the character output hook.

```
POKE &H168,&H80: POKE &H169,&H0F: POKE &H167,&H7E
```

It is not easy to find out which registers must be

preserved when a hook is called so you must save all registers, including CC, used in the hook routine. We didn't do this in the example above as this routine is also called by the normal character output routine when a PRINT #-2 is used. We therefore assumed that the line printer routine preserves the registers itself.

In our second RAM hook example we show how the new statement RAM hook can be used to reset the row state byte (\$151) to a value which forces a complete keyboard scan. The following code is made up of necessary equates, the routine used to reset the row state byte and a routine to set up the RAM hook.

```
HKNWST    EQU $19A            ; New statement RAM hook
KBROWS    EQU $151            ; Keyboard row state byte
*
* RSROWS - reset row state byte
*
* Register inputs NONE
*
RSROWS    PSHS A,CC           ; Save registers
           LDA #$7F            ; This value forces a scan
           STA KBROWS          ; of the keyboard
           PULS A,CC,PC        ; Restore and return
*
* HKUPNS - set up RAM hook
*
HKUPNS    LEAX RSROWS,PCR     ; Address of hook routine
           STX HKNWST+1        ; reset row state routine
           LDA #$7E            ; JMP opcode
           STA HKNWST          ; into hook
           RTS
```

Setting up this hook means that all key depressions will be recognised, even those on the same row. A similar technique can be used to disable the BREAK key thus stopping the user interrupting a program and to provide auto-repeat facilities on some or all keys. Both of these additions involve modifying RSROWS above so that the appropriate column bytes are modified.

The BREAK key can be disabled by adding the following code to the above program.

```
BRKCOL    EQU $154            ; Break row byte
BRKCLR    EQU $BF             ; To clear BREAK'S bit

           LDA BRKCOL          ; Pick up BREAK column
           ANDA #BRKCLR        ; Force BREAK bit to 0
           STA BRKCOL          ; and store it back
```

9.6 BASIC SYSTEM VARIABLES

In this final section we list the reserved memory locations used by the BASIC system and describe, very

briefly, what these locations are used for. As these are RAM locations, you may modify them using POKE but you must be very careful if you do so. If you make a mistake or set up an invalid value, you may hang the system. This means that you can do nothing except switch the machine off and on again to reset it and all your work currently in RAM will be lost.

We start with a list of variables held in the first 256 bytes of memory and accessed via direct addressing with the direct page register set to 00.

Address	Use
00	BREAK message flag. If negative print BREAK
01	String delimiting character
02	Another delimiting character
03	General count byte
04	Count of IFs seen while looking for ELSE
05	DIM flag
06	VALTYP - 0=numeric, 1=string
07	Garbage collection flag
08	Subscript allowed flag
09	INPUT/READ flag
0A	Arithmetic use
0B:0C	String pointer - first free temporary
0D:0E	String pointer - last used temporary
0F-18	Temporary results
19:1A	Pointer to start of BASIC text
1B:1C	Pointer to start of simple variables
1D:1E	Pointer to start of array variables
1F:20	End of storage in use
21:22	Stack base address
23:24	String space base address
25:26	Temporary pointer to new string
27:28	Address of top of RAM used by BASIC
29:2A	Last BASIC line number
2B:2C	Input line number
2D:2E	Old text pointer
2F:30	Another text pointer
31:32	DATA line number
33:34	Pointer for DATA
35:36	Pointer for INPUT
37-4E	Evaluation variables
4F	Floating point accumulator, FAC exponent
50-53	FAC mantissa
54	Sign of FAC
55	Temporary sign of FAC
56-5B	String descriptor temporaries
5C	Floating point argument, ARG exponent
5D-60	ARG mantissa
61	Sign of ARG
62-67	Miscellaneous use
68:69	Current line number
6A-6E	Device parameters used in PRINT
6F	Device number, 0-console, -1-cassette

```

-2-printer
70      End of file flag
71      Restart flag ($55 warm, other cold)
72:73   Warm start vector (points to NOP)
74:75   Top of RAM minus 1
76:77   Unused
78      Cassette file status (0-closed, 1-input,
        2-output)
79      Number of characters in CASBUF
7A:7B   Cassette buffer pointer
7C      Block type (0=header, 1=data
        FF=end of file)
7D      Block length
7E:7F   Address of cassette buffer
80      Block checksum
81      Checksum error flag
82      Pulse width counter
83      Sync bits counter
84      Bit phase flag
85      Last sine wave value
86      Used in SET, RESET, and POINT
87      Single character keyboard buffer
88:89   Current cursor address
8A:8B   16-bit zero
8C      Sound frequency
8D      Sound timer
8F      Cursor blink rate counter.
        Initial value = 32
90:91   Count of number of leader bytes
        Initial value 0080
92      Minimum cycle width of 1200Hz
        Initial value = 12
93      Minimum pulse width at 1200Hz
        Initial value = 0A
94      Maximum pulse width at 1200Hz
        Initial value = 12
95:96   Cassette motor delay value
97:98   Keyboard debounce delay value
        Initial value = 045E
99      Line printer comma field width
        Initial value = 10
9A      Line printer last comma field
        Initial value = 74
9B      Line printer width
        Initial value = 84
9C      Line printer head position
9D:9E   EXEC vector
9F:A0   INC $A7 ; CHRGET input routine
A1:A2   BNE $A5
A3:A4   INC $A6
A5-A7   LDA >0
A8-AA   JMP $BB26
AB-AE   Used by RND
AF      Program trace flag, 0-trace off
        non 0, trace on

```

B0:B1	Pointer to USR vector base
B2	Foreground colour
B3	Background colour
B4	Active colour
B5	Active colour
B6	Graphics mode
B7:B8	Top of current graphics screen
B9	Number of bytes in graphics row
BA:BB	Base address of current graphics screen
BC	Page number of graphics screen
BD:BE	Current X position
BF:C0	Current Y position
C1-DD	Used by graphics
DE	MUSIC octave
DF	MUSIC high volume
E0	MUSIC low volume
E1	MUSIC note value
E2	MUSIC tempo
E3:E4	MUSIC duration count
E5	MUSIC dotted note flag
E6-FF	Unused in Dragon 32

BASIC also uses a number of system variables between addresses 100 and 3FF. Their use is summarised in the table below.

Address	Use
100-102	SWI3 secondary vector
103-105	SWI2 secondary vector
106-108	SWI secondary vector
109-10B	NMI secondary vector
10C-10E	IRQ secondary vector
10F-111	FIRQ secondary vector
112:113	TIMER value
114	Unused
115-119	Random number seeds
11A-11F	Unused
120	Stub 0 - number of reserved words
121:122	address of reserved word table
123:124	address of dispatch table
125	number of functions
126:127	address of function table
128:129	address of function dispatch table
12A-133	Stub 1
134-147	USR address table
148	Auto line feed flag
149	Alpha lock flag 0=lower case FF=upper case
14A-150	Line printer EOL termination sequence
151-159	Keyboard matrix state table
15A	Right joystick X-value
15B	Right joystick, Y-value
15C	Left joystick, X-value
15D	Left joystick, Y-value
15E-1A8	RAM hooks

1A9-1D0	String buffer area
1D1	Cassette filename length
1D2-1D9	Cassette filename buffer
1DA-2D8	Cassette file data buffer
1DA-1E1	Cassette filename (in buffer)
1E2	Cassette file type, 0=program 1=data, 2=machine code
1E3	Cassette ASCII flag, 0=binary FF=ASCII file
1E4	Cassette gap flag, 0=continuous FF=gaps
1E5:1E6	Execution address of machine code file
1E7:1E8	Load address for ungapped machine code file
2D9-2DC	BASIC line input buffer preamble
2DD-3D8	BASIC line input buffer
3D9-3EA	Buffer space
3EB-3FF	Unused in Dragon 32

Reading list

This reading list is simply a list of books which may be of interest to the Dragon user who wants to follow up some of the ideas and techniques introduced in this book. As there are literally thousands of books on computing available, it does not pretend to be a complete list of all relevant books.

As the Dragon is very similar to the Tandy Color Computer, articles on this machine may be of interest to Dragon users. Many such articles have appeared in 'BYTE', a US computing magazine, and back issues of this may be available through your local library.

There are also many other magazines devoted to personal computing, including one specifically for Dragon users. Readers of this book will probably have their favourite periodical but we think that 'Practical Computing', 'Personal Computer World' and 'BYTE' are amongst the best of these journals.

1. GENERAL BACKGROUND

Sommerville I. 1983. 'Information Unlimited'. London : Addison-Wesley

Ullman, J.D. 1976. 'Fundamental Concepts of Programming Systems'. Reading, Mass. : Addison-Wesley

Wirth, N. 1976. 'Algorithms + Data Structures = Programs'. Englewood Cliffs, NJ : Prentice-Hall

Greenfield, J.D. & Wray, W.C. 1981. 'Using Microprocessors and Microcomputers - the 6800 Family'. New York : Wiley

Peatman, J.B. 1977. 'Microcomputer-Based Design'. New York : McGraw-Hill

2. ASSEMBLY LANGUAGE PROGRAMMING

Wakerly, J.F. 1981. 'Micro-computer Architecture and Programming'. New York : Wiley.

Leventhal, L.A. 1981. '6809 Assembly Language Programming'. New York : Osborne/McGraw-Hill

Zaks, R. & Labiak, W. 1982. 'Programming the 6809'. Sybex

3. DATA STRUCTURES

Knuth, D.E. 'Fundamental Algorithms'. Reading, Mass. : Addison-Wesley

Shave, M. 1975. 'Data Structures'. Maidenhead, Berks. : McGraw-Hill

4. GRAPHICS PROGRAMMING

Barden, W. 1982. 'Color Computer Graphics'. Fort Worth, Texas : Tandy Corp.

Foley, J.D. & Van Dam, A. 1979. 'Fundamentals of Interactive Computer Graphics'. Reading, Mass. Addison-Wesley

Inman, D. & Inman, K. 1983. 'Assembly Language Graphics for the TRS-80 Color Computer'. Virginia : Reston

5. I/O PROGRAMMING

Staugaard, J.R. 1981. '6809 Microcomputer Programming and Interfacing'. Indianapolis : Sams &, Co.

Artwick, B.A. 1980. 'Microcomputer Interfacing'. Englewood Cliffs, NJ : Prentice-Hall

Pritty, D.W. & Smeed, D.N. 1984 (to appear). 'Practical Electronic Interfacing to Popular Microcomputers'. London : Addison-Wesley

Lesea, A. & Zaks, R. 1978. 'Microprocessor Interfacing Techniques'. Sybex

Witten, I.H. 1980. 'Communicating with Microcomputers'. London : Academic Press

Andrews, M. 1982. 'Programming Microprocessor Interfaces for Control and Instrumentation'. Englewood Cliffs, NJ : Prentice-Hall

Appendix 1

MC6809E data sheet

Supplied courtesy of Motorola Semiconductors.

The information here has been carefully checked and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies. Motorola reserves the right to make changes to any products herein to improve reliability, function or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein. No licence is conveyed under patent rights in any form. When this document contains information on a new product, specifications herein are subject to change without notice.



MOTOROLA Semiconductors

Colvilles Road, Kelvin Estate - East Kilbride/Glasgow - SCOTLAND

8-BIT MICROPROCESSING UNIT

The MC6809E is a revolutionary high performance 8 bit microprocessor which supports modern programming techniques such as position independence, reentrancy, and modular programming.

This third-generation addition to the M6800 Family has major architectural improvements which include additional registers, instructions, and addressing modes.

The basic instructions of any computer are greatly enhanced by the presence of powerful addressing modes. The MC6809E has the most complete set of addressing modes available on any 8-bit microprocessor today.

The MC6809E has hardware and software features which make it an ideal processor for higher level language execution or standard controller applications. External clock inputs are provided to allow synchronization with peripherals, systems, or other MPUs.

MC6800 COMPATIBLE

- Hardware - Interfaces with All M6800 Peripherals
- Software - Upward Source Code Compatible Instruction Set and Addressing Modes

ARCHITECTURAL FEATURES

- Two 16 Bit Index Registers
- Two 16 Bit Indexable Stack Pointers
- Two 8 Bit Accumulators can be Concatenated to Form One 16 Bit Accumulator
- Direct Page Register Allows Direct Addressing Throughout Memory

HARDWARE FEATURES

- External Clock Inputs, E and Q, Allow Synchronization
- TSC Input Controls Internal Bus Buffers
- LIC Indicates Opcode Fetch
- AVMA Allows Efficient Use of Common Resources in a Multiprocessor System
- BUSY is a Status Line for Multiprocessing
- Fast Interrupt Request Input Stacks Only Condition Code Register and Program Counter
- Interrupt Acknowledge Output Allows Vectoring By Devices
- Sync Acknowledge Output Allows for Synchronization to External Event
- Single Bus-Cycle RESET
- Single 5-Volt Supply Operation
- NMI Inhibited After RESET Until After First Load of Stack Pointer
- Early Address Valid Allows Use With Slower Memories
- Early Write Data for Dynamic Memories

SOFTWARE FEATURES

- 10 Addressing Modes
 - M6800 Upward Compatible Addressing Modes
 - Direct Addressing Anywhere in Memory Map
 - Long Relative Branches
 - Program Counter Relative
 - True Indirect Addressing
 - Expanded Indexed Addressing
 - 0-, 5-, 8-, or 16-Bit Constant Offsets
 - 8- or 16-Bit Accumulator Offsets
 - Auto-Increment/Decrement by 1 or 2
- Improved Stack Manipulation
- 1464 Instruction with Unique Addressing Modes
- 8 x 8 Unsigned Multiply
- 16-Bit Arithmetic
- Transfer/Exchange All Registers
- Push/Pull Any Registers or Any Set of Registers
- Load Effective Address

MC6809E

HMOS

(HIGH-DENSITY N-CHANNEL, SILICON-GATE)

8-BIT MICROPROCESSING UNIT



L SUFFIX
CERAMIC PACKAGE
CASE 715

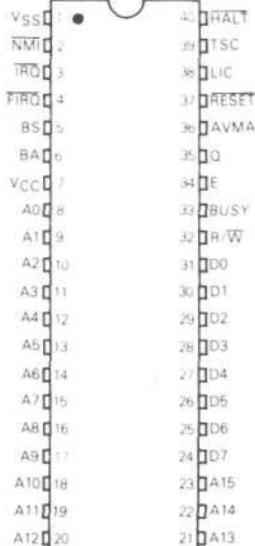


P SUFFIX
PLASTIC PACKAGE
CASE 711



S SUFFIX
CERDIP PACKAGE
CASE 734

PIN ASSIGNMENTS



MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	V _{CC}	-0.3 to +7.0	V
Input Voltage	V _{in}	-0.3 to +7.0	V
Operating Temperature Range MC6809E, MC68A09E, MC68B09E MC6809EC, MC68A09EC, MC68B09EC	T _A	T _L to T _H 0 to +70 -40 to +85	°C
Storage Temperature Range	T _{stg}	-55 to +150	°C

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit.

Reliability of operation is enhanced if unused inputs are tied to an appropriate logic voltage level (e.g., either V_{SS} or V_{CC}).

THERMAL CHARACTERISTICS

Characteristic	Symbol	Value	Unit
Thermal Resistance Ceramic	θ _{JA}	50	°C/W
Cerdp		60	
Plastic		100	

POWER CONSIDERATIONS

The average chip-junction temperature, T_J, in °C can be obtained from:

$$T_J = T_A + (P_D \cdot \theta_{JA}) \quad (1)$$

Where:

T_A = Ambient Temperature, °C

θ_{JA} = Package Thermal Resistance, Junction-to-Ambient, °C/W

P_D = P_{INT} + P_{PORT}

P_{INT} = I_{CC} × V_{CC}, Watts — Chip Internal Power

P_{PORT} = Port Power Dissipation, Watts — User Determined

For most applications P_{PORT} ≪ P_{INT} and can be neglected. P_{PORT} may become significant if the device is configured to drive Darlington bases or sink LED loads.

An approximate relationship between P_D and T_J (if P_{PORT} is neglected) is:

$$P_D = K - (T_J + 273)^\circ\text{C} \quad (2)$$

Solving equations 1 and 2 for K gives:

$$K = P_D \cdot (T_A + 273)^\circ\text{C} + \theta_{JA} \cdot P_D^2 \quad (3)$$

Where K is a constant pertaining to the particular part. K can be determined from equation 3 by measuring P_D (at equilibrium) for a known T_A. Using this value of K the values of P_D and T_J can be obtained by solving equations (1) and (2) iteratively for any value of T_A.

DC ELECTRICAL CHARACTERISTICS (V_{CC} = 5.0 V ± 5%, V_{SS} = 0 Vdc, T_A = T_L to T_H unless otherwise noted)

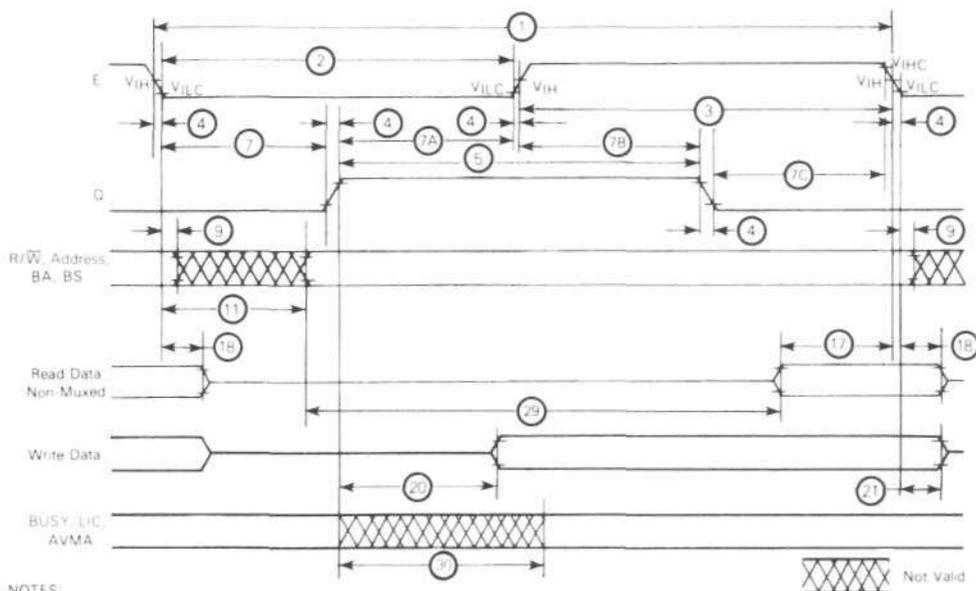
Characteristic	Symbol	Min	Typ	Max	Unit
Input High Voltage Logic, Q, RESET E	V _{IH} V _{IHR} V _{IHC}	V _{SS} + 2.0 V _{SS} + 4.0 V _{CC} - 0.75	—	V _{CC} V _{CC} V _{CC} + 0.3	V
Input Low Voltage Logic, Q, RESET E	V _{IL} V _{ILC}	V _{SS} - 0.3 V _{SS} - 0.3	—	V _{SS} + 0.8 V _{SS} + 0.4	V
Input Leakage Current (V _{in} = 0 to 5.25 V, V _{CC} = max)	I _{in}	—	—	2.5 100	µA
dc Output High Voltage (I _{Load} = -205 µA, V _{CC} = min) (I _{Load} = -145 µA, V _{CC} = min) (I _{Load} = -100 µA, V _{CC} = min)	V _{OH}	V _{SS} + 2.4 V _{SS} + 2.4 V _{SS} + 2.4	—	— — —	V
dc Output Low Voltage (I _{Load} = 2.0 mA, V _{CC} = min)	V _{OL}	—	—	V _{SS} + 0.5	V
Internal Power Dissipation (Measured at T _A = T _L in Steady State Operation)	P _{INT}	—	—	1.0	W
Capacitance* (V _{in} = 0, T _A = 25°C, f = 1.0 MHz)	C _{in}	—	10 30	15 50	pF
	C _{out}	—	10	15	pF
Frequency of Operation (E and Q Inputs)	f	0.1 0.1 0.1	—	1.0 1.5 2.0	MHz
Hi-Z (Off State) Input Current (V _{in} = 0.4 to 2.4 V, V _{CC} = max)	I _{TSI}	—	2.0 —	10 100	µA

* Capacitances are periodically tested rather than 100% tested.

BUS TIMING CHARACTERISTICS (See Notes 1, 2, 3, and 4)

Ident. Number	Characteristics	Symbol	MC6809E		MC68A09E		MC68B09E		Unit
			Min	Max	Min	Max	Min	Max	
1	Cycle Time	t_{CYC}	10	10	0.667	10	0.5	10	μs
2	Pulse Width, E Low	PW_{EL}	450	9500	295	9500	210	9500	ns
3	Pulse Width, E High	PW_{EH}	450	9500	280	9500	220	9500	ns
4	Clock Rise and Fall Time	t_r, t_f		25		25		20	ns
5	Pulse Width, Q High	PW_{QH}	450	9500	280	9500	220	9500	ns
7	Delay Time, E to Q Rise	t_{EQ1}	200		130		100		ns
7A	Delay Time, Q High to E Rise	t_{EQ2}	200		130		100		ns
7B	Delay Time, E High to Q Fall	t_{EQ3}	200		130		100		ns
7C	Delay Time, Q High to E Fall	t_{EQ4}	200		130		100		ns
9	Address Hold Time	t_{AH}	20		20		20		ns
11	Address Delay Time from E Low (BA, BS, R, W)	t_{AD}		200		140		110	ns
17	Read Data Setup Time	t_{DSR}	80		60		40		ns
18	Read Data Hold Time	t_{DHR}	10		10		10		ns
20	Data Delay Time from Q	t_{DDQ}		200		140		110	ns
21	Write Data Hold Time	t_{DHW}	30		30		30		ns
29	Usable Access Time	t_{ACC}	695		440		300		ns
30	Control Delay Time	t_{CD}		300		250		200	ns
	Interrupts, HALT, RESET, and TSC Setup Time (Figures 6, 7, 8, 9, 12, and 13)	t_{PCS}	200		140		110		ns
	TSC Drive to Valid Logic Level (Figure 13)	t_{TSV}		210		150		120	ns
	TSC Release MOS Buffers to High Impedance (Figure 13)	t_{TSR}		200		140		110	ns
	TSC Hi-Z Delay Time (Figure 13)	t_{TSD}		120		85		80	ns
	Processor Control Rise and Fall Time (Figure 7)	t_{PCR} , t_{PCF}		150		100		100	ns

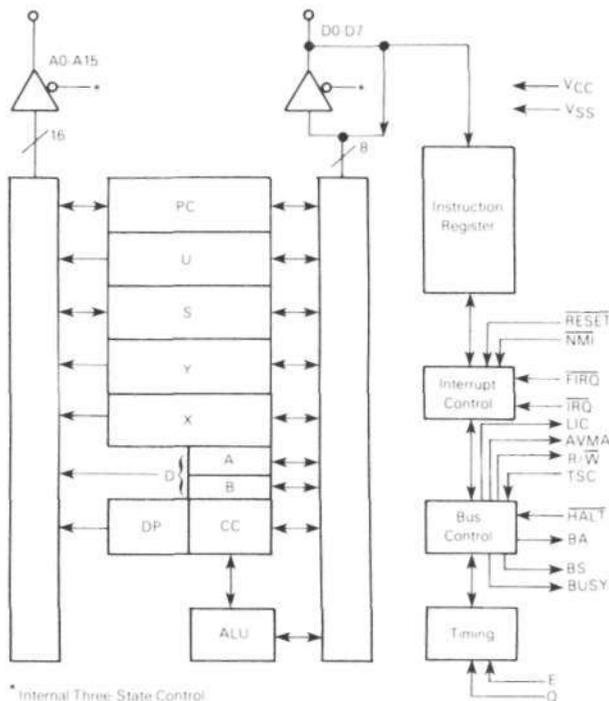
FIGURE 1 — READ/WRITE DATA TO MEMORY OR PERIPHERALS TIMING DIAGRAM



NOTES:

- 1 Voltage levels shown are $V_{L} \leq 0.4$ V, $V_{H} \geq 2.4$ V, unless otherwise specified.
- 2 Measurement points shown are 0.8 V and 2.0 V, unless otherwise specified.
- 3 Hold time t_{H} for BA and BS is not specified.
- 4 Usable access time is computed by $1 - 4 - 11 \text{ max} - 17$.

FIGURE 2 — EXPANDED BLOCK DIAGRAM



PROGRAMMING MODEL

As shown in Figure 4, the MC6809E adds three registers to the set available in the MC6800. The added registers include a direct page register, the user stack pointer, and a second index register.

ACCUMULATORS (A, B, D)

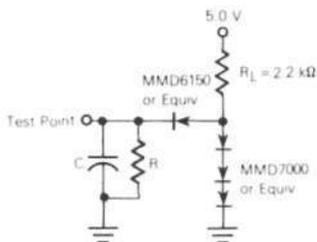
The A and B registers are general purpose accumulators which are used for arithmetic calculations and manipulation of data.

Certain instructions concatenate the A and B registers to form a single 16-bit accumulator. This is referred to as the D register, and is formed with the A register as the most significant byte.

DIRECT PAGE REGISTER (DP)

The direct page register of the MC6809E serves to enhance the direct addressing mode. The content of this register appears at the higher address outputs (A8-A15) during direct addressing instruction execution. This allows the direct mode to be used at any place in memory, under program control. To ensure M6800 compatibility, all bits of this register are cleared during processor reset.

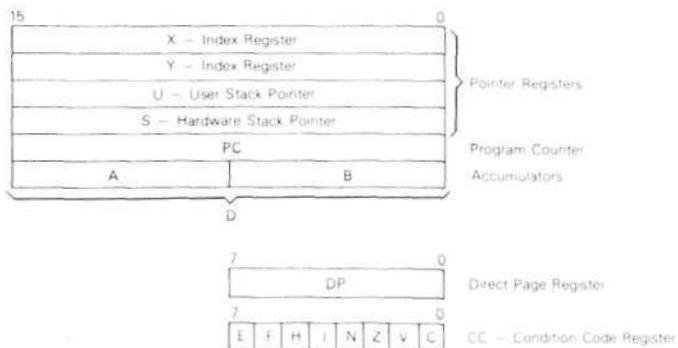
FIGURE 3 — BUS TIMING TEST LOAD



C = 30 pF for BA, BS, LIC, AVMA, BUSY
130 pF for D0-D7
90 pF for A0-A15, R/W

R = 11.7 kΩ for D0-D7
16.5 kΩ for A0-A15, R/W
24 kΩ for BA, BS, LIC, AVMA, BUSY

FIGURE 4 — PROGRAMMING MODEL OF THE MICROPROCESSING UNIT



INDEX REGISTERS (X, Y)

The index registers are used in indexed mode of addressing. The 16-bit address in this register takes part in the calculation of effective addresses. This address may be used to point to data directly or may be modified by an optional constant or register offset. During some indexed modes, the contents of the index register are incremented and decremented to point to the next item of tabular type data. All four pointer registers (X, Y, U, S) may be used as index registers.

STACK POINTER (U, S)

The hardware stack pointer (S) is used automatically by the processor during subroutine calls and interrupts. The user stack pointer (U) is controlled exclusively by the programmer. This allows arguments to be passed to and from subroutines with ease. The U register is frequently used as a stack marker. Both stack pointers have the same indexed mode addressing capabilities as the X and Y registers, but also support **Push** and **Pull** instructions. This allows the MC6809E to be used efficiently as a stack processor, greatly enhancing its ability to support higher level languages and modular programming.

NOTE

The stack pointers of the MC6809E point to the top of the stack in contrast to the MC6800 stack pointer, which pointed to the next free location on stack.

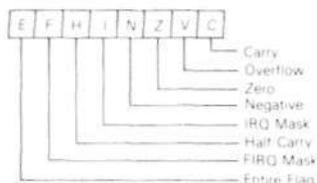
PROGRAM COUNTER

The program counter is used by the processor to point to the address of the next instruction to be executed by the processor. Relative addressing is provided allowing the program counter to be used like an index register in some situations.

CONDITION CODE REGISTER

The condition code register defines the state of the processor at any given time. See Figure 4.

FIGURE 5 — CONDITION CODE REGISTER FORMAT



CONDITION CODE REGISTER DESCRIPTION

BIT 0 (C)

Bit 0 is the carry flag and is usually the carry from the binary ALU. C is also used to represent a "borrow" from subtract like instructions (CMP, NEG, SUB, SBC) and is the complement of the carry from the binary ALU.

BIT 1 (V)

Bit 1 is the overflow flag and is set to a one by an operation which causes a signed two's complement arithmetic overflow. This overflow is detected in an operation in which the carry from the MSB in the ALU does not match the carry from the MSB-1.

BIT 2 (Z)

Bit 2 is the zero flag and is set to a one if the result of the previous operation was identically zero.

BIT 3 (N)

Bit 3 is the negative flag, which contains exactly the value of the MSB of the result of the preceding operation. Thus, a negative two's complement result will leave N set to a one.

BIT 4 (I)

Bit 4 is the $\overline{\text{IRQ}}$ mask bit. The processor will not recognize interrupts from the $\overline{\text{IRQ}}$ line if this bit is set to a one. NMI, $\overline{\text{FIRQ}}$, $\overline{\text{IRQ}}$, RESET, and SWI all set I to a one. SWI2 and SWI3 do not affect I.

BIT 5 (H)

Bit 5 is the half-carry bit, and is used to indicate a carry from bit 3 in the ALU as a result of an 8-bit addition only (ADC or ADD). This bit is used by the DAA instruction to perform a BCD decimal add adjust operation. The state of this flag is undefined in all subtract-like instructions.

BIT 6 (F)

Bit 6 is the $\overline{\text{FIRQ}}$ mask bit. The processor will not recognize interrupts from the $\overline{\text{FIRQ}}$ line if this bit is a one. NMI, $\overline{\text{FIRQ}}$, SWI, and RESET all set F to a one. $\overline{\text{IRQ}}$, SWI2, and SWI3 do not affect F.

BIT 7 (E)

Bit 7 is the entire flag, and when set to a one indicates that the complete machine state (all the registers) was stacked, as opposed to the subset state (PC and CC). The E bit of the stacked CC is used on a return from interrupt (RTI) to determine the extent of the unstacking. Therefore, the current E left in the condition code register represents past action.

PIN DESCRIPTIONS**POWER (VSS, VCC)**

Two pins are used to supply power to the part. VSS is ground or 0 volts, while VCC is +5.0 V, $\pm 5\%$.

ADDRESS BUS (A0-A15)

Sixteen pins are used to output address information from the MPU onto the address bus. When the processor does not require the bus for a data transfer, it will output address FFFF₁₆. $R/\overline{W} = 1$, and $BS = 0$, this is a "dummy access" or $\overline{\text{VMA}}$ cycle. All address bus drivers are made high-impedance when output bus available (BA) is high or when TSC is asserted. Each pin will drive one Schottky TTL load or four LSTTL loads and 90 pF.

DATA BUS (D0-D7)

These eight pins provide communication with the system bidirectional data bus. Each pin will drive one Schottky TTL load or four LSTTL loads and 130 pF.

READ/WRITE (R/ \overline{W})

This signal indicates the direction of data transfer on the data bus. A low indicates that the MPU is writing data onto the data bus. R/\overline{W} is made high impedance when BA is high or when TSC is asserted.

RESET

A low level on this Schmitt-trigger input for greater than one bus cycle will reset the MPU, as shown in Figure 6. The

reset vectors are fetched from locations FFFF₁₆ and FFFF₁₆ (Table 1) when interrupt acknowledge is true, ($\overline{\text{BA}} \bullet \text{BS} = 1$). During initial power on, the reset line should be held low until the clock input signals are fully operational.

Because the MC6809E RESET pin has a Schmitt-trigger input with a threshold voltage higher than that of standard peripherals, a simple R/C network may be used to reset the entire system. This higher threshold voltage ensures that all peripherals are out of the reset state before the processor

HALT

A low level on this input pin will cause the MPU to stop running at the end of the present instruction and remain halted indefinitely without loss of data. When halted, the BA output is driven high indicating the buses are high impedance. BS is also high which indicates the processor is in the halt state. While halted, the MPU will not respond to external real-time requests ($\overline{\text{FIRQ}}$, $\overline{\text{IRQ}}$) although NMI or RESET will be latched for later response. During the halt state, Q and E should continue to run normally. A halted state ($\overline{\text{BA}} \bullet \text{BS} = 1$) can be achieved by pulling HALT low while RESET is still low. See Figure 7.

BUS AVAILABLE, BUS STATUS (BA, BS)

The bus available output is an indication of an internal control signal which makes the MOS buses of the MPU high impedance. When BA goes low, a dead cycle will elapse before the MPU acquires the bus. BA will not be asserted when TSC is active; thus allowing dead cycle consistency.

The bus status output signal, when decoded with BA, represents the MPU state (valid with leading edge of Q).

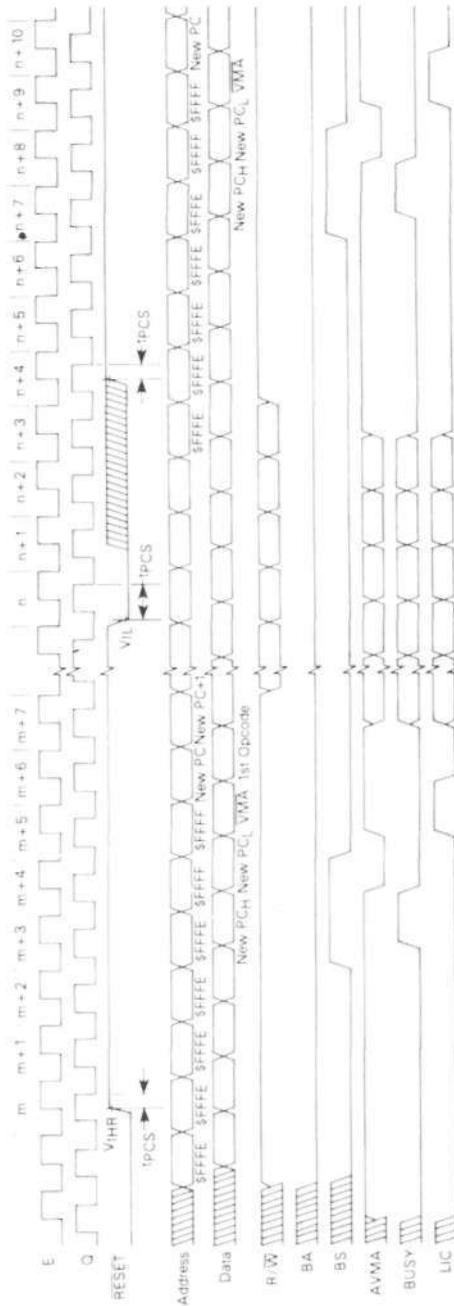
MPU State		MPU State Definition
BA	BS	
0	0	Normal (Running)
0	1	Interrupt or Reset Acknowledge
1	0	Sync Acknowledge
1	1	Halt Acknowledge

Interrupt Acknowledge is indicated during both cycles of a hardware vector fetch ($\overline{\text{RESET}}$, NMI, $\overline{\text{FIRQ}}$, $\overline{\text{IRQ}}$, SWI, SWI2, SWI3). This signal, plus decoding of the lower four address lines, can provide the user with an indication of which interrupt level is being serviced and allow vectoring by device. See Table 1.

TABLE 1 — MEMORY MAP FOR INTERRUPT VECTORS

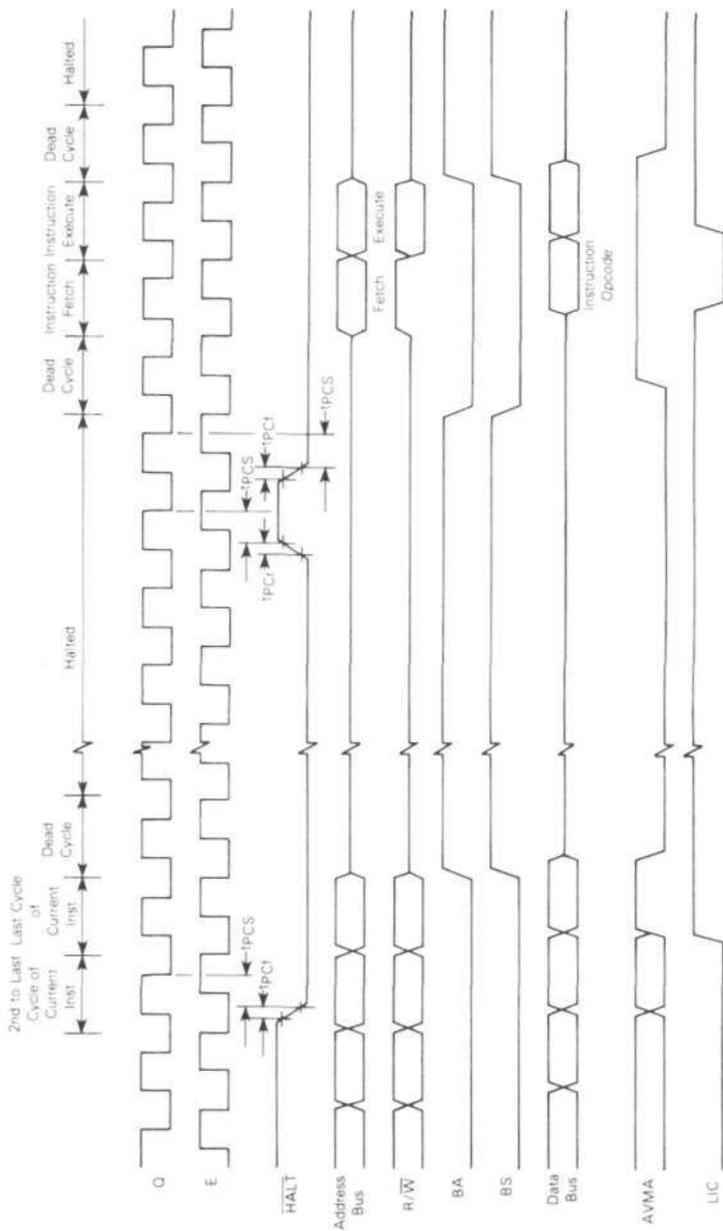
Memory Map For Vector Locations		Interrupt Vector Description
MS	LS	
FFFF	FFFF	RESET
FFFC	FFFD	NMI
FFFA	FFFB	SWI
FFF8	FFF9	$\overline{\text{IRQ}}$
FFF6	FFF7	$\overline{\text{FIRQ}}$
FFF4	FFF5	SWI2
FFF2	FFF3	SWI3
FFF0	FFF1	Reserved

FIGURE 6 – RESET TIMING



NOTE: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

FIGURE 7 — HALT AND SINGLE INSTRUCTION EXECUTION TIMING FOR SYSTEM DEBUG



NOTE: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

Sync Acknowledge is indicated while the MPU is waiting for external synchronization on an interrupt line.

