Technosys Research Laboratories Ltd

Postal Address
P.O. BOX 37 349
PARNELL AUCKLAND 1
NEW ZEALAND

Head Office
36 CHESHIRE STREET
PARNELL AUCKLAND 1
NEW ZEALAND

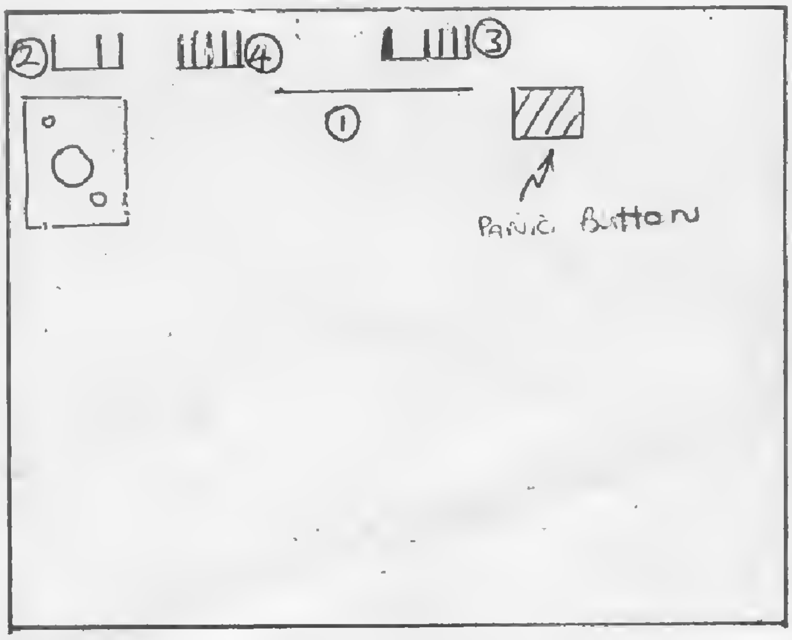
Telephone
798 304
AUCKLAND
(4 Lines)

HOW TO START YOUR PEGASUS

Check your equipment.

1 pegasus
1 power supply
1 R.F. modulator
1 keyboard
1 T.V.

Plug positions.



1. Keyboard plug, Plug in so metal chips face inwards.
2. Power supply plug . You can only plug this in one way.
3. Cassette recorder plug. You can plug this in one way only.
NOTE on play and record plugs, blue is microphone and yellow is ear.
4. RF modulator plug. You can plug this in one way only.

PLUGGING INTO TELEVISION.

1. Turn off television.
2. Remove aerial.
3. Replace with modulator connections.
4. Turn on television and turn to channel 3 or 4.
5. Turn on Pegasus.

6. Tune in television by the use of the fine tuner.
   If this does not work, try channels 1 to 4 on the television each in turn adjusting first the television fine tuner then by the use of a small flat screwdriver through the hole in the modulator,the modulator tuning.
   BEWARE THIS IS SENSITIVE.
7. Look for prompt. and menu.

Amber Pegasus 6809
TECHNOSYS RESEARCH LABORATORIES LTD.

8. Select by using the first letter of the item in the menu.

To select the menu type:
T for Tiny Basic
H for Hangman

G for Galaxy wars

M for Monitor
F for Forth
Controls of Galaxy Wars are:

F = up

K = Left

L = Right

9. To return to the Menu, hit the "Panic" button.

Handy Hint # 1

If you are unfamiliar with EPROMs the following information may be of use to you.

Firstly ensure that mains power is OFF before changing EPROM. Failure to do so may damage EPROM and/or Pegasus.

A screwdriver makes an excellent lever to gently pry the EPROM loose.

When replacing EPROMs note that they must all be aligned with the 'MON' (monitor EPROM , which should NEVER be removed).

Forth requires both sockets; Forth A in the socket that is farthest from 'MON' and Forth B in the middle socket.

Tiny BASIC, however, will work in either of the two sockets.

Aamber Pegasus - Introduction

Thank you for purchasing the Aamber Pegasus personal computer. We hope that you will have many hours of fun and learning from use of the computer. Experience with this, and other computers will enable you to better cope with the changing world that is ahead of us.

The machine that we are offering, while being approximately half the price of competitive products, offers much more capability in terms of expansion and ease of use. Initially we are supporting four languages with the Pegasus, these being ASSEMBLER, BASIC, FORTH and PASCAL. The computer can be expanded to 48K bytes of RAM, has cassettes for mass storage, and has the ability to interface to a wide range of peripherals, including environment controllers, which will be available from your dealer. The Pegasus is based on the popular Motorola 6809 microprocessor, which is generally considered to be the most powerful 8-bit micro in the world today. Because the processor is so versatile, the kit is suitable for people new to the computer world, as well as experienced users.

Once you have purchased the Pegasus, all you will require to expand the system for your further needs will be available from your local dealer, giving you the opportunity to let the system grow as your finances permit. We see the computer as being not only a starter kit, but also as the basis for a highly versatile and powerful computer system.

As part of our policy of continuing support for our products, you will receive a newsletter every six weeks, written by the same people who designed and made this computer a reality. The company is very interested in setting up interaction between the designers and the users, and thus will welcome correspondence, with the aim of receiving your suggestions and ideas on future add-ons for the Pegasus. Meritorious contributions will be published in the newsletter, the subscription for which is incorporated in the initial purchase price of the computer.

Now before using your computer, please read the manual! Enjoy yourselves, good computing.

[illegible]

Managing Director

Technosys Research Laboratories Ltd.

Introduction to Hardware

Welcome to the wonderful world of the 6809 microprocessor. The Pegasus uses the 6809 for two reasons. Firstly because of its simple but powerful structure (in terms of both hardware and software), and secondly because it is the most advanced microprocessor that is currently available for which there is an adequate range of supporting chips (hardware) and supporting software. We could, for instance, have used the 68000, or the Z8000, or even the 8086, all of which are more advanced processors, but none of these have adequate support (meaning that add-ons cost you very much more and take longer to become available.) Alternatively, we could have used one of the 'good old' breed of 6800, 8080, Z-80 or 6502 microprocessors, all of which have excellent support, but which in terms of today's technology may be best described as geriatric (all are in excess of five years old). This means that a product using these microprocessors is obsolete before it is even released, and that add-ons, although initially prolific, will rapidly cease to be forthcoming as everyone jumps on the new technology 'bandwagon' (as has already happened to the 8080 family).

Thus in chosing the 6809 for the Pegasus we have ensured that the product you invest your precious dollars in will not only be in the forefront of technology now, but will still be current enough in five years to have a continuing flow of add-ons and support.

This brings us to the primary design criterion behind Pegasus - <u>expandibility</u>. To date, most offerings of this type suffer one of two shortcomings. They are either cheap 'toys' which are rapidly outgrown and then discarded for next year's glossier model, or they are moderately sophisticated computers with a price tag beyond the reach of most home or school users. The toy becomes obsolete within a year and is hastily replaced by a new model (incompatible with last year's of course!) while the microcomputer may be expanded but starts at too high a basic cost.

Pegasus breaks new ground in this field by offering a simple low cost version to the hobbyist or school which may be expanded on four fronts:

(a) on the board (it starts out as a partially populated single board design).

(b) with extra boards (the case has room for three boards and any number may be added using the international standard SS-50 bus).

(c) as a plug-in to an existing SS-50 mainframe (i.e. as produced by Southwest Technical Products Ltd, or Gimix Inc. etc.)

(d) as a node in a large scale network linking up to 160 such units to a central microcomputer with fully shared resources (such a system will be released by Technosys Research Laboratories late in 1981).

In its basic form, the Pegasus offers:

- 1K bytes of RAM (560 reserved for video and system monitor)
- 32 characters by 0..16 lines (user selectable) video display
- full ASCII character set with Greek symbols (upper and lower case with descenders)
- 4K bytes EPROM containing System Monitor
- audio cassette interface (for home cassette recorder)
- individual character video inversion
- full ASCII keyboard interface (hex keypad also available)

- power supply (on-board regulator)
- RF modulator (for connection to any home TV set)
- large wire-wrap area for custom prototyping
- real-time clock calibrated in hours, minutes and seconds, with precision of 20 milliseconds
- user vectored interrupts

On board expansion permits:

- up to 4K bytes of RAM
- up to 12K bytes EPROM (20K modification also available)
- additional dual parallel port for user interfacing
- full SS-50 standard bus compatibility
- user programmable character set, giving 128 different characters in an 8 x 16 matrix, providing an effective on screen graphics resolution of 256 x 256 points.

Additional boards give:

- up to 48K bytes RAM
- floppy disk interface
- multiple parallel and serial interfaces
- special functions (speech synthesis, recognition, industrial control etc.)

The network system provides:

- up to 12 major nodes per central microcomputer sharing its disk, printer and processing resources
- each major node may be a dumb terminal, or have local intelligence (i.e. downloadable micro based unit)
- the intelligent nodes may be Aamber Pegasi units, or the forthcoming Aamber Intelligent Work Station (IWS) with high resolution graphics and self contained keyboard and screen

- any major node may be replaced by the Aamber network
  controller, which forms the root of a tree for up to 16
  additional units of the type described above. In this form
full terminal-terminal communications are supported, and
each Aamber Net may have local disk and printer resources
- see diagram

Thus the Pegasus may be used on its own or as part of a much larger network system, and a prospective network user (e.g. a school) may purchase a single unit to evaluate, and use it in a network mode later without penalty. This is true expandibility.

The Pegasus has been developed in New Zealand, by New Zealanders, for the New Zealand environment. No other computer, in any form, can make this claim. It was developed out of a need as expressed by people in industry, education and commerce for a versatile, expandible machine geared for the New Zealand situation. We feel that this computer not only meets these needs, but goes one step further. It was designed with the best features of currently available units in mind (i.e. the Apple, Pet, TRS-80, Superboard etc.), and we know that it is superior in capability and price terms to anything else on the market - there would have been no point in designing it, had it not been.

[illegible]

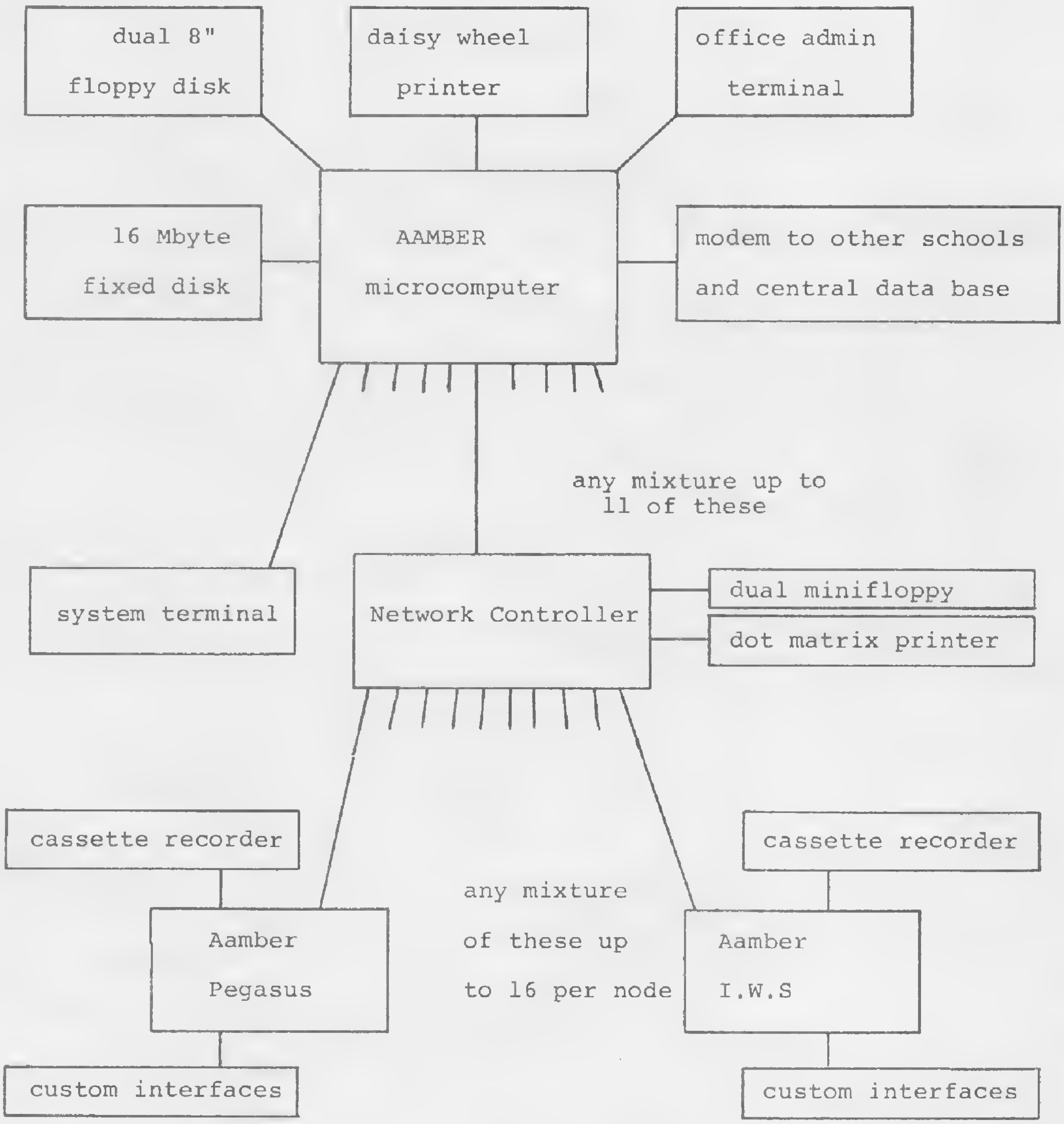
Director of Hardware Research
Technosys Research Laboratories Ltd
January 1981

Technosys Research Laboratories Ltd.