Halt Acknowledge is indicated when the MC6809E is in a halt condition.

NON MASKABLE INTERRUPT ($\overline{\text{NMI}}$)*

A negative transition on this input requests that a non-maskable interrupt sequence be generated. A non-maskable interrupt cannot be inhibited by the program and also has a higher priority than $\overline{\text{FIRQ}}$, $\overline{\text{IRQ}}$, or software interrupts. During recognition of an $\overline{\text{NMI}}$, the entire machine state is saved on the hardware stack. After reset, an $\overline{\text{NMI}}$ will not be recognized until the first program load of the hardware stack pointer (SI). The pulse width of $\overline{\text{NMI}}$ low must be at least one E cycle. If the $\overline{\text{NMI}}$ input does not meet the minimum set up with respect to Q, the interrupt will not be recognized until the next cycle. See Figure 8.

FAST-INTERRUPT REQUEST ($\overline{\text{FIRQ}}$)*

A low level on this input pin will initiate a fast interrupt sequence, provided its mask bit (F) in the CC is clear. This sequence has priority over the standard interrupt request ($\overline{\text{IRQ}}$) and is fast in the sense that it stacks only the contents of the condition code register and the program counter. The interrupt service routine should clear the source of the interrupt before doing an RTI. See Figure 9.

INTERRUPT REQUEST ($\overline{\text{IRQ}}$)*

A low level input on this pin will initiate an interrupt request sequence provided the mask bit (I) in the CC is clear. Since $\overline{\text{IRQ}}$ stacks the entire machine state, it provides a slower response to interrupts than $\overline{\text{FIRQ}}$. $\overline{\text{IRQ}}$ also has a lower priority than $\overline{\text{FIRQ}}$. Again, the interrupt service routine should clear the source of the interrupt before doing an RTI. See Figure 8.

CLOCK INPUTS E, Q

E and Q are the clock signals required by the MC6809E. Q must lead E, that is, a transition on Q must be followed by a similar transition on E after a minimum delay. Addresses will be valid from the MPU, t_{AD} after the falling edge of E, and data will be latched from the bus by the falling edge of E. While the Q input is fully TTL compatible, the E input directly drives internal MOS circuitry and, thus, requires a high level above normal TTL levels. This approach minimizes clock skew inherent with an internal buffer. Refer to **BUS TIMING CHARACTERISTICS** for E and Q and to Figure 10 which shows a simple clock generator for the MC6809E.

BUSY

BUSY will be high for the read and modify cycles of a read-modify-write instruction and during the access of the first byte of a double-byte operation (e.g., LDX, STD, ADDD). **BUSY** is also high during the first byte of any indirect or other vector fetch (e.g., jump extended, SWI indirect, etc.).

In a multiprocessor system, **BUSY** indicates the need to

defer the re-arbitration of the next bus cycle to insure the integrity of the above operations. This difference provides the indivisible memory access required for a "test and set" primitive, using any one of several read-modify-write instructions.

BUSY does not become active during PSH or PUL operations. A typical read-modify-write instruction (ASL) is shown in Figure 11. Timing information is given in Figure 12. **BUSY** is valid t_{CD} after the rising edge of Q.

AVMA

AVMA is the advanced VMA signal and indicates that the MPU will use the bus in the following bus cycle. The predictive nature of the **AVMA** signal allows efficient shared bus multiprocessor systems. **AVMA** is low when the MPU is in either a HALT or SYNC state. **AVMA** is valid t_{CD} after the rising edge of Q.

LIC

LIC (last instruction cycle) is high during the last cycle of every instruction, and its transition from high to low will indicate that the first byte of an opcode will be latched at the end of the present bus cycle. **LIC** will be high when the MPU is halted at the end of an instruction (i.e., not in CWA1 or RESET), in sync state, or while stacking during interrupts. **LIC** is valid t_{CD} after the rising edge of Q.

TSC

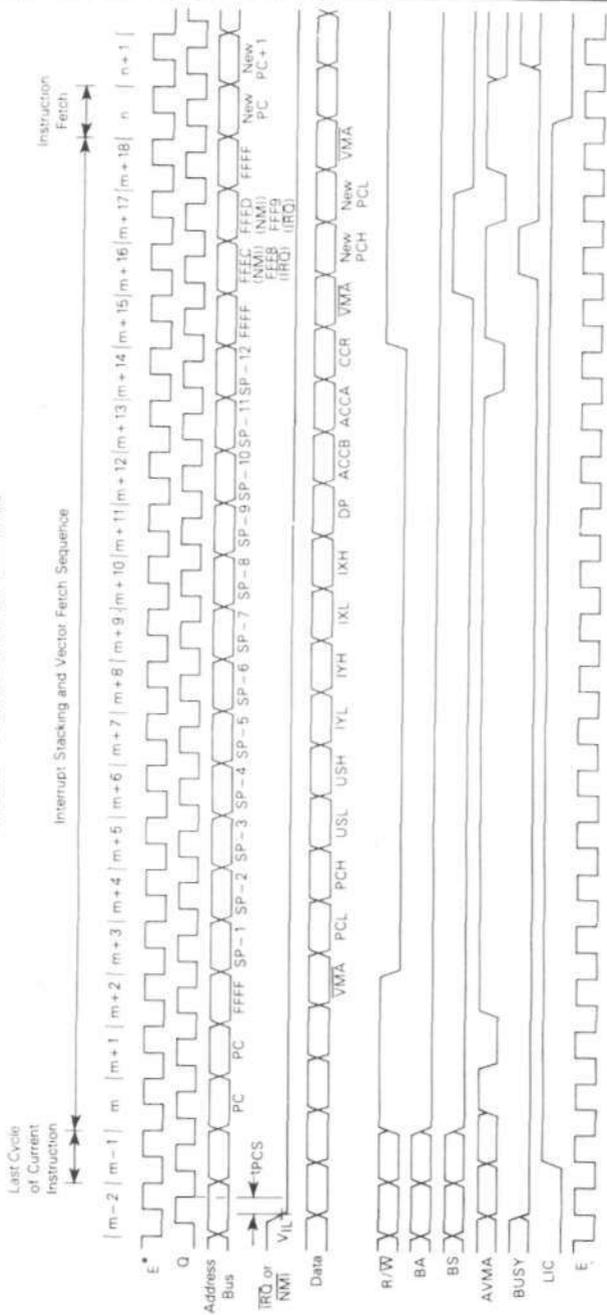
TSC (three-state control) will cause MOS address, data, and R/W buffers to assume a high-impedance state. The control signals (BA, BS, **BUSY**, **AVMA**, and **LIC**) will not go to the high-impedance state. **TSC** is intended to allow a single bus to be shared with other bus masters (processors or DMA controllers).

While E is low, **TSC** controls the address buffers and R/W directly. The data bus buffers during a write operation are in a high-impedance state until Q rises at which time, if **TSC** is true, they will remain in a high-impedance state. If **TSC** is held beyond the rising edge of E, then it will be internally latched, keeping the bus drivers in a high-impedance state for the remainder of the bus cycle. See Figure 13.

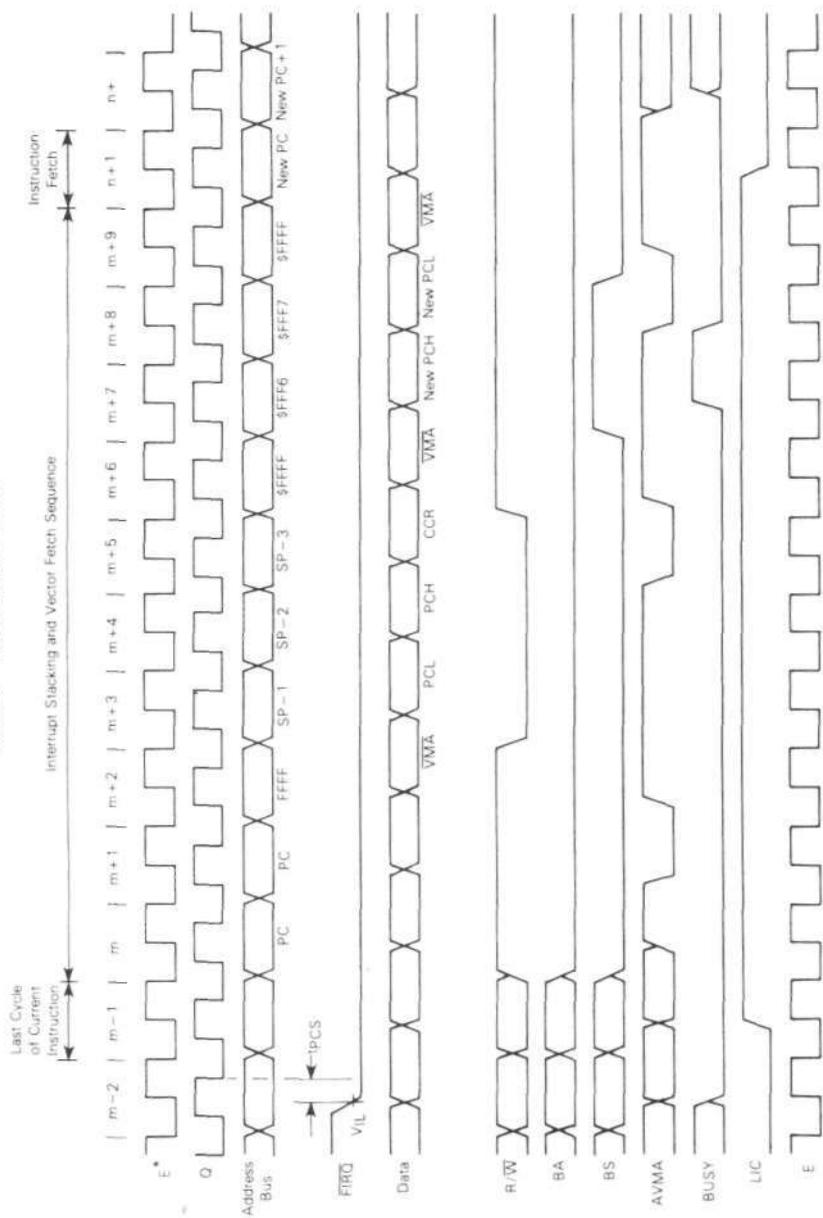
MPU OPERATION

During normal operation, the MPU fetches an instruction from memory and then executes the requested function. This sequence begins after **RESET** and is repeated indefinitely unless altered by a special instruction or hardware occurrence. Software instructions that alter normal MPU operation are SWI, SWI2, SWI3, CWA1, RTI, and SYNC. An interrupt or HALT input can also alter the normal execution of instructions. Figure 14 is the flowchart for the MC6809E.

* $\overline{\text{NMI}}$, $\overline{\text{FIRQ}}$, and $\overline{\text{IRQ}}$ requests are sampled on the falling edge of Q. One cycle is required for synchronization before these interrupts are recognized. The pending interrupt(s) will not be serviced until completion of the current instruction unless a SYNC or CWA1 condition is present. If $\overline{\text{IRQ}}$ and $\overline{\text{FIRQ}}$ do not remain low until completion of the current instruction, they may not be recognized. However, $\overline{\text{FIRQ}}$ is latched and need only remain low for one cycle. No interrupts are recognized or latched between the falling edge of **RESET** and the rising edge of BS indicating **RESET** acknowledge. See **RESET** sequence in the MPU flowchart in Figure 14.

FIGURE 8 – IRQ AND NMI INTERRUPT TIMING* E clock shown for reference only

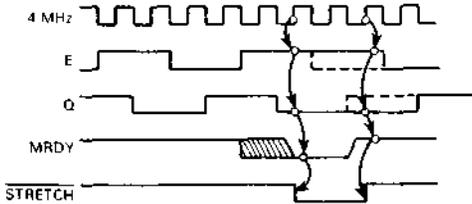
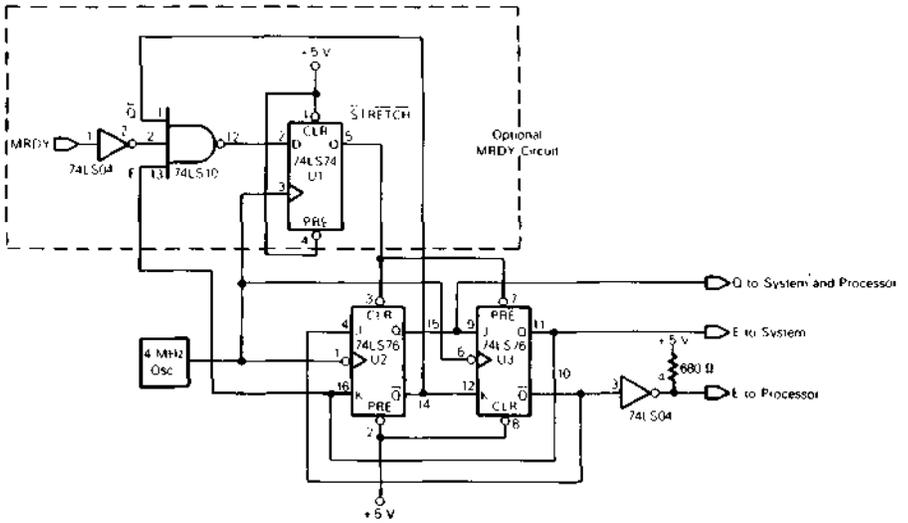
NOTE: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

FIGURE 9 – FIQ[®] INTERRUPT TIMING

* E clock shown for reference only

NOTE: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted

FIGURE 10 - CLOCK GENERATOR



NOTE: If optional circuit is not included the CLR and PRE inputs of U2 and U3 must be tied high

FIGURE 11 - READ-MODIFY-WRITE INSTRUCTION EXAMPLE (ASL EXTENDED INDIRECT)

Memory Location	Memory Contents	Contents Description
PC → \$0200	\$66	ASL Indexed Opcode
\$0201	\$9F	Extended Indirect Postbyte
\$0202	\$63	Indirect Address Hi-Byte
\$0203	\$00	Indirect Address Lo-Byte
\$0204		Next Main Instruction
\$6300	\$E3	Effective Address Hi-Byte
\$6301	\$D6	Effective Address Lo-Byte
\$E3D6	\$5C	Target Data

FIGURE 12 — BUSY TIMING

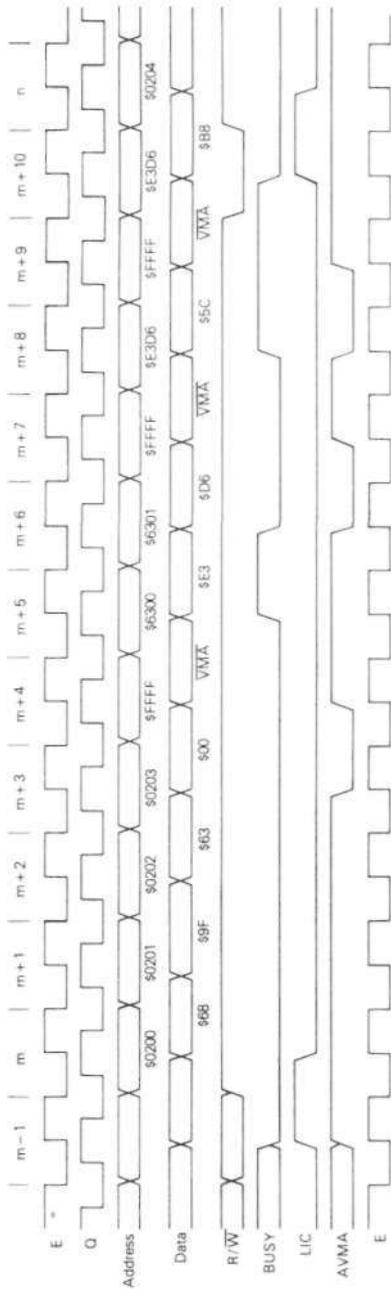
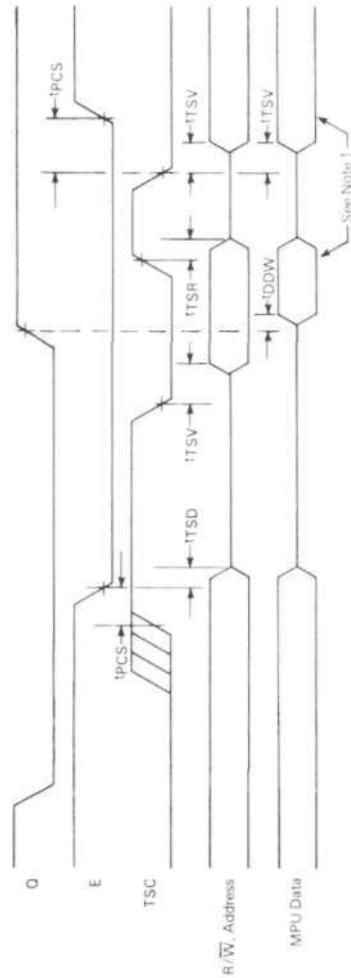
Last Cycle of
Current Instr.

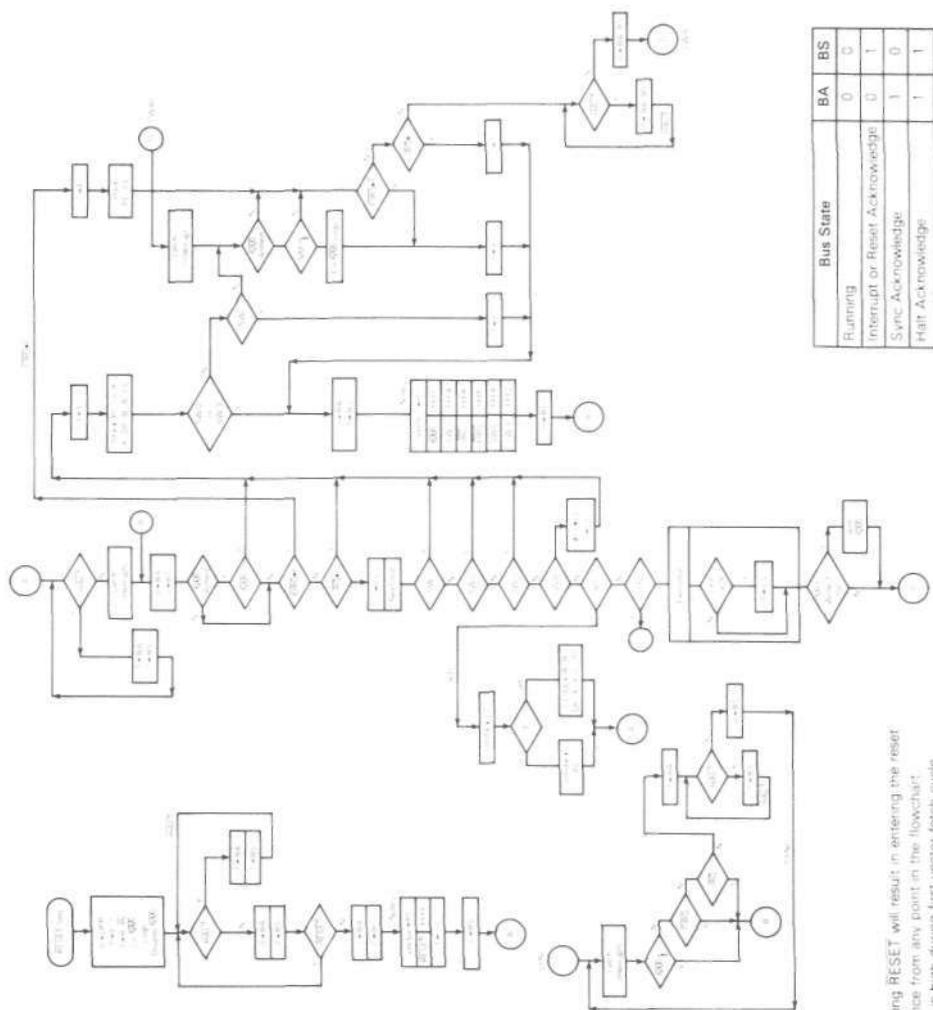
FIGURE 13 — TSC TIMING



NOTES:

1. Data will be asserted by the MPU only during the interval while R/\overline{W} is low and (E or Q) is high. A composite bus cycle is shown to give most cases of timing.
2. Timing measurements are referenced to and from a low voltage of 0.8 volts, and a high voltage of 2.0 volts, unless otherwise noted.

FIGURE 14 — FLOWCHART FOR MC6800E INSTRUCTIONS



NOTES 1. Asserting RESET will result in entering the reset sequence from any point in the flowchart.
 2. BUSY is high during first vector fetch cycle.

ADDRESSING MODES

The basic instructions of any computer are greatly enhanced by the presence of powerful addressing modes. The MC6809E has the most complete set of addressing modes available on any microcomputer today. For example, the MC6809E has 59 basic instructions; however, it recognizes 1464 different variations of instructions and addressing modes. The addressing modes support modern programming techniques. The following addressing modes are available on the MC6809E:

- Inherent (Includes Accumulator)
- Immediate
- Extended
 - Extended Indirect
- Direct
- Register
- Indexed
 - Zero-Offset
 - Constant Offset
 - Accumulator Offset
 - Auto Increment/Decrement
 - Indexed Indirect
- Relative
 - Short/Long Relative Branching
 - Program Counter Relative Addressing

INHERENT (INCLUDES ACCUMULATOR)

In this addressing mode, the opcode of the instruction contains all the address information necessary. Examples of inherent addressing are: ABX, DAA, SWI, ASRA, and CLRB.

IMMEDIATE ADDRESSING

In immediate addressing, the effective address of the data is the location immediately following the opcode (i.e., the data to be used in the instruction immediately following the opcode of the instruction). The MC6809E uses both 8- and 16-bit immediate values depending on the size of argument specified by the opcode. Examples of instructions with immediate addressing are:

```
LDA  #520
LDX  #$F000
LDY  #CAT
```

NOTE

signifies immediate addressing; \$ signifies hexadecimal value to the MC6809 assembler.

EXTENDED ADDRESSING

In extended addressing, the contents of the two bytes immediately following the opcode fully specify the 16-bit effective address used by the instruction. Note that the address generated by an extended instruction defines an absolute address and is not position independent. Examples of extended addressing include:

```
LDA  CAT
STX  MOUSE
LDD  $2000
```

EXTENDED INDIRECT

As a special case of indexed addressing (discussed below), one level of indirection may be added to extended addressing. In extended indirect, the two bytes following the postbyte of an indexed instruction contain the address of the data.

```
LDA  [CAT]
LDX  [$FFFE]
STU  [DOG]
```

DIRECT ADDRESSING

Direct addressing is similar to extended addressing except that only one byte of address follows the opcode. This byte specifies the lower eight bits of the address to be used. The upper eight bits of the address are supplied by the direct page register. Since only one byte of address is required in direct addressing, this mode requires less memory and executes faster than extended addressing. Of course, only 256 locations (one page) can be accessed without redefining the contents of the DP register. Since the DP register is set to \$00 on reset, direct addressing on the MC6809E is upward compatible with direct addressing on the M6800. Indirection is not allowed in direct addressing. Some examples of direct addressing are:

```
LDA  where DP = $00
LDB  where DP = $10
LDD  < CAT
```

NOTE

< is an assembler directive which forces direct addressing.

REGISTER ADDRESSING

Some opcodes are followed by a byte that defines a register or set of registers to be used by the instruction. This is called a postbyte. Some examples of register addressing are:

```
TFR  X, Y    Transfers X into Y
EXG  A, B    Exchanges A with B
PSHS A, B, X, Y Push Y, X, B and A onto S stack
PULU X, Y, D Pull D, X, and Y from U stack
```

INDEXED ADDRESSING

In all indexed addressing, one of the pointer registers (X, Y, U, S, and sometimes PC) is used in a calculation of the effective address of the operand to be used by the instruction. Five basic types of indexing are available and are discussed below. The postbyte of an indexed instruction specifies the basic type and variation of the addressing mode, as well as the pointer register to be used. Figure 15 lists the legal formats for the postbyte. Table 2 gives the assembler form and the number of cycles and bytes added to the basic values for indexed addressing for each variation.

FIGURE 15 - INDEXED ADDRESSING POSTBYTE REGISTER BIT ASSIGNMENTS

Post-Byte Register Bit							Indexed Addressing Mode
7	6	5	4	3	2	1 0	
0	R	R	d	d	d	d	EA = ,R + 5 Bit Offset
1	R	R	0	0	0	0	,R+
1	R	R	i	0	0	0	,R +
1	R	R	0	0	0	1	,- R
1	R	R	i	0	0	1	,-- R
1	R	R	i	0	1	0	EA = ,R + 0 Offset
1	R	R	i	0	1	0	EA = ,R + ACCB Offset
1	R	R	i	0	1	1	EA = ,R + ACCA Offset
1	R	R	i	1	0	0	EA = ,R + 8 Bit Offset
1	R	R	i	1	0	1	EA = ,R + 16 Bit Offset
1	R	R	i	1	0	1	EA = ,R + D Offset
1	x	x	i	1	1	0	EA = ,PC + 8 Bit Offset
1	x	x	i	1	1	0	EA = ,PC + 16 Bit Offset
1	R	R	i	1	1	1	EA = [,Address]

Addressing Mode Field

Indirect Field
(Sign Bit when b7 = 0)

Register Field: RR

x = Don't Care
d = Offset Bit
0 = Not Indirect
1 = Indirect

00 = X
01 = Y
10 = U
11 = S

ZERO-OFFSET INDEXED - In this mode, the selected pointer register contains the effective address of the data to be used by the instruction. This is the fastest indexing mode.

Examples are:

LDD O, X

LDA ,S

CONSTANT OFFSET INDEXED - In this mode, two complement offset and the contents of one of the pointer registers are added to form the effective address of the operand. The pointer register's initial content is unchanged by the addition.

Three sizes of offset are available:

5-bit (-16 to +15)

8-bit (-128 to +127)

16-bit (-32768 to +32767)

The two complement 5-bit offset is included in the postbyte and, therefore, is most efficient in use of bytes and cycles. The two complement 8-bit offset is contained in a single byte following the postbyte. The two complement 16-bit offset is in the two bytes following the postbyte. In most cases the programmer need not be concerned with the size of this offset since the assembler will select the optimal size automatically.

Examples of constant-offset indexing are:

LDA 23,X

LDX -2,S

LDY 300,X

LDU CAT,Y

TABLE 2 - INDEXED ADDRESSING MODE

Type	Forms	Non Indirect			Indirect		
		Assembler Form	Postbyte Opcode	+ ~ #	Assembler Form	Postbyte Opcode	+ ~ #
Constant Offset From R (2s Complement Offsets)	No Offset	,R	1RR00100	0 0	[,R]	1RR10100	3 0
	5-Bit Offset	n, R	0RRnnnnn	1 0	defaults to 8-bit		
	8-Bit Offset	n, R	1RR01000	1 1	[n, R]	1RR11000	4 1
	16-Bit Offset	n, R	1RR01001	4 2	[n, R]	1RR11001	7 2
Accumulator Offset From R (2s Complement Offsets)	A Register Offset	A, R	1RR00110	1 0	[A, R]	1RR10110	4 0
	B Register Offset	B, R	1RR00101	1 0	[B, R]	1RR10101	4 0
	D Register Offset	D, R	1RR01011	4 0	[D, R]	1RR11011	7 0
Auto Increment/Decrement R	Increment By 1	,R +	1RR00000	2 0	not allowed		
	Increment By 2	,R ++	1RR00001	3 0	[,R ++]	1RR10001	6 0
	Decrement By 1	,- R	1RR00010	2 0	not allowed		
	Decrement By 2	,- - R	1RR00011	3 0	[,- - R]	1RR10011	6 0
Constant Offset From PC (2s Complement Offsets)	8-Bit Offset	n, PCR	1xx01100	1 1	[n, PCR]	1xx11100	4 1
	16-Bit Offset	n, PCR	1xx01101	5 2	[n, PCR]	1xx11101	8 2
	Extended Indirect	-	-	-	[n]	10011111	5 2

R = X, Y, U or S
x = Don't Care

RR:
00 = X
01 = Y
10 = U
11 = S

+ and # indicate the number of additional cycles and bytes respectively for the particular indexing variation.

ACCUMULATOR-OFFSET INDEXED — This mode is similar to constant offset indexed except that the two's complement value in one of the accumulators (A, B, or D) and the contents of one of the pointer registers are added to form the effective address of the operand. The contents of both the accumulator and the pointer register are unchanged by the addition. The postbyte specifies which accumulator to use as an offset and no additional bytes are required. The advantage of an accumulator offset is that the value of the offset can be calculated by a program at run-time.

Some examples are:

```
LDA B, Y
LDX D, Y
LEAX B, X
```

AUTO INCREMENT/DECREMENT INDEXED — In the auto increment addressing mode, the pointer register contains the address of the operand. Then, after the pointer register is used, it is *incremented by one or two*. This addressing mode is useful in stepping through tables, moving data, or creating software stacks. In auto decrement, the pointer register is decremented prior to use as the address of the data. The use of auto decrement is similar to that of auto increment, but the tables, etc., are scanned from the high to low addresses. The size of the increment/decrement can be either one or two to allow for tables of either 8- or 16-bit data to be accessed and is selectable by the programmer. The pre-decrement, post-increment nature of these modes allows them to be used to create additional software stacks that behave identically to the U and S stacks.

Some examples of the auto increment/decrement addressing modes are:

```
LDA ,X+
STD ,Y++
LDB ,-Y
LDX ,--S
```

Care should be taken in performing operations on 16-bit pointer registers (X, Y, U, S) where the same register is used to calculate the effective address.

Consider the following instruction:

```
STX 0,X++ (X initialized to 0)
```

The desired result is to store a zero in locations \$0000 and \$0001, then increment X to point to \$0002. In reality, the following occurs:

```
0 → temp      calculate the EA; temp is a holding register
X + 2 → X      perform auto increment
X → (temp)    do store operation
```

INDEXED INDIRECT

All of the indexing modes, with the exception of auto increment/decrement by one or a ± 5 -bit offset, may have an additional level of indirection specified. In indirect addressing, the effective address is contained at the location specified by the contents of the index register plus any offset. In the example below, the A accumulator is loaded indirectly using an effective address calculated from the index register and an offset.

```
Before Execution
A = XX (don't care)
X = $F000
```

```
$0100 LDA [$10,X]   EA is now $F010
$F010 $F1           $F150 is now the
$F011 $50           new EA
$F150 SAA
```

After Execution

```
A = $AA (actual data loaded)
X = $F000
```

All modes of indexed indirect are included except those which are meaningless (e.g., auto increment/decrement by 1 indirect). Some examples of indexed indirect are:

```
LDA [X]
LDD [10,S]
LDA [B,Y]
LDD [X++]
```

RELATIVE ADDRESSING

The byte(s) following the branch opcode is (are) treated as a signed offset which may be added to the program counter. If the branch condition is true, then the calculated address (PC + signed offset) is loaded into the program counter. Program execution continues at the new location as indicated by the PC, short (one byte offset) and long (two bytes offset) relative addressing modes are available. All of memory can be reached in long relative addressing as an effective address interpreted modulo 2^{16} . Some examples of relative addressing are:

```
BEQ CAT (short)
BGT DOG (short)
CAT LBEQ RAT (long)
DOG LBGT RABBIT (long)
•
•
RAT NOP
RABBIT NOP
```

PROGRAM COUNTER RELATIVE

The PC can be used as the pointer register with 8- or 16-bit signed offsets. As in relative addressing, the offset is added to the current PC to create the effective address. The effective address is then used as the address of the operand or data. Program counter relative addressing is used for writing position independent programs. Tables related to a particular routine will maintain the same relationship after the routine is moved, if referenced relative to the program counter. Examples are:

```
LDA CAT, PCR
LEAX TABLE, PCR
```

Since program counter relative is a type of indexing, an additional level of indirection is available.

```
LDA [CAT, PCR]
LDU [DOG, PCR]
```

INSTRUCTION SET

The instruction set of the MC6809E is similar to that of the MC6800 and is upward compatible at the source code level. The number of opcodes has been reduced from 72 to 59, but because of the expanded architecture and additional addressing modes, the number of available opcodes (with different addressing modes) has risen from 197 to 1464.

Some of the new instructions are described in detail below.

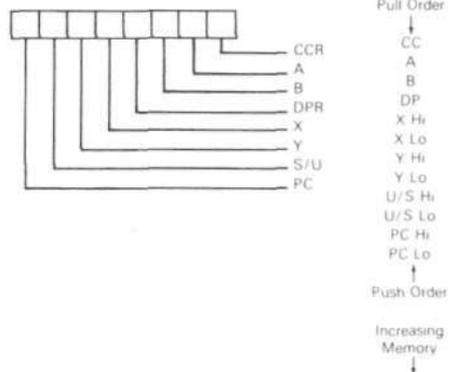
PSHU/PSHS

The push instructions have the capability of pushing onto either the hardware stack (S) or user stack (U) any single register or set of registers with a single instruction.

PULU/PULS

The pull instructions have the same capability of the push instruction, in reverse order. The byte immediately following the push or pull opcode determines which register or registers are to be pushed or pulled. The actual push/pull sequence is fixed, each bit defines a unique register to push or pull, as shown below.

Push/Pull Postbyte



TFR/EXG

Within the MC6809E, any register may be transferred to or exchanged with another of like size, i.e., 8-bit to 8-bit or 16-bit to 16-bit. Bits 4-7 of postbyte define the source register, while bits 0-3 represent the destination register. These are denoted as follows.

Transfer/Exchange Postbyte



Register Field

0000 = D (A B)	1000 = A
0001 = X	1001 = B
0010 = Y	1010 = CCR
0011 = U	1011 = DPR
0100 = S	
0101 = PC	

NOTE

All other combinations are undefined and INVALID.

LEAX/LEAY/LEAU/LEAS

The LEA (load effective address) works by calculating the effective address used in an indexed instruction and stores that address value, rather than the data at that address, in a pointer register. This makes all the features of the internal addressing hardware available to the programmer. Some of the implications of this instruction are illustrated in Table 3.

The LEA instruction also allows the user to access data and tables in a position independent manner. For example:

```
LEAX MSG1, PCR
LBSR  PDATA (Print message routine)
```

```
MSG1  FCC 'MESSAGE'
```

This sample program prints 'MESSAGE'. By writing MSG1, PCR, the assembler computes the distance between the present address and MSG1. This result is placed as a constant into the LEAX instruction which will be indexed from the PC value at the time of execution. No matter where the code is located when it is executed, the computed offset from the PC will put the absolute address of MSG1 into the X pointer register. This code is totally position independent.

The LEA instructions are very powerful and use an internal holding register (temp). Care must be exercised when using the LEA instructions with the auto increment and auto decrement addressing modes due to the sequence of internal operations. The LEA internal sequence is outlined as follows:

LEAa ,b+ (any of the 16-bit pointer registers X, Y, U, or S may be substituted for a and b.)

1. b → temp (calculate the EA)
2. b+1 → b (modify b, postincrement)
3. temp → a (load a)

LEAa , - b

1. b-1 → temp (calculate EA with predecrement)
2. b-1 → b (modify b, predecrement)
3. temp → a (load a)

TABLE 3 — LEA EXAMPLES

Instruction	Operation	Comment
LEAX 10, X	X + 10 → X	Adds 5-Bit Constant 10 to X
LEAX 500, X	X + 500 → X	Adds 16-Bit Constant 500 to X
LEAY A, Y	Y + A → Y	Adds 8-Bit A Accumulator to Y
LEAY D, Y	Y + D → Y	Adds 16-Bit D Accumulator to Y
LEAU -10, U	U - 10 → U	Subtracts 10 from U
LEAS -10, S	S - 10 → S	Used to Reserve Area on Stack
LEAS 10, S	S + 10 → S	Used to 'Clean Up' Stack
LEAX 5, S	S + 5 → X	Transfers As Well As Adds

Auto increment-by-two and auto decrement-by-two instructions work similarly. Note that LEAX, X+ does not change X; however LEAX, -X does decrement X. LEAX 1, X should be used to increment X by one.

MUL

Multiplies the unsigned binary numbers in the A and B accumulator and places the unsigned result into the 16-bit D accumulator. This unsigned multiply also allows multiple-precision multiplications.

LONG AND SHORT RELATIVE BRANCHES

The MC6809E has the capability of program counter relative branching throughout the entire memory map. In this mode, if the branch is to be taken, the 8- or 16-bit signed offset is added to the value of the program counter to be used as the effective address. This allows the program to branch anywhere in the 64K memory map. Position independent code can be easily generated through the use of relative branching. Both short (8 bit) and long (16 bit) branches are available.

SYNC

After encountering a sync instruction, the MPU enters a sync state, stops processing instructions, and waits for an interrupt. If the pending interrupt is non-maskable ($\overline{\text{NMI}}$) or maskable ($\overline{\text{IRQ}}$, $\overline{\text{IRQ}}$) with its mask bit (F or I) clear, the processor will clear the sync state and perform the normal interrupt stacking and service routine. Since $\overline{\text{FIRQ}}$ and $\overline{\text{IRQ}}$ are not edge-triggered, a low level with a minimum duration of three bus cycles is required to assure that the interrupt will be taken. If the pending interrupt is maskable ($\overline{\text{FIRQ}}$, $\overline{\text{IRQ}}$) with its mask bit (F or I) set, the processor will clear the sync state and continue processing by executing the next in-line instruction. Figure 16 depicts sync timing.

SOFTWARE INTERRUPTS

A software interrupt is an instruction which will cause an interrupt and its associated vector fetch. These software interrupts are useful in operating system calls, software debugging, trace operations, memory mapping, and software development systems. Three levels of SWI are available on this MC6809E and are prioritized in the following order: SWI, SWI2, SWI3.

16-BIT OPERATION

The MC6809E has the capability of processing 16-bit data. These instructions include loads, stores, compares, adds, subtracts, transfers, exchanges, pushes, and pulls.

CYCLE-BY-CYCLE OPERATION

The address bus cycle-by-cycle performance chart (Figure 16) illustrates the memory-access sequence corresponding to each possible instruction and addressing mode in the MC6809E. Each instruction begins with an opcode fetch. While that opcode is being internally decoded, the next program byte is always fetched. (Most instructions will use the next byte, so this technique considerably speeds throughput.) Next, the operation of each opcode will follow the flowchart. $\overline{\text{VMA}}$ is an indication of FFFF_{16} on the address bus, $\text{R}/\overline{\text{W}} = 1$ and $\text{BS} = 0$. The following examples illustrate the use of the chart.

Example 1: LBSR (Branch Taken) Before Execution SP = F000



CYCLE-BY-CYCLE FLOW

Cycle #	Address	Data	R/W	Description
1	8000	17	1	Opcode Fetch
2	8001	20	1	Offset High Byte
3	8002	00	1	Offset Low Byte
4	FFFF	*	1	$\overline{\text{VMA}}$ Cycle
5	FFFF	*	1	$\overline{\text{VMA}}$ Cycle
6	A000	*	1	Computed Branch Address
7	FFFF	*	1	$\overline{\text{VMA}}$ Cycle
8	FFFF	80	0	Stack High Order Byte of Return Address
9	FFFE	03	0	Stack Low Order Byte of Return Address

Example 2: DEC (Extended)

```

$8000      DEC      $A000
$A000      FCB      $80
  
```

CYCLE-BY-CYCLE FLOW

Cycle #	Address	Data	R/W	Description
1	8000	7A	1	Opcode Fetch
2	8001	A0	1	Operand Address, High Byte
3	8002	00	1	Operand Address, Low Byte
4	FFFF	*	1	$\overline{\text{VMA}}$ Cycle
5	A000	80	1	Read the Data
6	FFFF	*	1	$\overline{\text{VMA}}$ Cycle
7	FFFF	7F	0	Store the Decrement Data

* The data bus has the data at that particular address.

INSTRUCTION SET TABLES

The instructions of the MC6809E have been broken down into five different categories. They are as follows.

8-bit operation (Table 4)

16-bit operation (Table 5)

Index register/stack pointer instructions (Table 6)

Relative branches (long or short) (Table 7)

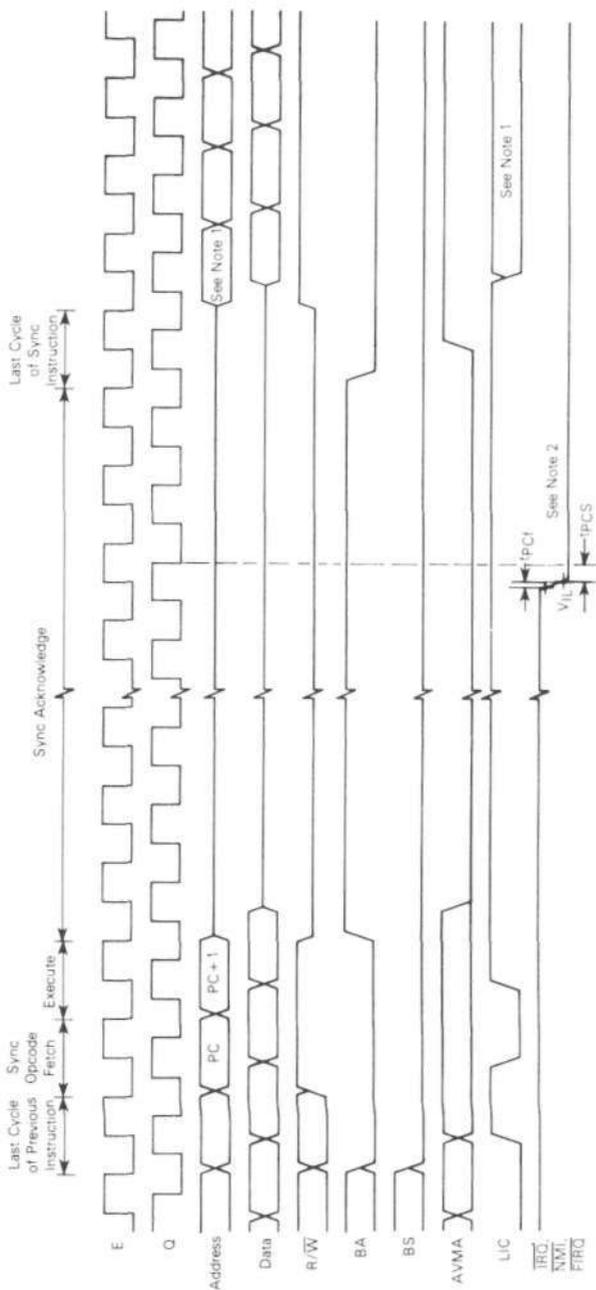
Miscellaneous instructions (Table 8)

Hexadecimal values for the instructions are given in Table 9.

PROGRAMMING AID

Figure 18 contains a compilation of data that will assist you in programming the MC6809E.

FIGURE 16 — SYNC TIMING

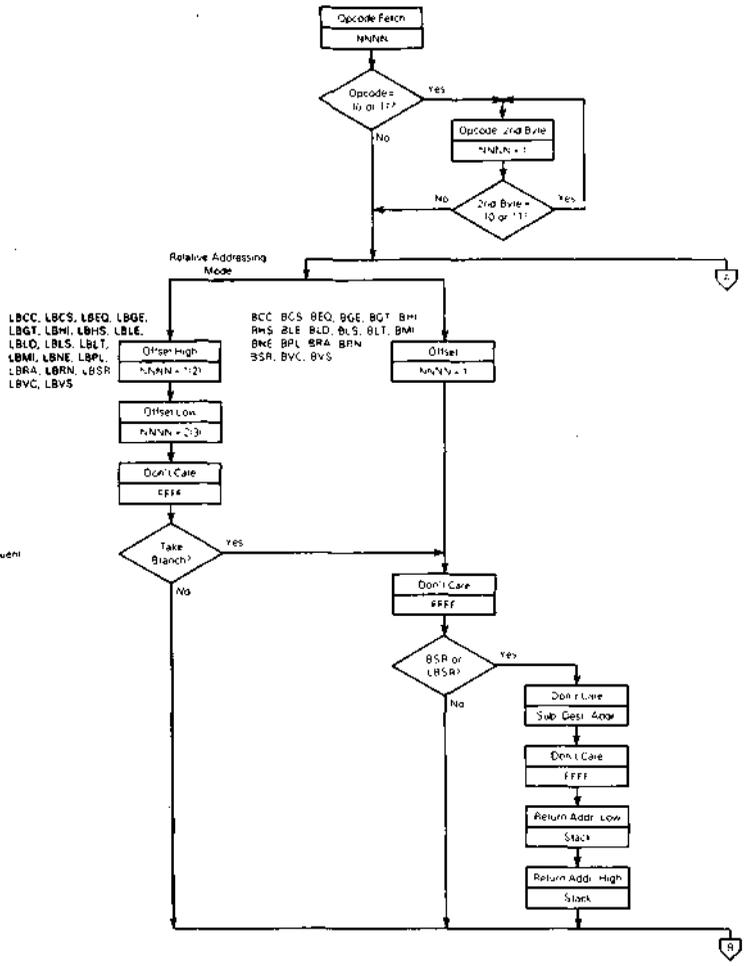


NOTES: 1. If the associated mask bit is set when the interrupt is requested, LIC will go low and this cycle will be an instruction fetch from address location PC + 1. However, if the interrupt is accepted (NMI or an unmasked FIRQ or IRQ), LIC will remain high and interrupt processing will start with this cycle as in Figures 8 and 9 (Interrupt Timing).

2. If mask bits are clear, IRQ and FIRQ must be held low for three cycles to guarantee that interrupt will be taken, although only one cycle is necessary to bring the processor out of SYNC.

3. Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

FIGURE 17 — CYCLE-BY-CYCLE PERFORMANCE (Sheet 1 of 5)



NOTES

- 1 Each state shows:

Data Bus	Offset High
Address Bus	NNNN + 12
- 2 Address NNNN is location of opcode
- 3 - If opcode is a two byte opcode subsequent addresses are in parenthesis (- 1)
- 4 Two-byte opcodes are highlighted

FIGURE 17 — CYCLE-BY-CYCLE PERFORMANCE (Sheet 3 of 5)

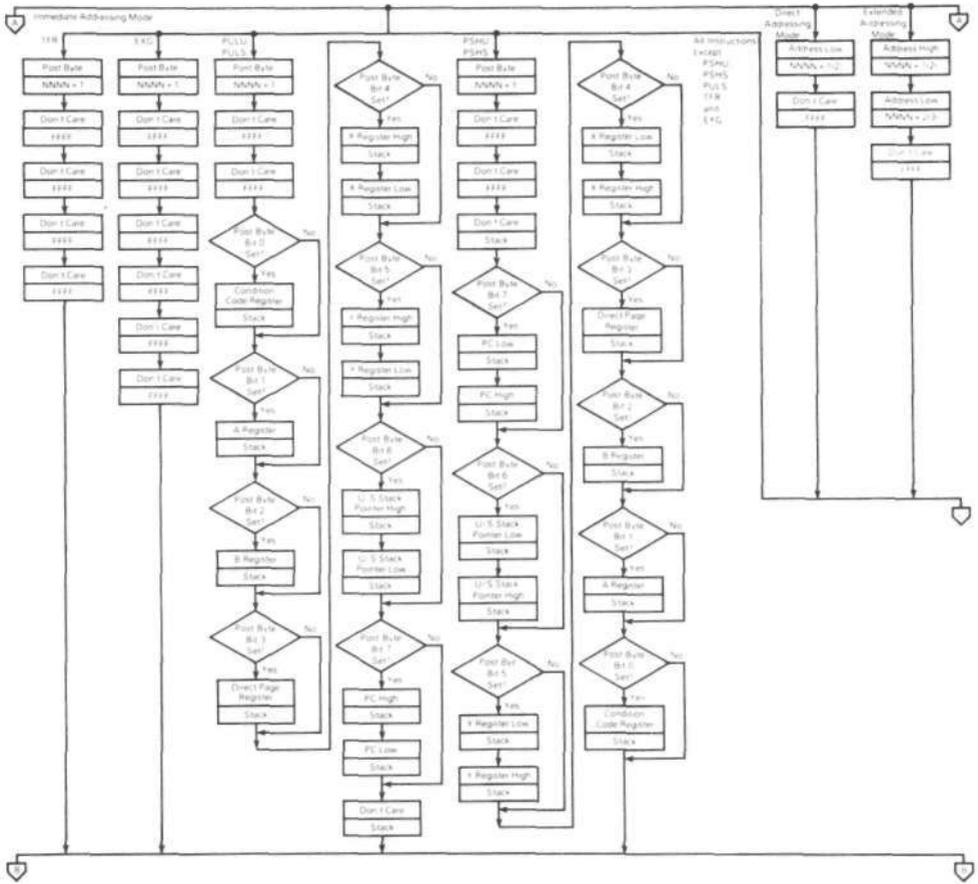


TABLE 4 – 8-BIT ACCUMULATOR AND MEMORY INSTRUCTIONS

Mnemonic(s)	Operation
ADCA, ADCB	Add memory to accumulator with carry
ADDA, ADDB	Add memory to accumulator
ANDA, ANDB	And memory with accumulator
ASL, ASLA, ASLB	Arithmetic shift of accumulator or memory left
ASR, ASRA, ASRB	Arithmetic shift of accumulator or memory right
BITA, BITB	Bit test memory with accumulator
CLR, CLRA, CLRB	Clear accumulator or memory location
CMPA, CMPB	Compare memory from accumulator
COM, COMA, COMB	Complement accumulator or memory location
DAA	Decimal adjust A accumulator
DEC, DECA, DECB	Decrement accumulator or memory location
EORA, EORB	Exclusive or memory with accumulator
EXG R1, R2	Exchange R1 with R2 (R1, R2 = A, B, CC, DP)
INC, INCA, INCB	Increment accumulator or memory location
LDA, LDB	Load accumulator from memory
LSL, LSLA, LSLB	Logical shift left accumulator or memory location
LSR, LSRA, LSRB	Logical shift right accumulator or memory location
MUL	Unsigned multiply (A x B → D)
NEG, NEGA, NEGB	Negate accumulator or memory
ORA, ORB	Or memory with accumulator
ROL, ROLA, ROLB	Rotate accumulator or memory left
ROR, RORA, RORB	Rotate accumulator or memory right
SBCA, SBCB	Subtract memory from accumulator with borrow
STA, STB	Store accumulator to memory
SUBA, SUBB	Subtract memory from accumulator
TST, TSTA, TSTB	Test accumulator or memory location
TFR R1, R2	Transfer R1 to R2 (R1, R2 = A, B, CC, DP)

NOTE: A, B, CC or DP may be pushed to (pulled from) either stack with PSHS, PSHU (PULS, PULU) instructions.

TABLE 5 – 16-BIT ACCUMULATOR AND MEMORY INSTRUCTIONS

Mnemonic(s)	Operation
ADDD	Add memory to D accumulator
CMPD	Compare memory from D accumulator
EXG D, R	Exchange D with X, Y, S, U or PC
LDD	Load D accumulator from memory
SEX	Sign Extend B accumulator into A accumulator
STD	Store D accumulator to memory
SUBD	Subtract memory from D accumulator
TFR D, R	Transfer D to X, Y, S, U or PC
TFR R, D	Transfer X, Y, S, U or PC to D

NOTE: D may be pushed (pulled) to either stack with PSHS, PSHU (PULS, PULU) instructions.

TABLE 6 – INDEX REGISTER/STACK POINTER INSTRUCTIONS

Instruction	Description
CMPX, CMPU	Compare memory from stack pointer
CMPX, CMPY	Compare memory from index register
EXG R1, R2	Exchange D, X, Y, S, U or PC with D, X, Y, S, U or PC
LEAS, LEAU	Load effective address into stack pointer
LEAX, LEAY	Load effective address into index register
LDS, LDU	Load stack pointer from memory
LDX, LDY	Load index register from memory
PSHS	Push A, B, CC, DP, D, X, Y, U, or PC onto hardware stack
PSHU	Push A, B, CC, DP, D, X, Y, S, or PC onto user stack
PULS	Pull A, B, CC, DP, D, X, Y, U or PC from hardware stack
PULU	Pull A, B, CC, DP, D, X, Y, S or PC from hardware stack
STS, STU	Store stack pointer to memory
STX, STY	Store index register to memory
TFR R1, R2	Transfer D, X, Y, S, U or PC to D; X, Y, S, U or PC
ABX	Add B accumulator to X (unsigned)

TABLE 7 – BRANCH INSTRUCTIONS

Instruction	Description
SIMPLE BRANCHES	
BEQ, LBEQ	Branch if equal
BNE, LBNE	Branch if not equal
BMI, LBMI	Branch if minus
BPL, LBPL	Branch if plus
BCS, LBCCS	Branch if carry set
BCC, LBCC	Branch if carry clear
BVS, LBVS	Branch if overflow set
BVC, LBVC	Branch if overflow clear
SIGNED BRANCHES	
BGT, LBGT	Branch if greater (signed)
BVS, LBVS	Branch if invalid 2's complement result
BGE, LBGE	Branch if greater than or equal (signed)
BEQ, LBEQ	Branch if equal
BNE, LBNE	Branch if not equal
BLE, LBLE	Branch if less than or equal (signed)
BVC, LBVC	Branch if valid 2's complement result
RLT, LBLT	Branch if less than (signed)
UNSIGNED BRANCHES	
BHI, LBHI	Branch if higher (unsigned)
BCC, LBCC	Branch if higher or same (unsigned)
BHS, LBHS	Branch if higher or same (unsigned)
BEQ, LBEQ	Branch if equal
BNE, LBNE	Branch if not equal
BLS, LBLS	Branch if lower or same (unsigned)
BCL, LBCL	Branch if lower (unsigned)
BLO, LBLO	Branch if lower (unsigned)
OTHER BRANCHES	
BSR, LBSR	Branch to subroutine
BRA, LBRA	Branch always
BRN, LBRN	Branch never

TABLE 8 – MISCELLANEOUS INSTRUCTIONS

Instruction	Description
ANDCC	AND condition code register
CWAI	AND condition code register, then wait for interrupt
NOP	No operation
ORCC	OR condition code register
JMP	Jump
JSR	Jump to subroutine
RTI	Return from interrupt
RTS	Return from subroutine
SWI, SWI2, SWI3	Software interrupt (absolute indirect)
SYNC	Synchronize with interrupt line

TABLE 9 — HEXADECFMAL VALUES OF MACHINE CODES

OP	Mnem	Mode	~	#	OP	Mnem	Mode	~	#	OP	Mnem	Mode	~	#
00	NEG	Direct ↑	6	2	30	LEAX	Indexed	4+	2+	60	NEG	Indexed ↑	6+	2+
01	*		31	LEAY	4+	2+	61	*						
02	*		32	LEAS	4+	2+	62	*						
03	COM		6	2	33	LEAU	Indexed	4+	2+	63	COM		6+	2+
04	LSR		6	2	34	PSHS	Immed	5+	2	64	LSR		6+	2+
05	*		35	PULS	5+	2	65	*						
06	ROR		6	2	36	PSHU	Immed	5+	2	66	ROR		6+	2+
07	ASR		6	2	37	PULU	Immed	5+	2	67	ASR		6+	2+
08	ASL, LSL		6	2	38	*	68	ASL, LSL	6+	2+				
09	ROL		6	2	39	RTS	Inherent	5	1	69	ROL		6+	2+
0A	DEC	6	2	3A	ABX	3	1	6A	DEC	6+	2+			
0B	*	3B	RTI	6/15	1	6B	*	6+	2+					
0C	INC	6	2	3C	CWAI	≥20	2	6C	INC	6+	2+			
0D	TST	6	2	3D	MUL	Inherent	11	1	6D	TST	6+	2+		
0E	JMP	3	2	3E	*	6E	JMP	3+	2+					
0F	CLR	Direct	6	2	3F	SWI	Inherent	19	1	6F	CLR	Indexed	6+	2+
10	Page 2	-	-	-	40	NEGA	Inherent	2	1	70	NEG	Extended ↑	7	3
11	Page 3	-	-	-	41	*	71	*						
12	NOP	Inherent	2	1	42	*	72	*						
13	SYNC	Inherent	≥4	1	43	COMA	2	1	73	COM	7		3	
14	*	44	LSRA	2	1	74	LSR	7	3					
15	*	45	*	75	*	7	3							
16	LBRA	Relative	5	3	46	RORA	2	1	76	ROR	7		3	
17	LBSR	Relative	9	3	47	ASRA	2	1	77	ASR	7		3	
18	*	48	ASLA, LSLA	2	1	78	ASL, LSL	7	3					
19	DAA	Inherent	2	1	49	ROLA	2	1	79	ROL	7		3	
1A	ORCC	Immed	3	2	4A	DECA	2	1	7A	DEC	7	3		
1B	*	-	-	-	4B	*	7B	*	7	3				
1C	ANDCC	Immed	3	2	4C	INCA	2	1	7C	INC	7	3		
1D	SEX	Inherent	2	1	4D	TSTA	2	1	7D	TST	7	3		
1E	EXG	Immed	8	2	4E	*	7E	JMP	4	3				
1F	TFR	Immed	6	2	4F	CLRA	Inherent	2	1	7F	CLR	Extended	7	3
20	BRA	Relative ↑	3	2	50	NEGB	Inherent ↑	2	1	80	SUBA	Immed ↑	2	2
21	BRN		3	2	51	*		81	CMPA	2	2			
22	BHI		3	2	52	*		82	SBCA	2	2			
23	BLS		3	2	53	COMB		2	1	83	SUBD		4	3
24	BHS, BCC		3	2	54	LSRB		2	1	84	ANDA		2	2
25	BLO, BCS		3	2	55	*		85	BITA	2	2			
26	BNE		3	2	56	RORB		2	1	86	LDA		2	2
27	BEG		3	2	57	ASRB		2	1	87	*		2	2
28	BVC		3	2	58	ASLB, LSLB		2	1	88	EORA		2	2
29	BVS		3	2	59	ROLB		2	1	89	ADCA		2	2
2A	BPL	3	2	5A	DECB	2	1	8A	ORA	2	2			
2B	BMI	3	2	5B	*	8B	ADDA	2	2					
2C	BGE	3	2	5C	INCB	2	1	8C	CMPX	4	3			
2D	BLT	3	2	5D	TSTB	2	1	8D	BSR	Immed	7	2		
2E	BGT	3	2	5E	*	8E	LDX	Relative	3	3				
2F	BLE	Relative	3	2	5F	CLRB	Inherent	2	1	8F	*	Immed	7	3

LEGEND:

- ~ Number of MPU cycles (less possible push pull or indexed-mode cycles)
- # Number of program bytes
- * Denotes unused opcode

FIGURE 18 - PROGRAMMING AID

Instruction	Forms	Addressing Modes												Description	5	4	3	2	1	0						
		Immediate			Direct			Indexed			Extended				Inherent			H	N	Z	V	C				
		Op	~	#		Op	~	#																		
ABX																3A	3	1		B ← X - X (Unsigned)	*	*	*	*	*	*
ADC	ADCA ADCB	89 C9	2	2	99 D9	4	2	A9 E9	4	2	B9 F9	5	3						A ← M + C - A B ← M + C - B	1	1	1	1	1	1	
ADD	ADDA ADDB ADDD	8B CB C3	2	2	9B DB D3	4	2	AB EB E3	4	2	B8 F8 F3	5	3						A ← M + A B ← M + B D ← M + M + 1 - D	1	1	1	1	1	1	
AND	ANDA ANDB ANDCC	84 C4 1C	2	2	94 D4	4	2	A4 E4	4	2	B4 F4	5	3						A ← M - A B ← M - B CC ← IMM - CC	*	1	1	0	*	*	
ASL	ASLA ASLB ASL													46 58	2	1	1		A ← M B ← M C ← 0	8	1	1	1	1	1	
ASR	ASRA ASRB ASR				07	6	2	67	6	2	77	7	3	47 57	2	1	1		A ← M B ← M C ← 0	8	1	1	*	1	*	
BIT	BITA BITB	85 C5	2	2	95 D5	4	2	A5 E5	4	2	B5 F5	5	3						Bit Test: A (M A A) Bit Test: B (M A B)	*	1	1	0	*	*	
CLH	CLRA CLRB CLR				0F	6	2	6F	6	2	7F	7	3	4F 5F	2	1	1		0 ← A 0 ← B 0 ← M	*	0	1	0	0	0	
CMP	CMPA CMPB CMPD CMPS CMPU CMPX CMPY	81 C1 10 83 11 8C 11 85 8C 10	2	2	91 D1 10 93 11 9C 11 93 9C 10	4	2	A1 E1 10 A3 11 AC 11 A3 AC 10	4	2	B1 F1 10 B3 11 BC 11 B3 BC 10	5	3						Compare M from A Compare M from B Compare M + M + 1 from D Compare M + M + 1 from S Compare M + M + 1 from U Compare M + M + 1 from X Compare M + M + 1 from Y	8	1	1	1	1	1	
COM	COMA COMB COM				03	6	2	63	6	2	73	7	3	43 53	2	1	1		A ← A B ← B M ← M	*	1	1	0	1	1	
CWAI		3C	2	2															CC ← IMM - CC (Wait for interrupt)	*	1	1	0	1	1	
DAA														19	2	1			Decimal Adjust A	*	1	1	0	1	1	
DEC	DECA DECB DEC				0A	6	2	6A	6	2	7A	7	3	4A 5A	2	1	1		A ← A - 1 B ← B - 1 M ← M - 1	*	1	1	1	*	*	
EDR	EDRA EDRB	8B CB	2	2	9B DB	4	2	AB EB	4	2	B8 F8	5	3						A ← M - A B ← M - B	*	1	1	0	*	*	
EXG	R1, R2	1E	8	2															R1 ← R2 R2 ← R1	*	*	*	*	*	*	
INC	INCA INCB INL				DC	6	2	6C	6	2	7C	7	3	4C 5C	2	1	1		A ← A + 1 B ← B + 1 M ← M + 1	*	1	1	1	*	*	
JMP					0E	3	2	6E	3	2	7E	4	3						EA ³ ← PC	*	*	*	*	*	*	
JSR					9D	7	2	AD	7	2	BD	8	3						Jump to Subroutine	*	*	*	*	*	*	
LD	LDA LDB LDD LDS LDU LDX LDY	86 C6 CC 10 CE 9E 10 8E	2	2	96 D6 5 D3 DE 9E 5 9E	4	2	A6 E6 5 E3 EE 5 AE	4	2	B6 F6 5 F3 FE 5 BE	5	3						M ← A M ← B M ← M - 1 - D M ← M - 1 - S M ← M - 1 - U M ← M + 1 - X M ← M + 1 - Y	*	1	1	0	*	*	
LEA	LEAS LEAU LEAX LEAY							32 33 30 31	4	2									EA ³ ← S EA ³ ← U EA ³ ← X EA ³ ← Y	*	*	*	*	*	*	

LEGEND

OP - Operation Code (Hexadecimal)

~ - Number of MPU Cycles

- Number of Program Bytes

+ - Arithmetic Plus

- - Arithmetic Minus

• - Multiply

M - Complement of M

← - Transfer Into

H - Half-carry (from bit 31)

N - Negative (sign bit)

Z - Zero result

V - Overflow, 2's complement

C - Carry from ALU

1 - Test and set if true, cleared otherwise

• - Not Affected

CC - Condition Code Register

Concatenation

V - Logical or

A - Logical and

↕ - Logical Exclusive or

FIGURE 18 — PROGRAMMING AID (CONTINUED)

Instruction	Forms	Addressing Modes									Description	5	3	2	1	0						
		Immediate			Direct			Indexed ¹									Extended			Inherent		
		Op	-	#	Op	-	#	Op	-	#	Op	-	#	Op	-	#	H	N	Z	V	C	
LSL	LSLA										48	2	1		*	*	*	*	*	*		
	LSLB LSL				08	6	2	68	6	2	78	7	3					*	*	*	*	*
LSR	LSRA										44	2	1		*	*	*	*	*			
	LSRB LSR				04	6	2	64	6	2	74	7	3					*	*	*	*	*
MUL											3D	11	1	A × B → D (Unsigned)		*	*	*	*	*	9	
NEG	NEGA										40	2	1	A ← 1 - A		*	*	*	*	*		
	NEGB NEG				00	6	2	60	6	2	70	7	3				*	*	*	*	*	
NOP											12	2	1	No Operation		*	*	*	*	*		
OR	ORA	8A	2	2	9A	4	2	AA	4	2	BA	5	3	A ∨ M → A		*	*	*	*	*		
	ORB	CA	2	2	DA	4	2	EA	4	2	FA	5	3	B ∨ M → B		*	*	*	*	*		
	ORCC	1A	3	2										CC ∨ IMM → CC		*	*	*	*	*	7	
PSH	PSHS	34	5	4	2									Push Registers on S Stack		*	*	*	*	*		
	PSHU	36	5	4	2									Push Registers on U Stack		*	*	*	*	*		
PUL	PULS	35	5	4	2									Pull Registers from S Stack		*	*	*	*	*		
	PULU	37	5	4	2									Pull Registers from U Stack		*	*	*	*	*		
ROL	ROLA										49	2	1		*	*	*	*	*			
	ROLB ROL				09	6	2	69	6	2	79	7	3					*	*	*	*	*
ROR	RORA										46	2	1		*	*	*	*	*			
	RORB ROR				06	6	2	66	6	2	76	7	3					*	*	*	*	*
RTI											3B	6	15	Return From Interrupt		*	*	*	*	*	7	
RTS											39	5	1	Return from Subroutine		*	*	*	*	*		
SBC	SBCA	B2	2	2	92	4	2	A2	4	2	B2	5	3	A - M - C → A		*	*	*	*	*		
	SBCB	C2	2	2	D2	4	2	E2	4	2	F2	5	3	B - M - C → B		*	*	*	*	*		
SEX											1D	2	1	Sign Extend B into A		*	*	*	*	*	0	
ST	STA				97	4	2	A7	4	2	B7	5	3	A → M		*	*	*	*	*		
	STB				D7	4	2	E7	4	2	F7	5	3	B → M		*	*	*	*	*	C	
	STD				DD	5	2	ED	5	2	FD	6	3	D → M M + 1		*	*	*	*	*	0	
	STS				10	6	3	10	6	3	10	7	4	S → M M + 1		*	*	*	*	*	0	
	STU				DF	5	2	EF	5	2	FF	6	3	U → M M + 1		*	*	*	*	*	0	
	STX				9F	5	2	AF	5	2	BF	6	3	X → M M + 1		*	*	*	*	*	0	
	STY				10	6	3	10	6	3	10	7	4	Y → M M + 1		*	*	*	*	*	0	
						9F			AF	6	3	BF					*	*	*	*	*	
																	*	*	*	*	*	
																	*	*	*	*	*	
SUB	SUBA	80	2	2	90	4	2	A0	4	2	B0	5	3	A - M → A		*	*	*	*	*		
	SUBB	CD	2	2	DD	4	2	E0	4	2	F0	5	3	B - M → B		*	*	*	*	*		
	SUBD	B3	4	3	93	6	2	A3	6	2	B3	7	3	D - M M + 1 → D		*	*	*	*	*		
SWI	SWI ⁶										3F	19	1	Software Interrupt 1		*	*	*	*	*		
	SWI ² ⁶										10	20	2	Software Interrupt 2		*	*	*	*	*		
											3F					*	*	*	*	*		
	SWI ³ ⁶										11	20	1	Software Interrupt 3		*	*	*	*	*		
											3F					*	*	*	*	*		
SYNC										13	≥ 4	1	Synchronize to Interrupt		*	*	*	*	*			
TFR	R1, R2	1F	6	2										R1 → R2 ²		*	*	*	*	*		
TST	TSTA										4D	2	1	Test A		*	*	*	*	*	0	
	TSTB										5D	2	1	Test B		*	*	*	*	*	0	
	TST				0D	6	2	8D	6	2	7D	7	3	Test M		*	*	*	*	*	0	

NOTES:

- This column gives a base cycle and byte count. To obtain total count, add the values obtained from the INDEXED ADDRESSING MODE table, Table 2.
- R1 and R2 may be any pair of 8 bit or any pair of 16 bit registers.
The 8 bit registers are: A, B, CC, DP.
The 16 bit registers are: X, Y, U, S, D, PC.
- EA is the effective address.
- The PSH and PUL instructions require 5 cycles plus 1 cycle for each **byte** pushed or pulled.
- 5(6) means: 5 cycles if branch not taken, 6 cycles if taken (Branch instructions)
- SWI sets I and F bits. SWI2 and SWI3 do not affect I and F.
- Conditions Codes set as a direct result of the instruction.
- Value of half-carry flag is undefined.
- Special Case — Carry set if b7 is SET.

FIGURE 18 – PROGRAMMING AID (CONTINUED)

Branch Instructions

Instruction	Forms	Addressing Mode		Description	5	3	2	1	0					
		OP	~ #							H	N	Z	V	C
BCC	BCC	24	3	2	•	•	•	•	•					
	LBCC	10	5(6)	4										
		24												
BCS	BCS	25	3	2	•	•	•	•	•					
	LBCS	10	5(6)	4										
		25												
BEQ	BEQ	27	3	2	•	•	•	•	•					
	LBEO	10	5(6)	4										
		27												
BGE	BGE	2C	3	2	•	•	•	•	•					
	LBGE	10	5(6)	4										
		2C												
BGT	BGT	2E	3	2	•	•	•	•	•					
	LBGT	10	5(6)	4										
		2E												
BHI	BHI	22	3	2	•	•	•	•	•					
	LBHI	10	5(6)	4										
		22												
BHS	BHS	24	3	2	•	•	•	•	•					
	LBHS	10	5(6)	4										
		24												
BLE	BLE	2F	3	2	•	•	•	•	•					
	LBLE	10	5(6)	4										
		2F												
BLO	BLO	25	3	2	•	•	•	•	•					
	LBLO	10	5(6)	4										
		25												

Instruction	Forms	Addressing Mode		Description	5	3	2	1	0					
		OP	~ #							H	N	Z	V	C
BLS	BLS	23	3	2	•	•	•	•	•					
	LBLS	10	5(6)	4										
		23												
BLT	BLT	2D	3	2	•	•	•	•	•					
	LBLT	10	5(6)	4										
		2D												
BMI	BMI	2B	3	2	•	•	•	•	•					
	LBMI	10	5(6)	4										
		2B												
BNE	BNE	26	3	2	•	•	•	•	•					
	LBNE	10	5(6)	4										
		26												
BPL	BPL	2A	3	2	•	•	•	•	•					
	LBPL	10	5(6)	4										
		2A												
BRA	BRA	20	3	2	•	•	•	•	•					
	LBRA	16	5	3										
BRN	BRN	21	3	2	•	•	•	•	•					
	LB RN	10	5	4										
		21												
BSR	BSR	8D	7	2	•	•	•	•	•					
	LBSR	17	9	3										
BVC	BVC	28	3	2	•	•	•	•	•					
	LBVC	10	5(6)	4										
		28												
BVS	BVS	29	3	2	•	•	•	•	•					
	LBVS	10	5(6)	4										
		29												

SIMPLE BRANCHES

	OP	~	#
BRA	20	3	2
LBRA	16	5	3
BRN	21	3	2
LB RN	1021	5	4
BSR	8D	7	2
LBSR	17	9	3

SIMPLE CONDITIONAL BRANCHES (Notes 1-4)

Test	True	OP	False	OP
N = 1	BMI	2B	BPL	2A
Z = 1	BEQ	27	BNE	26
V = 1	BVS	29	BVC	28
C = 1	BCS	25	BCC	24

SIGNED CONDITIONAL BRANCHES (Notes 1-4)

Test	True	OP	False	OP
r > m	BGT	2E	BLE	2F
r ≥ m	BGE	2C	BLT	2D
r = m	BEQ	27	BNE	26
r ≤ m	BLE	2F	BGT	2E
r < m	BLT	2D	BGE	2C

UNSIGNED CONDITIONAL BRANCHES (Notes 1-4)

Test	True	OP	False	OP
r > m	BHI	22	BLS	23
r ≥ m	BHS	24	BLO	25
r = m	BEQ	27	BNE	26
r ≤ m	BLS	23	BHI	22
r < m	BLO	25	BHS	24

NOTES

- All conditional branches have both short and long variations.
- All short branches are 2 bytes and require 3 cycles.
- All conditional long branches are formed by prefixing the short branch opcode with \$10 and using a 16-bit destination offset.
- All conditional long branches require 4 bytes and 6 cycles if the branch is taken or 5 cycles if the branch is not taken.
- 5(6) means: 5 cycles if branch not taken, 6 cycles if taken.