Aamber Network System

Large scale low cost implementation for educational use

(Note: equally applicable to commercial and industrial fields)



INTRODUCTION

Industrial Revolution

We are at the beginning of a new industrial revolution that is likely to have a greater impact on our way of life than any other single event in our industrial development. This revolution was created in the semiconductor industry, which is centred in Southern California, in an area affectionately known as 'Silicon Valley'. It is the microelectronic revolution.

Complex electronic circuits are now being constructed on tiny pieces of silicon (microelectronic 'chips') packaged in plastic or ceramic packages (dual-in-line or 'Dips'). From the humble beginnings of the germanium transistor in the late 1940's, we have advanced to the present day computer-on-a-chip, the microprocessor, containing the equivalent of up to 100,000 transistors.

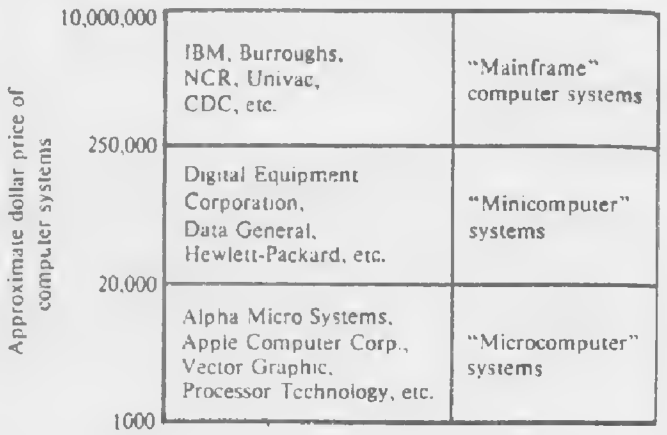
Evolution of Electronic Computers

DATE	DESIGNATION	SPECIAL DETAILS
1950	ENIAC	- entirely vacuum tubes - power hungry - unreliable - very expensive
1960	Beginning of semiconductor computers	
1965	PDP-8	- minicomputer - only $50,000 - moderate power consumption - applicable to laboratories and manufacturing production lines
1972	8080	- first microprocessor (microcomputer) - cheap - low power consumption - applicable to household applicances

1980 Advanced Microprocessor Components

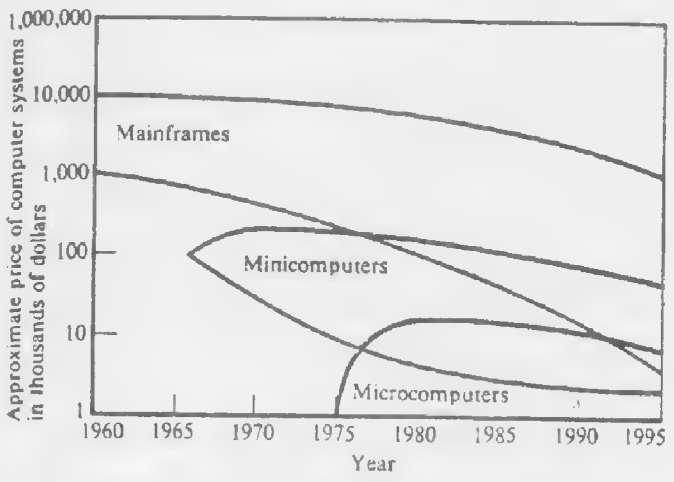
- low cost
- low power consumption
- rivaling conventional minicomputers in processing power

There is a three tier separation of computer products:


10,000,000
250,000
20,000
1000
Approximate dollar price of computer systems
IBM, Burroughs, NCR, Univac, CDC, etc.
"Mainframe" computer systems
Digital Equipment Corporation, Data General, Hewlett-Packard, etc.
"Minicomputer" systems
Alpha Micro Systems, Apple Computer Corp., Vector Graphic, Processor Technology, etc.
"Microcomputer" systems

The theoretical three tiers into which the computer industry can be divided. In reality, many products fall into two of the classifications, while continuous improvements keep altering products' classifications over time.

However, this price separation is misleading in terms of computer capabilities:



The computer industry divided into three tiers according to price. However, price can be misleading: it does not always reflect computing capability.

1969's $150,000 minicomputer is now equivalent to a $20,000 microcomputer and 1980's $150,000 minicomputer is equivalent to a mainframe. Mainframe, minicomputer and microcomputer are rapidly becoming labels only, with the internal electronics, and hence performance, rapidly blending. By 1990, it is estimated only 10% of computers will be in the mainframe category (e.g. future developments of Cray,) while the remaining 90% will be effectively microcomputers in physical size and cost (but not performance!!) i.e. central processing on one chip and only a handful of others for additional functions.

Differences between computer types are usually gauged by the following:

- speed	- time to execute instructions
- instruction type	- complexity of instructions
- physical size	- a chip or many boards
- cost	

with microcomputers traditionally scoring poorly in the first two, but extremely well in the latter two.

Consider the following analogy; one hundred years ago, the stagecoach was the principle mode of transport. It could travel at 25 mph carrying five passengers. Today, the Concorde can travel at 1300 mph carrying 200 passengers. In electronic terms, they are about the same because in only twenty years, electronic logic capacity has increased by a factor of 100,000 and logic speed by a factor of 1,000,000. Using the analogy, this would correspond to a Concorde carrying 500,000 passengers at 20,000,000 mph (costing 1c for an airline ticket).

Note that the microcomputer will never dominate all other computers because of the relentless need to make computers more powerful (for more advanced problems and ultimately intelligence). Consequently for every advance in microelectronic technology, two new products are produced:-

- a smaller cheaper version of 'yesterdays' computer
- a more powerful today's computer

Advances in computer science and programming coupled with those hardware advances can also produce breakthroughs into new types of computers, i.e. distributed and network processing.

Microprocessor Evolution

It was a case of accidential birth. The 'father', a firm called Datapoint wanted a computer-like chip to control a new product, the intelligent terminal. The 'mother', Intel, contracted to build it and succeeded. However, their product ran ten times slower than it was supposed to and Datapoint refused to buy it, leaving Intel quite literally holding the baby. Faced with shelving it or selling it, they called it the 8008 and put it on the market with no particular use in mind.
Overnight people realised they had a useful product and sales rocketed, thus the era of the microprocessor arrived.

All along, the industry has faced one substantial problem; how best to design and use these new chips. Advances in design techniques and circuit fabrication are constantly yielding more complex chips, but they are being designed for an unknown market and indeed they are creating the market as they are released. This is a hallmark of revolution, the unpredictable future. All the industry can do is design chips with features that it hopes will be desirable and then wait for sales. Nobody knows what the ideal microprocessor is or how it should be used and consequently, the microelectronics industry is undergoing a tremendous upheaval, and the pace is accelerating!

Microprocessor Family Tree

There are 3 major microprocessor manufacturers (excluding those like Texas who produce custom controller devices) -

$$\begin{tikzcd}
 & \underline{\text{ZILOG}} & \underline{\text{INTEL}} & \underline{\text{MOTOROLA}} & \\
\text{1972} & & 8008 \arrow[d] \arrow[dr, dashed] & & \\
\text{Time} \arrow[ddd] & \text{Z-80} \arrow[d] & 8080 \arrow[l, dashed] \arrow[d] & 6800 \arrow[d] & \text{\} original 3} \\
 & \text{Z-8} \arrow[d] & 8085 \arrow[d] & 6809 \arrow[d] & \text{\} second generation} \\
 & \text{Z8000} & 8086 & 68000 & \text{\} advanced 'minicomputer-like devices} \\
\text{1980} & & & &
\end{tikzcd}$$

It all began with the 8008 as previously described and Intel followed up with the 8080. Motorola and Zilog realised there existed a viable market and introduced their versions. From there, a natural evolution occurred to the present day 'minicomputer-like' microprocessors.
Many other companies are also involved (i.e. Texas, Fairchild, National, Synertek etc), but many of their original devices are not popular because of competition from the big 3.

A definite split is taking place in microprocessor architecture in terms of the size of information packets handled by the logic elements. The original 3 handled 8 binary bits, the second generation handled a mixture of 8 and 16 bits and the third generation handled 16 and 32 bit packets. Roughly speaking, computing power is proportional to the packet size ('word'). Main frames handle up to 64 bit words, while mini's are usually in the 12 to 32 bit range.

NOTE that the family tree is just a skeleton, and many more devices are produced by the big 3 and others which are microprocessors, but which are tailored for specific specialized roles (i.e. stand-alone controllers etc).

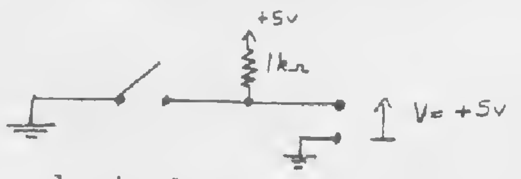
(Diagram of μP and its chips and packages etc on page 6)

NUMBER SYSTEMS

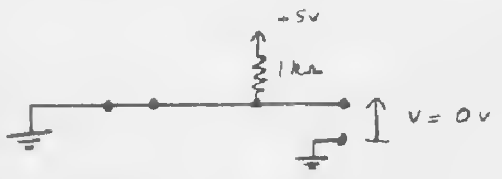
GENERAL CONCEPTS

A microprocessor is merely a glorified collection of logic gates and as such operates in exactly the same way as normal digital logic. All internal operations are in binary i.e. base 2, with two allowed states:

logic 1 = high = on = true
logic 0 = low = off = false
signified by the binary digits 1 and 0



logic 1



logic 0

however, our normal numbering system - decimal, has ten states signified by the digits 0-9 (base 10).

Large numbers are made up of multiples of these digits grouped together and weighted, i.e.

in decimal	13
means	1 lot of ten + 3 lots of one

This concept can be extended to the general case for any base and number by:

$$a_i b^i + \ldots + a_1 b^1 + a_0 b^0 + a_{-1} b^{-1} + \ldots\ldots + a_{-i} b^{-i}$$

where b = base
a = digit
i = digit position

i.e. consider 123 in decimal

this = $(1 * 10^2) + (2 * 10^1) + (3 * 10^0)$

similarly the same number in octal (base 8)

would = $(1 * 8^2) + (2 * 8^1) + (3 * 8^0)$

Computers always work internally in binary, but large base two numbers are extremely difficult for humans to visualize, i.e. 11010010 does not have immediately obvious significance.
Hence for convenience, we usually facilitate easy recognition and manipulation. At first glance, decimal would be the obvious choice, however, a little thought shows that binary to decimal conversion is

difficult and so we choose a base that is easily converted to and from binary,

e.g. octal or hexadecimal
(base 8) (base 16)

Binary	Octal	Decimal	hexadecimal
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F
10000	20	16	10

note: Hexadecimal weighting - x x x x

$16^3 16^2 16^1 16^0$

i.e. A F 6 E
is $(10 * 16^3) + (15 * 16^2) + (6 * 16^1) + (14 * 16^0)$ in base 10

Hexadecimal (base 16) is generally the base used for microcomputers as it offers easy binary to hexadecimal conversion and a very compact way of representing long binary numbers.

Base conversion

We have previously seen how to convert from any base to decimal by writing in the form -

$a_i b^i + \ldots.$

i.e A F in hexadecimal = $(10 * 16^1) + (10 * 16^0)$

= 10 + 16 + 15
= 175 in base 10

The reverse (decimal to another base) is accomplished similarly by breaking the decimal numbers up into multiples of b^i

i.e 16 in decimal = $2 * 8^1$
= 20 in octal

The generalized process is as follows:

> Write down the number and repeatedly divide by base to which it is to be converted, recording the remainder each time,
>
> e.g. for $(11)_{10}$ into binary -

Operation		Quotient		Remainder
11/2	=	5	+	1
5/2	=	2	+	1
2/2	=	1	+	0
1/2	=	0	+	1

therefore in binary = 1 0 1 1

This technique is just a failsafe method of determining a_i for each b_i.

Note, for conversion between bases niether of which are decimal, it's often easier (for humans) to convert to decimal first and then use the procedures above.

However, for conversion between binary and octal or hexadecimal, there exist quick methods, which can be seen by realizing that any octal digit can be defined by three unique binary digits, and any hexadecimal digit can be defined by four unique binary digits, i.e.

```
                        1 hex       1 hex      1 hex
                         digit       digit      digit
                    !           !           !           !
   binary digits -  x  x  x  x  x  x  x  x  x  x  x  x
                    !        !        !        !        !
                     1 octal 1 octal  1 octal  1 octal
                      digit   digit    digit    digit
```

```
such that       110111101001 in binary

can immediatly be written as (1101) (1110) (1001)
                               D      E      9 in hex
                        or   (110) (111) (101) (001)
                               6     7     5     1 in octal
```

Note that conversion to decimal as shown previously is painful in comparison.

Obviously the reverse process from octal or hex to binary is just as easily accomplished:

```
          hex to binary                        octal to binary
            F 3 C A                                7 3 4 6
  (1111) (0011) (1100) (1010)            (111) (011) (110) (110)
```

Hence the choice of hex or octal over decimal to represent large binary numbers in computer systems.

Note: The computer always works in binary alone - these and other conversions are solely for our convenience even in cases where they are done automatically by the computer.

Base Arithmetic

All normal arithmetic operations; addition, subtraction, multiplication and division can be performed in any number base, provided care is taken to do carrys and borrows correctly for the base in which you are working.

Addition

```
    Binary                       Hex

    10110                      F 3 D E
  + 01111                    + 7 5 A 1
   ------                    ---------
   100101                    1 6 9 7 F
   carry 2's                   carry 16's
```

Subtraction

```
    11101                      F A 3 0
  - 10110                    - E 1 9 D
   ------                     --------
    00111                      1 8 9 3
    borrow 2's                 borrow 16's
```

Multiplication

```
     1011                          F 2
   * 1101                        *  1 3
    -----                         -----
     1011                         2 D 6
    1011                          F 2 0
   1011
  --------                    ---------
  10001111                      1 1 F 6
```

Division

```
     101.1                        3 7.8
  10 1011.0                  3 F  D A 9.
     10                           B D
     ------                       -------
     0011                         1 D 9
       10                         1 B 8
     ----                         -------
      01                            2 1 0
       10                           2 1 0
      ---                         ------
       00                           0 0 0
```

You should try a few of these to ensure you understand how they work.

Two's Complement

It is difficult to produce hardware in computers to perform standard subtraction. Instead subtraction problems are first converted to an addition which can easily be done. The technique can be seen as follows:

for decimal 9 - 2 = 7

> but if the final carry is ignored 9 + 8 = 7 also
> and 8 is said to be the base complement (or tens
> complement of 2 since 8 + 2 = 10

Using this concept any subtraction can be performed as an addition by forming the base complement and adding (ignoring carry). The binary equivalent of this is called two's complement and the logic for this is extremely well suited for computers. The two's complement can easily be formed by the following process.

1. write down the desired no. to be two's complemented
2. change all the ones to zeros and all the zeros to ones
   - this is called the one's complement (inversion)
3. add 1 to the right most digit (least significant)

```
e.g.  minuend       11010     minuend         11010
      subtrahend  - 01001  or subtrahend    + 10111
      difference    -----     difference     ------
                    10001                    110001
                                           ignore carry
```

Note - That the answer can be positive or negative depending on whether the minuend or subrahend is larger. This is signified by the left most digit (most significant).

If it is a 1, then the answer is negative
if it is a 0, then the answer is positive

and the remaining digits represent its absolute value

```
   e.g.   011            011
        - 101            011
          ---            ---
          110            110
  borrow
  implying                   this is the absolute value of the
  answer negative            answer in 2's comp. form

                             digit is a 1 implying answer
                             negative, this is the sign of the
                             answer
                             (in decimal this gives 3 - 5 =-2)
```

This scheme yeilds the following range of numbers if an 8 digit binary value is used

2's comp. (hex equiv.)		Binary	Hex	Dec imal
0111 1111	(7 F)	0111 1111	7 F	+ 127
0111 1110	(7 E)	0111 1110	7 E	+ 126
0000 0001	(0 1)	0000 0001	0 1	+ 1
0000 0000	(0 0)	0000 0000	0 0	0
1111 1111	(F F)	-0000 0001	-0 1	1
1111 1110	(F E)	-0000 0010	-0 2	- 2
1000 0001	(8 1)	-0111 1111	-7 F	- 127
1000 0000	(8 0)	-1000 0000	-8 0	- 128

Thus the normal 8 digit binary gives a range in decimal from 0 to + 255 while 2's comp. binary gives a range in decimal from - 128 to + 127 (anything outside this range can be thought of as an overflow or underflow and is handled especially by the computer)

The beauty of this scheme (signed binary representation) is that it requires no special logic to take care of the sign; arithmetic operations simple proceed as normal, and the result will have the correct sign. Try this for yourself!

MICROPROCESSOR ORGANISATION

1. MEMORY ORGANISATION -

A computer processes information (data). All this data and the results of processing must be stored somewhere in memory. Since the computer works with binary data, the storage system must be bistable elements capable assuming one or two logic states on demand. They must furthermore have two essential properties -

1. Every stored binary digit must be held at some unique location. (otherwise it cannot be told apart from other digits).

2. Each location must be accessible to the computer so that it can read the information (or write in new information).

Microcomputers generally have two types of memory:

1. ROM - read only memory which cannot be changed (written) and is thus non-volatile.

2. RAM - random access memory which can be read or written and is volatile.

Note: RAM is a misleading name as both RAM and ROM are randomly accessable, i.e. any piece of information can be accessed at any time without going through a long list.

Computers do not merely operate on one digit at a time, but group them together into <u>words</u>, and all operations are performed on complete words. Individual binary digits are known as <u>bits</u>.

Word lengths generally range from 4 bits to 64 bits depending on the type of computer. With microprocessors, the most common size are 8 bit words, which are given the name <u>bytes</u>.

8 bit word = 1 byte

7 0

bit numbering

most significant bit (msb) least significant bit (lsb)

Larger microprocessors use 16 bit words i.e. 2 bytes long

high order byte low order byte

15 8 7 0

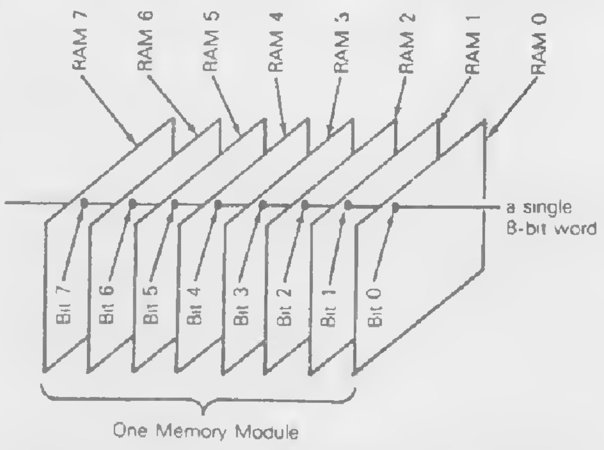
bit numbering

Note: 4 bit words are sometimes referred to as <u>nibbles</u>, but this is usually only when their name is explained

Since information is accessed in memory and since the computer requires information in word blocks, it follows that memories are organised into word lengths. At each accessible location in mammory (each address), there resides one word of information, and each word has a unique address.

This does not mean that memory is always manufactured in word lengths. Often memory is made with only one bit per location, so that the system designer can arrange a number of chips in parallel to give the word size he desires.

i.e.



For any one address 8 chips are accessed and one bit of the byte comes from each. In this manner, the same chips are just as suitable for 4 bit microcomputer memories as 16 bit memories.

The overall number of locations that a computer can access are collectively known as its <u>address space</u>. (65,536 for a typical microprocessor i.e. 2^{16})

However memory is rarely available at all of these locations. Sometimes the locations are used for other things and often the system only requires a small amount of memory, so this is physicaly located at the desired address, while the other addresses are vacant.

The amount of memory in a system is often referred to in 1024 location multiples; known as 1K blocks of memory.This corresponds to 10 binary bits being used for addresses, i.e. 2 to the power of 10 = 1024
Each different combination of the 10 bit binary number is used to specify a different address.

similarly	2048 is a 2K	(2^{11}) block
	16384 is a 16K	(2^{14}) block
and	65536 is a 64K	(2^{16}) block

Note: That each address within a block may contain a single bit or a whole word depending on how the microcomputer is organised

This is often denoted as follows:

1K x 1	means a 1024 block each location containing 1 bit
4K x 8	means a 4096 block each location containing 8 bits

2. DATA INTERPRETATION -

All information in memory is recalled and stored in the same way, and in the same form (binary). Thus, the microcomputer must have some way of interpreting the pattern of 0's and 1's to seperate the different types of data and instructions. Ways that a word can be interpreted are as follows:

a. pure numeric binary data
b. a data code subject to some arbitrary predefined interpretation (i.e. character code)
c. an instruction code, which the microprocessor can recognise as a command to perform some preset operation (e.g ADD)
d. as with (a), but part of a multiword data unit with some special weigting

1. Stand alone, pure binary data -
i.e an 8 bit word with decimal value from 0 - 255.

2. Interpreted binary data -
For 8 bits words, it may be necessary to interpret them in pairs to give 16 bit values, thus extending the range of numbers that can be represented from 255 to 65535. This can be extended to any lenght required. Signed binary numbers also fall into these catagories as discussed in appendix I. Another commonly used interpretation is binary coded decimal or BCD. this special representation uses 4 binary bits to represent each decimal digit (0 - 9),thus allowing easy manipulation of decimal information by binary computers.

Binary (BCD)		Decimal
0000		0
0001		1
0010		2
0011		3
0100		4
0101		5
0110		6
0111		7
1000		8
1001		9
1011		
:	not allowed	
1111		
0001	0000	10
0001	0001	11
	etc	

Note: the extra states corresponding to decimal 10 - 15 are not allowed

values in BCD representation.
Also that each byte contains two BCD digits (i.e. 00 - 99)

Although this representation makes storage and conversion easy it complicates arithmetic because BCD digits cannot be added and subtracted by normal binary rules.

3. Data Codes -

These are commonly used to allow the computer to store alphanumeric text rather than just numbers. This is accomplished by devising a code that assigns a distinct number in binary to each alphanumeric character.A standard code used is the American Standard Code for Information Interchange usuall refered to as ASCII (see appedix II), which is a seven bit code

i.e. $(41)_{16}$ - $(54)_{16}$

correspond to A and Z

Note: The 8th bit is usually used for an error checking procedure known as parity.

6809
PROGRAMMING
REFERENCE MANUAL

The following quick reference describes the sixty-seven instructions executed by the Motorola 6809 MPU. A few extra mnemonics have been added to clarify certain programming practices. The BZS mnemonics assembles into a BEQ, while the BZC assembles into a BNE instruction. The assembler will compute offsets for branch instructions and generate short or long branches as required relieving the programmer of the necessity to explicitly code long branches.

6809 Software Architecture

The 6809 microprocessor is a stack-oriented, one-address microprocessor containing two accumulators, four pointer registers, a direct page register, and a condition flag register. With the addition of more pointer registers and a powerful compliment of addressing modes, the 6809 is a major improvement over the 6800/6802 processors. Figure 1 is a programming model of the 6809.

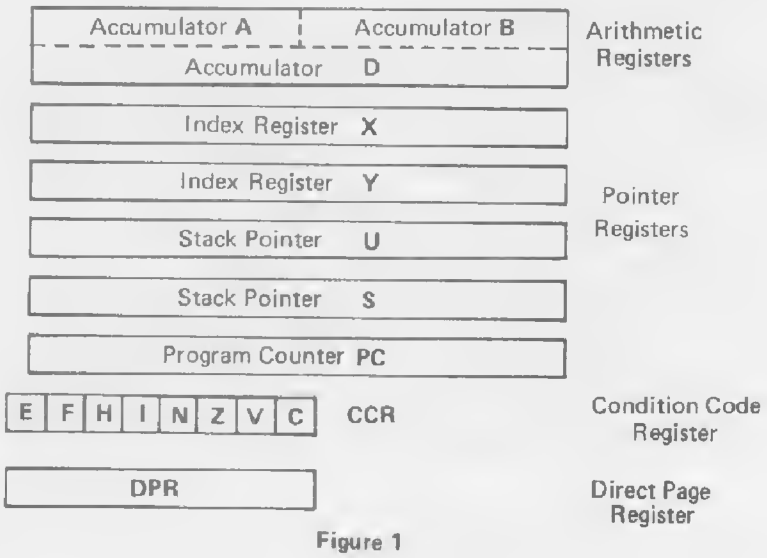


Figure 1

Arithmetic Registers

The 6809 has two 8-bit accumulators that are used to perform arithmetic and logic operations. For many operations the A and B accumulators can be treated like a single 16-bit accumulator, providing much improved performance in multiple-precision operations.