INDEXED ADDRESSING MODES

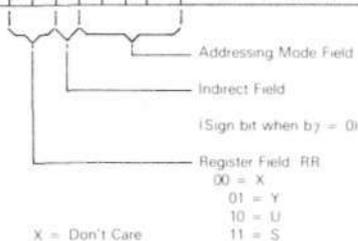
Type	Forms	Nondirect			Indirect		
		Assembler Form	Post-Byte Opcode	+ - #	Assembler Form	Post-Byte Opcode	+ - #
Constant Offset From R	No Offset	. R	1RR00100	0 0	[. R]	1RR10100	3 0
	5-Bit Offset	. n, R	0RRnmmmm	1 0	defaults to 8 bit		
	8-Bit Offset	. n, R	1RR01000	1 1	[n, R]	1RR11000	4 1
	16-Bit Offset	. n, R	1RR01001	4 2	[n, R]	1RR11001	7 2
Accumulator Offset From R	A - Register Offset	A, R	1RR00110	1 0	[A, R]	1RR10110	4 0
	B - Register Offset	B, R	1RR00101	1 0	[B, R]	1RR10101	4 0
	D - Register Offset	D, R	1RR01011	4 0	[D, R]	1RR11011	7 0
Auto Increment/Decrement R	Increment By 1	. R +	1RR00000	2 0	not allowed		
	Increment By 2	. R + +	1RR00001	3 0	[. R + +]	1RR10001	6 0
	Decrement By 1	. R	1RR00010	2 0	not allowed		
	Decrement By 2	. R	1RR00011	3 0	[. R]	1RR10011	6 0
Constant Offset From PC	8-Bit Offset	. n, PCR	1XX01100	1 1	[n, PCR]	1XX11100	4 1
	16-Bit Offset	. n, PCR	1XX01101	5 1	[n, PCR]	1XX11101	8 2
Extended Indirect	16-Bit Address				[n]	10011111	5 2

R = X, Y, U, or S
X = Don't Care

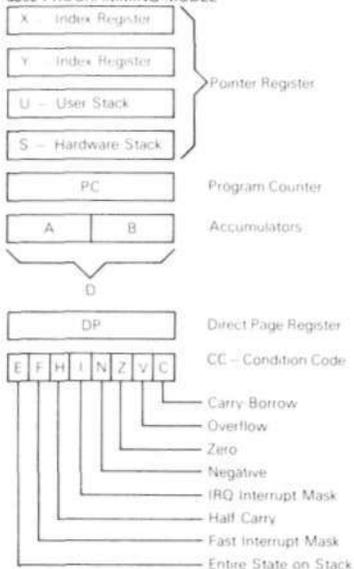
RR 00 = X 10 = U
01 = Y 11 = S

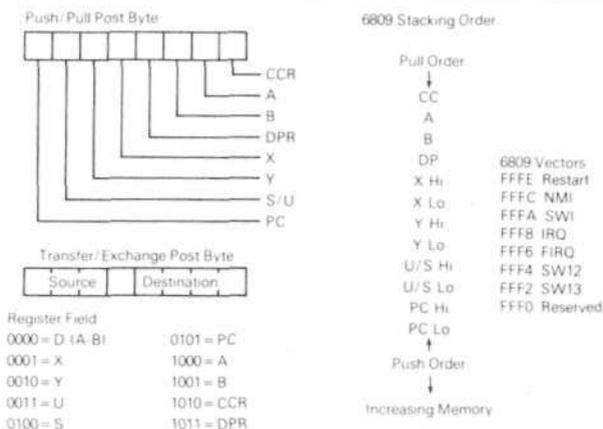
INDEXED ADDRESSING POSTBYTE REGISTER BIT ASSIGNMENTS

Post Byte Register Bit	Indexed Addressing Mode
7 6 5 4 3 2 1 0	
0 R R x x x x x	EA = . R + 5 Bit Offset
1 R R 0 0 0 0 0	. R +
1 R R 1 0 0 0 1	. R + +
1 R R 0 0 0 1 0	. R
1 R R 1 0 0 1 1	. R
1 R R 1 0 1 0 0	EA = . R + 0 Offset
1 R R 1 0 1 0 1	EA = . R + ACCB Offset
1 R R 1 0 1 1 0	EA = . R + ACCA Offset
1 R R 1 1 0 0 0	EA = . R + 8-Bit Offset
1 R R 1 1 0 0 1	EA = . R + 16-Bit Offset
1 R R 1 1 0 1 1	EA = . R + D Offset
1 x x 1 1 1 0 0	EA = . PC + 8 Bit Offset
1 x x 1 1 1 0 1	EA = . PC + 16 Bit Offset
1 R R 1 1 1 1 1	EA = [. Address]



6809 PROGRAMMING MODEL





ORDERING INFORMATION

Package Type	Frequency	Temperature Range	Order Number
Ceramic L Suffix	1.0 MHz	0°C to 70°C	MC6809EL
	1.0 MHz	-40°C to 85°C	MC6809ECL
	1.5 MHz	0°C to 70°C	MC68A09EL
	1.5 MHz	-40°C to 85°C	MC68A09ECL
	2.0 MHz	0°C to 70°C	MC68B09EL
	2.0 MHz	-40°C to 85°C	MC68B09ECL
Plastic P Suffix	1.0 MHz	0°C to 70°C	MC6809EP
	1.0 MHz	-40°C to 85°C	MC6809ECP
	1.5 MHz	0°C to 70°C	MC68A09EP
	1.5 MHz	-40°C to 85°C	MC68A09ECP
	2.0 MHz	0°C to 70°C	MC68B09EP
	2.0 MHz	-40°C to 85°C	MC68B09ECP
Cerdip S Suffix	1.0 MHz	0°C to 70°C	MC6809ES
	1.0 MHz	-40°C to 85°C	MC6809ECS
	1.5 MHz	0°C to 70°C	MC68A09ES
	1.5 MHz	-40°C to 85°C	MC68A09ECS
	2.0 MHz	0°C to 70°C	MC68B09ES
	2.0 MHz	-40°C to 85°C	MC68B09ECS

BETTER PROGRAM

BETTER program processing is available on all types listed.
Add suffix letters to part number.

Level 1 add "S" Level 2 add "D" Level 3 add "DS"
Level 1 "S" = 10 Temp Cycles (-1 to 25 to 150°C)
Hi Temp testing at T_A max.
Level 2 "D" = 168 Hour Burn-in at 125°C
Level 3 "DS" = Combination of Level 1 and 2

Appendix 2

SN74LS783 data sheet

Supplied courtesy of Motorola Semiconductors.

The information here has been carefully checked and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies. Motorola reserves the right to make changes to any products herein to improve reliability, function or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein. No licence is conveyed under patent rights in any form. When this document contains information on a new product, specifications herein are subject to change without notice.

MAXIMUM RATINGS ($T_A = 25^\circ\text{C}$ unless otherwise noted.)

Rating	Symbol	Value	Unit
Power Supply Voltage	V_{CC}	-0.5 to +7.0	Vdc
Input Voltage (Except Osc_{IN})	V_I	-0.5 to 10	Vdc
Input Current (Except Osc_{IN})	I_I	-30 to +5.0	mA
Output Voltage	V_O	-0.5 to +7.0	Vdc
Operating Ambient Temperature Range	T_A	0 to +70	$^\circ\text{C}$
Storage Temperature Range	T_{stg}	-65 to +150	$^\circ\text{C}$
Input Voltage Osc_{IN}	$V_{IOsc_{IN}}$	-0.5 to V_{CC}	Vdc
Input Current Osc_{IN}	$I_{IOsc_{IN}}$	-0.5 to +5.0	mA

RECOMMENDED OPERATING CONDITIONS

Rating	Symbol	Value	Unit
Power Supply Voltage	V_{CC}	4.75 to 5.25	Vdc
Operating Ambient Temperature Range	T_A	0 to +70	$^\circ\text{C}$

DC CHARACTERISTICS (Unless otherwise noted specifications apply over recommended power supply and temperature ranges.)

Characteristic	Symbol	Min	Typ	Max	Units
Input Voltage — High Logic State	V_{IH}	2.0	—	—	V
Input Voltage — Low Logic State	V_{IL}	—	—	0.8	V
Input Clamp Voltage ($V_{CC} = \text{Min}$, $I_{IN} = -18 \text{ mA}$) All Inputs Except Osc_{IN}	V_{IK}	—	—	-1.5	V
Input Current — High Logic State at Max Input Voltage ($V_{CC} = \text{Max}$, $V_{IN} = 5.25 \text{ V}$) V_{Clk} Input ($V_{CC} = \text{Max}$, $V_{IN} = 5.25 \text{ V}$) $DA0$ Input ($V_{CC} = \text{Max}$, $V_{IN} = 5.25 \text{ V}$) Osc_{OUT} Input ($V_{CC} = \text{Max}$, $V_{IN} = 7.0 \text{ V}$) All Other Inputs Except Osc_{IN}	I_I	—	—	200 100 250 100	μA
Input Current High Logic State ($V_{CC} = \text{Max}$, $V_{IN} = 2.7 \text{ V}$) All Inputs Except V_{Clk} , Osc_{IN}^*	I_{IH}	—	—	20	μA
Input Current — Low Logic State ($V_{CC} = \text{Max}$, $V_{IN} = 0.4 \text{ V}$) $DA0$ Input ($V_{CC} = \text{Max}$, $V_{IN} = 0.4 \text{ V}$) V_{Clk} Input ($V_{CC} = \text{Max}$, $V_{IN} = 0.4 \text{ V}$, $Osc_{IN} = \text{Gnd}$) Osc_{OUT} Input ($V_{CC} = \text{Max}$, $V_{IN} = 0.4 \text{ V}$) All Other Inputs Except Osc_{IN}	I_{IL}	—	—	-1.2 -60 -8 -4	mA
Output Voltage — High Logic State ($V_{CC} = \text{Min}$, $I_{OH} = -1.0 \text{ mA}$) $RAS0$, $RAS1$, CAS , WE ($V_{CC} = \text{Min}$, $I_{OH} = -0.2 \text{ mA}$) E , Q ($V_{CC} = \text{Min}$, $I_{OH} = -0.2 \text{ mA}$) All Other Outputs	$V_{OH(C)}$ $V_{OH(E)}$ V_{OH}	3.0 $V_{CC} - 0.75$ 2.7	—	—	V
Output Voltage — Low Logic State ($V_{CC} = \text{Min}$, $I_{OL} = 8.0 \text{ mA}$) $RAS0$, $RAS1$, CAS , WE ($V_{CC} = \text{Min}$, $I_{OL} = 4.0 \text{ mA}$) E , Q Outputs ($V_{CC} = \text{Min}$, $I_{OL} = 0.8 \text{ mA}$) V_{Clk} Output ($V_{CC} = \text{Min}$, $I_{OL} = 4.0 \text{ mA}$) All Other Outputs	$V_{OL(C)}$ $V_{OL(E)}$ $V_{OL(V)}$ V_{OL}	—	—	0.5 0.5 0.6 0.5	V
Power Supply Current	I_{CC}	—	180	230	mA
Output Short-Circuit Current	I_{OS}	30	—	225	mA

*Including Osc_{OUT} (when Osc_{IN} is grounded).

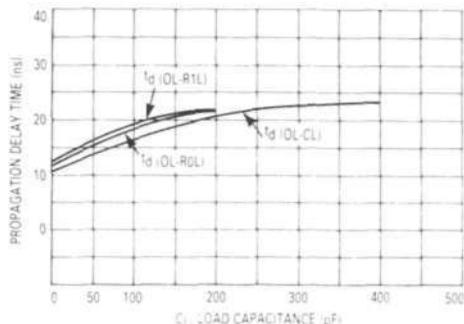
AC CHARACTERISTICS (4.75 V \leq V_{CC} \leq 5.25 V and 0 \leq T_A \leq 70°C, unless otherwise noted).

Characteristic	Symbol	Min	Typ	Max	Units	
Propagation Delay Times						
(See Circuit in Figure 9) Oscillator-In \rightarrow to Oscillator-Out \leftarrow	t _d (OL-OH)	—	3.0	—	ns	
Oscillator-In \leftarrow to Oscillator-Out \rightarrow	t _d (OH-OL)	—	20	—		
(C _L = 195 pF) A0 thru A15 to Z0, Z1, Z2 thru Z7	t _d (A-Z)	—	28	—		
(C _L = 30 pF) A0 thru A15, R/W to S0, S1, S3	t _d (A-S)	—	18	—		
(C _L = 95 pF) Oscillator-Out \rightarrow to RAS0 \leftarrow	t _d (OL-R0H)	—	20	—		
(C _L = 95 pF) Oscillator-Out \leftarrow to RAS0 \rightarrow	t _d (OL-R0L)	—	18	—		
(C _L = 95 pF) Oscillator-Out \rightarrow to RAS1 \leftarrow	t _d (OL-R1H)	—	22	—		
(C _L = 95 pF) Oscillator-Out \leftarrow to RAS1 \rightarrow	t _d (OL-R1L)	—	20	—		
(C _L = 195 pF) Oscillator-Out \rightarrow to CAS \leftarrow	t _d (OL-CH)	—	20	—		
(C _L = 195 pF) Oscillator-Out \leftarrow to CAS \rightarrow	t _d (OL-CL)	—	20	—		
(C _L = 195 pF) Oscillator-Out \rightarrow to WE \leftarrow	t _d (OL-WH)	—	22	—		
(C _L = 195 pF) Oscillator-Out \leftarrow to WE \rightarrow	t _d (OL-WL)	—	40	—		
(C _L = 100 pF) Oscillator-Out \rightarrow to E \leftarrow	t _d (OL-EH)	—	55	—		
(C _L = 100 pF) Oscillator-Out \leftarrow to E \rightarrow	t _d (OL-EL)	—	25	—		
(C _L = 100 pF) Oscillator-Out \rightarrow to Q \leftarrow	t _d (OL-QH)	—	55	—		
(C _L = 100 pF) Oscillator-Out \leftarrow to Q \rightarrow	t _d (OL-QL)	—	25	—		
(C _L = 30 pF) Oscillator-Out \leftarrow to VCIk \leftarrow	t _d (OH-VH)	—	50	—		
(C _L = 30 pF) Oscillator-Out \leftarrow to VCIk \rightarrow	t _d (OH-VL)	—	65	—		
(C _L = 195 pF) Oscillator-Out \rightarrow to Row Address	t _d (OL-AR)	—	36	—		
(C _L = 195 pF) Oscillator-Out \rightarrow to Column Address	t _d (OL-AC)	—	33	—		
(C _L = 15 pF) Oscillator-Out \rightarrow to DA0 \leftarrow Earliest(1)	t _d (OL-DH)	—	-15	—		
(C _L = 15 pF) Oscillator-Out \rightarrow to DA0 \leftarrow Latest(1)	t _d (OL-DL)	—	+15	—		
(C _L = 95 pF on RAS, C _L = 195 pF on CAS) CAS \rightarrow to RAS \leftarrow	t _d (CL-RH)	—	208	—		
Setup Time for A0 thru A15, R/W	Rate = -16 Rate = -8	t _{su} (A)	—	28 28	ns	
Hold Time for A0 thru A15, R/W	Rate = -16 Rate = -8	t _{hi} (A)	—	30 30	ns	
Width of HS Low ²		t _{wL} (HS)	2.0	5.0	6.0	μs

Notes: 1. When using the SAM with an MC6847, the rising edge of DA0 is confined within the range shown in the timing diagrams (unless the synchronizing process is incomplete.) The synchronizing process requires a maximum of 32 cycles of Osc_{OUT} for completion.

2. t_{wL}(HS) wider than 6.0 μs may yield more than 8 sequential refresh addresses.

**FIGURE 1 — PROPAGATION DELAY TIMES
VERSUS LOAD CAPACITANCE**



PIN DESCRIPTION TABLE

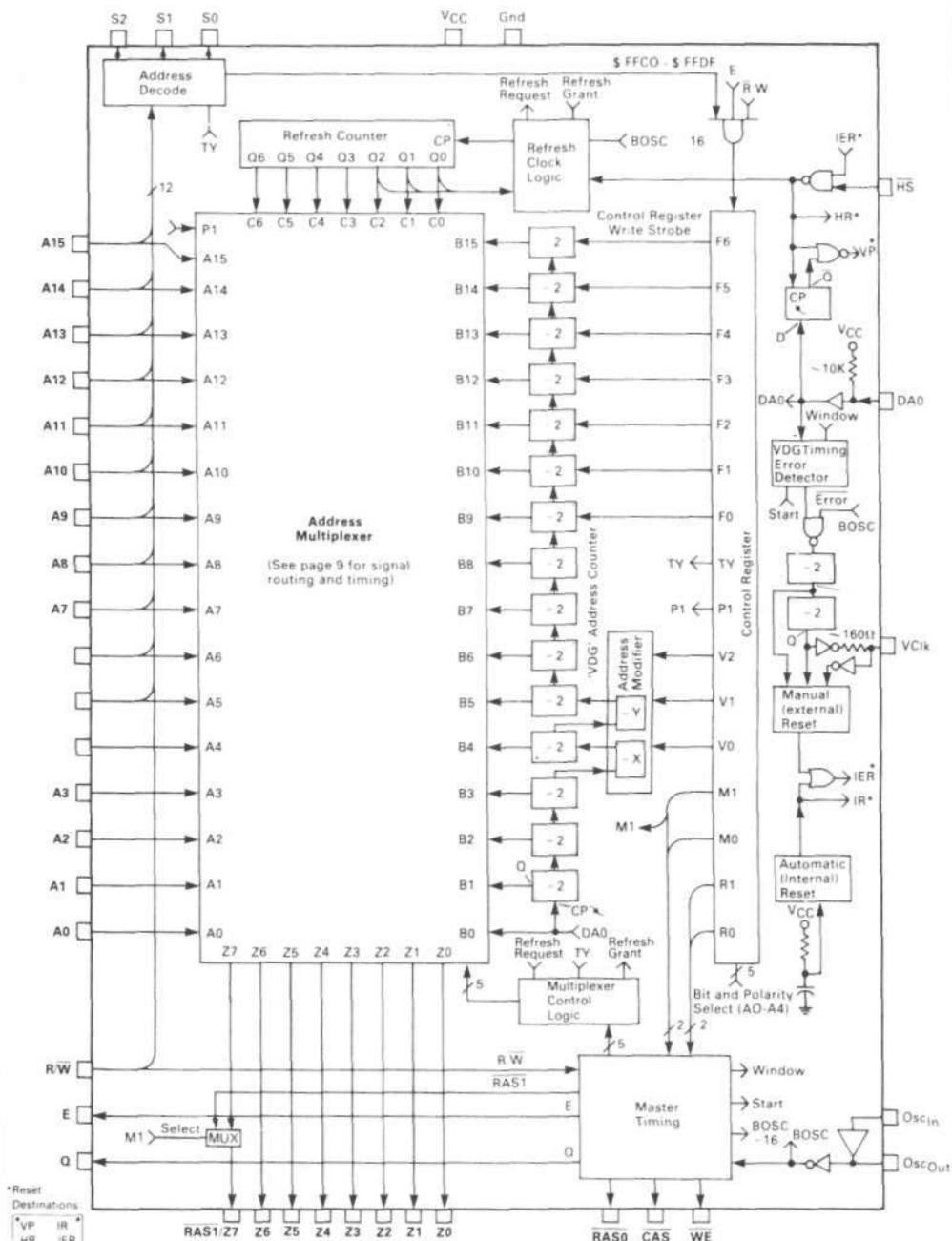
		Name	No.	Function
Input Pins	Power	V _{CC}	40	Apply + 5 volts ± 5%. SAM draws less than 230 mA.
		Gnd	20	Return Ground for + 5 volts.
	MPU Address and Control	A15	36	Most Significant Bit. MPU address bits A0-A15. These 16 signals come directly from the MPU and are used to directly address up to 64K memory locations or to indirectly address up to 96K memory locations. (See pages 17 and 18 for memory maps). Each input is approximately equivalent to one iow power Schottky load.
		A14	37	
		A13	38	
		A12	39	
		A11	1	
		A10	2	
		A9	3	
		A8	4	
		A7	24	
		A6	23	
		A5	22	
		A4	21	
		A3	19	
A2	18			
A1	17			
A0	16	Least Significant Bit.		
	R/W	15	MPU READ or WRITE. This signal comes directly from the MPU and is used to enable writing to the SAM control register, dynamic RAM (via WE), and to enable device select #0.	
VDG Control	Osc _{in}	5	Apply 14.31818* MHz crystal and 2.5–30 pF trimmer to ground. See page 12.	
	DA0	8	Display Address DA0. The primary function of this pin is to input the least significant bit of a 18-bit video display address. The more significant 15-bits are outputs from an internal 15-bit counter which is clocked by DA0. The secondary function of this pin is to indirectly input the logic level of the VDG "FS" (field synchronization pulse) for vertical video address updating.	
	HS	9	Horizontal Synchronization. The primary function of this pin is to detect the falling edge of VDG "HS" pulse in order to initiate eight dynamic RAM refresh cycles. The secondary function is to reset up to 4 least significant bits of the internal video address counter.	
	VClk	7	VDG Clock. The primary function of this pin is to output a 3.579545 MHz square wave** to the VDG "Clk" pin. The secondary function resets the SAM when this VClk pin is pulled to logic "0" level, acting as an input.	
	Osc _{Out}	6	Apply 1.5 kΩ resistor to 14.31818* MHz crystal and 33 pF capacitor to ground. See page 12.	
Device Selects	S2	25	Most Significant Bit (Device Select Bits). The binary value of S2, S1, S0 selects one of eight "chunks" of MPU address space (numbers 0 through 7). Varying in length, these "chunks" provide efficient memory mapping for ROMs, RAMs, Input/Output devices, and MPU Vectors. (Requires 74LS 138-type demultiplexer).	
	S1	26		
	S0	27		Least Significant Bit
MPU Clocks	E	14	E (Enable Clock) "E" and "Q" are 90° out of phase and are both used as MPU clocks for the MC6809E. For the MC6800 and MC6801E, only "E" is used. "E" is also used for many MC6800 peripheral chips.	
	Q	13	Q (Quadrature Clock).	
RAM Address	Z7†	35	Most Significant Bit First, the least significant address bits from the MPU or "VDG" are presented to Z0–Z5 (4K x 1 RAMs) or Z0–Z6 (16K x 1 RAMs) or Z0–Z7 (64K x 1 RAMs). Next, the most significant address bits from the MPU or "VDG" are presented to Z0–Z5 (4K x 1 RAMs) or Z0–Z6 (16K x 1 RAMs) or Z0–Z7 (64K x 1 RAMs). Note that for 4K x 1 and 16K x 1 RAMs, Z7 (Pin 35) is not needed for address information. Therefore, Pin 35 is used for a second row address select which is labeled (RAS1).	
	Z6†	34		
	Z5†	33		
	Z4†	32		
	Z3†	31		
	Z2†	30		
	Z1†	29		
	Z0†	28		Least Significant Bit.
RAM Control	RAS1†	35	Row Address Strobe One. This pulse strobes the least significant 6, 7 or 8 address bits into dynamic RAMs in Bank #1.	
	RAS0†	12	Row Address Strobe Zero. This pulse strobes the least significant 6, 7 or 8 address bits into dynamic RAMs in Bank #0.	
	CAS†	11	Column Address Strobe. This pulse strobes the most significant 6, 7 or 8 address bits into dynamic RAMs.	
	WE†	10	Write Enable. When low, this pulse enables the MPU to write into dynamic RAM.	

*14.31818 MHz is 4 times 3.579545 MHz television color subcarrier. Other frequencies may be used. (See page 12.)

**When VDG and SAM are not yet synchronized the "square wave" will stretch (see page 10.)

† Due to fast transitions, ferrite beads in series with these outputs may be necessary to avoid high frequency (~ 60 MHz) resonances.

FIGURE 4 — SAM BLOCK DIAGRAM



SAM BLOCK DIAGRAM DESCRIPTION

MPU Addresses (A0 - A15):

These 16 signals come directly from the MPU and are used to directly address up to 64K memory locations (K = 1024) or to indirectly address up to 96K memory locations, by using a paging bit "P" (see pages 17 and 18 for memory maps.) Each input is approximately equivalent to one low power Schottky load.

VDG Address Counter (B0 - B15):

These 16 signals are derived from one input (DA0) which is the least significant bit of the VDG address. Most of the counter is simply binary. However, to duplicate the various addressing modes of the MC6847 VDG, ADDRESS MODIFIER logic is used. Selected by three VDG mode bits (V2, V1, and V0) from the SAM CONTROL REGISTER, eight address modifications are obtained as shown in Figure 5.

Also, notice that bits B9-B15 may be loaded from bits F0-F6 from the CONTROL REGISTER. This allows the starting address of the VDG display to be offset (in 1/2K increments) from \$0000 to \$FFFF. B9-B15 are loaded when a VERTICAL PRE-LOAD (VP) pulse is generated. VP goes active (high) when HS from the VDG rises if DA0 is high (or a high impedance.) This condition should occur only while the TV electron beam is in vertical blanking and is simply implemented by connecting FS and MS together on the MC6847. The VP pulse also clears bits B1 - B8.

Finally, a HORIZONTAL RESET (HR) pulse may also affect the counter by clearing bits B1 - B3 or B1 - B4 when HS from the VDG is LOW (see Figure 5.) The HR pulse should occur only while the TV electron beam is in horizontal blanking.

In summary, DA0 clocks the VDG ADDRESS COUNTER; HR initializes the horizontal portion and VP initializes the vertical portion of the VDG ADDRESS COUNTER.

REfresh Address Counter (C0 - C6):

A seven bit binary counter with outputs labeled C0 - C6 supplies bursts of eight* sequential addresses triggered by a HS high to low transition. Thus, while the TV electron beam is in horizontal blanking, eight sequential addresses are accessed. Likewise, the next eight addresses are accessed during the next horizontal blanking period, etc. In this manner, all 128 addresses are refreshed in less than 1.1 milliseconds.

Address Multiplexer:

Occupying a large portion of the block diagram in Figure 4, is the address multiplexer which outputs bits Z0-Z7 (as addresses to dynamic RAM's.) Inputs to the address multiplexer include the VDG address (B0 - B15) the REfresh address (C0 - C6) and the MPU address (A0 - A15) or (A0 - A14 plus one paging bit "P".) The paging bit "P" is one bit in the SAM CONTROL REGISTER that is used in place of A15 when memory map TYPe #0 is selected (via the SAM CONTROL REGISTER "TY" bit.)

Figure 6 shows which inputs are routed to Z0 - Z7 and when the routing occurs relative to one SAM machine cycle. Notice that Z7 and RAS1 share the same pin. Z7 is selected if "M1" in the SAM CONTROL REGISTER IS HIGH (Memory size = 64K.)

Address Decode:

At the top left of Figure 4, is the Address Decode block. Outputs S2, S1, and S0 form a three bit encoded binary word(S). Thus S may be one of eight values (0 through 7) with each value representing a different range of MPU addresses. (To enable peripheral ROM's or I/O, decode the S2, S1, and S0 bits into eight separate signals by using a 74LS138, 74LS155 or 74LS156. Notice that S2, S1, and S0 are not gated with any timing signals such as E or Q.)

Along with the A5 - A15 inputs is the MEMORY MAP TYPe bit (TY.) This bit is soft-programmable (as are all 16 bits in the SAM CONTROL REGISTER,) and selects one of two memory maps. Memory map #0 is intended to be used in systems that are primarily ROM based. Whereas, memory map #1 is intended for a primarily RAM based system with 64K contiguous RAM locations (minus 256 locations.) The various meanings of S2, S1, S0 are tabulated in Figure 16 (page 19) and again on pages 17 and 18.

In addition to S2, S1, and S0 outputs is a decode of \$FFC0 through \$FFDF which, when gated with E and R/W, results in the write strobe for the SAM CONTROL REGISTER.

SAM Control Register

As shown in Figure 4, the CONTROL REGISTER has 16 "outputs":

VDG Addressing Modes:	V2, V1, V0	MPU Rate:	R1, R0
VDG Address Offset:	F6, F5, F4, F3, F2, F1, F0	Memory Size (RAM):	M1, M0
32K Page Switch:	P	Memory Map TYPe:	TY

When the SAM is reset (see page 10,) all 16 bits are cleared. To set any one of these 16 bits, the MPU simply writes to a unique** odd address (within \$FFC1 through \$FFDF.) To clear any one of these 16 bits, the MPU

* If RS is held low longer than 8 μ s, then the number of sequential addresses in one refresh "BURST" is proportional to the time interval during which RS is low.

** See pages 17 or 18 for specific addresses.

† In this document, the "\$" symbol always precedes hexadecimal characters.

simply writes to a unique** even address (within \$FFCO through \$FFDE.) Note that the data on the MPU data bus is irrelevant.

Inputs to the control register include A4, A3, A2, A1 (which are used to select which one of 16 bits is to be cleared or set), A0 (which determines the polarity ... clear or set,) and R W, E and \$FFCO - \$FFDF (which restrict the method, timing and addresses for changing one of the 16 bits.) For more detailed descriptions of the purposes of the 16 control bits, refer to related sections in the BLOCK DIAGRAM DESCRIPTION (pages 8 through 12) and the PROGRAMMING GUIDE (pages 14 through 18).

** See pages 17 or 18 for specific addresses.

FIGURE 5 — VDG ADDRESS MODIFIER

Mode			Division Variables		Bits Cleared by HS (low)
V2	V1	V0	X	Y	
0	0	0	1	12	B1-B4
0	0	1	3	1	B1-B3
0	1	0	1	3	B1-B4
0	1	1	2	1	B1-B3
1	0	0	1	2	B1-B4
1	0	1	1	1	B1-B3
1	1	0	1	1	B1-B4
1	1	1	1	1	None (DMA MODE)

FIGURE 6 — SIGNAL ROUTING for ADDRESS MULTIPLEXER

Memory Size		Signal Source	Row/Column	Signals Routed to Z0-Z7								Timing (Figure 2)	
M1	M0			Z7	Z6	Z5	Z4	Z3	Z2	Z1	Z0		
4K	0	0	MPU	ROW	*	A6	A5	A4	A3	A2	A1	A0	T7-TA
				COL	*	L	A11	A10	A9	A8	A7	A6	TA-TF
			VDG	ROW	*	B6	B5	B4	B3	B2	B1	B0	TF-T2
				COL	*	L	B11	B10	B9	B8	B7	B6	T2-T7
			REF	ROW	*	C6	C5	C4	C3	C2	C1	C0	TF-T2
				COL	*	L	L	L	L	L	L	L	T2-T7
16K	0	1	MPU	ROW	*	A6	A5	A4	A3	A2	A1	A0	T7-TA
				COL	*	A13	A12	A11	A10	A9	A8	A7	TA-TF
			VDG	ROW	*	B6	B5	B4	B3	B2	B1	B0	TF-T2
				COL	*	B13	B12	B11	B10	B9	B8	B7	T2-T7
			REF	ROW	*	C6	C5	C4	C3	C2	C1	C0	TF-T2
				COL	*	L	L	L	L	L	L	L	T2-T7
64K (dynamic)	1	0	MPU	ROW	A7	A6	A5	A4	A3	A2	A1	A0	T7-TA
				COL	P/A15***	A14	A13	A12	A11	A10	A9	A8	TA-TF
			VDG	ROW	B7	B6	B5	B4	B3	B2	B1	B0	TF-T2
				COL	B15	B14	B13	B12	B11	B10	B9	B8	T2-T7
			REF	ROW	L	C6	C5	C4	C3	C2	C1	C0	TF-T2
				COL	L	L	L	L	L	L	L	L	T2-T7
64K (static)	1	1	MPU	ROW	A7	A6	A5	A4	A3	A2	A1	A0	T7-T9
				COL	P/A15***	A14	A13	A12	A11	A10	A9	A8	T9-TF
			VDG	ROW	B7	B6	B5	B4	B3	B2	B1	B0	TF-T1
				COL	B15	B14	B13	B12	B11	B10	B9	B8	T1-T7
			REF	ROW	L	C6	C5	C4	C3	C2	C1	C0	TF-T1
				COL	L	L	L	L	L	L	L	L	T1-T7

Notes: 'L' implies logical LOW level.

*Z7 functions as RAS1 and its level is address dependent. For example, when using two banks of 16K x 1 RAMs, RAS0 is active for addresses

\$0000 to \$3FFF and RAS1 is active for addresses \$4000 to \$7FFF.

***If Map Type = 0, then page bit 'P' is the output (otherwise A15).

Internal Reset

By lowering V_{CC} below 0.6 volts for at least one millisecond, a **complete** SAM reset is initiated and is completed within 500 nanoseconds after V_{CC} rises above 4.25 volts.

NOTE: In some applications, (for example, multiple "VDG-RAM" systems controlled by a single MPU)

multiple SAM ICs can be synchronized as follows:

- Drive all SAM's from one external oscillator.
- Stop external oscillator.
- Lower V_{CC} below 0.6 volts for at least 1.0 millisecond.
- Raise V_{CC} to 5.0 volts.
- Start external oscillator.
- Wait at least 500 nanoseconds.

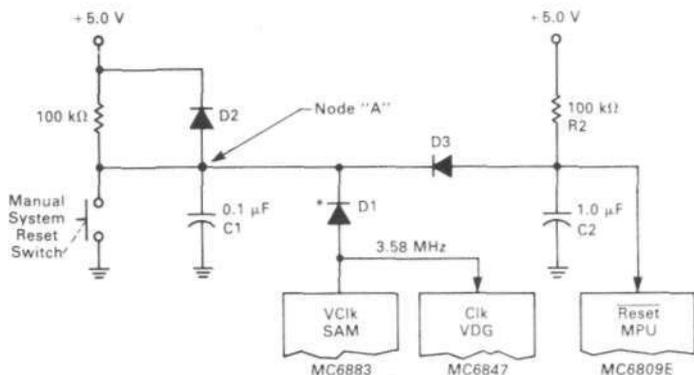
Now, the "E" clocks from all SAM's should be in-phase.

External Reset

When the VCik pin on SAM is forced below 0.8 volts for at least eight cycles of "oscillator-out", the SAM becomes **partially** reset. That is, all bits in the SAM control register are cleared. However, signals such as \overline{RAS} , \overline{CAS} , \overline{WE} , E or Q are **not** stopped (as they are with an **internal** reset), since the SAM must maintain dynamic RAM refresh even during this external reset period.

Figure 7 shows how VCik can be pulled low through diode D1 when node "A" is low.* When node "A" is high, only the backbiased capacitance of diode D1 loads the 3.58 MHz on VCik. Diode D2 helps discharge C1 (Power-on-Reset capacitor) when power is turned off. Diode D3 allows the MPU reset time constant $R2C2$ to be greater than the SAM reset time constant. Thereby, ensuring **release** of the SAM reset **prior** to attempting to program the SAM control register.

FIGURE 7 — EXTERNAL RESET CIRCUITRY



VDG Synchronization

In order for the VDG and MPU to share the same dynamic RAM (see page 13,) the **VDG clock must be stopped** until the VDG data fetch and MPU data fetch are synchronized as shown in Figure 12. Once synchronized, the VDG clock resumes its 3.579545 MHz rate and is not stopped again unless an extreme temperature change (or SAM reset) occurs. When stopped, the VDG clock remains stopped for **no more than 32 Osc_{Out} cycles** (approximately 2 microseconds.)

In the block diagram in Figure 4, DA0 enters a block labeled VDG Timing Error Detector. If DA0 rises **between** time reference points** τ_A and τ_C , then Error is high and VCik is the result of dividing BOSC (Buffered Osc_{Out} \approx 14 MHz) by four. However, if DA0 rises **outside** the time Window τ_A to τ_C , then Error goes LOW and the VDG stops. A START pulse at time reference point τ_B (center of Window) restarts the VDG . . . properly synchronized.

*Use a diode with sufficiently low forward voltage drop to meet V_{IL} requirement at VCik.

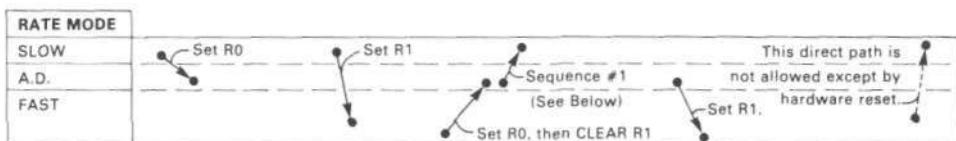
**See timing diagrams on page 5 and 6.

Changing the MPU Rate (by changing SAM control register bits R0, R1).

Two bits in the SAM control register determine the period of both "E" and "Q" MPU clocks. Three rate modes are implemented as follows:

RATE MODE	R1	R0	
SLOW	0	0	The frequency of "E" (and "Q") is $f_{\text{crystal}} \div 16$. This rate mode is automatically selected when the SAM is reset. Note that system timing is least critical in this "SLOW" rate mode.
A.D. (Address Dependent)	0	1	The frequency of "E" (and "Q") is either $f_{\text{crystal}} \div 16$ or $f_{\text{crystal}} \div 8$, depending on the address the MPU is presenting.
FAST	1	X	The frequency of "E" (and "Q") is $f_{\text{crystal}} \div 8$. This is accomplished by stealing the time that is normally used for VDG-REFRESH, and using this time for the MPU. Note: Neither VDG display nor dynamic RAM refresh are available in the "FAST" rate mode. (Both are available in SLOW and A.D. rate modes).

When changing between any two of the three rate modes, the following procedures must be followed to ensure that MPU timing specifications are met:



May be ANY address from \$0000 to \$7FFF.

SEQUENCE #1:

```
7D 00 00 TST # $0000 ... Synchronizes STA instruction to write during T2-TG (See Figure #8).*
21 00 BRN 00
B7 FF D6 STA # $FFD6 ... Clears bit R0
```

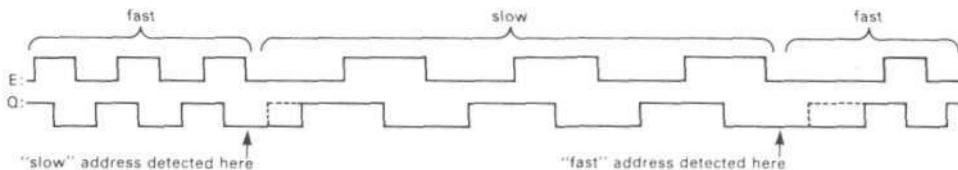
*Note: "TST" instruction affects MC6809E condition code register.

Changing the MPU Rate (In Address Dependent Mode)

When the SAM control register bits "R1", and "R0" are programmed to "0" and "1", respectively, the Address Dependent Rate Mode is selected. In this mode, the $\div 16$ MPU rate is automatically used when addressing within \$0000 to \$7FFF* or \$FF00 to \$FF1F ranges. Otherwise the $\div 8$ MPU rate is automatically used. (Refer to Figure 8 for sample "E" and "Q" waveforms yielding $\div 8$ to $\div 16$ and $\div 16$ to $\div 8$ rate changes). This mode often nearly doubles the MPU throughput while still providing transparent VDG and dynamic, RAM refresh functions. For example, since much of the MPU's time may be spent performing internal MPU functions (address = \$FFFF)**, accessing ROM (address = \$8000 to \$FEFF) or accessing I/O (address = \$FF20 — \$FF5F), the faster $f_{\text{crystal}} \div 8$ MPU rate may be used much of the time.