The 6809 performs all arithmetic operations in two's complement format.

Pointer Registers

The 6809 has four 16-bit pointer registers that are used as base address registers for indexed-mode addressing. The various combinations available with indexed mode addressing allows all four pointer registers to be used as explicit stack pointers. In addition, two pointer registers are also implicit stack pointers. The U and S registers have a series of PUSH and PULL instructions to facilitate stack programming. The S Stack pointer register is implicitly used by the 6809 hardware for subroutine calls and interrupts.

Program Counter

The 6809 maintains an internal 16-bit program counter. At any given time, the PC may be thought of as a pointer to the next instruction to be executed. Indexed addressing has two modes available that utilize the program coutner as a base address pointer.

Condition Code Register

The condition Code Register is conceptually an eight-bit register that contains the processor condition flags. The bit positions of the condition flags are shown in figures 1 and 2. A detailed description of each flag follows.

Bit Ø is the carry flag (C), and represents the binary carry-out from an arithmetic or shift type operation. For these operations this flag is an unsigned overflow. In general, load-type and logical operations do not effect the carry bit.

Bit 1 is the two's compliment overflow flag (V), and is set by an operation that causes two's compliment arithmetic overflow. Loads, stores, and logical operations clear V, while arithmetic operations set V appropriately.

Since all arithmetic operations are of limited precision (8 or 16 bits) it is possible to generate invalid results whrn performing arithmetic operations. For example, when performing an 8-bit addition, it is possible to add 75_{10} (01001011_2) to 85_{10} (01010101_2) and get the invalid result -96_{10} (10100000_2).

What has occurred is that the carry out of the most significant bit (the sign bit) is different from the carry into the sign bit, hence the sign (and the value) of the result is invalid. It is under these conditions that the two's compliment overflow flag is set. As another example, consider performing an arithmetic left shift on 96_{10} (01100000_2). The result is -64_{10} (11000000_2). Since the signed result is invalid, the V flag is set.

Bit 2 is the zero flag (Z) and is set whenever the result of an operation is zero. After compare operations, this bit represents the equal condition. Arithmetic, load, store, and logical operations set this bit appropriately.

Bit 3 is the sign (N) and is set whenever the most significant bit of the result is set. For arithmetic operations this flag is set if valid negative two's compliment result is obtained. Note that two's compliment branches use both the N and V bits, and the branch is taken whether the result is valid or not.

Bit 4 is the IRQ mask bit (I). The processor will not recognize IRQ interrupts if this bit is set. The IRQ interrupt acknowledge sequence sets the I bit to mask subsequent IRQ requests until the IRQ service routine completes. An RTI instruction will restore the state of the I flag from the stack.

Bit 5 is the half-carry bit (H). This bit is used after 8-bit add operations to indicate the carry out of bit3 in the arithmetic unit. This bit is used by the DAA instruction to perform packed decimal (BCD) add adjust. In general, the H flag state is undefined after non-add -perations, and add-type instructions on 16-bit quantities.

Bit 6 is the FIRQ interrupt mask bit (F). This bit affects the FIRQ interrupt in the same manner that the I bit affects IRQ.

Bit 7 is the Entire State flag (E), and is used by the RTI instruction to determine how much of the machine state to load from the stack. Two saved states are defined: the entire state (E = 1) in which all registers are pushed on the stack, and the subset state (E = Ø) in which only the program counter and the condition flags are pushed on the stack. In general the state of the E flag is undefined except after an interrupt.

Figure 2

bit Ø-	C	-Carry Flag
bit 1-	V	-Two's Compliment Overflow Flag
bit 2-	Z	-Zero Flag
bit 3-	N	-Sign Flag
bit 4-	I	-IRQ Mask Flag
bit 5-	H	-Half-carry Flag
bit 6-	F	-FIRQ Mask Flag
bit 7-	E	-Entire State Flag

Direct Page Register

The Direct Page Register (DPR) is an 8-bit register that is used to provide significant 8 bits of the 16 bit address generated by instructions using direct addressing. This register is initialized to zero at RESET time.

MEMORY ADDRESSING MODES

INHERENT

EXAMPLE: MUL

Inherent addressing includes those instructions which have no addressing options.

ACCUMULATOR

EXAMPLE: CLRA, CLRB

Accumulator addressing includes those instructions which operate on an accumulator.

ABSOLUTE

EXAMPLE: LDA $8ØØ4

Absolute addressing refers to an exact 16-bit location in the memory address space, and is especially useful for transactions with peripherals (I/O).

There are three program-selectable modes of absolute addressing, namely: Direct, Extended, and Extended Indirect. Certain instructions (SW1,SW2,SW3), and the interrupts, use an inherent absolute address to function similarly to Extended Indirect mode addressing. These instructions are said to have "Absolute Indirect" addressing.

DIRECT

EXAMPLE LDA CAT

Direct addressing uses the immediate byte of the instruction as a one-byte pointer into a single 256-byte "page" of memory. (The term "page" refers to one of the 256 possible combinations of the high-order address bits.) The particular page in use is fixed by loading the Direct Page Register with the desired high-order byte (by transferring from or exchanging with another register.) Thus, the effective address consists of a high-order byte (from the Direct Page Register) catenated with a low-order byte (from the instruction).

This mode may allow economies of both program space and execution time as compared to other absolute or indexed modes.

EXTENDED

EXAMPLE: LDA CAT

Extended addressing uses a 16-bit immediate value (and thus contained in the two bytes following the last byte of the op code) as the exact memory address value.

EXTENDED INDIRECT

EXAMPLE: LDA ($FØØØ)

Extended indirect addressing uses a 16-bit immediate value as an absolute address from which to recover the effective address.

This mode is inherently used by interrupts to vector to the handling routine; and may be used to create vector tables in a customized system which allow the use of standard software packages.

Although Extended Indirect is a logical extension of Extended addressing, this mode is implemented using an encoding of the post-byte under the indexed addressing group.

REGISTER

EXAMPLE: TFR X,Y

Register addressing refers to the selection of various on-board registers.

INDEXED

The 6809 includes extremely powerful indexing capabilities. There are five indexable registers (X,Y,S,U, AND PC) with many options (constant-offset, accumulator offset using A,B, or D, auto-increment or decrement or indirection.) These options are selected by complex coding of the first byte after the op code byte(s) of indexed-mode instructions. Most 6800 indexed-mode instructions will map into an equivalent two bytes on the 6809.

CONSTANT-OFF SET INDEXED

EXAMPLE: LDA Ø,X

Constant-offset indexing uses an optional two's complement offset contained ineither the post byte of the instruction as a bit-field or as a immediate value. This offset may be an absolute quantity, a symbol, or an expression and may range from zero to a 16-bit binary value which may be specified either positive or negative with an absolute value less or equal to 2^{16}. The offset is temporarily added to the pointer value from the selected register (X,Y,U,S, OR PC); the result is the effective address which points into memory.

A number of hardware modes are available to reduce the number of instruction bytes for various options. The majortiy of 6800 indexed-mode instructions will still need only two bytes on the 6809.

The notation THERE, PCR causes the assembler to compute the relative destance between the location of the symbol THERE elsewhere inthe program counter. The computed value is used as an immediate value in the instruction, indexed from the program-counter. This notation is painlessly position-independent.

Becuase a 16-bit offset is allowed, the (necessarily absolute) address of the indexable data may be carried as a constant value in the indexing instructions. This would allow the "index register" to be simultaneously used for indexing and counting using LEA.

CONSTANT-OFFSET INDEXED INDIRECT

EXAMPLE: LDA (Ø,X)

Constant-offset indexed indirect addressing functions in two stages (like all indirects). First an indexed address is formed by temporarily adding the offset-value contained in the addressing byte(s) to the value from the selected pointer register (X,Y,S,I, or PC). Second, this address is used to recover a two-byte absolute pointer which is used as the "effective address".

This mode allow the programmer to use a "table of pointers" data structure, or to do I/0 through absolute values stored on the stack.

ACCUMULATOR INDEXED

EXAMPLE: LDA A,X

Accumulator-indexed indirect addressing uses an accumulator (A,B, or D) as a two's complement offset which is temporarily added to the value from the selected pointer register (X,Y,S, or U) to form the effective address.

ACCUMULATOR INDEXED INDIRECT

EXAMPLE: LDA (A,X)

Accumulator-indexed indirect addressing uses an accumulator (A,B, or D) as a two's complement offset which is temporarily added to the value from the selected pointer register (X,Y,S, or U). The resulting pointer is then used to recover another pointer from memory (thus, the indirect designation) which is then used as the effective address.

AUTO-INCREMENT

EXAMPLE: LDA Ø,X+ LDX Ø,X++

Auto-increment addressing uses the value in the selected pointer register (X,Y,S or U) to address a one or two byte value in memory. The register is then incremented by one

AUTO-INCREMENT Cont -

(single +) or two (two +'s). No offset is permitted (a constant offset of Ø is enforced].

AUTO-INCREMENT INDIRECT

EXAMPLE: LDA (Ø,X++)

Auto-increment indirect addressing uses the value in the selected pointer register (X,Y,S, or U) to recover an address value from memory. This value is used as the effective address. The register is then incremented by two (++); the indirected increment by one is invalid. No offset is permitted (a constant offset of Ø is enforced).

AUTO-DECREMENT

EXAMPLE: LDA Ø,-X LDX Ø,--X

Auto-decrement addressing first decrements the selected pointer register (X,Y,D, or U) by one (-) or two (--) as selected by the user. The resulting value is then used as the effective address. No offset is permitted (a constant offset of Ø is enforced).

AUTO-DECREMENT INDIRECT

EXAMPLE: LDA (Ø,--X)

Auto-decrement indirect addressing first decrements the selected pointer register by two (--). Auto-decrement by one indirect is prohibited in the assembly language. The resulting value is used to recover a pointer value from memory; this value is used as the effective address. No offset is permitted (a constant offset of Ø is enforced).

RELATIVE
EXAMPLE: BRA POLE

(Short) Relative addressing adds the value of the immediate byte of the instruction (an 8-bit two's complement value) to the value of the program counter to produce an absolute address. This addressing mode is always postion-independent.

LONG RELATIVE
EXAMPLE: BRA CAT

Long Relative addressing adds the value of the immediate bytes of theinstructions (a 16-bit two's complement value) to the value of the program counter to produce an absolute address. This addressing mode is always position-independent.

SWTPC 68/09 QUICK REFERENCE

ABX	Add ACCB into IX
DESCRIPTION:	Add the 8-bit unsigned value in Accumulator B into the X index register.
CONDITION CODES:	Not Affected
ADDRESSING MODE:	Inherent

ADC	Add With Carry
DESCRIPTION	Adds the carry flag and the memory byte into an 8-bit register.
CONDITION CODE:	H,N,Z,V,C
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

ADD	Add Without Carry
DESCRIPTION:	Adds memory into register.
CONDITION CODE:	H,N,Z,V,C
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended
AND	Logical AND
DESCRIPTION:	Performs the logical "AND" operation between the contents of a register and the contents of memory.
CONDITION CODES:	N,Z,V
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended
ANDCC	Logical AND Into Condition Code Register
DESCRIPTION:	Performs a logical "AND" between the condition register and the immedate byte and places the result in the condition code register.
CONDITION CODES:	The fianl state of the condition codes is governed by the immediate code register.
ADDRESSING; MODES:	Immediate
ASL	Arithmetic Shift Left
DESCRIPTION	Shifts all bits of the operand one place to the left. Bit Ø is loaded with a zero. Bit 7 of the operand is shifted into the carry flag.
CONDITION CODES:	N,Z,V,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended

ASR	Arithmetic Shift Right
DESCRIPTION:	Shifts all bits of the operand right one place. Bit 7 is held constant.
CONDITION CODES:	N,Z,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended
BCC	Branch on Carry Clear
DESCRIPTION:	Tests the state of the C bit and causes a branch if C is clear.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative
BCS	Branch on Carry Set
DESCRIPTION:	Tests the state of the C bit and causes a branch if C is set.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative
BEQ	Branch on Equal
DESCRIPTION:	Used after a subtract or compare operation, this instruction will branch if the register is equal to the memory operand.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BGE	Branch on Greater or Equal
DESCRIPTION:	Used after a subtract or compare operation on signed binary values, this instruction will branch if the register was greater than or equal to the memory operand,
CONDITION CODES	
CODES:	Not Affected
ADDRESSING	
MODES:	Releative, Long Relative

BGT	Branch on Greater
DESCRIPTION:	Used after a subtract or compare operation on signed binary values, this instruction will branch if the register was greater than the memory operand.
CONDITION	
CODES:	Not Affected
ADDRESSING	
MODES:	Relative, Long Relative

BHI	Branch if Higher
DESCRIPTION:	Used after a subtract or compare operation on unsigned binary values this instruction will branch if the register was higher than the memory operand.
CONDITION	
CODES:	Not Affected
ADDRESSING	
MODES:	Relative, Long Relative

BHS	Branch if Higher or Same
DESCRIPTION:	When used after a subtract or compare on unsigned binary values, this instruction will branch if register was higher than or same as the memory operand.
CONDITION	
CODES:	Not Affected
ADDRESSING	
MODES:	Relative, Long Relative

BLE	Branch on Less or Equal
DESCRIPTION:	Used after a subtract or compare operation on signed binary values, this instruction will branch if the register was less than or equal to the memory operand.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long relative

BLO	Branch on Lower
DESCRIPTION:	When used after a subtract or compare on unsigned binary values, this instruction will branch if the register was lower than the memory operand.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BLS	Branch on Lower or Same
DESCRIPTION:	Used after a subtract or compare operation signed binary values, this instruction will branch if the register was lower than or the same as the memory operand.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BLT	Branch on Less
DESCRIPTION:	Used after a subtract or compare operation on signed binary values, this instruction will branch if the register was less than the memory operand.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BIT	Bit Test
DESCRIPTION:	Performs the logical "AND" of the contents of a register and the contents of memory and modifies condition codes accordingly. The contents of the register are not affected.
CONDITION CODES:	N,Z.V
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

BMI	Branch on Minus
DESCRIPTION:	Used after an operation on signed binary values, this instruction will branch if the result is negative.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BNE	Branch Not Equal
DESCRIPTION:	Used after a subtract or compare operation, this instruction will branch if the register is not equal to the memory operand.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BPL	Branch on Plus
DESCRIPTION:	Used after an operation signed binary values, this instruction will branch if the result is positive.
CONDITION CODES:	Not Affected
ADDRESSING CODES:	Relative, Long Relative

BRA	Branch
DESCRIPTION:	Causes an unconditional branch
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BRN	Branch never
DESCRIPTION:	Does not cuase branch. This instruction is essentially a NO-OP.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BSR	Branch to Subroutine
DESCRIPTION:	The program counter is pushed onto the stack, and control is passed to the subroutine.
CONDITION CODES:	Not affected
ADDRESSING MODES:	Relative, Long Relative

BVC	Branch on Overflow Clear
DESCRIPTION:	Tests the state of the V bit and causes a branch if the V bit is set
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BVS	Branch on Overflow Set
DESCRIPTION:	Tests the state of the V bit and causes a branch if the V bit is clear.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative
BZC	Branch on Zero Clear
DESCRIPTION:	Tests the state of the Z bit and cuases a branch if the Z bit is clear.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative
BZS	Branch on Zero Set
DESCRIPTION:	Tests the state of the Z bit and causes a branch if the Z bit is set.
CLR	Clear
DESCRIPTION:	The register or memory is loaded with ØØØØØØØØ. The C-flag is cleared for 6800 compatibility.
CONDITION CODES:	N,Z,V,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended
CMP	Compare Memory to a Register
DESCRIPTION:	Compares the contents of M to the contents of the specified register and sets appropriate condition codes.
CONDITION CODES:	N,Z,C,V
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

COM	Complement
DESCRIPTION:	Replaces the contents of a register or memory with its one's complement. The carry flag is set for 6800 compatibility.
CONDITION CODES:	N,Z,V,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended
CWAI	Clear and Wait for Interrupt
DESCRIPTION:	The CWAI instruction ANDs and immediate byte with the condition code register (which may clear interrupt masks), stacks the entire machine state on the hardware stack then looks for an interrupt. When a (non-maks) interrupt occurs, no further machine states will be saved before vectoring to the interrupt handling routine.
CONDITION CODE:	Possibly Cleared by the immediate byte.
ADDRESSING MODE:	Immediate
DAA	Decimal Addition Adjust
DESCRIPTION:	This instruction can be used after the addition of two binary-coded decimal numbers to insure that the result is in the proper binary-coded decimal format, and that the carry bit is set correctly. This instruction should be used after an ADD or an ADC instruction, with the result held in the A register.
CONDITION CODES:	N,Z,C
ADDRESSING MODE:	Inherent

DEC	Decrement
DESCRIPTION:	Subtract one from the operand. The carry flag is not affected, thus allowing DEC to be a loop-counter in multiple-precision computations.
CONDITION CODES:	N,Z,V
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended

JMP	Jump
DESCRIPTION:	Program control is transferred to the effective address.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Direct, Indexed, Extended

JSR	Jump to Subroutine
DESCRIPTION:	Program control is transferred to the Effective Address after storing the return address on the system stack.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Direct, Indexed, Extended

LD	Load Register from Memory
DESCRIPTION:	Load the contents of the addressed memory into the register.
CONDITION CODES:	N,Z,V
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

LEA Load Effective Address

DESCRIPTION: Form the effective address to data using the memory addressing mode. Load that address, not the data itself into the pointer register.

CONDITION CODES: LEAX and LEAY affect Z to allow use as counters and for 6800 INX/DEX compatibility. LEAU and LEAS do not affect Z to allow for cleaning up to the stack while returning Z as a parameter to a calling routine, and for 6800 INS/DES compatibility.

ADDRESSING MODE: Indexed.

LSL Logical Shift Left

DESCRIPTION: Shifts all bits of the operand one place to the left. Bit Ø is loaded with a zero. Bit 7 is shifted into the carry flag.

CONDITION CODES: N.Z,V,C

ADDRESSING MODES: Accumulator, Direct, Indexed, Extended

LSR Logical Shift Right

DESCRIPTION: Perfroms a logical shift right on the operand. Shifts a zero into bit 7 and bitØ into the carry flag.

LSR Cont:-

CONDITION CODES:	N,Z,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended
MUL	Multiply Accumulators
DESCRIPTION:	Multiply the unsigned binary numbers in the A and B registers and place the result in the D register.
CONDITION CODES:	Z,C
ADDRESSING MODES:	Inherent
NEG	Negate
DESCRIPTION:	Replaces the operand with its two's complement. Note that 80_{16} is replaced by itself and only in this case is V set. The value $ØØ_{16}$ is also replaced by itself and only in this case is C cleared.
CONDITION CODES:	N,Z,V,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended
NOP	No Operation
DESCRIPTION:	This is a single-byte instruction that causes only the program counter to be incremented. No other registers or memory contents are affected.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

OR	Inclusive OR
DESCRIPTION:	Perfroms an "Inclusive OR" operation between the contents of a register and the contents of memory and the result is stored in the register.
CONDITION CODES:	N,Z,V
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended
ORCC	Inclusive OR into condition code register
DESCRIPTION:	Perfroms an "Inclusive OR" operation between the condition code register and the immediate byte and the result is placed in the condition code register. This instruction may be used to Set interrupt masks.
CONDITION CODES:	The final state of the condition codes is determined by the immediate byte specified.
ADDRESSING MODE:	Immediate
PHSH	Push Registers on the System Stack
DESCRIPTION:	Any subset of the MPU registers are pushed onto the system stack.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Register
PSHU	Push Registers on the User Stack
DESCRIPTION:	Any subset of the MPU registers are pushed onto the user stack.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Register

ROL	Rotate Left
DESCRIPTION:	Rotate all bits of the operand one place left through the carry flag; this is a nine-bit rotation.
CONDITION CODES:	N,Z,V,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended
ROR	Rotate Right
DESCRIPTION:	Rotates all bits of the operand right one place through the carry flag; this is a nine-bit rotation.
CONDITION CODES:	N,Z,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended
RTI	Return from Interrupt
DESCRIPTION:	The saved machine state is recovered from the hardware stack and control is returned to the interrupted program.
CONDITION CODES:	Recovered from Stack
ADDRESSING MODES:	Inherent

RTS	Return from Subroutine
DESCRIPTION:	Program control is returned from the subroutine to the calling program. The return address is pulled from the stack.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Inherent
SBC	Subtract with Borrow
DESCRIPTION:	Subtracts the contents of memory and the borrow from the contents of a register, and places the result in that register.
CONDITION CODES:	N,Z,V,C
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended
SEX	Sign Extended
DESCRIPTION:	This instruction transforms a signed binary eight-bit value in the B register into a signed binary sixteen-bit value in the D register.
CONDITION CODES:	N,Z
ADDRESSING MODES:	Inherent
ST	Store Register Into Memory
DESCRIPTION:	Writes the contents of an MPU register into a a memory location.
CONDITION CODES:	N,Z,V
ADDRESSING MODES:	Direct, Indexed, Extended

SUB	Subtract Memory from Register
DESCRIPTION:	Subtracts the value in memory from the contents of a register.
CONDITION CODES:	N,Z,V,C
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

SWI	Software Interrupt
DESCRIPTION:	All of the MPU registers are pushed onto the hardware stack and control is transferred through the SWI vector.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Inherent

SWI2	Software Interrupt 2
DESCRIPTION:	All of the MPU registers are pushed onto the hardware stack and control is transferred through the SWI2 vector.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Inherent

SWI3	Software Interrupt 3
DESCRIPTION:	All of the MPU registers are pushed onto the hardware stack and control is transferred through the SWI3 vector.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Inherent

SYNC	Synchronize to External Event
DESCRIPTION:	When a SYNC instruction is executed, the MPU enters a SYNCING state, stops processing instructions, and waits on an interrupt. When an interrupt occurs, the SYNCING state is cleared and processing continues. If the interrupt is enabled, the processor will perform the interrupt routine. If the interrupt is masked, the processor simply continues to the next instruction.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Inherent

TFR	Transfer Register to Register
DESCRIPTION:	Transfer a Source to a destination register. Registers may only be transferred between registers of like size; i.e., 8-bit to 8-bit, and 16 ti 16.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Register

TST	Test
DESCRIPTION:	Set condition code flags N and Z according to the contents of the operand and clear the V flag.
CONDITION CODES:	N,Z,V
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended

6809 STACKING ORDER

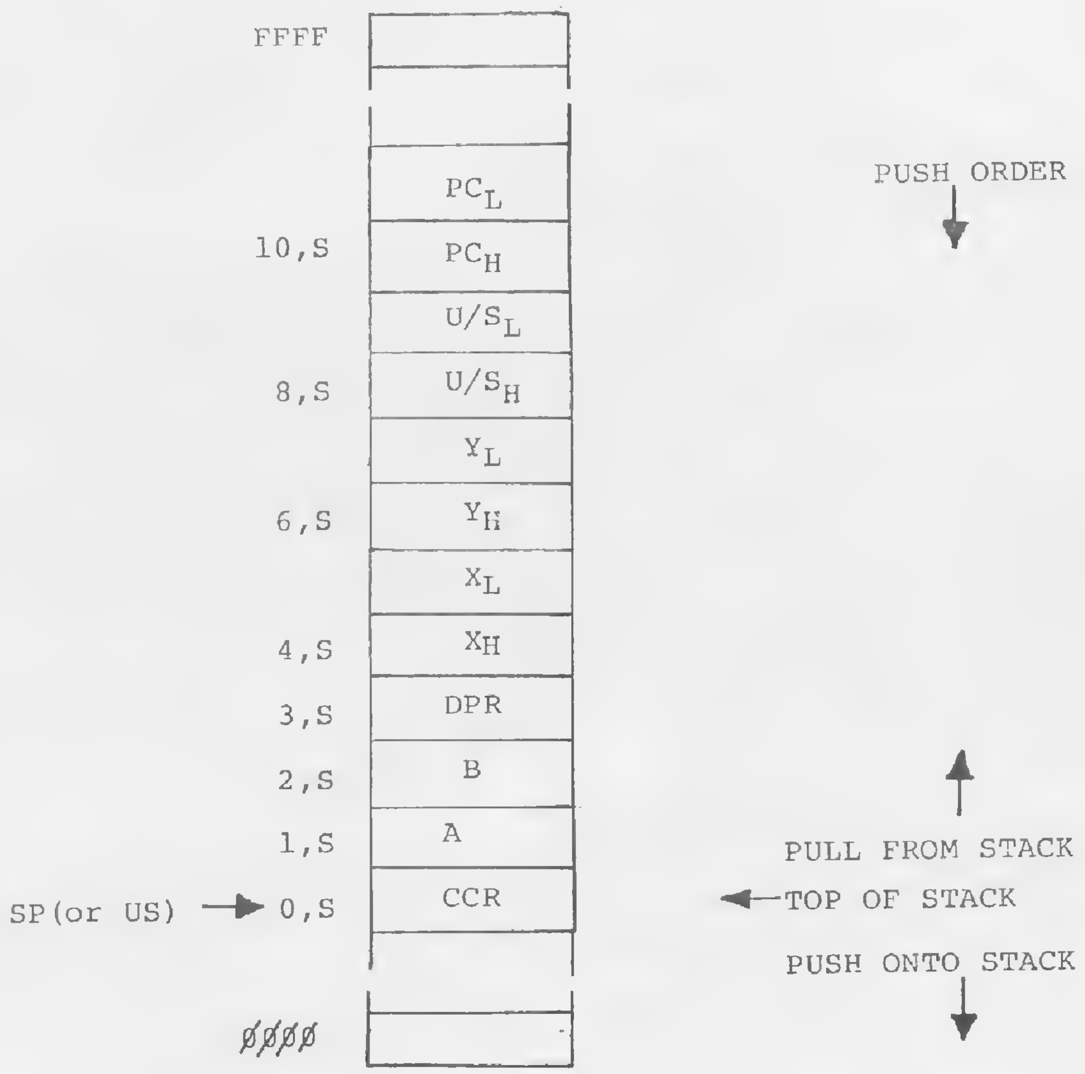


Figure 7: 6809 Push/Pull and Interrupt Stacking Order

SOFTWARE INCOMPATABILITIES WITH 6800/6802

1. The new stacking order on the 6809 exchanges the order of ACCA and ACCB; this allowa ACCA to stack as the MS byte of the pair.