Note: The VDG operates normally when using the SLOW or A.D. rate modes. However, in the FAST rate mode, the VDG is not allowed access to the dynamic RAM.

FIGURE 8 — RATE CHANGE E AND Q WAVEFORMS

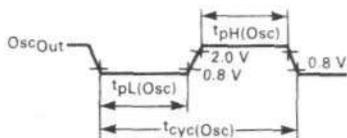


*When using Memory Map 0, addresses \$0000 to \$7FFF may access Dynamic RAM.

**The MC6809 outputs \$FFFF on A0-A15 when no other valid addresses are being presented.

Oscillator

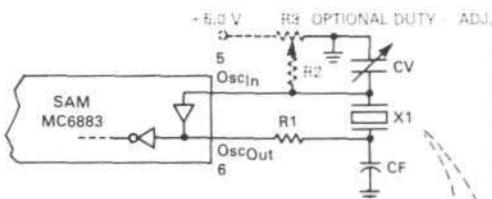
In Figure 4, an amplifier between Osc_{In} and Osc_{Out} provides the gain for oscillation (using a crystal as shown in Figure 9.) Alternately, Pin 5 (Osc_{In}) may be grounded while Pin 6 (Osc_{Out}) may be driven at low-power Schottky levels as shown in Figure 10. Also, see V_{IH} , V_{IL} on page 2.



AC Specifications*

	Max	Typ	Min	Units
$t_{pH}(Osc)$	—	30	22	ns
$t_{pL}(Osc)$	—	30	22	ns
$t_{cyc}(Osc)$	—	70	62.4	ns

FIGURE 9 — CRYSTAL OSCILLATOR



Suggested Component Values

Freq. MHz	CV*	CF*	R1*	R2*	R3*	X1
14.31818	2.5-30 pF	33 pF	1.5 k Ω	~100K	10K	*
16.0000	2.5-30 pF	33 pF	1.5 k Ω	~100K	10K	*

Recommended Crystal Parameters

	14.31818 MHz**	16.0000 MHz**
R_S	10 $\Omega \pm 2.0 \Omega$	10 $\Omega \pm 2.0 \Omega$
CO	5.0 pF ± 1.5 pF	6.0 pF ± 1.0 pF
C1	0.0245 pF $\pm 15\%$	0.0319 pF $\pm 15\%$
L1	5.05 mH	3.1 mH
Q	50K ± 10 K	40K ± 10 K

Calibration Tolerance: 0.002% at 26°C

Temperature Tolerance: 0.001% 0°C to 70°C

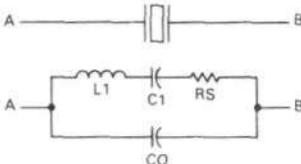
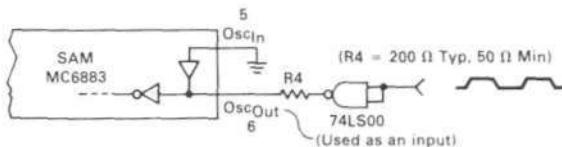


FIGURE 10 — TTL CLOCK INPUT



*Optimum values depend on characteristics of the crystal (X1). For many applications, VCik must be 3.579545 MHz \pm 50 Hz! Hence, Osc_{Out} must be made similarly "drift resistant" (by balancing temperature coefficients of X1, CV, CF, R1, R2 and R3).

**Specifically cut for MC6883 are International Crystal Manufacturing, Inc. Crystals (#167568 for 14.31818 MHz or #167569 for 16.0 MHz). However, other crystals may be used.

THEORY OF OPERATION

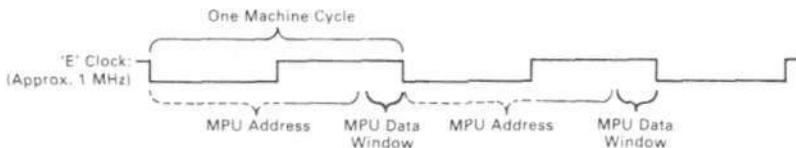
Video or No Video

Although the MC6883 may be used as a dynamic RAM controller **without** a video display*, most applications are likely to include a MC6847 video display generator (VDG). Therefore, this document emphasizes MC6883 with MC6847 systems.

Shared RAM (with interleaved DMA)

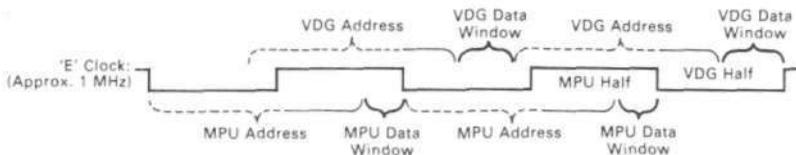
To minimize the number of RAM and interface chips, both the MPU and VDG share common dynamic RAM. Yet, the use of common RAM creates an apparent difficulty. That is, the MPU and VDG must both access the RAM without contention. This difficulty is overcome by taking advantage of the timing and architecture of Motorola MPU's (MC6800, MC6801E, MC6809E, MC68000). Specifically, **all** MPU accesses of external memory **always** occur in the **latter half** of the machine cycle, as shown below:

FIGURE 11 — MOTOROLA MPU TIMING



Similarly, the MC6847 (non-interlaced) VDG transfers a data byte in a half machine cycle (E or $\Phi 2$). Thus, when properly positioned, VDG and MPU RAM accesses interleave without contention as shown below:

FIGURE 12 — MOTOROLA MPU WITH VDG TIMING



This Interleaved Direct Memory Access (IDMA) is synchronized via the MC6883 by centering the VDG data window half-way between MPU data windows.**

The result is a shared RAM system without MPU/VDG RAM access contention, with both MPU and VDG running uninterrupted at normal operating speed, each transparent to the other.

RAM Refresh

Dynamic RAM refresh is accomplished by accessing eight*** sequential addresses every 64*** microseconds until 128 consecutive addresses have been accessed. To avoid RAM access contention between REFRESH and MPU, each of the 128 refresh accesses occupies the "VDG half" of the interleaved DMA (IDMA). Furthermore, refresh accesses occur only during the television retrace period (at which time the VDG doesn't need to access RAM).

In summary, the VDG, MPU and MC6883's Refresh Counter all transparently access the common dynamic RAM without contention or interruption.

Why IDMA?

Use of the interleaved direct memory access results in fast modification to variable portions of display RAM, by the MPU, without any distracting flashes on the screen (due to RAM access contention.) In addition, the MPU is not slowed down nor stopped by the MC6883; thereby, assuring accurate software timing loops without costly additional hardware timers. Furthermore, additional hardware and software to give "access permission" to the MPU is eliminated since the MPU may access RAM at **any** time.

* Only 1 pin, (DA0) out of 40 pins is dedicated to the video display.

** See VDG synchronization (page 10) for more detail.

*** When not using a MC6847, HS may be wired low for continuous transparent refresh.

"Systems On Silicon" Concept

Total Timing

For most applications, the SAM can supply complete system timing from its on-chip precision 14.31818 MHz oscillator. This includes buffered MPU clocks (E and Q), VDG clock, color subcarrier (3.58 MHz), row address select (RAS), column address select (CAS) and write enable (WE).

Total Address Decode

For most applications, the SAM plus a "1 of 8 decoder" chip completely decodes I/O, ROM and RAM chip selects without wasting memory address space and without needlessly chopping-up contiguous address space. Chip selects are positioned in address space to allow three types of memory (RAM, local ROM and cartridge ROM) independent room for growth. For example, RAM may grow from address \$0000-up, cartridge ROM may grow from address \$FEFF-down and local ROM may grow from \$FBFF-down. Alternately, if the application requires minimum ROM and maximum contiguous RAM, a second choice of two memory maps places RAM from \$0000 to \$FEFF. (See pages 17 and 18.)

In both memory maps all I/O, MPU vectors, SAM control registers, and some reserved address spaces are efficiently contained between addresses \$FF00 and \$FFFF.

How Much RAM?

Using nine SAM pins (Z0 - Z7 and RAS0) the following combinations require no additional address logic.

FIGURE 13 — RAM CONFIGURATIONS

Address:		Chip Select:	
MSB	LSB		
Z5Z4Z3Z2Z1Z0	RAS0	} One or two banks of 4K x 8 (like MCM4027's)
Z5Z4Z3Z2Z1Z0	RAS1 (= Z7)	
Z6Z5Z4Z3Z2Z1Z0	RAS0	} One or two banks of 16K x 8 (like MCM4116's)
Z6Z5Z4Z3Z2Z1Z0	RAS1 (= Z7)	
Z7Z6Z5Z4Z3Z2Z1Z0	RAS0	One bank of 64K x 8 (like MCM6665's)

PROGRAMMING GUIDE

SAM — Programmability

The SAM contains a 16-bit control register which allows the MC6809E to program the SAM for the following options:

VDG Addressing Mode	3-bits
VDG Address Offset	7-bits
32K Page Switch	1-bit
MPU Rate	2-bits
Memory Size	2-bits
Map Type	1-bit

Note that when the SAM is **reset** by first applying power or by manual hardware reset,† all control register bits are **cleared** (to a logic "0").

VDG Addressing Mode

Three bits (V2, V1, V0) control the sequence of DISPLAY ADDRESSES generated by the SAM (which are used to scan dynamic RAM for video information). For example, if you wish to display Dynamic RAM data as INTERNAL ALPHANUMERIC VIDEO, you should program‡ the MC6847 for the INTERNAL ALPHANUMERIC MODE and CLEAR BITS V2, V1 and V0 in the SAM. The table on the following page summarizes the available modes:

† See Figure 7 for manual reset circuit.

‡ Typically, part of a PIA (MC6821) at location \$FF22 is used to control MC6847 modes. (See MC6847 Data Sheet.)

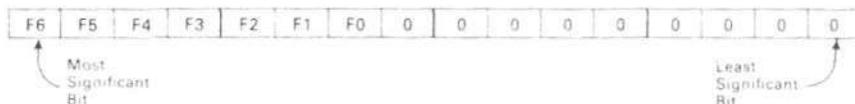
Mode Type	MC6847 Mode					SAM Mode		
	G/A	GM2	GM1	GM0 EXT/I	CSS	V2	V1	V0
Internal Alphanumerics	0	X	X	0	X	0	0	0
External Alphanumerics	0	X	X	1	X	0	0	0
OSemigraphics — 4	0	X	X	0	X	0	0	0
Semigraphics — 6	0	X	X	1	X	0	0	0
Semigraphics — 8*	0	X	X	0	X	0	1	0
Semigraphics — 12*	0	X	X	0	X	1	0	0
Semigraphics — 24*	0	X	X	0	X	1	1	0
Full Graphics — 1C	1	0	0	0	X	0	0	1
Full Graphics — 1R	1	0	0	1	X	0	0	1
Full Graphics — 2C	1	0	1	0	X	0	1	0
Full Graphics — 2R	1	0	1	1	X	0	1	1
Full Graphics — 3C	1	1	0	0	X	1	0	0
Full Graphics — 3R	1	1	0	1	X	1	0	1
Full Graphics — 6C	1	1	1	0	X	1	1	0
Full Graphics — 6R	1	1	1	1	X	1	1	0
Direct Memory Access*	X	X	X	X	X	1	1	1

*SR 512 & 524 modes are not described in the MC6847 Data Sheet. See appendix A.

*DMA is identical to 6R except as shown in Figure 5 on page 9.

VDG Address Offset

Seven bits (F6, F5, F4, F3, F2, F1 and F0) determine the **Starting Address** for the video display. The "Starting Address" is defined as "the address corresponding to data displayed in the **Upper Left** corner of the TV screen". The "Starting Address" is shown below in binary:



Note that the "Starting Address" may be placed anywhere within the 64K address space with a resolution of 1/2K (the size of one alphanumeric page).

The F6-F0 bits take effect during the TV vertical synchronization pulse (i.e., when \overline{FS} from MC6847 is low).

Page Switch

One bit (P1) is used "in place of" A15 from the MC6809E in order to refer access within \$0000-\$7FFF to one of two 32K byte **pages** of RAM. If the system does **not** use more than 32K bytes of RAM, P1 can be ignored.**

**When using 4K x 1 RAMS, two banks of eight IC's are allowed. This accounts for Addresses \$0000-1FFF. Also, this same RAM can be addressed at \$2000-\$3FFF, \$4000-\$5FFF and \$6000-\$7FFF.

MPU Rate

Two bits (R1, R0) control the clock rate to the MC6809E MPU. The options are:

RATE (FREQUENCY OF "E" CLOCK)	R1	R0
0.9 MHz (Crystal Frequency ÷ 16) Slow	0	0
0.9/1.8 MHz (Address Dependent Rate)	0	1
1.8 MHz (Crystal Frequency ÷ 8) Fast	1	X

(Typical Crystal Frequency = 14.31818 MHz)

In the "address dependent rate" mode, accesses to \$0000-\$7FFF and \$FF00-\$FF1F are slowed to 0.9 MHz (crystal frequency ÷ 16) and all other addresses are accessed at 1.8 MHz (crystal frequency ÷ 8.)

Memory Size

Two bits (M1 and M0) determine RAM memory size. The options are:

SIZE	M1	M0
One or two banks of 4K × 1 dynamic RAMs	0	0
One or two banks of 16K × 1 dynamic RAMs	0	1
One bank of 64K × 1 dynamic RAMs	1	0
Up to 64K static RAM*	1	1

*Requires a latch for demultiplexing the RAM address.

IMPORTANT!

Note: Be sure to program the SAM for the correct memory size **before** using RAM (i.e., for a subroutine stack).

Map Type

One bit (TY) is used to select between two memory map configurations.

Refer to pages 17, 18 and 19 for details. When using Map Type "TY = 1", only the "Slow" MPU rate may be used. Future versions of the SAM may allow use of all rates.

Writing To The SAM Control Register

Any bit in the control register (CR) may be set by writing to a specific unique address. Each bit has two unique addresses . . . writing to the **even #** address **clears** the bit and writing to the **odd #** address **sets** the bit. (Data on the data bus is irrelevant in this procedure.) The specific addresses are tabulated on pages 17 and 18.

If desired, a short routine may be written to program the SAM CR "a word at a time". For example, the following routine copies "B" bits from "A" register to SAM CR addresses beginning with address "X".

SAM1	46	ROR	A
	24	06	BCC SAM2
	30	01	INX (LEAX1,X)
	A7	80	STA 0,X
SAM2	20	02	BRA SAM3
	A7	81	STA 0,X
SAM3	5A	F2	DEC B
	26	F2	BNE SAM1
	39		RTS

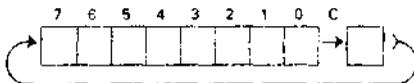


FIGURE 15 — MEMORY MAP (TYPE #1)

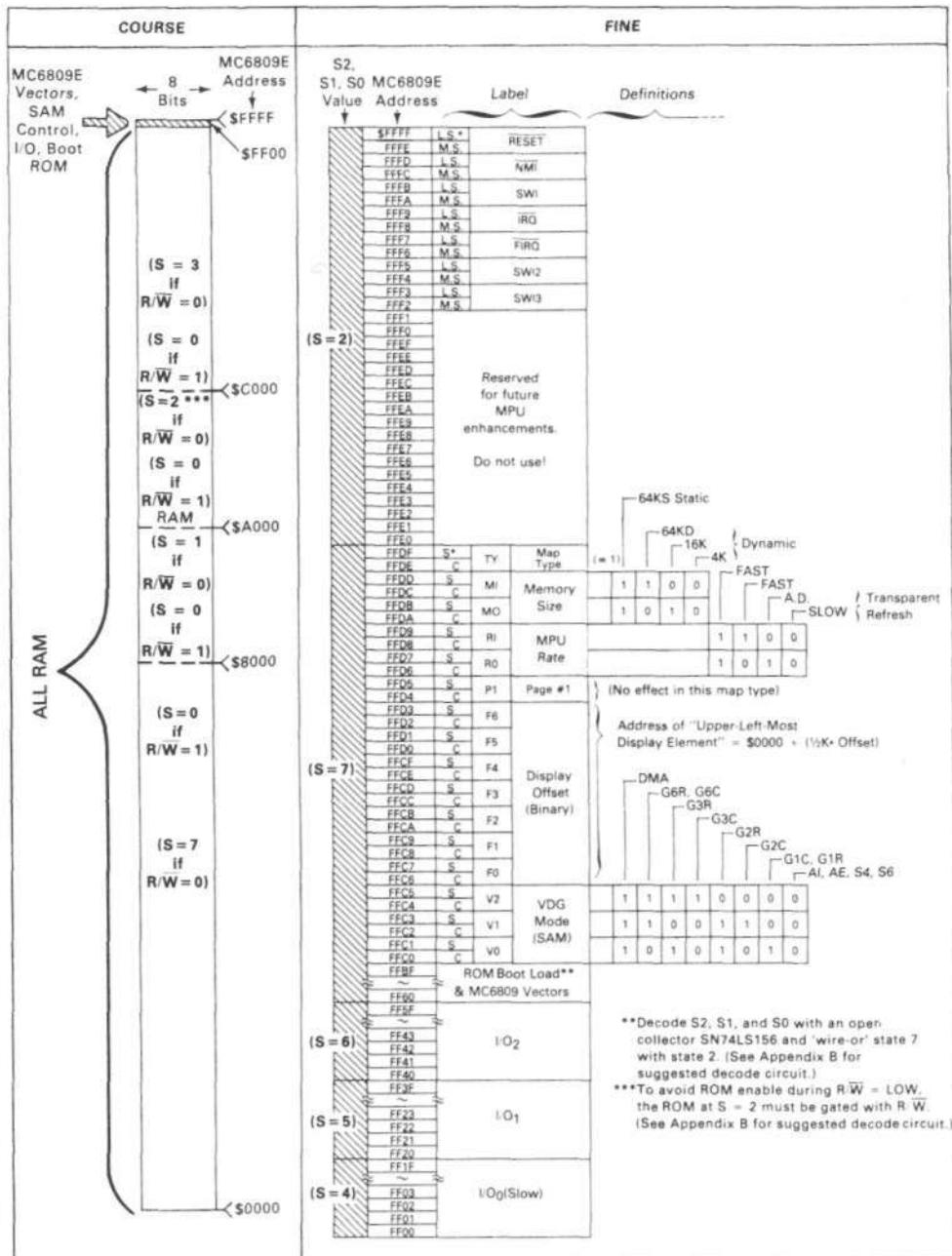


FIGURE 16 — MEMORY ALLOCATION TABLE
(Also, see the memory MAPs on pages 17 and 18.)

Type # 0: (Primarily for ROM based systems)

Address Range	$S = 4(S2) + 2$ (S1) + S0 S Value	Intended Use
\$FFF2 to FFFF	2	MC6809E Vectors: Reset, NMI, SWI, IRQ, FIRQ, SWI2, SWI3.
FFE0 to FFF1	2	Reserved for future MPU enhancements.
FFC0 to FFDF	7	SAM Control Register: V0, V2, F0 - F6, P, R0, R1, M0, M1, TY.
FF60 to FFBF	7	Reserved for future control register enhancements.
FF40 to FF5F	6	I/O ₂ : Input Output (PIAs, ACIAs, etc.) To subdivide, use A0 - A4.
FF20 to FF3F	5	I/O ₁ : Input Output (PIAs, ACIAs, etc.) To subdivide, use A0 - A4.
FF00 to FF1F	4	I/O ₀ : Input Output (PIAs, ACIAs, etc.) To subdivide, use A0 - A4.
C000 to FEFF	3	ROM2: 16K addresses. External cartridge ROM*.
A000 to BFFF	2	ROM1: 8K addresses. Internal ROM*. Note that MC6809E vector addresses select this ROM*.
8000 to 9FFF	1	ROM0: 8K addresses. Internal ROM*.
0000 to 7FFF	0 if $R\bar{W} = 1$ 7 if $R\bar{W} = 0$	RAM: 32K addresses. RAM shared by MPU and VDG.

*Not restricted to ROM. For example, RAM or I/O may be used here.

Type # 1: (Primarily for RAM based systems)

Address Range	$S = 4(S2) + 2$ (S1) + S0 S Value	Intended Use
\$FFF2 to FFFF	2	MC6809E Vectors: Reset, NMI, SWI, IRQ, FIRQ, SWI2, SWI3.
FFE0 to FFF1	2	Reserved for future MPU enhancements.
FFC0 to FFDF	7	SAM Control Register: V0 - V2, F0 - F6, P, R0, R1, M0, M1, TY.
FF60 to FFBF	7	Small ROM: Boot load program and initial MC6809 vectors.
FF40 to FF5F	6	I/O ₂ : Input Output (PIAs, ACIAs, etc.) To subdivide, use A0-A4.
FF20 to FF3F	5	I/O ₁ : Input Output (PIAs, ACIAs, etc.) To subdivide, use A0 - A4.
FF00 to FF1F	4	I/O ₀ : Input Output (PIAs, ACIAs, etc.) To subdivide, use A2 - A4.
0000 to FEFF	0 if $R\bar{W} = 1$	RAM: 64K(1 - 256) addresses, shared by MPU and VDG. (If $R\bar{W} = 0$ then $S = 3$ for \$C000-\$FEFF; $S = 2$ for \$A000-\$BFFF; $S = 1$ for \$8000-\$9FFF and $S = 7$ for \$0000-\$7FFF.)

APPENDIX A

VDG/SAM Video Display System Offers 3 New Modes

by
Paul Fletcher

There are three new modes created when the VDG and SAM are used together in a video display system. These modes offer alphanumeric compatibility with 8 color low-to-high resolution graphics, 64H*64V, 64H*96V, 64H*192V. The new modes S8, S12, and S24 are created by placing the VDG in the Alpha Internal mode and having the SAM in a 2K, 3K or 6K full color graphics mode. In all modes the VDG's S/A and Inv. pins are connected to data bits DD7 and DD6 to allow switching on the fly between Alpha and Semigraphics and between inverted and non-inverted alpha. This method is used in most VDG systems to obtain maximum flexibility.

The three modes divide the standard 8*12 dot box used by the VDG for the standard alpha and semigraphics modes into eight 4*3 dot boxes for the S8 mode, twelve 4*2 dot boxes for the S12 mode, and twenty-four 4*1 dot boxes for the S24 mode. Figure 17 shows the arrangement of these boxes. One byte is needed to control two horizontally consecutive boxes. It therefore takes four bytes for the S8, six bytes for the S12, and 12 bytes for the S24 mode to control the entire 8*12 dot box. These two horizontally consecutive boxes have four combinations of luminance controlled by bits B0 - B3. For conven-

ience B2 should be made equal to B0 and B3 should be made equal to B1. This eliminates a screen placement problem which would cause other codes to change patterns when moved vertically on the screen. The illuminated boxes can be one of eight colors which are controlled by B4 - B6 (see Figure 18). The bytes needed to control all the boxes in the 8*12 dot box must be spaced 32 address spaces apart in the display RAM because of the addressing scheme originally used in the VDG and duplicated by the SAM. This means to place an alphanumeric character on the TV screen it requires 4, 6, or 12 bytes depending on the mode used. These bytes are placed 32 memory locations apart in the display RAM (see Figure 18). This multiple byte format allows the mixing of character rows of different characters in the same 8*12 dot box creating new characters and symbols. It also allows overlining and underlining in eight colors by switching to semigraphics at the correct time.

These new modes optimize the memory versus screen density tradeoffs for RF performance on color TVs. This could make them the most versatile of all the modes depending on the users creativity and the software sophistication.

APPENDIX B

Memory Decode for "MAP TYPE = 1"

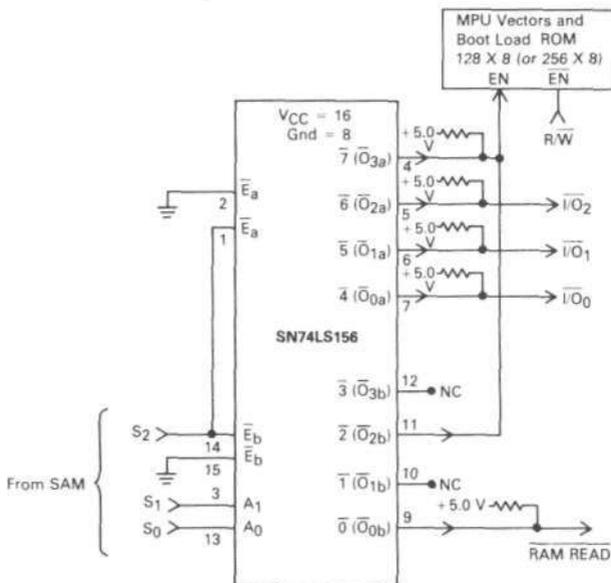
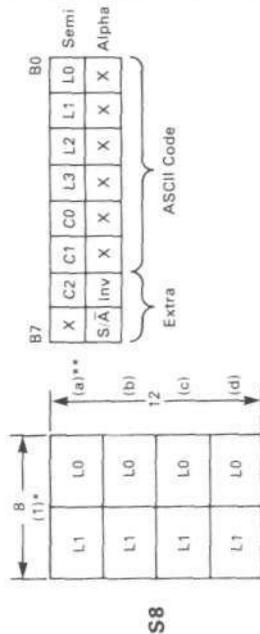


FIGURE 18 — S8 DISPLAY FORMAT EXAMPLES



LX	C2	C1	C0	Color
0	X	X	X	Black
1	0	0	0	Green
1	0	0	1	Yellow
1	0	1	0	Blue
1	0	1	1	Red
1	1	0	0	Buff
1	1	0	1	Cyan
1	1	1	0	Magenta
1	1	1	1	Orange

B3,B1	B2,B0	4
0	0	Off
0	1	Color
1	0	Color
1	1	Color

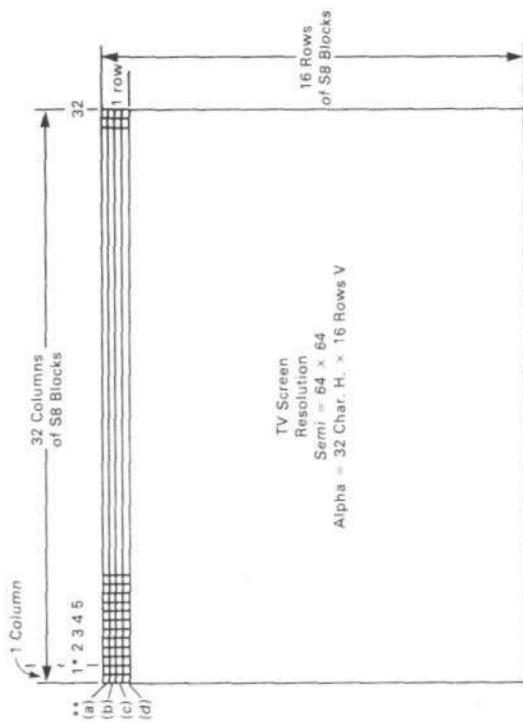
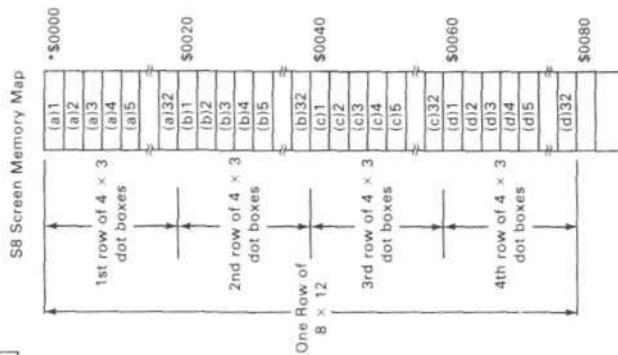


FIGURE 20 — EQUIVALENT OF OSCILLATOR INPUT AND OUTPUT

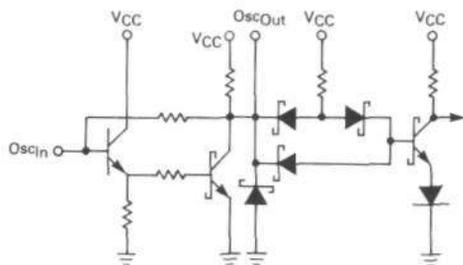


FIGURE 21 — DA0 INPUT

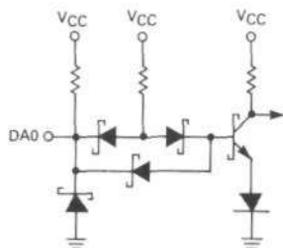


FIGURE 22 — VCik INPUT/OUTPUT

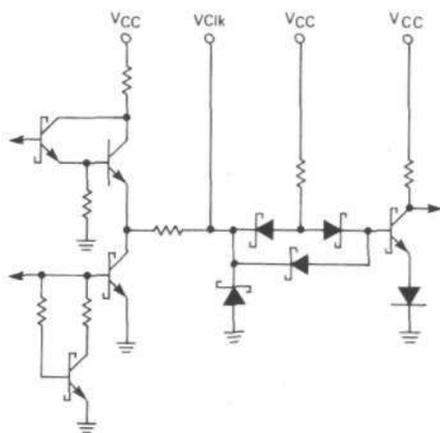


FIGURE 23 — E AND Q OUTPUTS

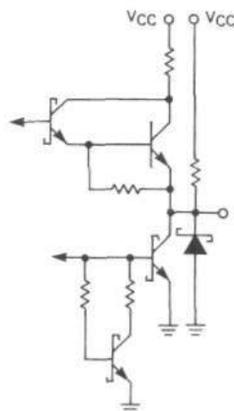


FIGURE 24 — TYPICAL INPUT

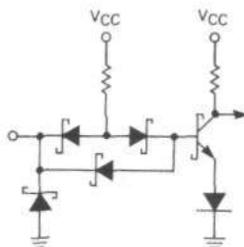


FIGURE 25 — TYPICAL OUTPUT

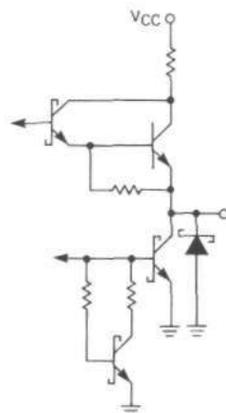
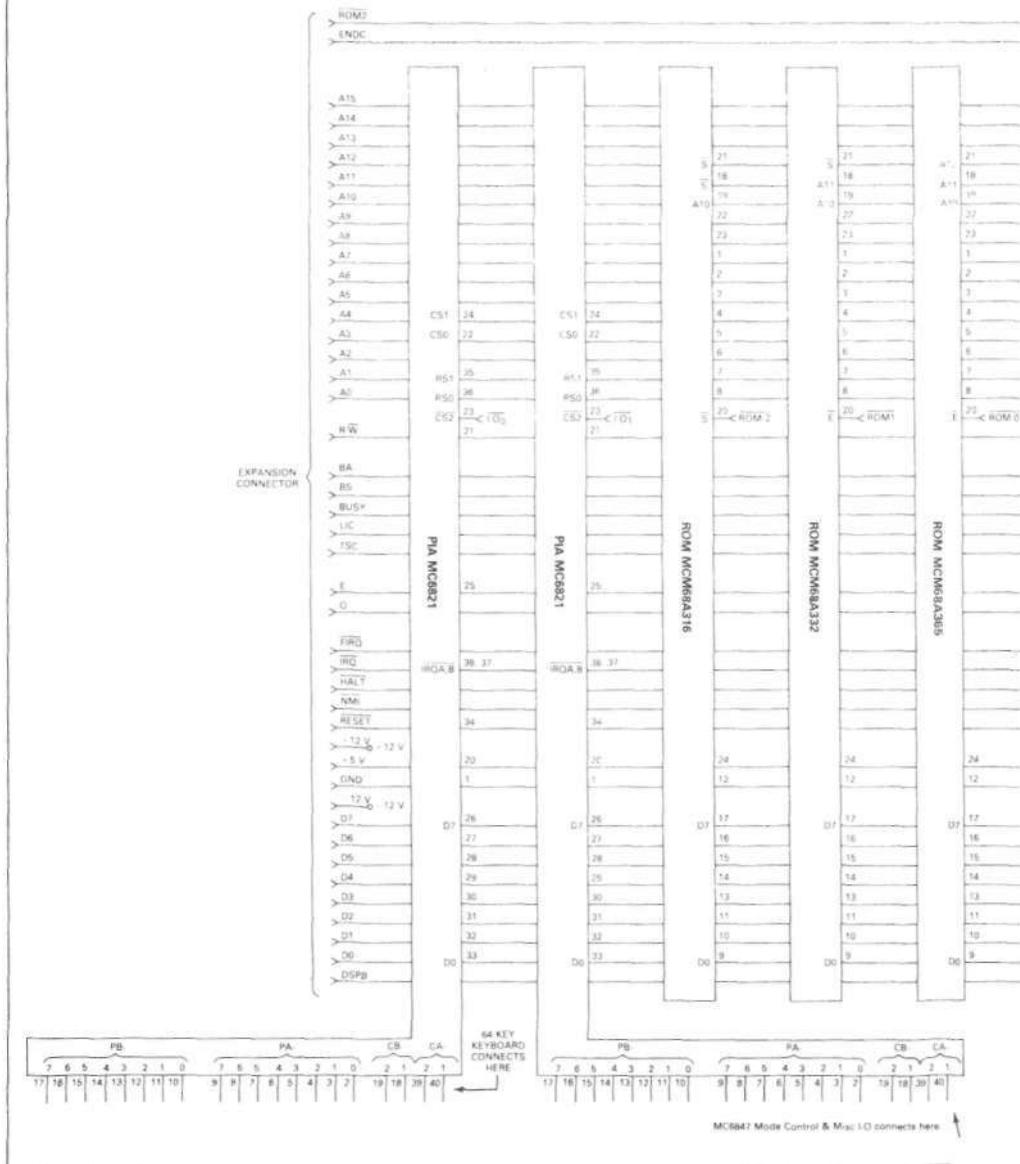
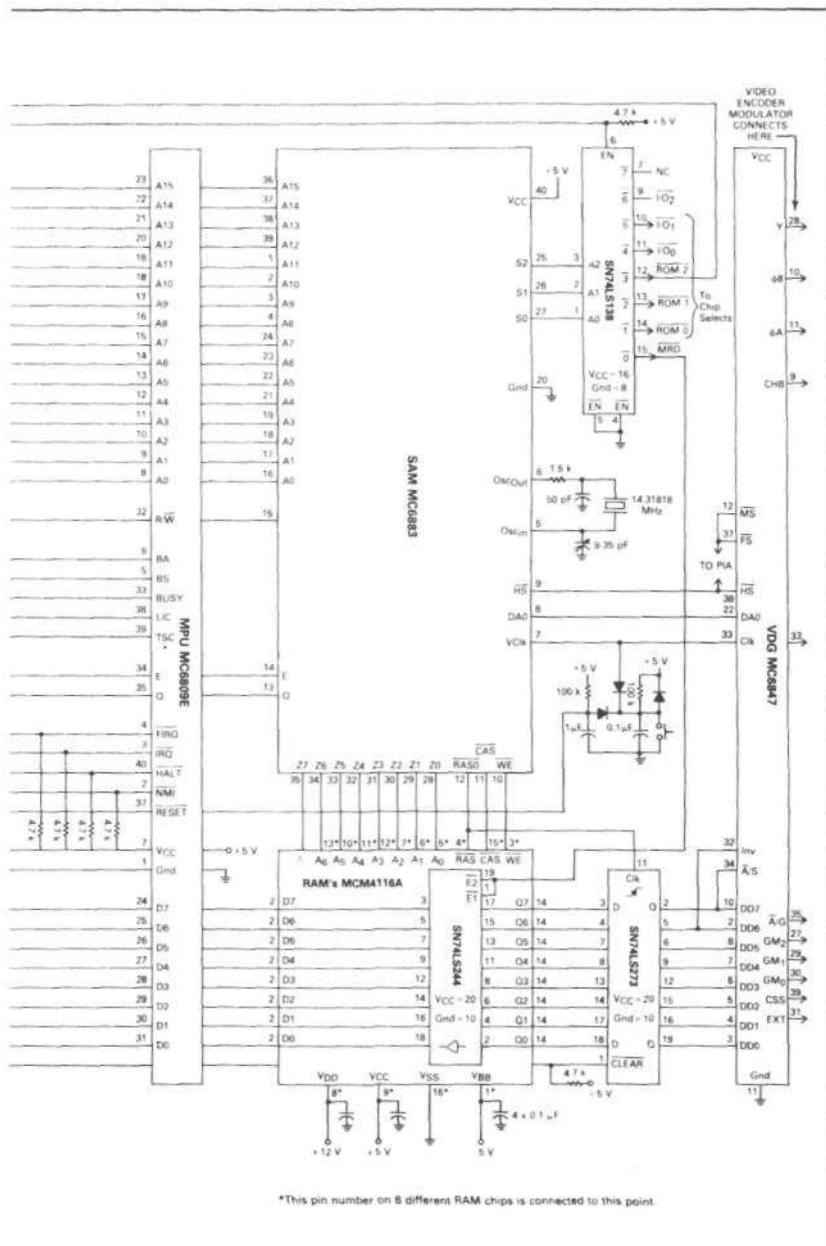


FIGURE 19 — EXAMPLE of MC6809E, MC6883 and MC6847 COMPUTER





Appendix 3

MC6847 data sheet

Supplied courtesy of Motorola Semiconductors.