2. The new stacking length on the 6809 invalidates previous 6800 code which displayed IX or PC from the stack.

3. Additional stacking length on the 6809 stacks five more bytes for each NMI, IRQ, or SWI when compared to 6800/6802.

4. The 6809 stack pointer points directly at the last item placed on the stack, instead of the location before the lsot itme as in 6800/6802. In general this is not a problem since the most-usual method of pointing at the stack in the 6800/6802 is to execute a TSX. The TSX increments the value during the transfer, making X point directly at the last item on the stack.

   The stack pointer may thus be initialized one location higher on the 6809 than in the 6800/6802; similarly, comparison values may need to be one location higher.

   Any 6800/6802 program which does all stack manipulation through X (i.e., LDX #CAT, TXS instead of LDS #CAT) will have an exactly correct stack translation when assembled for 6809.

5. The 6809 uses the two high-order condition cose register bits. Consequently, these will not, in general, appear as 1's as on the 6800/6802.

6. The 6809 TST instruction does not affect the C flag, while 6800/6802 TST does clear the flag.

SOFTWARE IMCOMPATABILITIES WITH 6800/6802

7. The 6809 right shifts (ASR,LSR,ROR) do not affect V; the 6800/6802 shifts set V = $b_7 + b_6$.

8. The 6809 H-flag is not defined as having any particular state after subtract-like operations (CMP, NEG, SBC, SUB); the 6809/6800 clear the H-flag under these conditions.

9. The 6800/6802 CPX instruction compared MS byte, then LS byte; consequently only the Z-flag was set correctly for branching. The 6809 instruction CMPX set all flags correctly.

10. The 6809 instruction LEA may or may not affect the Z-flag depending upon which register is being loaded; LEAX and LEAY do affect the Z-flag while LEAS abd LEAU do not. Thus, the User Stack does not emulate the index registers in this respect.

6809
PROGRAMMING
REFERENCE MANUAL

The following quick reference describes the sixty-seven instructions executed by the Motorola 6809 MPU. A few extra mnemonics have been added to clarify certain programming practices. The BZS mnemonics assembles into a BEQ, while the BZC assembles into a BNE instruction. The assembler will compute offsets for branch instructions and generate short or long branches as required relieving the programmer of the necessity to explicitly code long branches.

6809 Software Architecture

The 6809 microprocessor is a stack-oriented, one-address microprocessor containing two accumulators, four pointer registers, a direct page register, and a condition flag register. With the addition of more pointer registers and a powerful compliment of addressing modes, the 6809 is a major improvement over the 6800/6802 processors. Figure 1 is a programming model of the 6809.

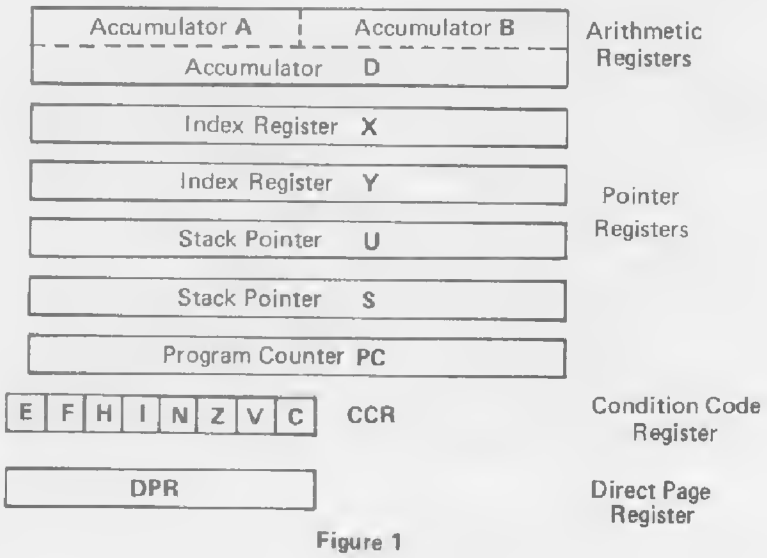


Figure 1

Arithmetic Registers

The 6809 has two 8-bit accumulators that are used to perform arithmetic and logic operations. For many operations the A and B accumulators can be treated like a single 16-bit accumulator, providing much improved performance in multiple-precision operations.

The 6809 performs all arithmetic operations in two's complement format.

Pointer Registers

The 6809 has four 16-bit pointer registers that are used as base address registers for indexed-mode addressing. The various combinations available with indexed mode addressing allows all four pointer registers to be used as explicit stack pointers. In addition, two pointer registers are also implicit stack pointers. The U and S registers have a series of PUSH and PULL instructions to facilitate stack programming. The S Stack pointer register is implicitly used by the 6809 hardware for subroutine calls and interrupts.

Program Counter

The 6809 maintains an internal 16-bit program counter. At any given time, the PC may be thought of as a pointer to the next instruction to be executed. Indexed addressing has two modes available that utilize the program coutner as a base address pointer.

Condition Code Register

The condition Code Register is conceptually an eight-bit register that contains the processor condition flags. The bit positions of the condition flags are shown in figures 1 and 2. A detailed description of each flag follows.

Bit Ø is the carry flag (C), and represents the binary carry-out from an arithmetic or shift type operation. For these operations this flag is an unsigned overflow. In general, load-type and logical operations do not effect the carry bit.

Bit 1 is the two's compliment overflow flag (V), and is set by an operation that causes two's compliment arithmetic overflow. Loads, stores, and logical operations clear V, while arithmetic operations set V appropriately.

Since all arithmetic operations are of limited precision (8 or 16 bits) it is possible to generate invalid results whrn performing arithmetic operations. For example, when performing an 8-bit addition, it is possible to add 75_{10} (01001011_2) to 85_{10} (01010101_2) and get the invalid result -96_{10} (10100000_2).

What has occurred is that the carry out of the most significant bit (the sign bit) is different from the carry into the sign bit, hence the sign (and the value) of the result is invalid. It is under these conditions that the two's compliment overflow flag is set. As another example, consider performing an arithmetic left shift on 96_{10} (01100000_2). The result is -64_{10} (11000000_2). Since the signed result is invalid, the V flag is set.

Bit 2 is the zero flag (Z) and is set whenever the result of an operation is zero. After compare operations, this bit represents the equal condition. Arithmetic, load, store, and logical operations set this bit appropriately.

Bit 3 is the sign (N) and is set whenever the most significant bit of the result is set. For arithmetic operations this flag is set if valid negative two's compliment result is obtained. Note that two's compliment branches use both the N and V bits, and the branch is taken whether the result is valid or not.

Bit 4 is the IRQ mask bit (I). The processor will not recognize IRQ interrupts if this bit is set. The IRQ interrupt acknowledge sequence sets the I bit to mask subsequent IRQ requests until the IRQ service routine completes. An RTI instruction will restore the state of the I flag from the stack.

Bit 5 is the half-carry bit (H). This bit is used after 8-bit add operations to indicate the carry out of bit3 in the arithmetic unit. This bit is used by the DAA instruction to perform packed decimal (BCD) add adjust. In general, the H flag state is undefined after non-add -perations, and add-type instructions on 16-bit quantities.

Bit 6 is the FIRQ interrupt mask bit (F). This bit affects the FIRQ interrupt in the same manner that the I bit affects IRQ.

Bit 7 is the Entire State flag (E), and is used by the RTI instruction to determine how much of the machine state to load from the stack. Two saved states are defined: the entire state (E = 1) in which all registers are pushed on the stack, and the subset state (E = Ø) in which only the program counter and the condition flags are pushed on the stack. In general the state of the E flag is undefined except after an interrupt.

Figure 2

bit Ø-	C	-Carry Flag
bit 1-	V	-Two's Compliment Overflow Flag
bit 2-	Z	-Zero Flag
bit 3-	N	-Sign Flag
bit 4-	I	-IRQ Mask Flag
bit 5-	H	-Half-carry Flag
bit 6-	F	-FIRQ Mask Flag
bit 7-	E	-Entire State Flag

Direct Page Register

The Direct Page Register (DPR) is an 8-bit register that is used to provide significant 8 bits of the 16 bit address generated by instructions using direct addressing. This register is initialized to zero at RESET time.

MEMORY ADDRESSING MODES

INHERENT

EXAMPLE: MUL

Inherent addressing includes those instructions which have no addressing options.

ACCUMULATOR

EXAMPLE: CLRA, CLRB

Accumulator addressing includes those instructions which operate on an accumulator.

ABSOLUTE
EXAMPLE: LDA $8ØØ4

Absolute addressing refers to an exact 16-bit location in the memory address space, and is especially useful for transactions with peripherals (I/O).

There are three program-selectable modes of absolute addressing, namely: Direct, Extended, and Extended Indirect. Certain instructions (SW1,SW2,SW3), and the interrupts, use an inherent absolute address to function similarly to Extended Indirect mode addressing. These instructions are said to have "Absolute Indirect" addressing.

DIRECT
EXAMPLE LDA CAT

Direct addressing uses the immediate byte of the instruction as a one-byte pointer into a single 256-byte "page" of memory. (The term "page" refers to one of the 256 possible combinations of the high-order address bits.) The particular page in use is fixed by loading the Direct Page Register with the desired high-order byte (by transferring from or exchanging with another register.) Thus, the effective address consists of a high-order byte (from the Direct Page Register) catenated with a low-order byte (from the instruction).

This mode may allow economies of both program space and execution time as compared to other absolute or indexed modes.

EXTENDED
EXAMPLE: LDA CAT

Extended addressing uses a 16-bit immediate value (and thus contained in the two bytes following the last byte of the op code) as the exact memory address value.

EXTENDED INDIRECT

EXAMPLE: LDA ($FØØØ)

Extended indirect addressing uses a 16-bit immediate value as an absolute address from which to recover the effective address.

This mode is inherently used by interrupts to vector to the handling routine; and may be used to create vector tables in a customized system which allow the use of standard software packages.

Although Extended Indirect is a logical extension of Extended addressing, this mode is implemented using an encoding of the post-byte under the indexed addressing group.

REGISTER

EXAMPLE: TFR X,Y

Register addressing refers to the selection of various on-board registers.

INDEXED

The 6809 includes extremely powerful indexing capabilities. There are five indexable registers (X,Y,S,U, AND PC) with many options (constant-offset, accumulator offset using A,B, or D, auto-increment or decrement or indirection.) These options are selected by complex coding of the first byte after the op code byte(s) of indexed-mode instructions. Most 6800 indexed-mode instructions will map into an equivalent two bytes on the 6809.

CONSTANT-OFF SET INDEXED

EXAMPLE: LDA ∅,X

Constant-offset indexing uses an optional two's complement offset contained ineither the post byte of the instruction as a bit-field or as a immediate value. This offset may be an absolute quantity, a symbol, or an expression and may range from zero to a 16-bit binary value which may be specified either positive or negative with an absolute value less or equal to 2^{16}. The offset is temporarily added to the pointer value from the selected register (X,Y,U,S, OR PC); the result is the effective address which points into memory.

A number of hardware modes are available to reduce the number of instruction bytes for various options. The majortiy of 6800 indexed-mode instructions will still need only two bytes on the 6809.

The notation THERE, PCR causes the assembler to compute the relative destance between the location of the symbol THERE elsewhere inthe program counter. The computed value is used as an immediate value in the instruction, indexed from the program-counter. This notation is painlessly position-independent.

Becuase a 16-bit offset is allowed, the (necessarily absolute) address of the indexable data may be carried as a constant value in the indexing instructions. This would allow the "index register" to be simultaneously used for indexing and counting using LEA.

CONSTANT-OFFSET INDEXED INDIRECT

EXAMPLE: LDA (Ø,X)

Constant-offset indexed indirect addressing functions in two stages (like all indirects). First an indexed address is formed by temporarily adding the offset-value contained in the addressing byte(s) to the value from the selected pointer register (X,Y,S,I, or PC). Second, this address is used to recover a two-byte absolute pointer which is used as the "effective address".