The information here has been carefully checked and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies. Motorola reserves the right to make changes to any products herein to improve reliability, function or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein. No licence is conveyed under patent rights in any form. When this document contains information on a new product, specifications herein are subject to change without notice.



MOTOROLA Semiconductors

Colvilles Road, Kelvin Estate - East Kilbride/Glasgow - SCOTLAND

Advance Information

VIDEO DISPLAY GENERATOR (VDG)

The Motorola MC6847 Video Display Generator (VDG) provides a means of interfacing the Motorola M6800 microprocessor family (or similar products) to a commercially available color or black and white television receiver. Applications of the VDG include video games, bioengineering displays, education, communications and any place graphics are required.

The VDG reads data from memory and produces a composite video signal which will allow the generation of alphanumeric or graphic displays. The generated composite video may be up modulated to either Channel 3 or 4 by using the compatible MC1372 (TV Chroma and Video modulator). The up modulated signal is suitable for application to the antenna of a color TV. A typical TV game is indicated in Figure 1.

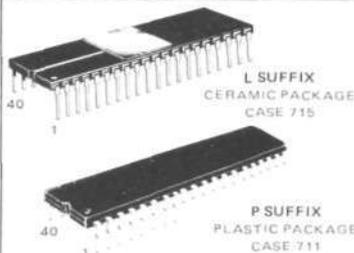
- Generates four different alphanumeric display modes and eight graphic display modes
- Compatible with the M6800 family
- Compatible with the MC1372 modulator
- The alphanumeric modes display 32 characters per line by 16 lines
- An internal multiplexer allows the use of either the internal ROM or an external character generator
- An external character generator can be used to extend the internal character set for "limited graphic" shapes
- A Mask Programmable internal character generator ROM is available on special order (Appendix A)
- One display mode offers 8-color 64 x 32 density graphics in an alphanumeric display mode
- One display mode offers 4-color 64 x 48 density graphics in an alphanumeric display mode
- All alphanumeric modes have a selectable video inverse
- Generates full video signal
- Generates R-Y and B-Y signals for external color modulator
- Full-graphic modes offer 64 x 64, 128 x 64, 128 x 96, 128 x 192, or 256 x 192 densities
- Full-graphic modes allow 2-color or 4-color data structures
- Full-graphic modes use one of two 4-color sets or one of two 2-color sets
- Available in either an interlace mode (NTSC Standard) or a non-interlace mode

MC6847
NONINTERLACE
MC6847Y
INTERLACE

MOS

(N-CHANNEL, SILICON GATE)

VIDEO
DISPLAY
GENERATOR



PIN ASSIGNMENT

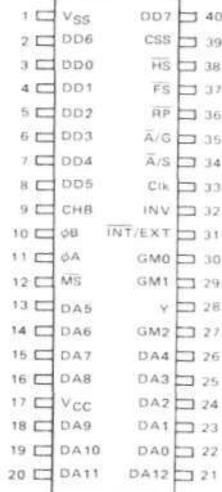
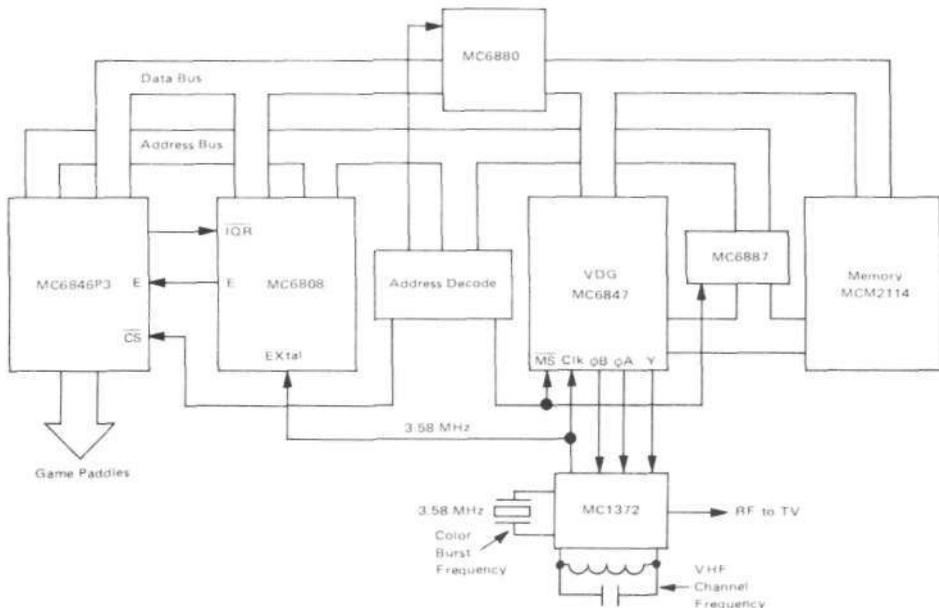


FIGURE 1 – BLOCK DIAGRAM OF USE OF THE VDG IN A TV GAME



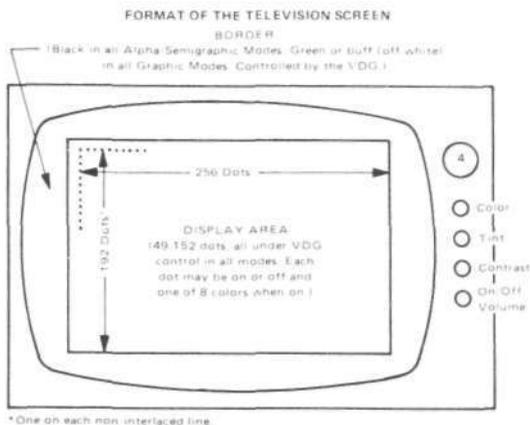
Mnemonic	Pin Numbers
VCC	17
VSS	1
CLK	33
DA0-DA12	22, 23, 24, 25, 26 13, 14, 15, 16, 18, 19, 20, 21
DD0-DD5	3, 4, 5, 6, 7, 8
DD6, DD7	2, 40
φA, φB, Y	11, 10, 28
CHB	9
RP	36
HS	38
INV	32
INT/EXT	31
A/S	34
MS	12
A/G	35
FS	37
CSS	39
GM0-GM2	30, 29, 27

Function
+5V
Ground
Color burst clock 3.579545 MHz (input)
Address lines to display memory, high impedance during memory select (MS)
Data from display memory RAM or ROM
Data from display memory in graphic mode; data also in alpha external mode; color data in alpha semigraphic 4 or 6
Chrominance and luminance analog (R-Y, B-Y, Y) output to RF modulator (MC1372)
Chroma bias; reference φA and φB levels
Row preset – Output to provide timing for external character generator.
Horizontal Sync – Output to provide timing for external character generator.
Inverts video in all alpha modes
Switches to external ROM in alpha mode and between SEMIG-4 and SEMIG-6 in semigraphics
Alpha/Semigraphics; selects between alpha and semigraphics in alpha mode
Memory select forces VDG address buffers to high-impedance state
Switches between alpha and graphic modes
Field Synchronization goes low at bottom of active display area.
Color set select; selects between two alpha display colors or between two color sets in semigraphics 6 and full graphics
Graphic mode select; select one of eight graphic modes.

ELECTRICAL SPECIFICATIONS

ABSOLUTE MAXIMUM RATINGS

Rating	Value
Supply Voltage (V_{CC})	-0.3 to +7.0V
Input Voltage any Pin	-0.3 to +7.0V
Operating Temperature	0°C to 70°C
Storage Temperature	-65°C to 150°C
Power Dissipation	TBD

DC (STATIC) CHARACTERISTICS - ($V_{CC} = 5.0V \pm 5\%$, $V_{SS} = 0.0V$, $T_A = 0^\circ\text{C}$ to 70°C unless otherwise noted)

Characteristic	Symbol	Min	Typ.	Max.	Unit
Input High Voltage Cik Other Inputs	V_{IH}	$V_{SS} + 2.4$ $V_{SS} + 2.0$	-	V_{CC} V_{CC}	Vdc
Input Low Voltage Cik Other Inputs	V_{IL}	$V_{SS} - 0.3$ $V_{SS} - 0.3$	-	$V_{SS} + 0.4$ $V_{SS} + 0.8$	Vdc
Input Leakage Current CLK, GM0-GM2, INV, INT/EXT, MS, V_{SS} , DD0-DD7, A/S, A/G	I_{in}	-	-	2.5	μA dc
Three-State (Off State) Input Current DA0-DA12	I_{LO}	-	-	10	μA dc
Output High Voltage ($C_{Load} = 30\text{ pF}$, $I_{Load} = -100\text{ }\mu\text{A}$)	V_{OH}	2.4	-	-	Vdc
Output High Voltage ($C_{Load} = 55\text{ pF}$, $I_{Load} = -100\text{ }\mu\text{A}$)	V_{OH}	2.4	-	-	Vdc
Output Load Voltage ($C_{Load} = 30\text{ pF}$, $I_{Load} = 1.6\text{ mA}$)	V_{OL}	-	-	$V_{SS} + 0.4$	Vdc
Output Low Voltage ($C_{Load} = 55\text{ pF}$, $I_{Load} = 1.6\text{ mA}$)	V_{OL}	-	-	$V_{SS} + 0.4$	Vdc
Output High Current (Sourcing) ($V_{OH} = 2.4\text{ V}$)	I_{OH}	-100	-	-	μA dc
Output Low Current (Sinking) ($V_{OL} = 0.4\text{ Vdc}$)	I_{OL}	1.6	-	-	mAdc
Input Capacitance ($V_{in} = 0$, $T_A = 25^\circ\text{C}$, $f = 1.0\text{ MHz}$)	C_{in}	-	-	7.5	pF
Chroma Bias Voltage ($C_{Load} = 20\text{ pF}$, R Load = 200 k ohm, $V_{CC} = 4.75 - 5.25\text{ V}$)	V_R	-	$0.3 V_{CC}$	-	Vdc

DC (STATIC) CHARACTERISTICS — ($V_{CC} = 5.0 \text{ V} \pm 5\%$, $V_{SS} = 0.0 \text{ V}$, $T_A = 0^\circ\text{C}$ to 70°C unless otherwise noted)

Characteristic	Symbol	Min	Typ.	Max.	Unit
Chroma ϕ A Voltage (C _{Load} = 20 pF, R _{Load} = 200 k ohm)	V _{CϕA} V _{H1} V _O V _{LO}	—	V _R + 0.1 V _{CC} V _R V _R - 0.1 V _{CC}	—	Vdc
Chroma ϕ B Voltage (C _{Load} = 20 pF, R _{Load} = 200 k ohm)	V _{CϕB}	—	V _R + 0.1 V _{CC} V _R V _R - 0.05 V _{CC} V _R - 0.1 V _{CC}	—	Vdc
Luminance Y Voltage (C _{Load} = 20 pF, R _{Load} = 200 k ohm)	V _Y V _S V _{BLANK} V _{BLACK}	—	0.2 V _{CC} 0.75 V _S 0.7 V _S	—	Vdc
Voltage White Low (Voltage White Medium) (Voltage White High)	V _{WL} V _{WM} V _{WH}	—	0.62 V _S 0.5 V _S 0.38 V _S	—	Vdc

AC (Dynamic) CHARACTERISTICS — ($V_{CC} = 5.0 \text{ V} \pm 5\%$, $T_A = 0^\circ\text{C}$ to 70°C)

Characteristic	Symbol	Min	Typ.	Max.	Unit
Clk Frequency Clk Duty Cycle	f Clk _{dc}	3.579535	3.579545	3.579555	MHz
Chroma Phase Delay (measured with respect to "Y" output)					ns
ϕ A ϕ B	t _{YA} t _{YB}	—	200 200	—	
Luminance Rise Time Luminance Fall Time	t _{ry} t _{fy}	—	60 50	—	ns
Chroma Rise and Fall Times (ϕ A Rise Time) (ϕ A Fall Time) (ϕ B Rise Time) (ϕ B Fall Time)	t _{rϕA} t _{fϕA} t _{rϕB} t _{fϕB}	—	60 60 60 60	—	ns
Field Sync. (FS) (Pulse Width)	t _{WFS}	—	2.03	—	ms
Row Present (RP) (Pulse Width) (Delay From HS)	t _{WRP} t _{HSRP}	—	0.98 0.98	—	μ s μ s
Horizontal Sync (HS)	t _{WHS}	—	4.9	—	μ s

FIGURE 2 – VIDEO AND CHROMINANCE RELATIONSHIPS OUTPUT WAVEFORM

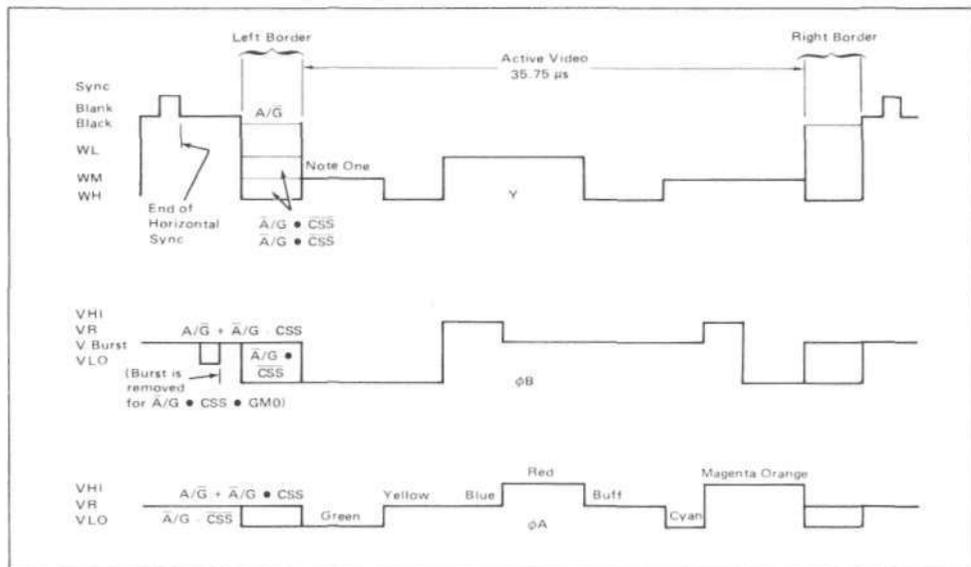
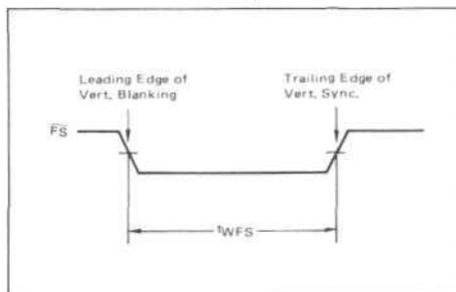
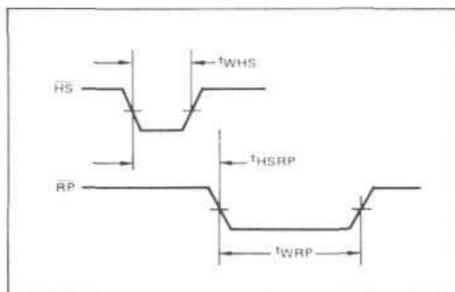


FIGURE 3 – TIMING DIAGRAMS

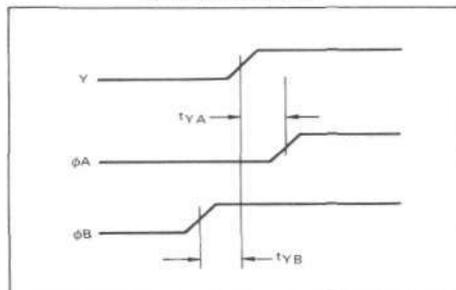
A – Field Sync



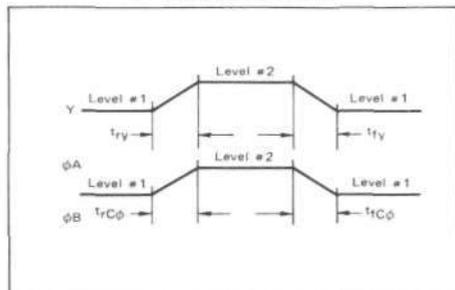
B – Row Preset



C – Chroma Phase Delay



D – Video Rise & Fall Times



VDG SIGNAL DESCRIPTION

Address Output Lines (DA0-DA12) — Thirteen address lines are used by the VDG to scan the display memory. The starting address of the display memory is located at the upper left corner of the display screen. As the television sweeps from the left to right and top to bottom, the VDG increments the RAM display address. These lines are TTL compatible and may be forced into a high impedance state whenever the \overline{MS} pin goes low.

Data Inputs (DD0-DD7) — Eight TTL compatible data lines are used to input data from RAM to be processed by the VDG. The data is interpreted and transformed into luminance Y (Pin 28) and color outputs ϕA and ϕB (Pin 11 and Pin 10).

Power Inputs — V_{CC} requires +5 volts. V_{SS} requires zero volts and is normally ground. The tolerance and current requirements of the VDG are specified in the Electrical Characteristics.

Video Outputs (ϕA , ϕB , Y , CHB) — These four analog outputs are used to transfer luminance and color information to a standard NTSC color television receiver, either via the MC1372 RF modulator or directly into Y , ϕA , ϕB television video inputs.

LUMINANCE (Y) — This six level analog output contains composite sync., blanking and four levels of video luminance.

ϕA — This three level analog output is used in combination with ϕB and Y outputs to specify one of eight colors.

ϕB — This four level analog output is used in combination with ϕA and Y outputs to specify one of eight colors. Additionally, one analog level is used to specify the time of the color burst reference signal.

CHROMA BIAS (CHB) — This pin is an analog output and provides a D.C. reference corresponding to the quiescent value of ϕA and ϕB . CHB is used to guarantee good thermal tracking and minimize the variation between the parts.

Synchronizing Inputs (\overline{MS} , CLK)

Three-State Control — (\overline{MS}) is a TTL compatible input which, when low, forces the VDG address lines into a high impedance state. This may be done to allow other devices (such as an MPU) to address the display memory (RAM).

Clock (CLK) — The VDG clock input (CLK) requires a 3.579545 MHz (standard) TV crystal frequency square wave. The duty cycle of this clock must be between 45

and 55 percent since it controls the width of alternate dots on the television screen. The MC1372 RF modulator may be used to supply the 3.579545 MHz clock and has provisions for a duty cycle adjustment.

Synchronizing Outputs (\overline{FS} , \overline{HS} , \overline{RP}) — Three TTL compatible outputs provide circuits, exterior to the VDG, with timing references to the following internal VDG states:

FIELD SYNC — (\overline{FS}) — The high to low transition of the \overline{FS} output coincides with the end of active display area. During this time interval an MPU may have total access to the display RAM without causing undesired flicker on the screen. The Low to High transition of \overline{FS} coincides with the trailing edge of the vertical synchronization pulse.

HORIZONTAL SYNC — (\overline{HS}) — The \overline{HS} pulse is in coincident with the horizontal synchronization pulse furnished to the television receiver by the VDG. The high to low transition of the \overline{HS} output coincides with the leading edge of the horizontal synchronization pulse.

ROW PRESET — (\overline{RP}) — If desired, an external character generator ROM may be used with the VDG. However, an external four bit counter must be added to supply row selection. The counter is clocked by the \overline{HS} signal and cleared by the \overline{RP} signal.

Mode Control Lines (Input) ($\overline{A/G}$, $\overline{A/S}$, $\overline{INT/EXT}$, $GM0$, $GM1$, $GM2$, CSS , INV) — Eight TTL compatible inputs are used to control the operating mode of the VDG. $\overline{A/S}$, $\overline{INT/EXT}$, CSS and INV may be changed on a character by character basis. The CSS pin is used to select between two possible alphanumeric colors; when the VDG is in the alphanumeric mode and between two color sets when the VDG is in the semigraphics 6 and full Graphic mode. Table 1 illustrates the various modes that can be obtained using the mode control lines.

DISPLAY MODES

The VDG is capable of generating 12 distinct display modes (refer to Table 1). The color set selection and invert pins will allow variations on certain modes. The VDG will display two alphanumeric modes with two compatible semigraphic modes or display one of eight full graphic modes. A detailed description of the various modes of operation follows. A summary of major modes can be found in Table 2.

ALPHANUMERIC DISPLAY MODES — All alphanumeric modes occupy an 8 x 12 dot character matrix box and there are 32 x 16 character boxes per TV frame. Each horizontal dot (dot-clock) corresponds to one-half the period duration of the 3.58 MHz clock and each vertical dot is one scan line. One of two colors for the lighted dots may be selected by the color set select pin. An internal ROM will generate 64 ASCII display characters in a standard 5 x 7 box. Six bits of the eight-bit data word are used for the ASCII character generator and the two bits not used can be used to implement inverse video or color switching on a character by character basis. A 512 word display memory is required for this class of display.

The ALPHA SEMIGraphics -4 mode translates bits zero through three into a 4 x 6 dot element in the standard 8 x 12 dot box. Three data bits may be used to select one of eight colors for the entire character box. The extra bit is available to implement mode switching on the fly. A 512 word display memory is required. A density of 64 x 32 elements is available in the display area. The element area is four dot-clocks wide by six lines high.

The ALPHA SEMIGraphic -6 mode maps six 4 x 4 dot elements into the standard 8 x 12 dot alphanumeric box, a screen density of 64 x 48 elements is available. Six bits are used to generate this map and two data bits may be used to select one of four colors in the display box. The element area is four dot-clocks wide by four lines high.

FULL GRAPHIC MODE — There are eight full graphic modes available from the VDG. These modes require 1K to 6K bytes of memory. The eight full-graphic modes include an outside color border in one of two colors depending upon the color set select pin (CSS). The CSS pin selects one of two sets of four colors in the four color graphic modes.

The 64 x 64 Color Graphics Mode — The 64 x 64 color graphics mode generates a display matrix of 64 elements wide by 64 elements high. Each element may be one of four colors. A 1K x 8 display memory is required. Each pictel equals four dot-clocks by three scan lines.

The 128 x 64 Graphics Mode — The 128 x 64 graphics mode generates a matrix 128 elements wide by 64 elements high. Each element may be either ON or OFF. However, the entire display may be one of two colors, selected by using the color set select pin. A 1K x 8 display memory is required. Each pictel equals two dot-clocks by three scan lines.

The 128 x 64 Color Graphics Mode — The 128 x 64 color graphics mode generates a display matrix 128 elements wide by 64 elements high. Each element may be one of four colors. A 2K x 8 display memory is required. Each pictel equals two dot-clocks by three scan lines.

The 128 x 96 Graphics Mode — The 128 x 96 graphics mode generates a display matrix 128 elements wide by 96 elements high. Each element may be either ON or OFF. However, the entire display may be one of two colors selected by using the color select pin. A 2K x 8 display memory is required. Each pictel equals two dot-clocks by two scan lines.

The 128 x 96 Color Graphics Mode — The 128 x 96 color graphics mode generates a display 128 elements wide by 96 elements high. Each element may be one of four colors. A 3K x 8 display memory is required. Each pictel equals two dot-clocks by two scan lines.

The 128 x 192 Graphics Mode — The 128 x 192 graphics mode generates a display matrix 128 elements wide by 192 elements high. Each element may be either ON or OFF, but the ON elements may be one of two colors selected with color set select pin. A 3K x 8 display memory is required. Each pictel equals two dot-clocks by one scan line.

The 128 x 192 Color Graphics Mode — The 128 x 192 color graphics mode generates a display 128 elements wide by 192 elements high. Each element may be one of four colors. A 6K x 8 display memory is required. A detailed description of the VDG modes is given in Table 3. Each pictel equals two dot-clocks by one scan line.

The 256 x 192 Graphics Mode — The 256 x 192 graphics mode generates a display 256 elements wide by 192 elements high. Each element may be either ON or OFF, but the ON element may be one of two colors selected with the color set select pin. A 6K x 8 display memory is required. Each pictel equals one dot-clock by one scan line.

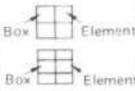
TABLE 1 – TABLE OF MODE CONTROL LINES (INPUTS)

\bar{A}/G	\bar{A}/S	\overline{INT}/EXT	INV	GM2	GM1	GM0	ALPHA/GRAPHIC MODE SELECT
0	0	0	0	X	X	X	Internal Alphanumerics
0	0	0	1	X	X	X	Internal Alphanumerics Inverted
0	0	1	0	X	X	X	External Alphanumerics
0	0	1	1	X	X	X	External Alphanumerics Inverted
0	1	0	X	X	X	X	Semigraphics - 4
0	1	1	X	X	X	X	Semigraphics - 6
1	X	X	X	0	0	0	64 x 64 Color Graphics
1	X	X	X	0	0	1	128 x 64 Graphics
1	X	X	X	0	1	0	128 x 64 Color Graphics
1	X	X	X	0	1	1	128 x 96 Graphics
1	X	X	X	1	0	0	128 x 96 Color Graphics
1	X	X	X	1	0	1	128 x 192 Graphics
1	X	X	X	1	1	0	128 x 192 Color Graphics
1	X	X	X	1	1	1	256 x 192 Graphics

TABLE 2 – SUMMARY OF MAJOR MODES

MAJOR MODE ONE

TABLE OF ALPHA MINOR MODES

Title	Memory	Colors	Display Elements
Alphanumeric (Internal)	512 x 8	2	
Alphanumeric (External)	512 x 8	2	
Alpha Semig-4	512 x 8	8	
Alpha Semig-6	512 x 8	4	

MAJOR MODE TWO

TABLE OF MINOR GRAPHICS MODES

Title	Memory	Colors	Comments
64 x 64 Color Graphic	1K x 8	4	Matrix 64 x 64 Elements
128 x 64 Graphics*	1K x 8	2	Matrix 128 elements wide by 64 elements high
128 x 64 Color Graphic	2K x 8	4	
128 x 96 Graphics*	1.5K x 8	2	Matrix 128 elements wide by 96 elements high
128 x 96 Color Graphic	3K x 8	4	
128 x 192 Graphics*	3K x 8	2	Matrix 128 elements wide by 192 elements high
128 x 192 Color Graphic	6K x 8	4	
256 x 192 Graphics*	6K x 8	2	Matrix 256 elements wide by 192 elements high

*Graphics mode turns on or off each element. The color may be one of two.

TABLE 3 — DETAILED DESCRIPTION OF VDG MODES

REF	VDG PINS					COLOR			TV SCREEN		VDG DATA BUS	COMMENTS	
	RT0	RT5	INT/EXT	GM2	GM1	GM0	CSB	INV	Character Color	Background			Display Mode
1	0	0	0	X	X	X	1	0	Green	Black	32 Characters across	8 bits 12 bits	The ALPHANUMERIC INTERNAL mode uses 32 vertical characters generated by a MADPDS TUWKVZ(1) - SP - 818B (10" x 6.5" 2468 BB) connected to the mode pins (A, G, A3, INT/EXT, GM2, GM1, GM0, CSB or INV).
1	0	0	1	X	X	X	0	0	Black	Black	32 Characters across	7 bits 12 bits	The ALPHANUMERIC EXTERNAL mode uses an external character generator as described in the manual which will set to 256 different eight dot, 12 dot, characters. They are displayed across characters.
1	0	1	0	X	X	X	0	0	Black	Black	64 Display elements across	5, 3, 2	The SEMI-GRAPHICS FOUR mode uses an internal "course graphics" generator in which a vertical eight dot is a hollow dot in divided pins four equal parts. The number of each set is proportional to a corresponding bit on the 100 data bus. The color of the character pairs is determined by four bits.
1	0	1	1	X	X	X	0	0	Black	Black	32 Display elements down	5, 3, 2	The SEMI-GRAPHICS FOUR mode uses an internal "course graphics" generator in which a vertical eight dot is a hollow dot in divided pins four equal parts. The number of each set is proportional to a corresponding bit on the 100 data bus. The color of the character pairs is determined by four bits.
1	1	1	0	X	X	0	0	0	Color	Color	64 Display elements across	5, 3, 2	The SEMI-GRAPHICS SIX mode uses an internal "course graphics" generator in which a vertical eight dot is a hollow dot in divided pins four equal parts. The number of each set is proportional to a corresponding bit on the 100 data bus. The color of the character pairs is determined by four bits.
1	1	1	1	X	X	0	0	0	Color	Color	48 Display elements down	5, 3, 2	The SEMI-GRAPHICS SIX mode uses an internal "course graphics" generator in which a vertical eight dot is a hollow dot in divided pins four equal parts. The number of each set is proportional to a corresponding bit on the 100 data bus. The color of the character pairs is determined by four bits.
1	1	X	X	0	0	0	0	0	Color	Color	64 Display elements across	5, 3, 2, 1, 0	The GRAPHICS ONE mode uses a maximum of 1024 bytes of display RAM in which one set of bits specifies one picture element.
1	1	X	X	0	0	0	0	0	Color	Color	64 Display elements down	5, 3, 2, 1, 0	The GRAPHICS ONE mode uses a maximum of 1024 bytes of display RAM in which one set of bits specifies one picture element.
1	1	X	X	0	1	0	0	0	Color	Color	128 Display elements across	5, 3, 2, 1, 0	The GRAPHICS TWO mode uses a maximum of 2048 bytes of display RAM in which one set of bits specifies one picture element.
1	1	X	X	0	1	0	0	0	Color	Color	128 Display elements down	5, 3, 2, 1, 0	The GRAPHICS TWO mode uses a maximum of 2048 bytes of display RAM in which one set of bits specifies one picture element.
1	1	X	X	0	1	0	1	0	Color	Color	256 Display elements across	5, 3, 2, 1, 0	The GRAPHICS THREE mode uses a maximum of 4096 bytes of display RAM in which one set of bits specifies one picture element.
1	1	X	X	0	1	0	1	0	Color	Color	256 Display elements down	5, 3, 2, 1, 0	The GRAPHICS THREE mode uses a maximum of 4096 bytes of display RAM in which one set of bits specifies one picture element.
1	1	X	X	1	0	0	0	0	Color	Color	192 Display elements across	5, 3, 2, 1, 0	The GRAPHICS SIX mode uses a maximum of 6144 bytes of display RAM in which one set of bits specifies one picture element.
1	1	X	X	1	0	0	0	0	Color	Color	192 Display elements down	5, 3, 2, 1, 0	The GRAPHICS SIX mode uses a maximum of 6144 bytes of display RAM in which one set of bits specifies one picture element.

Appendix 4

MC6821 data sheet

Supplied courtesy of Motorola Semiconductors.

The information here has been carefully checked and is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies. Motorola reserves the right to make changes to any products herein to improve reliability, function or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein. No licence is conveyed under patent rights in any form. When this document contains information on a new product, specifications herein are subject to change without notice.


MOTOROLA

SEMICONDUCTORS

Colvilles Road, Kelvin Estate-East Kilbridge/Glasgow-SCOTLAND

PERIPHERAL INTERFACE ADAPTER (PIA)

The MC6821 Peripheral Interface Adapter provides the universal means of interfacing peripheral equipment to the M6800 family of microprocessors. This device is capable of interfacing the MPU to peripherals through two 8-bit bidirectional peripheral data buses and four control lines. No external logic is required for interfacing to most peripheral devices.

The functional configuration of the PIA is programmed by the MPU during system initialization. Each of the peripheral data lines can be programmed to act as an input or output, and each of the four control/interrupt lines may be programmed for one of several control modes. This allows a high degree of flexibility in the overall operation of the interface.

- 8-Bit Bidirectional Data Bus for Communication with the MPU
- Two Bidirectional 8-Bit Buses for Interface to Peripherals
- Two Programmable Control Registers
- Two Programmable Data Direction Registers
- Four Individually-Controlled Interrupt Input Lines; Two Usable as Peripheral Control Outputs
- Handshake Control Logic for Input and Output Peripheral Operation
- High-Impedance Three-State and Direct Transistor Drive Peripheral Lines
- Program Controlled Interrupt and Interrupt Disable Capability
- CMOS Drive Capability on Side A Peripheral Lines
- Two TTL Drive Capability on All A and B Side Buffers
- TTL-Compatible
- Static Operation

MAXIMUM RATINGS

Characteristics	Symbol	Value	Unit
Supply Voltage	V_{CC}	-0.3 to +7.0	V
Input Voltage	V_{in}	-0.3 to +7.0	V
Operating Temperature Range MC6821, MC68A21, MC68B21 MC6821C, MC68A21C, MC68B21C	T_A	T_L to T_H 0 to 70 -40 to +85	$^{\circ}$ C
Storage Temperature Range	T_{stg}	-55 to +150	$^{\circ}$ C

THERMAL CHARACTERISTICS

Characteristic	Symbol	Value	Unit
Thermal Resistance Ceramic Plastic Cerdip	θ_{JA}	50 100 60	$^{\circ}$ C/W

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum-rated voltages to this high-impedance circuit. Reliability of operation is enhanced if unused inputs are tied to an appropriate logic voltage (i.e., either V_{SS} or V_{CC}).

MC6821

(1.0 MHz)

MC68A21

(1.5 MHz)

MC68B21

(2.0 MHz)

MOS

 (N-CHANNEL, SILICON-GATE,
DEPLETION LOAD)

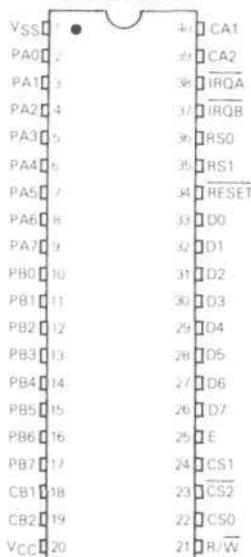
PERIPHERAL INTERFACE ADAPTER


 L SUFFIX
CERAMIC PACKAGE
CASE 715

 S SUFFIX
CERDIP PACKAGE
CASE 734

 P SUFFIX
PLASTIC PACKAGE
CASE 711

PIN ASSIGNMENT



POWER CONSIDERATIONS

The average chip-junction temperature, T_J , in $^{\circ}\text{C}$ can be obtained from:

$$T_J = T_A + (P_D \cdot \theta_{JA}) \quad (1)$$

Where:

T_A = Ambient Temperature, $^{\circ}\text{C}$

θ_{JA} = Package Thermal Resistance, Junction-to-Ambient, $^{\circ}\text{C}/\text{W}$

$P_D = P_{\text{INT}} + P_{\text{PORT}}$

$P_{\text{INT}} = I_{\text{CC}} \times V_{\text{CC}}$, Watts – Chip Internal Power

P_{PORT} = Port Power Dissipation, Watts – User Determined

For most applications $P_{\text{PORT}} \ll P_{\text{INT}}$ and can be neglected. P_{PORT} may become significant if the device is configured to drive Darlington bases or sink LED loads.

An approximate relationship between P_D and T_J (if P_{PORT} is neglected) is:

$$P_D = K - (T_J + 273^{\circ}\text{C}) \quad (2)$$

Solving equations 1 and 2 for K gives:

$$K = P_D \cdot (T_A + 273^{\circ}\text{C}) + \theta_{JA} \cdot P_D^2 \quad (3)$$

Where K is a constant pertaining to the particular part. K can be determined from equation 3 by measuring P_D (at equilibrium) for a known T_A . Using this value of K the values of P_D and T_J can be obtained by solving equations (1) and (2) iteratively for any value of T_A .