This mode allow the programmer to use a "table of pointers" data structure, or to do I/0 through absolute values stored on the stack.

ACCUMULATOR INDEXED

EXAMPLE: LDA A,X

Accumulator-indexed indirect addressing uses an accumulator (A,B, or D) as a two's complement offset which is temporarily added to the value from the selected pointer register (X,Y,S, or U) to form the effective address.

ACCUMULATOR INDEXED INDIRECT

EXAMPLE: LDA (A,X)

Accumulator-indexed indirect addressing uses an accumulator (A,B, or D) as a two's complement offset which is temporarily added to the value from the selected pointer register (X,Y,S, or U). The resulting pointer is then used to recover another pointer from memory (thus, the indirect designation) which is then used as the effective address.

AUTO-INCREMENT

EXAMPLE: LDA Ø,X+ LDX Ø,X++

Auto-increment addressing uses the value in the selected pointer register (X,Y,S or U) to address a one or two byte value in memory. The register is then incremented by one

AUTO-INCREMENT Cont -

(single +) or two (two +'s). No offset is permitted (a constant offset of Ø is enforced].

AUTO-INCREMENT INDIRECT

EXAMPLE: LDA (Ø,X++)

Auto-increment indirect addressing uses the value in the selected pointer register (X,Y,S, or U) to recover an address value from memory. This value is used as the effective address. The register is then incremented by two (++); the indirected increment by one is invalid. No offset is permitted (a constant offset of Ø is enforced).

AUTO-DECREMENT

EXAMPLE: LDA Ø,-X LDX Ø,--X

Auto-decrement addressing first decrements the selected pointer register (X,Y,D, or U) by one (-) or two (--) as selected by the user. The resulting value is then used as the effective address. No offset is permitted (a constant offset of Ø is enforced).

AUTO-DECREMENT INDIRECT

EXAMPLE: LDA (Ø,--X)

Auto-decrement indirect addressing first decrements the selected pointer register by two (--). Auto-decrement by one indirect is prohibited in the assembly language. The resulting value is used to recover a pointer value from memory; this value is used as the effective address. No offset is permitted (a constant offset of Ø is enforced).

RELATIVE	
EXAMPLE:	BRA POLE
	(Short) Relative addressing adds the value of the immediate byte of the instruction (an 8-bit two's complement value) to the value of the program counter to produce an absolute address. This addressing mode is always postion-independent.
LONG RELATIVE	
EXAMPLE:	BRA CAT
	Long Relative addressing adds the value of the immediate bytes of theinstructions (a 16-bit two's complement value) to the value of the program counter to produce an absolute address. This addressing mode is always position-independent.

SWTPC 68/09 QUICK REFERENCE

ABX	Add ACCB into IX
DESCRIPTION:	Add the 8-bit unsigned value in Accumulator B into the X index register.
CONDITION CODES:	Not Affected
ADDRESSING MODE:	Inherent
ADC	Add With Carry
DESCRIPTION	Adds the carry flag and the memory byte into an 8-bit register.
CONDITION CODE:	H,N,Z,V,C
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

ADD	Add Without Carry
DESCRIPTION:	Adds memory into register.
CONDITION CODE:	H,N,Z,V,C
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended
AND	Logical AND
DESCRIPTION:	Performs the logical "AND" operation between the contents of a register and the contents of memory.
CONDITION CODES:	N,Z,V
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended
ANDCC	Logical AND Into Condition Code Register
DESCRIPTION:	Performs a logical "AND" between the condition register and the immedate byte and places the result in the condition code register.
CONDITION CODES:	The fianl state of the condition codes is governed by the immediate code register.
ADDRESSING; MODES:	Immediate
ASL	Arithmetic Shift Left
DESCRIPTION	Shifts all bits of the operand one place to the left. Bit Ø is loaded with a zero. Bit 7 of the operand is shifted into the carry flag.
CONDITION CODES:	N,Z,V,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended

ASR	Arithmetic Shift Right
DESCRIPTION:	Shifts all bits of the operand right one place. Bit 7 is held constant.
CONDITION CODES:	N,Z,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended
BCC	Branch on Carry Clear
DESCRIPTION:	Tests the state of the C bit and causes a branch if C is clear.
CONDITION CODES:	Not Affected.
ADDRESSING MODES:	Relative, Long Relative
BCS	Branch on Carry Set
DESCRIPTION:	Tests the state of the C bit and causes a branch if C is set.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative
BEQ	Branch on Equal
DESCRIPTION:	Used after a subtract or compare operation, this instruction will branch if the register is equal to the memory operand.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BGE	Branch on Greater or Equal
DESCRIPTION:	Used after a subtract or compare operation on signed binary values, this instruction will branch if the register was greater than or equal to the memory operand,
CONDITION CODES	
CODES:	Not Affected
ADDRESSING	
MODES:	Releative, Long Relative
BGT	Branch on Greater
DESCRIPTION:	Used after a subtract or compare operation on signed binary values, this instruction will branch if the register was greater than the memory operand.
CONDITION	
CODES:	Not Affected
ADDRESSING	
MODES:	Relative, Long Relative
BHI	Branch if Higher
DESCRIPTION:	Used after a subtract or compare operation on unsigned binary values this instruction will branch if the register was higher than the memory operand.
CONDITION	
CODES:	Not Affected
ADDRESSING	
MODES:	Relative, Long Relative
BHS	Branch if Higher or Same
DESCRIPTION:	When used after a subtract or compare on unsigned binary values, this instruction will branch if register was higher than or same as the memory operand.
CONDITION	
CODES:	Not Affected
ADDRESSING	
MODES:	Relative, Long Relative

BLE	Branch on Less or Equal
DESCRIPTION:	Used after a subtract or compare operation on signed binary values, this instruction will branch if the register was less than or equal to the memory operand.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long relative

BLO	Branch on Lower
DESCRIPTION:	When used after a subtract or compare on unsigned binary values, this instruction will branch if the register was lower than the memory operand.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BLS	Branch on Lower or Same
DESCRIPTION:	Used after a subtract or compare operation signed binary values, this instruction will branch if the register was lower than or the same as the memory operand.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BLT	Branch on Less
DESCRIPTION:	Used after a subtract or compare operation on signed binary values, this instruction will branch if the register was less than the memory operand.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BIT	Bit Test
DESCRIPTION:	Performs the logical "AND" of the contents of a register and the contents of memory and modifies condition codes accordingly. The contents of the register are not affected.
CONDITION CODES:	N,Z.V
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

BMI	Branch on Minus
DESCRIPTION:	Used after an operation on signed binary values, this instruction will branch if the result is negative.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BNE	Branch Not Equal
DESCRIPTION:	Used after a subtract or compare operation, this instruction will branch if the register is not equal to the memory operand.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BPL	Branch on Plus
DESCRIPTION:	Used after an operation signed binary values, this instruction will branch if the result is positive.
CONDITION CODES:	Not Affected
ADDRESSING CODES:	Relative, Long Relative

BRA	Branch
DESCRIPTION:	Causes an unconditional branch
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative
BRN	Branch never
DESCRIPTION:	Does not cuase branch. This instruction is essentially a NO-OP.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative
BSR	Branch to Subroutine
DESCRIPTION:	The program counter is pushed onto the stack, and control is passed to the subroutine.
CONDITION CODES:	Not affected
ADDRESSING MODES:	Relative, Long Relative
BVC	Branch on Overflow Clear
DESCRIPTION:	Tests the state of the V bit and causes a branch if the V bit is set
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative

BVS	Branch on Overflow Set
DESCRIPTION:	Tests the state of the V bit and causes a branch if the V bit is clear.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative
BZC	Branch on Zero Clear
DESCRIPTION:	Tests the state of the Z bit and cuases a branch if the Z bit is clear.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Relative, Long Relative
BZS	Branch on Zero Set
DESCRIPTION:	Tests the state of the Z bit and causes a branch if the Z bit is set.
CLR	Clear
DESCRIPTION:	The register or memory is loaded with ØØØØØØØØ. The C-flag is cleared for 6800 compatibility.
CONDITION CODES:	N,Z,V,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended
CMP	Compare Memory to a Register
DESCRIPTION:	Compares the contents of M to the contents of the specified register and sets appropriate condition codes.
CONDITION CODES:	N,Z,C,V
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

COM	Complement
DESCRIPTION:	Replaces the contents of a register or memory with its one's complement. The carry flag is set for 6800 compatibility.
CONDITION CODES:	N,Z,V,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended
CWAI	Clear and Wait for Interrupt
DESCRIPTION:	The CWAI instruction ANDs and immediate byte with the condition code register (which may clear interrupt masks), stacks the entire machine state on the hardware stack then looks for an interrupt. When a (non-maks) interrupt occurs, no further machine states will be saved before vectoring to the interrupt handling routine.
CONDITION CODE:	Possibly Cleared by the immediate byte.
ADDRESSING MODE:	Immediate
DAA	Decimal Addition Adjust
DESCRIPTION:	This instruction can be used after the addition of two binary-coded decimal numbers to insure that the result is in the proper binary-coded decimal format, and that the carry bit is set correctly. This instruction should be used after an ADD or an ADC instruction, with the result held in the A register.
CONDITION CODES:	N,Z,C
ADDRESSING MODE:	Inherent

DEC	Decrement
DESCRIPTION:	Subtract one from the operand. The carry flag is not affected, thus allowing DEC to be a loop-counter in multiple-precision computations.
CONDITION CODES:	N,Z,V
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended

JMP	Jump
DESCRIPTION:	Program control is transferred to the effective address.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Direct, Indexed, Extended

JSR	Jump to Subroutine
DESCRIPTION:	Program control is transferred to the Effective Address after storing the return address on the system stack.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Direct, Indexed, Extended

LD	Load Register from Memory
DESCRIPTION:	Load the contents of the addressed memory into the register.
CONDITION CODES:	N,Z,V
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

LEA — Load Effective Address

DESCRIPTION: Form the effective address to data using the memory addressing mode. Load that address, not the data itself into the pointer register.

CONDITION CODES: LEAX and LEAY affect Z to allow use as counters and for 6800 INX/DEX compatibility. LEAU and LEAS do not affect Z to allow for cleaning up to the stack while returning Z as a parameter to a calling routine, and for 6800 INS/DES compatibility.

ADDRESSING MODE: Indexed.

LSL — Logical Shift Left

DESCRIPTION: Shifts all bits of the operand one place to the left. Bit Ø is loaded with a zero. Bit 7 is shifted into the carry flag.

CONDITION CODES: N.Z,V,C

ADDRESSING MODES: Accumulator, Direct, Indexed, Extended

LSR — Logical Shift Right

DESCRIPTION: Perfroms a logical shift right on the operand. Shifts a zero into bit 7 and bitØ into the carry flag.

LSR Cont:-

CONDITION CODES:	N,Z,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended
MUL	Multiply Accumulators
DESCRIPTION:	Multiply the unsigned binary numbers in the A and B registers and place the result in the D register.
CONDITION CODES:	Z,C
ADDRESSING MODES:	Inherent
NEG	Negate
DESCRIPTION:	Replaces the operand with its two's complement. Note that 80_{16} is replaced by itself and only in this case is V set. The value $ØØ_{16}$ is also replaced by itself and only in this case is C cleared.
CONDITION CODES:	N,Z,V,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended
NOP	No Operation
DESCRIPTION:	This is a single-byte instruction that causes only the program counter to be incremented. No other registers or memory contents are affected.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

OR	Inclusive OR
DESCRIPTION:	Perfroms an "Inclusive OR" operation between the contents of a register and the contents of memory and the result is stored in the register.
CONDITION CODES:	N,Z,V
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended
ORCC	Inclusive OR into condition code register
DESCRIPTION:	Perfroms an "Inclusive OR" operation between the condition code register and the immediate byte and the result is placed in the condition code register. This instruction may be used to Set interrupt masks.
CONDITION CODES:	The final state of the condition codes is determined by the immediate byte specified.
ADDRESSING MODE:	Immediate
PHSH	Push Registers on the System Stack
DESCRIPTION:	Any subset of the MPU registers are pushed onto the system stack.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Register
PSHU	Push Registers on the User Stack
DESCRIPTION:	Any subset of the MPU registers are pushed onto the user stack.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Register

ROL	Rotate Left
DESCRIPTION:	Rotate all bits of the operand one place left through the carry flag; this is a nine-bit rotation.
CONDITION CODES:	N,Z,V,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended
ROR	Rotate Right
DESCRIPTION:	Rotates all bits of the operand right one place through the carry flag; this is a nine-bit rotation.
CONDITION CODES:	N,Z,C
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended
RTI	Return from Interrupt
DESCRIPTION:	The saved machine state is recovered from the hardware stack and control is returned to the interrupted program.
CONDITION CODES:	Recovered from Stack
ADDRESSING MODES:	Inherent