DC ELECTRICAL CHARACTERISTICS ($V_{\text{CC}} = 5.0 \text{ Vdc} \pm 5\%$, $V_{\text{SS}} = 0$, $T_A = T_L$ to T_H unless otherwise noted)

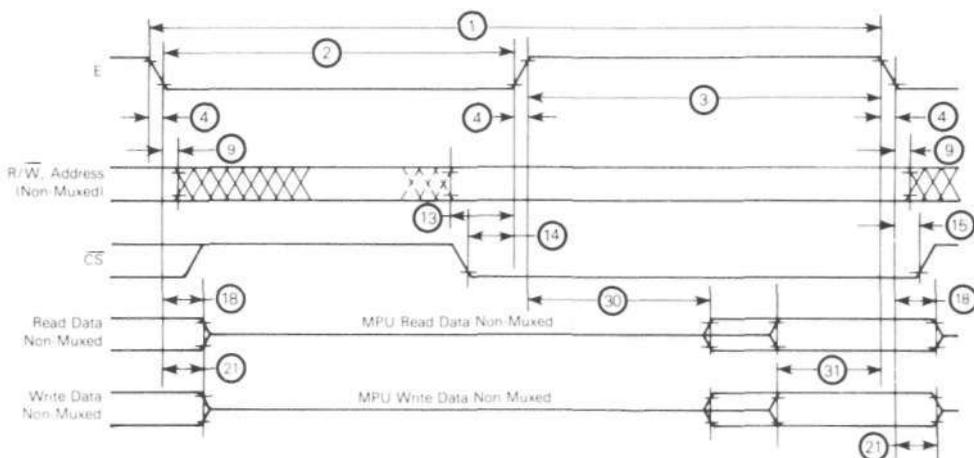
Characteristic	Symbol	Min	Typ	Max	Unit	
BUS CONTROL INPUTS (R/W, Enable, RESET, RS0, RS1, CS0, CS1, CS2)						
Input High Voltage	V_{IH}	$V_{\text{SS}} + 2.0$	–	V_{CC}	V	
Input Low Voltage	V_{IL}	$V_{\text{SS}} - 0.3$	–	$V_{\text{SS}} + 0.8$	V	
Input Leakage Current ($V_{\text{in}} = 0$ to 5.25 V)	I_{in}	–	1.0	2.5	μA	
Capacitance ($V_{\text{in}} = 0$, $T_A = 25^{\circ}\text{C}$, $f = 1.0 \text{ MHz}$)	C_{in}	–	–	7.5	pF	
INTERRUPT OUTPUTS (IRQA, IRQB)						
Output Low Voltage ($I_{\text{Load}} = 3.2 \text{ mA}$)	V_{OL}	–	–	$V_{\text{SS}} + 0.4$	V	
Three-State Output Leakage Current	I_{OZ}	–	1.0	10	μA	
Capacitance ($V_{\text{in}} = 0$, $T_A = 25^{\circ}\text{C}$, $f = 1.0 \text{ MHz}$)	C_{out}	–	–	5.0	pF	
DATA BUS (D0-D7)						
Input High Voltage	V_{IH}	$V_{\text{SS}} + 2.0$	–	V_{CC}	V	
Input Low Voltage	V_{IL}	$V_{\text{SS}} - 0.3$	–	$V_{\text{SS}} + 0.8$	V	
Three-State Input Leakage Current ($V_{\text{in}} = 0.4$ to 2.4 V)	I_{Iz}	–	2.0	10	μA	
Output High Voltage ($I_{\text{Load}} = -205 \mu\text{A}$)	V_{OH}	$V_{\text{SS}} + 2.4$	–	–	V	
Output Low Voltage ($I_{\text{Load}} = 1.6 \text{ mA}$)	V_{OL}	–	–	$V_{\text{SS}} + 0.4$	V	
Capacitance ($V_{\text{in}} = 0$, $T_A = 25^{\circ}\text{C}$, $f = 1.0 \text{ MHz}$)	C_{in}	–	–	12.5	pF	
PERIPHERAL BUS (PA0-PA7, PB0-PB7, CA1, CA2, CB1, CB2)						
Input Leakage Current ($V_{\text{in}} = 0$ to 5.25 V)	R/W, RESET, RS0, RS1, CS0, CS1, CS2, CA1, CB1, Enable	I_{in}	–	1.0	2.5	μA
Three-State Input Leakage Current ($V_{\text{in}} = 0.4$ to 2.4 V)	PB0-PB7, CB2	I_{Iz}	–	2.0	10	μA
Input High Current ($V_{\text{IH}} = 2.4 \text{ V}$)	PA0-PA7, CA2	I_{IH}	–200	–400	–	μA
Darlington Drive Current ($V_{\text{D}} = 1.5 \text{ V}$)	PB0-PB7, CB2	I_{OH}	–1.0	–	–10	mA
Input Low Current ($V_{\text{IL}} = 0.4 \text{ V}$)	PA0-PA7, CA2	I_{IL}	–	–1.3	–2.4	mA
Output High Voltage ($I_{\text{Load}} = -200 \mu\text{A}$) ($I_{\text{Load}} = -10 \mu\text{A}$)	PA0-PA7, PB0-PB7, CA2, CB2 PA0-PA7, CA2	V_{OH}	$V_{\text{SS}} + 2.4$ $V_{\text{CC}} - 1.0$	–	–	V
Output Low Voltage ($I_{\text{Load}} = 3.2 \text{ mA}$)		V_{OL}	–	–	$V_{\text{SS}} + 0.4$	V
Capacitance ($V_{\text{in}} = 0$, $T_A = 25^{\circ}\text{C}$, $f = 1.0 \text{ MHz}$)		C_{in}	–	–	10	pF
POWER REQUIREMENTS						
Internal Power Dissipation (Measured at $T_A = T_L$)		P_{INT}	–	–	550	mW

BUS TIMING CHARACTERISTICS (See Notes 1 and 2)

Ident. Number	Characteristic	Symbol	MC6821		MC68A21		MC68B21		Unit
			Min	Max	Min	Max	Min	Max	
1	Cycle Time	t_{cyc}	1.0	10	0.67	10	0.5	10	μs
2	Pulse Width, E Low	PW_{EL}	430	—	280	—	210	—	ns
3	Pulse Width, E High	PW_{EH}	450	—	280	—	220	—	ns
4	Clock Rise and Fall Time	t_r, t_f	—	25	—	25	—	20	ns
9	Address Hold Time	t_{AH}	10	—	10	—	10	—	ns
13	Address Setup Time Before E	t_{AS}	80	—	60	—	40	—	ns
14	Chip Select Setup Time Before E	t_{CS}	80	—	60	—	40	—	ns
15	Chip Select Hold Time	t_{CH}	10	—	10	—	10	—	ns
18	Read Data Hold Time	t_{DHR}	20	50*	20	50*	20	50*	ns
21	Write Data Hold Time	t_{DHW}	10	—	10	—	10	—	ns
30	Output Data Delay Time	t_{DDR}	—	290	—	180	—	150	ns
31	Input Data Setup Time	t_{DSW}	165	—	80	—	60	—	ns

* The data bus output buffers are no longer sourcing or sinking current by t_{DHR}/t_{DHW} max (High Impedance).

FIGURE 1 — BUS TIMING



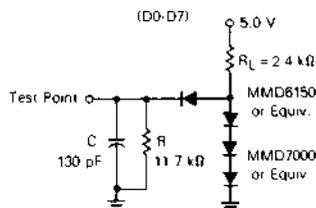
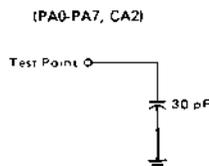
Notes

1. Voltage levels shown are $V_L \leq 0.4$ V, $V_H \geq 2.4$ V, unless otherwise specified.
2. Measurement points shown are 0.8 V and 2.0 V, unless otherwise specified.

PERIPHERAL TIMING CHARACTERISTICS ($V_{CC} = 5.0 \text{ V} \pm 5\%$, $V_{SS} = 0 \text{ V}$, $T_A = T_L$ to T_H unless otherwise specified)

Characteristic	Symbol	MC6821		MC68A21		MC68B21		Unit	Reference Fig. No
		Min	Max	Min	Max	Min	Max		
Data Setup Time	t_{PDS}	200	-	135	-	100	-	ns	6
Data Hold Time	t_{PDH}	0	-	0	-	0	-	ns	6
Delay Time, Enable Negative Transition to CA2 Negative Transition	t_{CA2}	-	1.0	-	0.670	-	0.500	μs	3, 7, 8
Delay Time, Enable Negative Transition to CA2 Positive Transition	t_{RS1}	-	1.0	-	0.670	-	0.500	μs	3, 7
Rise and Fall Times for CA1 and CA2 Input Signals	t_r, t_f	-	1.0	-	1.0	-	1.0	μs	8
Delay Time from CA1 Active Transition to CA2 Positive Transition	t_{RS2}	-	2.0	-	1.35	-	1.0	μs	3, 8
Delay Time, Enable Negative Transition to Data Valid	t_{PDW}	-	1.0	-	0.670	-	0.5	μs	3, 9, 10
Delay Time, Enable Negative Transition to CMOS Data Valid PA0-PA7, CA2	t_{CMOS}	-	2.0	-	1.35	-	1.0	μs	4, 9
Delay Time, Enable Positive Transition to CB2 Negative Transition	t_{CB2}	-	1.0	-	0.670	-	0.5	μs	3, 11, 12
Delay Time, Data Valid to CB2 Negative Transition	t_{DC}	20	-	20	-	20	-	ns	3, 10
Delay Time, Enable Positive Transition to CB2 Positive Transition	t_{RS1}	-	1.0	-	0.670	-	0.5	μs	3, 11
Control Output Pulse Width, CA2, CB2	PW_{CT}	500	-	375	-	250	-	ns	3, 11
Rise and Fall Time for CB1 and CB2 Input Signals	t_r, t_f	-	1.0	-	1.0	-	1.0	μs	12
Delay Time, CB1 Active Transition to CB2 Positive Transition	t_{RS2}	-	2.0	-	1.35	-	1.0	μs	3, 12
Interrupt Release Time, IRQA and IRQB	t_{IR}	-	1.60	-	1.10	-	0.85	μs	5, 14
Interrupt Response Time	t_{RS3}	-	1.0	-	1.0	-	1.0	μs	5, 13
Interrupt Input Pulse Time	PW_I	500	-	500	-	500	-	ns	13
RESET Low Time*	t_{RL}	1.0	-	0.66	-	0.5	-	μs	15

*The RESET line must be high a minimum of 1.0 μs before addressing the PIA.

FIGURE 2 — BUS TIMING TEST LOADS

FIGURE 4 — CMOS EQUIVALENT TEST LOAD

FIGURE 3 — TTL EQUIVALENT TEST LOAD

(PA0-PA7, PB0-PB7, CA2, CB2)

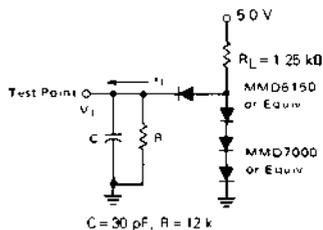
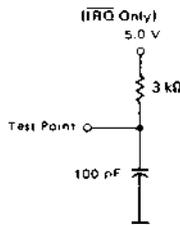

FIGURE 5 — NMOS EQUIVALENT TEST LOAD


FIGURE 6 — PERIPHERAL DATA SETUP AND HOLD TIMES
(Read Mode)

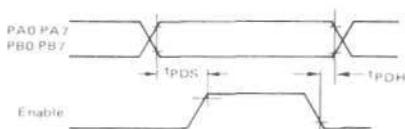


FIGURE 8 — CA2 DELAY TIME
(Read Mode: CRA-5 = 1, CRA-3 = CRA-4 = 0)

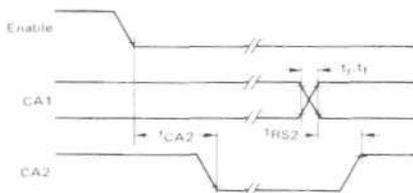
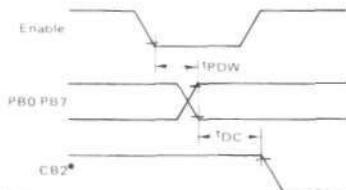
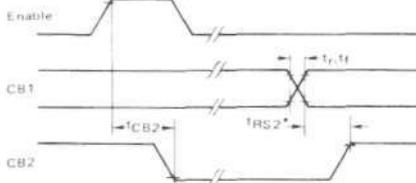


FIGURE 10 — PERIPHERAL DATA AND CB2 DELAY TIMES
(Write Mode: CRB-5 = CRB-3 = 1, CRB-4 = 0)



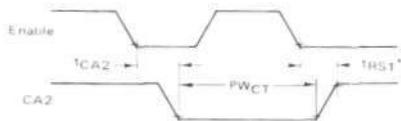
*CB2 goes low as a result of the positive transition of Enable.

FIGURE 12 — CB2 DELAY TIME
(Write Mode: CRB-5 = 1, CRB-3 = CRB-4 = 0)



*Assumes part was deselected during any previous E pulse.

FIGURE 7 — CA2 DELAY TIME
(Read Mode: CRA-5 = CRA-3 = 1, CRA-4 = 0)

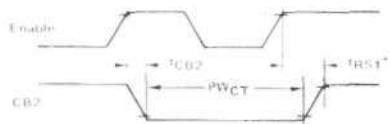


*Assumes part was deselected during the previous E pulse.

FIGURE 9 — PERIPHERAL CMOS DATA DELAY TIMES
(Write Mode: CRA-5 = CRA-3 = 1, CRA-4 = 0)

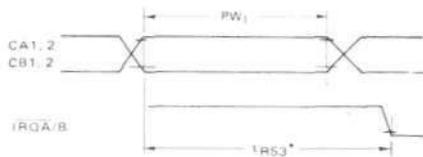


FIGURE 11 — CB2 DELAY TIME
(Write Mode: CRB-5 = CRB-3 = 1, CRB-4 = 0)



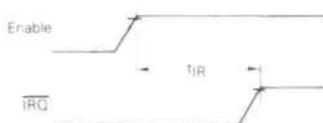
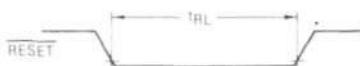
*Assumes part was deselected during the previous E pulse.

FIGURE 13 — INTERRUPT PULSE WIDTH AND \overline{IRQ} RESPONSE



*Assumes Interrupt Enable Bits are set.

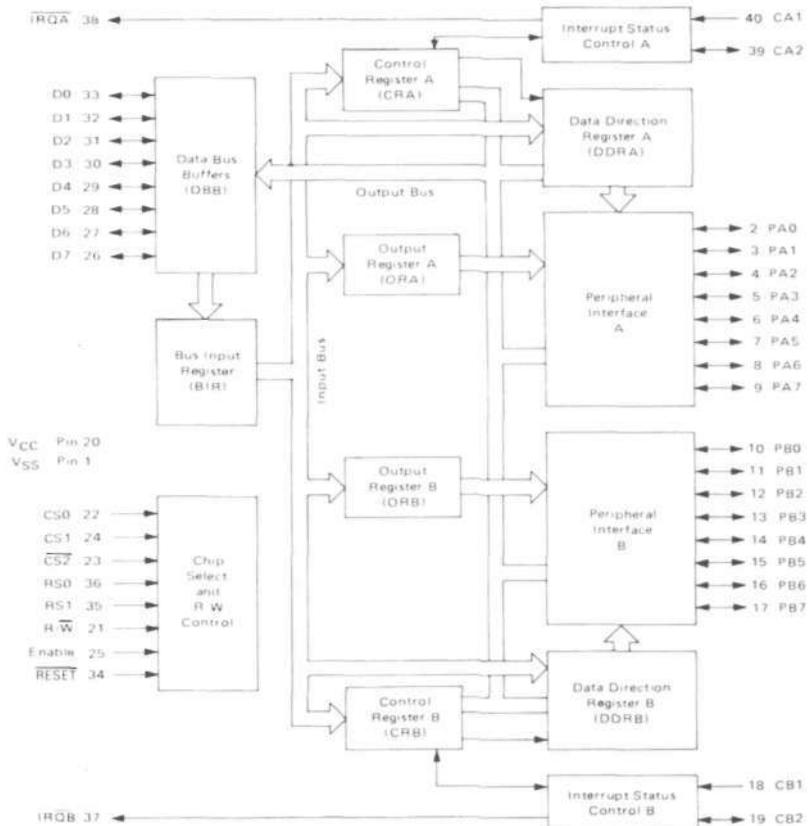
Note: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

FIGURE 14 — $\overline{\text{IRQ}}$ RELEASE TIMEFIGURE 15 — $\overline{\text{RESET}}$ LOW TIME

*The $\overline{\text{RESET}}$ line must be a V_{IH} for a minimum of $1.0 \mu\text{s}$ before addressing the PIA.

Note: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

FIGURE 16 — EXPANDED BLOCK DIAGRAM



PIA INTERFACE SIGNALS FOR MPU

The PIA interfaces to the M6800 bus with an 8-bit bidirectional data bus, three chip select lines, two register select lines, two interrupt request lines, a read/write line, an enable line and a reset line. To ensure proper operation with the MC6800, MC6802, or MC6808 microprocessors, VMA should be used as an active part of the address decoding.

Bidirectional Data (D0-D7) — The bidirectional data lines (D0-D7) allow the transfer of data between the MPU and the PIA. The data bus output drivers are three-state devices that remain in the high-impedance (off) state except when the MPU performs a PIA read operation. The read/write line is in the read (high) state when the PIA is selected for a read operation.

Enable (E) — The enable pulse, E, is the only timing signal that is supplied to the PIA. Timing of all other signals is referenced to the leading and trailing edges of the E pulse.

Read/Write (R/W) — This signal is generated by the MPU to control the direction of data transfers on the data bus. A low state on the PIA read/write line enables the input buffers and data is transferred from the MPU to the PIA on the E signal if the device has been selected. A high on the read/write line sets up the PIA for a transfer of data to the bus. The PIA output buffers are enabled when the proper address and the enable pulse E are present.

RESET — The active low $\overline{\text{RESET}}$ line is used to reset all register bits in the PIA to a logical zero (low). This line can be used as a power-on reset and as a master reset during system operation.

Chip Selects (CS0, CS1, and CS2) — These three input signals are used to select the PIA. CS0 and CS1 must be high and CS2 must be low for selection of the device. Data transfers are then performed under the control of the enable and read/write signals. The chip select lines must be stable

for the duration of the E pulse. The device is deselected when any of the chip selects are in the inactive state.

Register Selects (RS0 and RS1) — The two register select lines are used to select the various registers inside the PIA. These two lines are used in conjunction with internal Control Registers to select a particular register that is to be written or read.

The register and chip select lines should be stable for the duration of the E pulse while in the read or write cycle.

Interrupt Request ($\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$) — The active low Interrupt Request lines ($\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$) act to interrupt the MPU either directly or through interrupt priority circuitry. These lines are "open drain" (no load device on the chip). This permits all interrupt request lines to be tied together in a wire-OR configuration.

Each Interrupt Request line has two internal interrupt flag bits that can cause the Interrupt Request line to go low. Each flag bit is associated with a particular peripheral interrupt line. Also, four interrupt enable bits are provided in the PIA which may be used to inhibit a particular interrupt from a peripheral device.

Servicing an interrupt by the MPU may be accomplished by a software routine that, on a prioritized basis, sequentially reads and tests the two control registers in each PIA for interrupt flag bits that are set.

The interrupt flags are (zeroed) as a result of an MPU Read Peripheral Data Operation of the corresponding data register. After being cleared, the interrupt flag bit cannot be enabled to be set until the PIA is deselected during an E pulse. The E pulse is used to condition the interrupt control lines (CA1, CA2, CB1, CB2). When these lines are used as interrupt inputs, at least one E pulse must occur from the inactive edge to the active edge of the interrupt input signal to condition the edge sense network. If the interrupt flag has been enabled and the edge sense circuit has been properly conditioned, the interrupt flag will be set on the next active transition of the interrupt input pin.

PIA PERIPHERAL INTERFACE LINES

The PIA provides two 8-bit bidirectional data buses and four interrupt/control lines for interfacing to peripheral devices.

Section A Peripheral Data (PA0-PA7) — Each of the peripheral data lines can be programmed to act as an input or output. This is accomplished by setting a "1" in the corresponding Data Direction Register bit for those lines which are to be outputs. A "0" in a bit of the Data Direction Register causes the corresponding peripheral data line to act as an input. During an MPU Read Peripheral Data Operation, the data on peripheral lines programmed to act as inputs appears directly on the corresponding MPU Data Bus lines. In the input mode, the internal pullup resistor on these lines represents a maximum of 1.5 standard TTL loads.

The data in Output Register A will appear on the data lines that are programmed to be outputs. A logical "1" written into the register will cause a "high" on the corresponding data

line while a "0" results in a "low." Data in Output Register A may be read by an MPU "Read Peripheral Data A" operation when the corresponding lines are programmed as outputs. This data will be read properly if the voltage on the peripheral data lines is greater than 2.0 volts for a logic "1" output and less than 0.8 volt for a logic "0" output. Loading the output lines such that the voltage on these lines does not reach full voltage causes the data transferred into the MPU on a Read operation to differ from that contained in the respective bit of Output Register A.

Section B Peripheral Data (PB0-PB7) — The peripheral data lines in the B Section of the PIA can be programmed to act as either inputs or outputs in a similar manner to PA0-PA7. They have three-state capability, allowing them to enter a high-impedance state when the peripheral data line is used as an input. In addition, data on the peripheral data lines

PB0-PB7 will be read properly from those lines programmed as outputs even if the voltages are below 2.0 volts for a "high" or above 0.8 V for a "low". As outputs, these lines are compatible with standard TTL and may also be used as a source of up to 1 milliampere at 1.5 volts to directly drive the base of a transistor switch.

Interrupt Input (CA1 and CB1) — Peripheral input lines CA1 and CB1 are input only lines that set the interrupt flags of the control registers. The active transition for these signals is also programmed by the two control registers.

Peripheral Control (CA2) — The peripheral control line CA2 can be programmed to act as an interrupt input or as a

peripheral control output. As an output, this line is compatible with standard TTL; as an input the internal pullup resistor on this line represents 1.5 standard TTL loads. The function of this signal line is programmed with Control Register A.

Peripheral Control (CB2) — Peripheral Control line CB2 may also be programmed to act as an interrupt input or peripheral control output. As an input, this line has high input impedance and is compatible with standard TTL. As an output it is compatible with standard TTL and may also be used as a source of up to 1 milliampere at 1.5 volts to directly drive the base of a transistor switch. This line is programmed by Control Register B.

INTERNAL CONTROLS

INITIALIZATION

A RESET has the effect of zeroing all PIA registers. This will set PA0-PA7, PB0-PB7, CA2 and CB2 as inputs, and all interrupts disabled. The PIA must be configured during the restart program which follows the reset.

There are six locations within the PIA accessible to the MPU data bus: two Peripheral Registers, two Data Direction Registers, and two Control Registers. Selection of these locations is controlled by the RS0 and RS1 inputs together with bit 2 in the Control Register, as shown in Table 1.

Details of possible configurations of the Data Direction and Control Register are as follows:

TABLE 1 — INTERNAL ADDRESSING

RS1	RS0	Control Register Bit		Location Selected
		CRA-2	CRB-2	
0	0	1	X	Peripheral Register A
0	0	0	X	Data Direction Register A
0	1	X	X	Control Register A
1	0	X	1	Peripheral Register B
1	0	X	0	Data Direction Register B
1	1	X	X	Control Register B

X: Don't Care

PORT A-B HARDWARE CHARACTERISTICS

As shown in Figure 17, the MC6821 has a pair of I/O ports whose characteristics differ greatly. The A side is designed to drive CMOS logic to normal 30% to 70% levels, and incorporates an internal pullup device that remains connected even in the input mode. Because of this, the A side requires more drive current in the input mode than Port B. In contrast, the B side uses a normal three-state NMOS buffer which cannot pullup to CMOS levels without external resistors. The B side can drive extra loads such as Darlington's without problem. When the PIA comes out of reset, the A port represents inputs with pullup resistors, whereas the B side (input mode also) will float high or low, depending upon the load connected to it.

Notice the differences between a Port A and Port B read operation when in the output mode. When reading Port A, the actual pin is read, whereas the B side read comes from an output latch, ahead of the actual pin.

CONTROL REGISTERS (CRA and CRB)

The two Control Registers (CRA and CRB) allow the MPU to control the operation of the four peripheral control lines CA1, CA2, CB1, and CB2. In addition they allow the MPU to enable the interrupt lines and monitor the status of the interrupt flags. Bits 0 through 5 of the two registers may be written or read by the MPU when the proper chip select and register select signals are applied. Bits 6 and 7 of the two registers are read only and are modified by external interrupts occurring on control lines CA1, CA2, CB1, or CB2. The format of the control words is shown in Figure 18.

DATA DIRECTION ACCESS CONTROL BIT (CRA-2 and CRB-2)

Bit 2, in each Control Register (CRA and CRB), determines selection of either a Peripheral Output Register or the corresponding Data Direction Register when the proper register select signals are applied to RS0 and RS1. A "1" in bit 2 allows access of the Peripheral Interface Register, while a "0" causes the Data Direction Register to be addressed.

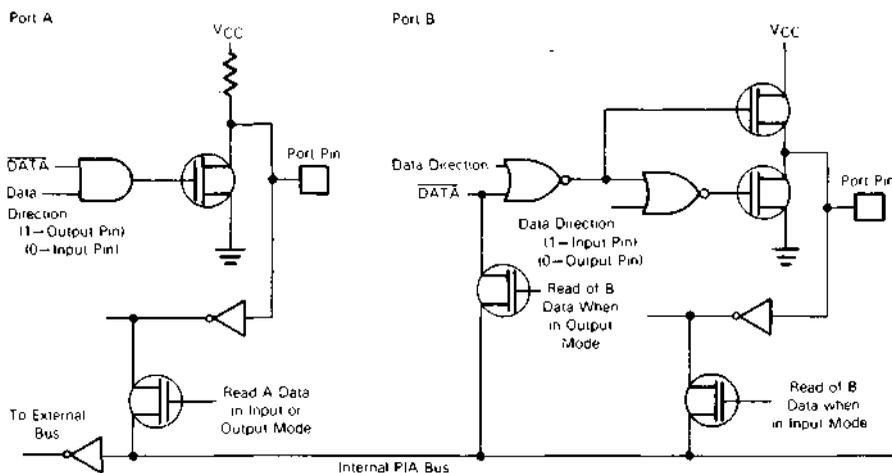
Interrupt Flags (CRA-6, CRA-7, CRB-6, and CRB-7) — The four interrupt flag bits are set by active transitions of signals on the four Interrupt and Peripheral Control lines when those lines are programmed to be inputs. These bits cannot be set directly from the MPU Data Bus and are reset indirectly by a Read Peripheral Data Operation on the appropriate section.

Control of CA2 and CB2 Peripheral Control Lines (CRA-3, CRA-4, CRA-5, CRB-3, CRB-4, and CRB-5) — Bits 3, 4, and 5 of the two control registers are used to control the CA2 and CB2 Peripheral Control lines. These bits determine if the control lines will be an interrupt input or an output control signal. If bit CRA-5 (CRB-5) is low, CA2 (CB2) is an interrupt input line similar to CA1 (CB1). When CRA-5 (CRB-5) is high, CA2 (CB2) becomes an output signal that may be used to control peripheral data transfers. When in the output mode, CA2 and CB2 have slightly different loading characteristics.

Control of CA1 and CB1 Interrupt Input Lines (CRA-0, CRB-1, CRA-1, and CRB-1) — The two lowest-order bits of the control registers are used to control the interrupt input lines CA1 and CB1. Bits CRA-0 and CRB-0 are used to

enable the MPU interrupt signals \overline{IRQA} and \overline{IRQB} , respectively. Bits CRA-1 and CRB-1 determine the active transition of the interrupt input signals CA1 and CB1.

FIGURE 17 — PORT A AND PORT B EQUIVALENT CIRCUITS



ORDERING INFORMATION

MC68A21CP

Motorola Integrated Circuit

M6800 Family

Blanks = 1.0 MHz

A = 1.5 MHz

B = 2.0 MHz

Device Designation

In: M6800 Family

Temperature Range

Blank = 0° to +70°C

C = -40 to +85°C

Package

P = Plastic

S = Cerdip

L = Ceramic

BETTER PROGRAM

Better program processing is available on all types listed. Add suffix letters to part number.

Level 1 add 'S' Level 2 add 'D' Level 3 add 'DS'

Level 1 'S' = 10 Temp Cycles = -1 to 25 to 150°C.
Hi Temp testing at $T_{D,max}$

Level 2 'D' = 168 Hour Burn in at 125°C

Level 3 'DS' = Combination of Level 1 and 2

Speed	Device	Temperature Range
1.0 MHz	MC6821P,L,S	0 to 70°C
	MC6821CP,CL,CS	-40 to +85°C
1.5 MHz	MC68A21P,L,S	0 to +70°C
	MC68A21CP,CL,CS	-40 to +85°C
2.0 MHz	MC68B21P,L,S	0 to +70°C

FIGURE 18 — CONTROL WORD FORMAT

Determine Active CA1 (CB1) Transition for Setting Interrupt Flag IRQA(B1) — (bit 7)

- b1 = 0 IRQA(B1) set by high-to-low transition on CA1 (CB1)
- b1 = 1 IRQA(B1) set by low-to-high transition on CA1 (CB1)

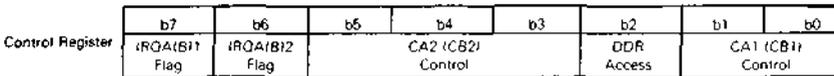
IRQA(B) 1 Interrupt Flag (bit 7)

Goes high on active transition of CA1 (CB1). Automatically cleared by MPU Read of Output Register A(B). May also be cleared by hardware Reset.

CA1 (CB1) Interrupt Request Enable/Disable

- b0 = 0 Disables IRQA(B) MPU Interrupt by CA1 (CB1) active transition¹
- b0 = 1 Enable IRQA(B) MPU Interrupt by CA1 (CB1) active transition.

¹ IRQA(B) will occur on next (MPU generated) positive transition of b0 if CA1 (CB1) active transition occurred while interrupt was disabled



IRQA(B)2 Interrupt Flag (bit 6)

When CA2 (CB2) is an input, IRQA(B) goes high on active transition CA2 (CB2); Automatically cleared by MPU Read of Output Register A(B). May also be cleared by hardware Reset

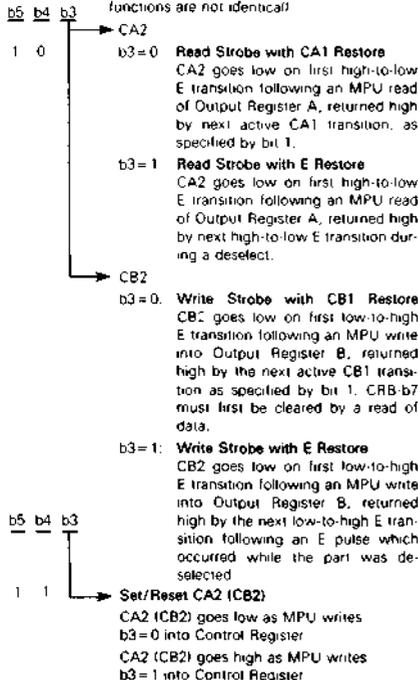
CA2 (CB2) Established as Output (b5 = 1). IRQA(B) 2 = 0; not affected by CA2 (CB2) transitions.

Determines Whether Data Direction Register Or Output Register is Addressed

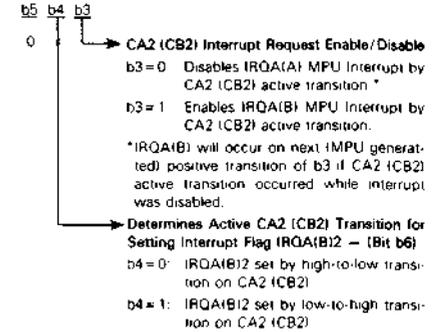
- b2 = 0 Data Direction Register selected
- b2 = 1 Output Register selected.

CA2 (CB2) Established as Output by b5 = 1

(Note that operation of CA2 and CB2 output functions are not identical)



CA2 (CB2) Established as Input by b5 = 0



**Orchard House,
Cleeve Moor,
Cumbria. CA25 6LN**

Appendix 5

The Dragon 64

The major design aim of the Dragon 64 was to ensure upward compatibility with the Dragon 32 and yet provide a machine with enhanced facilities. These extra facilities are:

- (1) An additional 32K of RAM.
- (2) An RS232 (serial) interface.
- (3) Auto-repeating keys.

As the Dragon 64 is virtually identical to the Dragon 32 in most other respects, we confine ourselves to describing these extra features and detailing the differences between the two machines.

1. SWITCHING IN RAM

The SAM chip, described in Appendix 2, can operate in two modes called map type 0 and map type 1. This allows the chip to map addresses to either a 32K or to a 64K RAM address space and this facility means that compatibility between the Dragon 64 and the Dragon 32 can be maintained.

Unlike the Dragon 32, the 64 can operate in both map types provided by the SAM chip and yet still use Extended Color BASIC. On power up, the 64 is configured like a 32. In other words, it is in map type 0 which provides access to 32K of RAM, addressed from 0000 to 7FFF, 16K of BASIC ROM, addressed from 8000 to BFFF, and 16K (minus 256 bytes for I/O devices, vectors, etc.) of expansion space addressed from C000 to FFFF. Switching in the extra RAM involves switching to map type 1 which gives a 64K RAM address space (less 256 bytes) from 0000 to FFFF.

The extra 32K of RAM therefore 'overlays' the BASIC ROM and expansion addresses which means that neither the Extended Color BASIC ROM nor the cartridge ROM can be accessed. It is therefore necessary to 'bootstrap' the machine by first copying a small program into RAM which selects a 'new' BASIC ROM and copies its contents into map type 1. Naturally, the new BASIC ROM will reside in map type 0 and therefore the bootstrap

operates by reading a byte (from the ROM) in map type 0, switching to map type 1, writing the byte, switching back to map type 0 to read the next ROM byte and so on.

Because the new BASIC does not need to reside in the same address space as the 'old' BASIC, it can occupy the addresses C000 to FEFF. This relocation of the BASIC results in 48K of RAM being made available for system/user use. Naturally, if the BASIC interpreter is not required, e.g. when using the OS-9 operating system or machine code programs, the full 64K of RAM is available to the programmer.

The bootstrap procedure is invoked by EXEC on its own if no other EXECs have been used. The Dragon 32's default entry in the EXEC vector has been replaced by the entry point of the Dragon 64's bootstrap routine. Alternatively, if an EXEC has been used, EXEC 48000 calls the bootstrap entry routine directly.

The 64K mode can be distinguished from the 32K mode by the fact that the cursor flashes blue rather than black. One point worth noting is that booting into the 64K mode will not wipe out an existing BASIC program as it behaves like a CLEAR command. Subsequent resets after the initial 64K coldboot will perform a 64K warmboot so that the system remains in 64K mode and does not revert to 32K mode.

Some extra 'housekeeping' bytes which are unused in the Dragon 32 are used to keep track of the boot state. FLAG64 (11A) indicates whether this is a warmboot or a coldboot. A warmboot (FLAG64 = \$55) indicates that the new BASIC has already been copied into RAM in which case a further two bytes (11B:11C) hold a 16-bit checksum that was calculated during the BASIC copy

A checksum is a value that is calculated by adding together the values of the bytes in the BASIC area of RAM. If one or more of these bytes are changed, the value of the checksum will change. Therefore, if the BASIC system in RAM is changed, the change can be detected because its checksum will not be the same as the checksum for the original BASIC system.

When the system is reset, this checksum is checked against a recalculated checksum of the BASIC RAM area. If these do not agree then a coldboot, which copies BASIC into RAM, is initiated. This avoids the problem of users or programs poking the BASIC in RAM thus causing the system to crash and then performing a warmboot which would not restore the correct BASIC system.

Because the BASIC in the 64K mode resides in RAM, it is possible to experiment with it. Obviously, great care has to be taken with such experimentation as it is all too easy to accidentally crash the system. The restoration of the original BASIC can be avoided if the checksum is recalculated from the 'experimental' BASIC so that it appears that the BASIC system is unchanged.

A routine which carries out this recalculation is shown below.

```

* RECSUM - re-checksum the BASIC RAM and
*         update the system checksum(CSUM64)
*
* Register inputs NONE
*
CSUM64    EQU $11B        ; System checksum
RECSUM    PSHS X,D        ; Save registers
          LDX #$C000      ; Base of 64K BASIC
          LDD #$0         ; Zero running total
NXTADD    ADDD ,X++       ; Add to checksum
          CMPX #$FF00     ; until end of
          BLO NXTADD      ; BASIC reached
          STD CSUM64      ; Update system checksum
          PULS X,D,PC     ; Restore and return

```

The original BASIC can be restored by clearing FLAG64 and then resetting the machine.

The 64K mode bootstrap is contained in the 'old' BASIC ROM at address BF49 onwards. The ROM part of the bootstrap copies the RAM part of the bootstrap into the cassette buffer since this will not be used during a boot, and then jumps into the RAM part to complete the boot sequence. Once loaded, the secondary reset vector (72:73) is set up to point to the 64K mode bootstrap so that subsequent resets will invoke the bootstrap automatically.

Once in 64K mode, there is no easy way to return to the 32K mode since a reverse bootstrap has not been provided. Whilst expanding into extra RAM (the 64K bootstrap) holds no danger, trying to contract back to 32K of RAM may cause the existing program/variables to be overwritten. It may appear that a safe reversion technique, which works with programs contained wholly in the bottom 32K of RAM is as shown below:

```

CLEAR 200,32766 'Default 32K settings
POKE &H72,&HB4  ' Restore normal 32K
POKE &H73,&H4F  ' secondary reset vector

```

In fact, a reset after these statements does not cause reversion to 32K mode. To revert requires resetting the interrupt vectors and changing various other addresses and pointers.

2. THE RS232 INTERFACE

An RS232 serial interface (via a 7 pin DIN connector) is provided as standard with the Dragon 64 and can be used in both the 32K mode and the 64K mode. This facility supports the additional commands DLOAD and DLOADM which enable BASIC programs, in ASCII format,

and machine code programs to be downloaded into the Dragon 64 from a host computer. These extra commands are very similar in operation to their cassette file equivalents CLOAD and CLOADM. However, DLOAD and DLOADM are limited to loading files of a particular format and cannot be used for more general inter-computer communication.

The download facility of the Dragon 64 is supported by three low-level routines, the entry points of which are contained in the I/O jump tables. These routines are, in actual fact, of more general use for serial I/O and a brief description of each is given below.

```
* SERIN - Read a byte (8 bits) from the serial port
*
* Register inputs NONE
* Register outputs A - returns byte read
* Registers destroyed NONE
*
* SEROUT - Send a byte (8 bits) to the serial port
*
* Register inputs A - byte to be output
* Registers destroyed NONE
*
* SERSET - set up serial port baud rate
*
* Register inputs B - baud rate select byte
* Register outputs CC.C = 0 if select byte OK
*                   CC.C = 1 if select byte out of range
* Registers destroyed B,X,CC
*
* The routine supports 7 baud rate select values:
*   B = 0 -> 110 baud
*   B = 1 -> 300 baud
*   B = 2 -> 600 baud
*   B = 3 -> 1200 baud
*   B = 4 -> 2400 baud
*   B = 5 -> 4800 baud
*   B = 6 -> 9600 baud
* The default baud rate on power-up is 1200 baud
```

The entry points in the I/O jump table for these routines are:

```
802A  SERIN
802D  SEROUT
8030  SERSET
```

2.1 Using an RS232 terminal with a Dragon 64

The above routines can be used in conjunction with the character input/output RAM hooks described in section 9.5.1, to replace the normal Dragon keyboard and screen with an RS232 terminal. In this example, we show how the character input RAM hook at 16A may be used to

redirect input to SERIN and the character output hook at 167 to redirect output to the SEROUT routine. However, these RAM hooks are also called for cassette and printer I/O so you must inspect DEVNUM (6F) to avoid redirecting their I/O.

The following program demonstrates this technique:

```

*
* Redirect console I/O to RS232 terminal
*
* Program equates
*
HKCHRO    EQU $167           ; Character output hook
HKCHRI    EQU $16A           ; Character input hook
SERIN     EQU $802A          ; Serial input entry point
SEROUT    EQU $802D          ; Serial output entry point
DEVNUM    EQU $6F            ; Device number location
          ORG $4E21
*
* SETIO - set up console I/O redirection
* Register inputs NONE
* Registers destroyed A,X,CC
*
SETIO     LEAX INCH,PCR       ; Set up address of input
          ; routine
          STX HKCHRI+1       ; and redirect console input
          LEAX OUTCH,PCR     ; Do the same for
          STX   HKCHRO+1     ; console output
          LDA #$7E           ; Opcode for JMP
          STA HKCHRI         ; placed in I/O
          STA HKCHRO         ; RAM hooks
*
* Use a call to SERSET here for baud rate setting
* if not a 1200 baud device
          RTS                 ; Return
*
* INCH - input a character from RS232 port
* Register inputs NONE
* Register outputs DEVNUM = 0 -> A contains character
          ; DEVNUM <> 0 -> A unaffected
*
INCH      TST DEVNUM         ; Is this console input
          BNE INXIT
          JSR SERIN          ; Yes, read RS232
*
* At this point, we have input the character from
* the RS232 port and therefore wish to avoid returning
* the code which will input from the normal keyboard.
* Simplest solution is to remove the return address
* from the stack
*
          LEAS 2,S           ; Remove latest return
INXIT     RTS                 ; Return

```

* OUTCH - output a character to RS232 port

* Register inputs A - contains character

```

OUTCH      TST DEVNUM          ; Is this console output
           BNE OUTXIT
           JSR SEROUT          ; Output to RS232
           LEAS 2,S            ; Remove return address
OUTXIT     RTS                 ; Return
  
```

2.2 Using a serial printer with the Dragon 64
 The RS232 port can also be used as the standard printer interface instead of the Centronics (parallel) interface. Which of these two options is selected is determined by location PRNSEL (3FF). A 0 (default) value in this location selects the parallel interface, non-0 selects the serial interface. Therefore:

```

POKE &H3FF,1 'Selects serial printer
POKE &H3FF,0 'Selects parallel printer
  
```

In addition to this printer select byte, there are two other bytes (3FD:3FE) which specify an end-of-line delay period since some printers (especially serial) require this. The time delay is in increments of 10 milliseconds. For example:

```

POKE &H3FE,100
  
```

will provide a delay of 100×10 milliseconds = 1 second. The minimum delay (default) is 0 and the maximum delay is 655.35 seconds.

2.3 Configuring the RS232 interface

The pinout of the RS232 connector is shown in Figure A5.1. The device that drives this interface is an R6551 Asynchronous Communication Interface Adapter (ACIA) which, like the PIA, is a programmable device. In its default configuration, this device is programmed to produce 1 start bit, 8 data bits and 2 stop bits with no parity at a baud rate of 1200 baud.

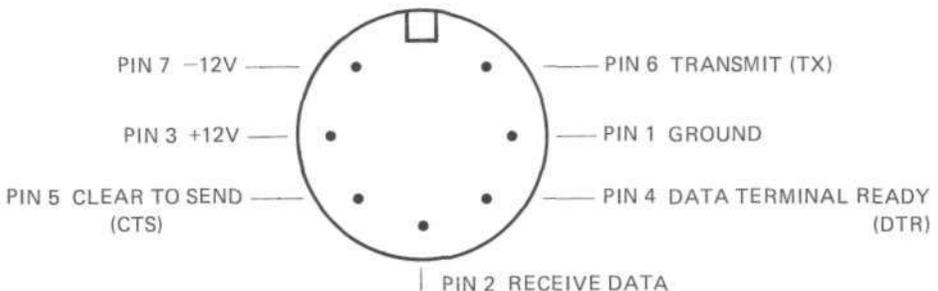


Fig. A5.1 RS232 Pin out connections

Like all I/O devices in the Dragon 64, the ACIA is memory-mapped and occupies the following address space:

Address	Register
FF04	Transmit data register (on write cycle)
FF04	Receive data register (on read cycle)
FF05	Status register
FF06	Command register
FF07	Control register

Because the ACIA is a sophisticated device with many options, it is not possible to cover the operation of this chip in detail here. In most instances, it is easier to configure hardware (printers, terminals, etc.) to the default configuration since this is a once and for all operation compared to configuration by software as this is necessary every time the Dragon is switched on. However, we do provide the following BASIC statement which can be used to select the baud rate of the device.

```
POKE &HFF07,((PEEK(&HFF07) AND &HF0) OR B)
```

The variable B holds a value which specifies the baud rate of the device connected to the RS232 interface. Possible values are:

B-value	Baud rate
1	50
2	75
3	110
4	135
5	150
6	300
7	600
8	1200
9	1800
10	2400
11	3600
12	4800
13	7200
14	9600
15	19200

3. THE KEYBOARD AUTO-REPEAT FACILITY

This facility is provided in the 64K mode only and is not implemented when the Dragon 64 is operating in 32K mode. The reason for this is to maintain compatibility with existing software, such as the DREAM assembler, which provide their own auto-repeat facilities.

The auto-repeat makes use of the 50Hz (60Hz) interrupt as the timing reference which determines the delay before repeating the key and which also controls

the rate of repeat. A byte location, REPDLY (11F) in RAM contains the inter-repeat delay value. The default value of REPDLY is 5 giving an auto-repeat of 10 characters per second. The same value is also used to control the delay before starting the repeat but, in this case, its value is multiplied by 8 giving a normal delay of 0.8 seconds before auto-repeat starts. By altering the value in REPDLY with a POKE statement, you may increase or decrease the auto-repeat rate/delay.

Auto-repeat is incorporated in the 64K mode by redirecting the secondary interrupt IRQ vector to an additional piece of code (entry point FE18) which is dedicated to updating the keyboard delay value. After executing this code, a jump is made to the normal interrupt service routine. Therefore, the auto-repeat facility can be disabled by reinstating the normal interrupt service routine entry point (DD3D) into the secondary IRQ vector as described in section 8.2.3. Conversely, auto-repeat can be incorporated into the 32K mode by altering the secondary IRQ vector to jump to the normally unused keyboard delay update code located at BF20 in the 'old' BASIC ROM.

There are two further 'housekeeping' bytes which are also used in the keyboard repeat process. These are LSTKEY (11D) which keeps a copy of the last key code returned by the keyboard polling routine and CNTDWN (11E) which contains the updated delay value which, when it reaches zero, triggers the code which resets the keyboard rollover table thus causing the current key depressing to be recognised as a new depression. A mechanism to do this has already been described in Chapter 8.

4. DRAGON 64/32 DIFFERENCES

The most important differences between these two machines have already been covered in the previous sections in this appendix. However, there are a few other minor differences and these are summarised below.

4.1 Differences in BASIC

The only differences between the two machines in terms of Extended Color BASIC is that DLOAD and DLOADM are implemented in the Dragon 64 and that the USR call bug described in Chapter 5 has now been corrected. Recall that this bug meant that USR calls 1 to 9 were not recognised without a padding character so that USR1 had to be written as USR01, USR2 as USR02, etc. Now, you must write these as USR1, USR2, etc. as USR01, USR02, etc. are all (correctly) taken to be syntax errors.

Another slight difference is that the functions VARPTR and MEM have been altered so that they return an unsigned 16-bit value. Previously, if these functions were used with an argument which was greater than 32767

they returned a negative result because negative integers are represented in two's complement notation and have values from 32768 to 65535. Now a positive result is always returned.

4.2 RAM usage

The Dragon 64 in 64K mode provides an extra 16K bytes for user/variable storage with 16K bytes used by the BASIC and I/O space. The BASIC RAM area can also be used for those applications which do not need a resident BASIC interpreter such as the OS-9 operating system and machine code programs. In addition, some unused bytes in the system pages of the Dragon 32 are now used by the Dragon 64. These are locations 11A to 11F inclusive and 3FD to 3FF inclusive.

4.3 ROM usage

There are very few differences between the BASIC ROM entry points in the Dragon 32 and the Dragon 64 when in 32K mode. All the dispatch addresses given in Appendix 7 remain the same. However, there are a number of ROM patches to previously unused areas of ROM in the Dragon 64 which repair some of the bugs in the Dragon 32. More major differences apply to the area of ROM which contains I/O driver code as, obviously, additions have been made to support the RS232 interface, the latent auto-repeat facility and the 64K mode bootstrap.

In the 64K mode, the entry points to the system routines are now in RAM in the address space C000 to FEFF so, obviously, old entry addresses are completely incompatible. However, throughout most of this address space there is a simple relationship between the old ROM entry points and the new ROM entry point since they are offset by 4000 (hex) bytes. In other words, the direct jump table is located at C000 onwards, the indirect jump table at E000 onwards, etc. This simple relationship is maintained until the area of RAM that corresponds to the old ROM initialisation sequence (B39B/F39B) from which point the relationship no longer holds.

The reason for this is that RESET automatically selects map type 0 and therefore enters the 32K mode ROM where the initialisation code (RESET service routine) resides. This code is not duplicated in the BASIC RAM area.

Appendix 6

The ASCII character set

The table below shows the characters in the ASCII character set and their associated values. Notice that the character values are given in octal (base 8) rather than decimal notation. Each octal digit represents 3 bits from 000 to 111 and the octal representation means that the bit pattern of each character may be readily deduced.

000	nul	001	soh	002	stx	003	etx	004	eot	005	enq
006	ack	007	bel	010	bs	011	ht	012	nl	013	vt
014	np	015	cr	016	so	017	si	020	dle	021	dcl
022	dc2	023	dc3	024	dc4	025	nak	026	syn	027	etb
030	can	031	em	032	sub	033	esc	034	fs	035	gs
036	rs	037	us	040	sp	041	!	042	"	043	#
044	␣	045	%	046	&	047	'	050	(051)
052	*	053	+	054	,	055	-	056	.	057	/
060	0	061	1	062	2	063	3	064	4	065	5
066	6	067	7	070	8	071	9	072	␣	073	,
074	<	075	=	076	>	077	?	100	@	101	A
102	B	103	C	104	D	105	E	106	F	107	G
110	H	111	I	112	J	113	K	114	L	115	M
116	N	117	O	120	P	121	Q	122	R	123	S
124	T	125	U	126	V	127	W	130	X	131	Y
132	Z	133	[134	\	135]	136	^	137	
140	\	141	a	142	b	143	c	144	d	145	e
146	f	147	g	150	h	151	i	152	j	153	k
154	l	155	m	156	n	157	o	160	p	161	q
162	r	163	s	164	t	165	u	166	v	167	w
170	x	171	y	172	z	173	{	174		175	}
176	~	177	del								

Appendix 7

Dragon-specific tables

This appendix is made up of detailed, Dragon-specific information collected together in a tabular form. Rather than include such information in the text, we have collected a number of tables together in this appendix. First, we summarise the functions and connections of the individual bits in the Dragon's PIA registers.

1. PIA SUMMARY

- (1) P0DDRA - A-side data direction register PIA0
All bits in this register DA0-DA7 are set to 0 meaning that the corresponding bits in the PDR are inputs.
- (2) P0PDRA - A-side peripheral data register PIA0
Bits PA0-PA6 are connected to keyboard rows 0 to 6. Bits PA7 is a joystick comparison input and bits PA0 and PA1 are connected to the right and left joystick buttons respectively. PA0 and PA1 are also shared by keyboard rows 1 and 2 thus the keyboard must be disabled when joysticks are used.
- (3) P0CRA - A-side control register PIA0
The table below shows the functions of the bits in this register.

CRA0	CA1 control, 0->disable IRQA, 1->enable IRQA
CRA1	CA1 control, 0->set IRQA1 on HI to LO, 1->set IRQA1 on LO to HI.
CRA2	0->P0DDRA, 1->P0PDRA
CRA3	CA2 control, 0->CA2 LO, 1->CA2 HI
CRA4	1 -> CA2 in CRA3 in bit follow mode
CRA5	as above
CRA6	IRQA2 flag, not used
CRA7	IRQA1 flag
- (4) CA1 - Horizontal sync interrupt input (63.5 microseconds)
- (5) CA2 - LSB of two analog multiplexor select lines

- (6) P0DDRB - B-side data direction register PIA0
All bits in this registers are set to 1 thus configuring the corresponding bits in P0PDRB as outputs.
- (7) P0PDRB - B-side peripheral data register PIA0
This register is shared by the keyboard input and printer data lines. Bits PB0-PB7 are either connected to keyboard matrix columns 0 to 7 or are printer data bits 0 to 7.
- (8) P0CRB - B-side control register PIA0
The table below summarises the functions of the bits in this register.

CRB0	CB1 control 0 -> disable IRQB, 1 -> enable IRQB
CRB1	CB1 control 0 -> set IRQB1 on Hi to LO, 1 set IRQB1 on LO to HI
CRB2	0 -> P0DDRB, 1 -> P0PDRB
CRB3	CB2 control, 0 -> CB2 LO, 1 -> CB2 HI
CRB4	=1 CB2 in CRB3 bit follow mode
CRB5	=1 CB2 in CRB3 bit follow mode
CRB6	IRQB2 flag, not used
CRB7	IRQB1 flag

- (9) CB1 - field sync interrupt (20ms, 50Hz)
- (10) CB2 - MSB of analog multiplexor select lines
- (11) P1DDRA - A-side data direction register PIA1
Bit 0 in this register is 0 thus configuring bit 0 in the peripheral data register as an input. All other bits are 1, configuring associated peripheral data bits as outputs.
- (12) P1PDRA - A-side peripheral data register PIA1
Bits 2 to 7 in this register correspond to bits 0-5 of a 6-bit DAC input value. Bit 0 is a cassette data bit input and bit 1 is the printer strobe output.
- (13) P1CRA - A-side control register PIA1
The functions of the bits in this register are summarised in the table below.

CRA0	CA1 control, 0->disable IRQA, 1 -> enable IRQA
CRA1	CA1 control, 0->set IRQA1 on HI to LO, 1 -> set IRQA1 on LO to HI
CRA2	0 -> P1DDRA, 1->P1DDRA
CRA3	CA2 control, 0->CA2 LO, 1->CA2 HI
CRA4	=1 -> CA2 in CRA3 bit follow mode
CRA5	=1 -> CA2 in CRA3 bit follow mode

CRA6 IRQA2 flag, not used
 CRA7 IRQA1 flag

- (14) CA1 - printer acknowledge interrupt input, not used
- (15) CA2 - cassette motor control, 0 -> off, 1 -> on
- (16) P1DDRB - B-side data direction register PIA1
 Bits 0 to 2 in this register are 0, configuring associated bits as inputs. Bits 3-7 are 1, configuring associated bits as outputs.
- (17) P1PDRB - B-side peripheral data register PIA1
 Bits 3 to 7 in this register are VDG control lines. Bit 0 is a printer busy input, bit 1 is used for single bit sound and bit 2 is a RAM type detect bit. If it is 0, available RAM is 32K or 64K type. If it is 1, available RAM is 16K type. In the Dragon 64, bit 2 is programmed as an output to select between the 32K mode BASIC ROM (bit 2 = 1, default) and the 64K mode BASIC ROM (bit 2 = 0).
- (18) P1CRB - B-side peripheral control register PIA1
 The table below summarises the functions of the bits in this register.

CRB0 As P0CRB
 CRB1 As P0CRB
 CRB2 0->P1DDRB, 1->P1PDRB
 CRB3 CB2 control, 0->CB2 LO, 1->CB2 HI
 CRB4 =1 CB2 in CRB3 bit follow mode
 CRB5 =1 CB2 in CRB3 bit follow mode
 CRB6 IRQB2 flag, not used
 CRB7 IRQB1 flag

- (19) CB1 - ROM cartridge interrupt detect
- (20) CB2 - sound source enable

2. RESERVED WORD TABLE

The following tables list the reserved words of Extended Color BASIC, their internal tokens and the addresses of associated action routines.

Reserved word	Token	Dispatch address
FOR	80	8448
GO(TO/SUB)	81	85B9
REM	82	8616
'	83	8616
ELSE	84	8616

IF	85	8647
DATA	86	8613
PRINT	87	903D
ON (GOTO / SUB)	88	8675
INPUT	89	872B
END	8A	8532
NEXT	8B	8829
DIM	8C	8A8B
READ	8D	8777
LET	8E	86BC
RUN	8F	85A5
RESTORE	90	8514
RETURN	91	85F3
STOP	92	8539
POKE	93	8E9D
CONT	94	8560
LIST	95	8EAA
CLEAR	96	8571
NEW	97	8415
DEF	98	9C81
CLOAD	99	B6D5
CSAVE	9A	B683
OPEN	9B	B829
CLOSE	9C	B64D
LLIST	9D	8EA4
SET	9E	B9D3
RESET	9F	BA04
CLS	A0	BA60
MOTOR	A1	B982
SOUND	A2	BA9B
AUDIO	A3	BADF
EXEC	A4	B771
SKIPF	A5	B81F
DELETE	A6	9D61
EDIT	A7	9965
TRON	A8	9AD9
TROFF	A9	9ADA
LINE	AA	A749
PCLS	AB	A8C0
PSET	AC	A6EF
PRESET	AD	A6F3
SCREEN	AE	A9FE
PCLEAR	AF	AA19
COLOR	BO	A8D4
CIRCLE	B1	B238
PAINT	B2	AC87
GET	B3	AAF0
PUT	B4	AAF3
DRAW	B5	E051
PCOPY	B6	AABE
PMODE	B7	A9AF
PLAY	B8	ADBD
DLOAD	B9	A049
RENUM	BA	9DFA

TAB(BB	N/A
TO	BC	N/A
SUB	BD	N/A
FN	BE	N/A
THEN	BF	N/A
NOT	C0	N/A
STEP	C1	N/A
OFF	C2	N/A
+	C3	N/A
-	C4	N/A
*	C5	N/A
/	C6	N/A
©	C7	N/A
AND	C8	N/A
OR	C9	N/A
>	CA	N/A
=	CB	N/A
<.	CC	N/A
USING	CD	N/A

The tokens which have a dispatch address 'N/A' are normally handled by other action routines and are not the first word of a BASIC statement. For example, the MOTOR action routine looks for OFF or ON, the FOR action routine looks for STEP and arithmetic and logical operators are handled by an expression evaluation routine.

This evaluation routine uses its own dispatch table to initiate actions for each operator and intrinsic function.

Operator/Function	Token	Dispatch address
+	C3	910E
-	C4	9105
*	C5	9275
/	C6	933C
©	C7	96A0
AND	C8	8A12
OR	C9	8A11
SGN	FF80	9425
INT	FF81	9499
ABS	FF82	943E
POS	FF83	9ADE
RND	FF84	9772
SQR	FF85	9697
LOG	FF86	923C
EXP	FF87	9713
SIN	FF88	97D1
COS	FF89	97CB
TAN	FF8A	9816
ATN	FF8B	9877
PEEK	FF8C	8E96
LEN	FF8D	8DC7
STR\$	FF8E	8C40

VAL	FF8F	8E5C
ASC	FF90	8DE6
CHR\$	FF91	8DD2
EOF	FF92	B801
JOYSTK	FF93	BB0D
FIX	FF94	9956
HEX\$	FF95	A00E
LEFT\$	FF96	8DF1
RIGHT\$	FF97	8E0E
MID\$	FF98	8E15
POINT	FF99	BA45
INKEY\$	FF9A	B797
MEM	FF9B	8C31
VARPTR	FF9C	9AF4
INSTR	FF9D	9BB4
TIMER	FF9E	9D59
PPOINT	FF9F	A6C7
STRING\$	FFA0	9B84
USR	FFA1	9D1D

3. I/O JUMP TABLES

As we have indicated in the text, system I/O routines may be accessed via a direct jump table starting at address 8000 and an indirect jump table at address A000. The routines accessible through the direct jump are listed below:

Address	Routine
8000	Hardware initialisation
8003	Software initialisation
8006	Keyboard input
8009	Cursor blinking
800C	Screen output
800F	Printer output
8012	Joystick input
8015	Cassette on
8018	Cassette off
801B	Write leader to cassette
801E	Byte output to cassette
8021	Cassette on for reading
8024	Byte input from cassette
8027	Bit input from cassette
802A	Read a byte from another computer
802D	Send a byte to another computer
8030	Select baud rate of communication line

The routines accessible via the indirect jump table are:

Address	Routine
A000	Keyboard input
A002	Character output routine

A004	Cassette on for reading
A006	Block input from tape
A008	Block output to tape
A00A	Joystick input
A00C	Write leader to cassette

Almost all of these routines have been described in the text. Those which we have not described are the three routines used when the Dragon is set up to communicate with another machine and support the DLOAD/DLOADM features of Extended Color BASIC. These features are not supported in the Dragon 32 but are available in the Dragon 64.

The character output routine which may be accessed through indirect jump address A002 is a general-purpose routine which can output a character to either the screen, the printer, or the cassette. The character to be output should be in register A and location 6F (DEVNUM) should be set up with a device number. To output to the screen, DEVNUM should be 0, to the cassette, DEVNUM should be -1 and if output to the printer is required, DEVNUM should be -2.

Appendix 8

The disk operating system

The internal workings of the Dragon's disk operating system (DOS) are too complicated for us to describe in detail here. Therefore, we restrict ourselves to a brief description of the differences between the normal Dragon 32 system requirements and the system requirements of the DOS.

1. DOS WORK SPACE

The Dragon's DOS reallocates the memory area normally occupied by the first graphics page, 600 to BFF, for its own use. This means that the graphics pages and BASIC program/variable areas are offset by 1536 (decimal) bytes from their normal position as shown in Figure 1.3. Fortunately, Extended Color BASIC has been designed with this in mind so it has no effect on existing BASIC programs other than reducing the free space available to them by 1.5K.

This 1.5K DOS work space is used for variables, buffers, etc. and a breakdown of its organisation is provided in the table below.

Address	Usage
600-604	Temporary storage
605-609	Controller variables
60A	Default drive number
60B:60C	Address of FWRITE buffer
60D:60E	AUTO, current line number
60F:610	AUTO, current increment
611	RUN/LOAD flag
612	FREAD/FLREAD flag
613	AUTO flag, 00=OFF, FF=ON
614	ERROR flag, 00=OFF, FF=ON
615:616	ERROR destination line number
617:618	Line number in ERROR
619	ERROR type
61A:61B	Address of start of statement in ERROR
61C-621	Drive 1 details
622-627	Drive 2 details
628-62D	Drive 3 details
62E-633	Drive 4 details
634-63A	Disk buffer 1 details
63B-641	Disk buffer 2 details

```

642--648   Disk buffer 3 details
649--64F   Disk buffer 4 details
650--682   Current drive information
683--696   USR vector table (relocated from 134--147)
697--6AA   Drive descriptor table
6Ab--6BC   Directory sector status
6BD--7F2   File control blocks (10)
            Each FCB is 32 bytes long
7F3--7FF   Temporary variables
800--BFF   Disk buffers (4)
            Each disk buffer is 256 bytes long

```

In addition to the workspace area 600 to BFF, the DOS makes use of some of the base page locations which are unused by Extended Color BASIC. These are:

Address	Use
EA	Disk command byte
EB	Drive unit number
EC	Track number
ED	Sector number
EE:EF	Address of disk buffer area
F0	Disk status byte
F1	Current file control block number
F2	Number of bytes in disk buffer area
F3	Number of bytes to transfer to/from buffer
F4	Record length flag, 00=don'tcare, FF=do care
F5	Read/write flag, 00=read, 01=write, FF=verify
F6	IRQ time out flag, 00=check for time out Non-00=skip time out check

2. DOS LINKS WITH BASIC

The Dragon's DOS extends the facilities of Extended Color BASIC using the same techniques as described in section 9.5. Most of these extended facilities are directly associated with floppy disk file management such as SAVE, LOAD, BACKUP, etc. However, there are a number of other commands which extend the BASIC itself such as AUTO, ERROR, GOTO, WAIT, etc. These disk and BASIC commands are described in detail in the booklet supplied with the DOS and we do not describe them further here. Rather, we concentrate on the details of how these new commands are linked to the existing BASIC system.

As we described in section 9.1.1, part of the initial power-on/reset sequence is to look for a disk controller cartridge and, if it is present, jump to its initialisation routine (entry point C002). The DOS initialisation routine sets up a second command interpretation vector table (stub) from 12A-133 inclusive with each entry point set up as shown in the table below.

Address	Contents	Use
12A	1A	Number of DOS reserved words
12B:12C	DED4	Address of DOS word list
12D:12E	C64C	Address of DOS word dispatch routine
12F	07	Number of DOS functions
130:131	DEBB	Address of functions list
132:133	C667	Address of functions dispatch routine

The third stub acts as the terminator and occupies RAM at 134-147. This means that the USR vector table which normally occupies that address space has to be relocated into the DOS workspace (683-696).

The following tables list the reserved words/functions of the DOS, their internal tokens and the addresses of associated action routines.

Reserved word	Token	Dispatch address
AUTO	CE	DADC
BACKUP	CF	C520
BEEP	DO	DB90
BOOT	D1	DC03
CHAIN	D2	D503
COPY	D3	D332
CREATE	D4	D725
DIR	D5	DA35
DRIVE	D6	DC3C
DSKINIT	D7	C397
FREAD	D8	D7FB
FWRITE	D9	D8A5
ERROR	DA	DC49
KILL	DB	D774
LOAD	DC	D4A7
MERGE	DD	D3E5
PROTECT	DE	D781
WAIT	DF	DBC1
RENAME	E0	D7A5
SAVE	E1	D53F
SREAD	E2	DC79
SWRITE	E3	DCC8
VERIFY	E4	DD36
FROM	E5	89B4
FLREAD	E6	D7C7
SWAP	E7	DBD5

Function	Token	Dispatch address
LOF	A2	DD88
FREE	A3	DDA3
ERL	A4	DDBB
ERR	A5	DDC1
HIMEM	A6	DDC7
LOC	A7	DD7A
FRE\$	A8	DDCB

In addition to these new reserved words and functions,

some of the existing BASIC commands such as CLOSE and RUN now relate to the DOS. Such commands are linked into the DOS via the RAM hooks described in section 9.5.1. There are several new entries in the RAM hooks as summarised in the table below.

Ram hook	Jump address
15E-160	D902
167-169	D8FA
176-178	D917
17C-17E	D960
182-184	D720
188-18A	DD4D
191-193	C69E
194-196	D490

Index

- A register, 22, 23
- A-D conversion, 220
- accumulator offset indexed
 - addressing, 34
- action routines, 229
- add instructions, 47
- alphanumeric display modes, 159
- analogue multiplexor, 208, 220
- and instructions, 52
- animation, 180
- animation delay, 182
- arithmetic expression
 - evaluation, 86
- arithmetic instructions, 46
- arithmetic shift
 - instructions, 54
- arrays, 107
- ASCII, 7
- assembler directives, 75
- assembler facilities, 69
- assembler memory map, 76
- assembler program counter, 77
- assembly code,
 - hand-translation, 39, 75
- assembly language program
 - development, 67
- assembly language
 - programming
 - advantages, 66
 - disadvantages, 66
- assignment statements, 84
- auto increment/decrement, 11, 24, 73
- auto increment/decrement
 - indexed addressing, 33
- B register, 22, 23
- BASIC
 - adding new commands, 239
 - array descriptors, 230
 - array storage, 229
 - dump program, 226
 - graphics display, 158
 - I/O routines, 146
 - information
 - representation, 230
 - largest number, 232
 - number representation, 230
 - program storage, 226
 - reserved words, 228
 - smallest number, 232
 - system variables, 244-248
 - text display, 152
 - variable storage, 229
 - variables, 234
- binary arithmetic, 4
- binary numbers, 3
- binary search, 220
- bit test instructions, 55
- BITIN, 216
- BLKIN, 216
- BLKOUT, 216
- branch instructions, 57
- BREAK key, disabling of, 244
- bus, 1, 37
 - connections, 222
- byte test instructions, 56
- byte, 4
- cartridge expansion port, 221
- CASOFF, 216

- CASON, 215
- cassette control, 213
- cassette routines, entry points, 217
- CBIN, 216
- CBOUT, 216
- Centronics interface, 205
- character input routine, 102
- character output routine, 103
- character representation, 7, 159
- character strings, 132
 - fundamental operations, 133
 - manipulation routines, 137-142
 - representation techniques, 134
 - storage area, 133
- checkerboard pattern, 162
- CHKHP, 137
- CHRGET, 240
- clear instructions, 49
- clear screen, 173
- CLOADM statement, 149
- clock, 1
- CLS, 173
- CMPSTR, 140
- cold-start initialisation, 225
- colour conversion program, 179
- colour graphics
 - 1 mode, 163
 - 2 mode, 163
 - 3 mode, 164
 - 6 mode, 164
 - display modes, 161
- colour set, 154, 162
 - selection, 172
- command interpretation vector
 - defining new, 241 (stub), 239
- comments field, 74
- compare instructions, 56
- complement arithmetic, 5, 6
- complement instructions, 53
- compound conditional expressions, 94
- computer architecture, 2
- condition code (CC) register, 11, 26, 40, 124
- conditional constructs, 89
- constant offset indexed addressing, 34
- COPY1B, 181
- COPY2B, 177
- CPSTR, 137
- CSRDON, 216
- CSS, 172
- cursor
 - addressing, 160
 - blinking, 160
- D register, 23
- DAC, 213, 220
- DACOUT, 210
- data highway, 1
- data movement instructions, 41
- decimal adjust instruction, 50
- decimal arithmetic, 8, 9
- decimal system, 3
- DELAY, 182
- digital-to-analogue conversion, 210
- direct addressing, 28, 72, 78, 147
- dispatch table, 229
- DP register, 25
- Dragon block diagram, 15
- Dragon logo, 175
 - animation, 182
 - colour data table, 180
 - data table, 176
 - flame data table, 181
 - grid pattern, 176
 - repetition, 178
- Dragon memory map, 17
- Dragon memory organisation, 16-19
- DREAM assembler, 70
- dynamic RAM, 16
- EQU directive, 77
- exchange instructions, 43
- EXEC statement, 148
- EXEC vector, 148
- extended addressing, 27, 72

- FCB/FCC directive, 78
- FDB directive, 79
- FIB, 131
- Fibonacci numbers, 130
- FILLER, 178
- FIRQ, 190
- floating point accumulator (FAC), 232
- for loops, 97
- FREE SP, 135, 136, 139
- frequency shift keying, 213
- FULL6R, 172

- garbage collection, 135
- general purpose register, 10
- GETSP, 135, 136, 138
- GIVABF, 232, 234
- GMODE, 171
- goto statements, 101
- graphics
 - display hardware, 152
 - hardware setup, 170
 - modes, 153
 - page selection, 173
 - segment size, 156
 - symbol design, 175
 - symbols, 174
 - test rig, 158
 - utilities, 169

- heap, 135
- hexadecimal notation, 7, 8
- hexadecimal system, 3
- high-level language programming, 65

- I/O hardware diagram, 187
- I/O programming techniques, 193
- Iliffe vectors, 109
- immediate addressing, 27, 72, 144
- index registers, 23-24
- index registers for array accessing, 108
- indexed addressing, 31, 72
- indirect addressing, 29, 30, 73
- input and output, 101
- instruction format, 26, 27, 74
- INTCNV, 232, 234
- integer representation, 5
- interrupt handling
 - instructions, 62, 191
- interrupt mask, 188
- interrupt priority, 189, 191
- interrupt processing, M6809, general, 189
- interrupt service routine, 188, 197
- interrupt vector, 188, 190, 193
- interrupt-driven I/O transfer, 195
- interrupts, 188
 - adding new service routines, 196
 - Dragon-specific, 192
 - software, 191
- INV control bit, 161
- IRQ, 189, 197
- IRQSET, 198

- JOYIN, 219
- joystick
 - buttons, 219
 - control, 218
 - I/O, 194
 - selection, 221
 - values, 219
- JOYSTK, 218
- jump instructions, 63
- jump tables, 80, 145
 - direct, 146
 - indirect, 146
 - initialisation of, 147

- keyboard
 - auto-repeat, 204
 - control, 201
 - matrix, 201
 - scanning, 202, 220, 244
 - state bytes, 203
- label field, 71
- load effective address instructions, 44

load instructions, 42
 local variables, 129
 logic instructions, 51
 logical shift instructions, 54
 LOGOTL, 177
 loop constructs, 96
 low-level language programming, 65
 LPOUT, 206

M6809 programming model, 21
 M6809 registers, 21
 machine code loader, 82
 machine code monitor, 111-120
 machine instructions, 11, 12, 38
 MAXMIN, 129
 memory addresses, notation, 41
 memory-mapped input/output, 36
 mnemonic field, 71
 MOVFM, 232, 235
 multi-armed conditionals, 93
 multiplication and division, 88
 multiply instruction, 49

negate instructions, 50
 NEWDSP, 240
 NEWSSET, 241
 NMI, 190
 no-operation instruction, 63
 number conversion, 232
 numerals, 3

operand field, 72
 optimisation, 68, 91, 93, 98, 100, 106
 or instructions, 52
 ORG directive, 80

PAGEX, 173
 parameter passing in registers, 123

using parameter area, 125
 using stack, 125
 PATGEN, 162
 PC-relative addressing, 143
 PCLS, 158
 PEEK and POKE, 88
 PIA, 155, 193, 198
 control lines, 199
 control register, 199
 data direction register, 199
 data register, 199
 equate table, 201
 PIAs, Dragon-specific, 200
 PMODE, 157
 polled I/O transfer, 194
 position-independent code, 35, 78, 142
 postbyte, 27, 31, 32
 power-up/reset actions, 224
 print routine, 243
 printer control, 205
 printer control lines, 205
 printer routine parameters, 206
 problem solving, 121
 processor architecture, 9
 program counter, 12
 PULL, 14
 pull instructions, 46
 PUSH, 14
 push instructions, 45
 PUT directive, 81

R32C0L, 174
 RAM hooks, 242-244
 read-only memory, 16
 real number representation, 231
 recursion, 125, 130, 131
 register addressing, 29, 72
 register, 9, 10
 relative addressing, 35
 relative branch instructions, 143
 repeat loops, 100
 RESET, 190
 resolution graphics

1 mode, 165
 2 mode, 165
 3 mode, 166

- 6 mode, 166
- modes, 164
- return instruction, 64
- returning results on the
 - stack, 128
- RMB directive, 80
- rotate instructions, 54

- SAM chip, 156
- SAM control register, 156
- SAMMOD, 170
- SAMSET, 172
- SCREEN, 157
- SEM18, 172
- Semigraphics
 - 4 mode, 167
 - 6 mode, 167
 - 8 mode, 168
 - 12 mode, 166
 - 24 mode, 166
 - character organisation, 167
 - display modes, 166
- shift instructions, 53
- sign bit, 5
- sign extend instruction, 50
- signed conditional branch
 - instructions, 59
- simple conditional branch
 - instructions, 59
- single-armed conditionals, 90
- SNDSEL, 209
- sound control, 207
- sound output generation, 211
- sound source selection, 209
- sound source, single bit, 212
- sound sources, 208
- special purpose register, 10
- SQUARE, 123, 126, 128
- stack, 12, 13
- stack frame, 126, 127
- stack pointer, 13
- stack pointer registers, 24, 25
- STINIT, 136
- store instructions, 43
- STRCAT, 140
- string descriptor, 233
- string representation, 233
- subroutines, 105, 121
 - assembly language, 122
 - call sequence, 129
 - disadvantages of BASIC, 121
 - entry/exit sequence, 129
 - parameter passing, 106
 - register conventions, 106
- SUBSTR, 141
- subtract instructions, 48
- synchronous address
 - multiplexor, 14

- tape file format, 214
- tape leader
 - format, 214
 - length, 215
- tape verification, 215
- test instructions, 55
- transfer instructions, 43
- two's complement, 6
- two-armed conditionals, 92
- two-dimensional arrays, 108, 110
- TXPCH, 207

- unconditional branch
 - instructions, 58
- unconditional I/O transfer, 194
- unsigned conditional branch
 - instructions, 60
- USR calls, 149, 235
 - bugs, 149
 - numeric parameters, 235
 - string parameters, 236
 - string results, 237
- USRASC, 236
- USRINC, 237

- VALTYP, 234
- VARPTR, 234
- VDG, 153, 159
 - control lines, 154, 155
- VDG/SAM addresses, 169

VDGMOD, 170	X register, 23
video RAM, 155	
	Y register, 23
warm-start initialisation, 225	
while loops, 99	zero offset indexed addressing, 34
word, 4	

Calling all DRAGON owners

If you have ever wanted to get more from your machine, but have been held back by a lack of documentation on its internal hardware and software, **INSIDE THE DRAGON** is the answer to your problem.

The authors have assembled in this book a wealth of information not previously available in one place. No one who wants to do more with their Dragon than merely play games can afford to be without it.

Among the topics covered are:

- * The architecture of the M6809 – the chip at the heart of the machine
- * programming the M6809
- * input/output hardware
- * graphics hardware
- * the Dragon 64
- * the disc operating system

Also included in the book are the manufacturer's data sheets for the M6809 processor, the SN74LS783 multiplexer, the MC6847 video display generator, and the MC6821 interface adaptor; invaluable aids to the serious user.

The Dragon is similar in many areas to the Tandy Color Computer so owners of that machine will find plenty in the book to interest them also.

GB £ NET +007.95

ISBN 0-201-14523-5



9 780201 145236



Addison-Wesley Publishing Company