RTS	Return from Subroutine
DESCRIPTION:	Program control is returned from the subroutine to the calling program. The return address is pulled from the stack.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Inherent
SBC	Subtract with Borrow
DESCRIPTION:	Subtracts the contents of memory and the borrow from the contents of a register, and places the result in that register.
CONDITION CODES:	N,Z,V,C
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended
SEX	Sign Extended
DESCRIPTION:	This instruction transforms a signed binary eight-bit value in the B register into a signed binary sixteen-bit value in the D register.
CONDITION CODES:	N,Z
ADDRESSING MODES:	Inherent
ST	Store Register Into Memory
DESCRIPTION:	Writes the contents of an MPU register into a a memory location.
CONDITION CODES:	N,Z,V
ADDRESSING MODES:	Direct, Indexed, Extended

SUB	Subtract Memory from Register
DESCRIPTION:	Subtracts the value in memory from the contents of a register.
CONDITION CODES:	N,Z,V,C
ADDRESSING MODES:	Immediate, Direct, Indexed, Extended

SWI	Software Interrupt
DESCRIPTION:	All of the MPU registers are pushed onto the hardware stack and control is transferred through the SWI vector.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Inherent

SWI2	Software Interrupt 2
DESCRIPTION:	All of the MPU registers are pushed onto the hardware stack and control is transferred through the SWI2 vector.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Inherent

SWI3	Software Interrupt 3
DESCRIPTION:	All of the MPU registers are pushed onto the hardware stack and control is transferred through the SWI3 vector.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Inherent

SYNC	Synchronize to External Event
DESCRIPTION:	When a SYNC instruction is executed, the MPU enters a SYNCING state, stops processing instructions, and waits on an interrupt. When an interrupt occurs, the SYNCING state is cleared and processing continues. If the interrupt is enabled, the processor will perform the interrupt routine. If the interrupt is masked, the processor simply continues to the next instruction.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Inherent
TFR	Transfer Register to Register
DESCRIPTION:	Transfer a Source to a destination register. Registers may only be transferred between registers of like size; i.e., 8-bit to 8-bit, and 16 ti 16.
CONDITION CODES:	Not Affected
ADDRESSING MODES:	Register
TST	Test
DESCRIPTION:	Set condition code flags N and Z according to the contents of the operand and clear the V flag.
CONDITION CODES:	N,Z,V
ADDRESSING MODES:	Accumulator, Direct, Indexed, Extended

6809 STACKING ORDER

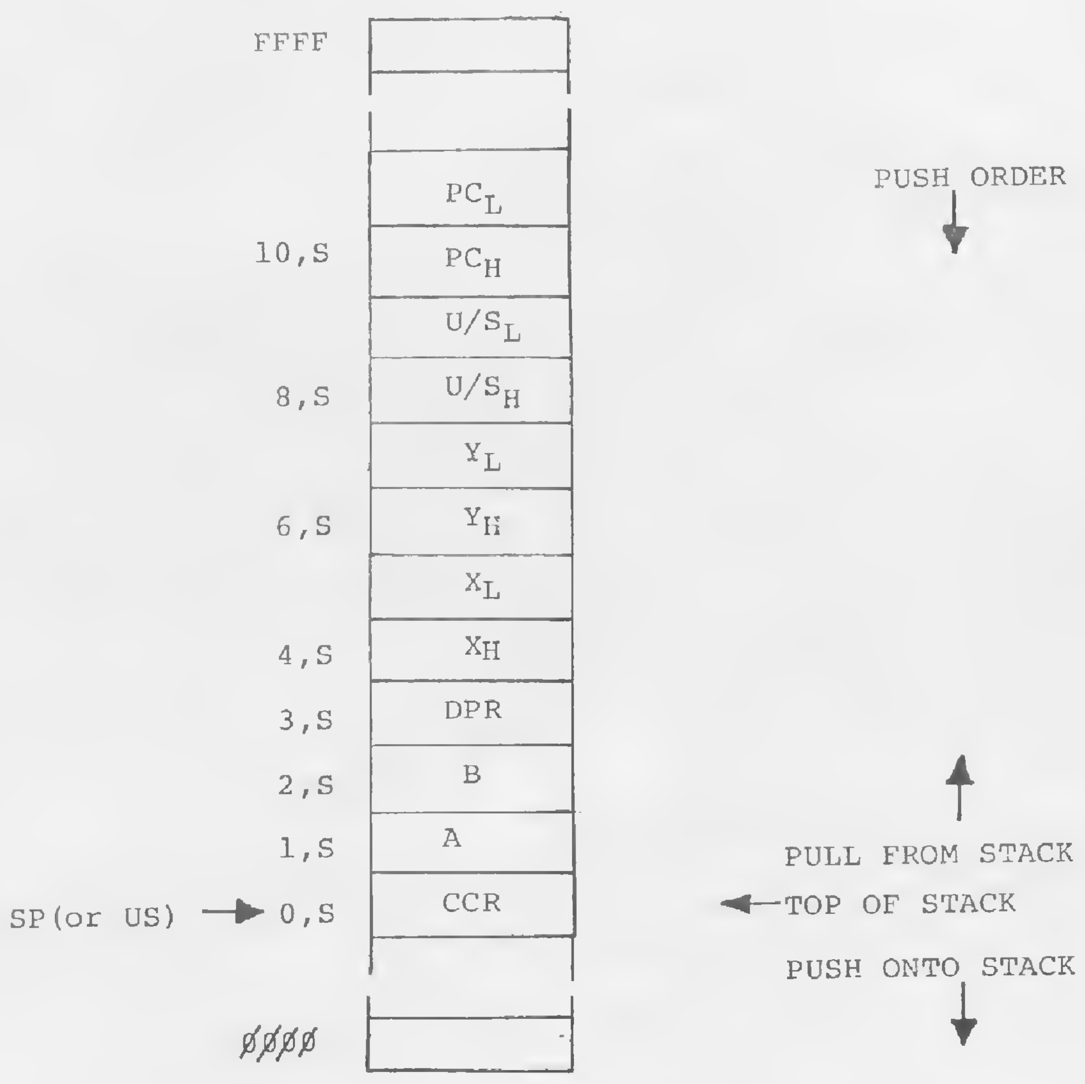


Figure 7: 6809 Push/Pull and Interrupt Stacking Order

SOFTWARE INCOMPATABILITIES WITH 6800/6802

1. The new stacking order on the 6809 exchanges the order of ACCA and ACCB; this allowa ACCA to stack as the MS byte of the pair.

2. The new stacking length on the 6809 invalidates previous 6800 code which displayed IX or PC from the stack.

3. Additional stacking length on the 6809 stacks five more bytes for each NMI, IRQ, or SWI when compared to 6800/6802.

4. The 6809 stack pointer points directly at the last item placed on the stack, instead of the location before the lsot itme as in 6800/6802. In general this is not a problem since the most-usual method of pointing at the stack in the 6800/6802 is to execute a TSX. The TSX increments the value during the transfer, making X point directly at the last item on the stack.

   The stack pointer may thus be initialized one location higher on the 6809 than in the 6800/6802; similarly, comparison values may need to be one location higher.

   Any 6800/6802 program which does all stack manipulation through X (i.e., LDX #CAT, TXS instead of LDS #CAT) will have an exactly correct stack translation when assembled for 6809.

5. The 6809 uses the two high-order condition cose register bits. Consequently, these will not, in general, appear as 1's as on the 6800/6802.

6. The 6809 TST instruction does not affect the C flag, while 6800/6802 TST does clear the flag.

THEORY OF OPERATION - TECHNICAL SUMMARY

Page numbers refer to the circuit diagrams.

Page.1

An AC supply of between 6 and 15 volts is rectified and clamped to between 0 and 5 volts by the diode resistor combination. The 74LS14 Schmitt inverter provides rectangular pulses from this at 50Hz.

A falling edge then triggers both halves of a 74LS123 monostable producing a 1ms low going $\overline{\text{vsync}}$ pulse from one and a 2ms low going pulse from the other. These pulses are synchronized to $\overline{\text{hsync}}$ by one half of the 74LS74 ensuring that $\overline{\text{vsync}}$ and $\overline{\text{hsync}}$ are locked together. The rising edge of this second pulse clocks the other half of the 74LS74 to produce an active high output delayed by 1ms from the termination of $\overline{\text{vsync}}$. This in turn is gated with $\overline{\text{hsync}}$ by a 74LS32 OR gate to produce $\overline{\text{firq}}$ interrupt pulses to the processor once every $\overline{\text{hsync}}$ pulse while the output of the 74LS74 remains high. This is cleared under software control at the completion of each video frame via the clr bit of the 6821 PIA.

The $\overline{\text{hsync}}$ pulse is derived from the master microprocessor clock (e) which is divided by 64 by two 74LS93 counters. Their outputs are gated by a 74LS21 and 74LS00 to produce an 8us $\overline{\text{hsync}}$ pulse every 64us as required for video synchronization.

Page.2

The 6809 microprocessor has a 4MHz crystal providing a master clock (e) of 1MHz. Reset is accomplished at power on by an RC delay network and two 74LS14 Schmitt inverters. The non-maskable interrupt is utilized as an abort feature by coupling it to the on board PANIC push button. This button is debounced by an RC network and two more Schmitt inverters.

The main address and data buses are buffered by a 74LS245 and two 74LS241's to provide expansion capability via the SS-50 bus edge connector.

Page.3

The system chip selects are provided by three 74LS139's and some additional gating. Initially the top two address lines,a14 and a15 are decoded into four 16k blocks which are further decoded by the other gates. The bottom 16k block is decoded by another half of a 74LS139 and a12 and a13 to provide four 4k blocks. The bottom two of these ($\overline{\text{rom2}}$ and $\overline{\text{rom3}}$) may be used for two optional roms (see memory map). Note that the position of these two roms is selectable by jumpers on board.

The top 16k block is further decoded by half a 74LS139 and a12 and a13 to provide 4k block selects for $\overline{\text{rom1}}$, $\overline{\text{rom2}}$, and $\overline{\text{rom3}}$. $\overline{\text{Rom1}}$ is the system monitor rom and occupies the top 4k of the memory map, $\overline{\text{rom2}}$ and $\overline{\text{rom3}}$ are the optional roms described above. The middle 4k of this 16k block is used for I/O operations and is further divided by another half 74LS139 to give chip selects for $\overline{\text{pia1}}$, $\overline{\text{pia2}}$ and $\overline{\text{char ram}}$.

Note that these only occupy half of the 4k I/O block, the other half being reserved for I/O on additional boards. The next 16k block down is only partially utilized. The top 4k of it is decoded by half a 74LS139 to give four chip selects for $\overline{\text{ram1}}$ through $\overline{\text{ram4}}$ (1k each). These are the onboard ram chips, and the upper half of $\overline{\text{ram1}}$ is used for video character storage.
The rest of the circuitry, the 74LS21, 74LS08 and 74LS00 are used to provide a $\overline{\text{sel}}$ strobe when none of the onboard devices are selected.

Page.4

$\overline{\text{Rom1}}$, $\overline{\text{rom2}}$, and $\overline{\text{rom3}}$ select 2532 EPROMS as described above. The 6821 PIA is dedicated to system usage. Its lines are used to control the keyboard, cassette and video circuitry. Eight lines of port A are used for two purposes. During video scan they control which row of each character is being displayed and during keyboard scan they set up which row of the key matrix is being scanned. The remaining two lines of port A are used for the cassette interface. One provides serial data out via an RC network to the recorders microphone input and the other receives data in from the recorders earphone output. This data is first clipped by diodes and then squared by the 74LS14 and 74LS04.
Three lines of port B are used to provide the video control signals page, char and blank. A fourth control pulse, clr, is provided by the pia's handshaking output. The remaining four port B data lines are configured as inputs from the keyboard columns. These are also fed to the 74LS20 so that any

low transition (keypressed) will result in a pia handshaking interrupt.

Page.5

The second (optional)pia is completely available to the user and all port lines are brought out to an edge connector.
The expandable on board ram consists of six 2114's selected by $\overline{\text{ram2}}$ through $\overline{\text{ram4}}$.

Page.6

The 74LS20 and the 74LS08 gating provides a select pulse to the 2114 ram chips if one of three conditions is met. Firstly if page is low (signifying an access to the character generator ram), secondly in $\overline{\text{raml}}$ is low (signifying a normal processor ram access) and thirdly if any access to $(\text{F600 - F7FF})_{16}$ occurs (signifying a video scan cycle). The read/write to this ram is organized so that only reads occur during page access via the 74LS00 gating (see software theory of operation).
The 74LS245 gates the raml access in the normal fashion. During video access data from the ram is fed to the programmable character generator ram and the rom character generator (66710). These provide the necessary video data and character row selects are derived from the pia as previously described.
On the programmable character ram, bit seven of the input data selects between the ram chips (via a 74LS86) and the r/w is

gated via a 74LS32 so that it is always high (read) except for when the char ram select is active.

Page.7

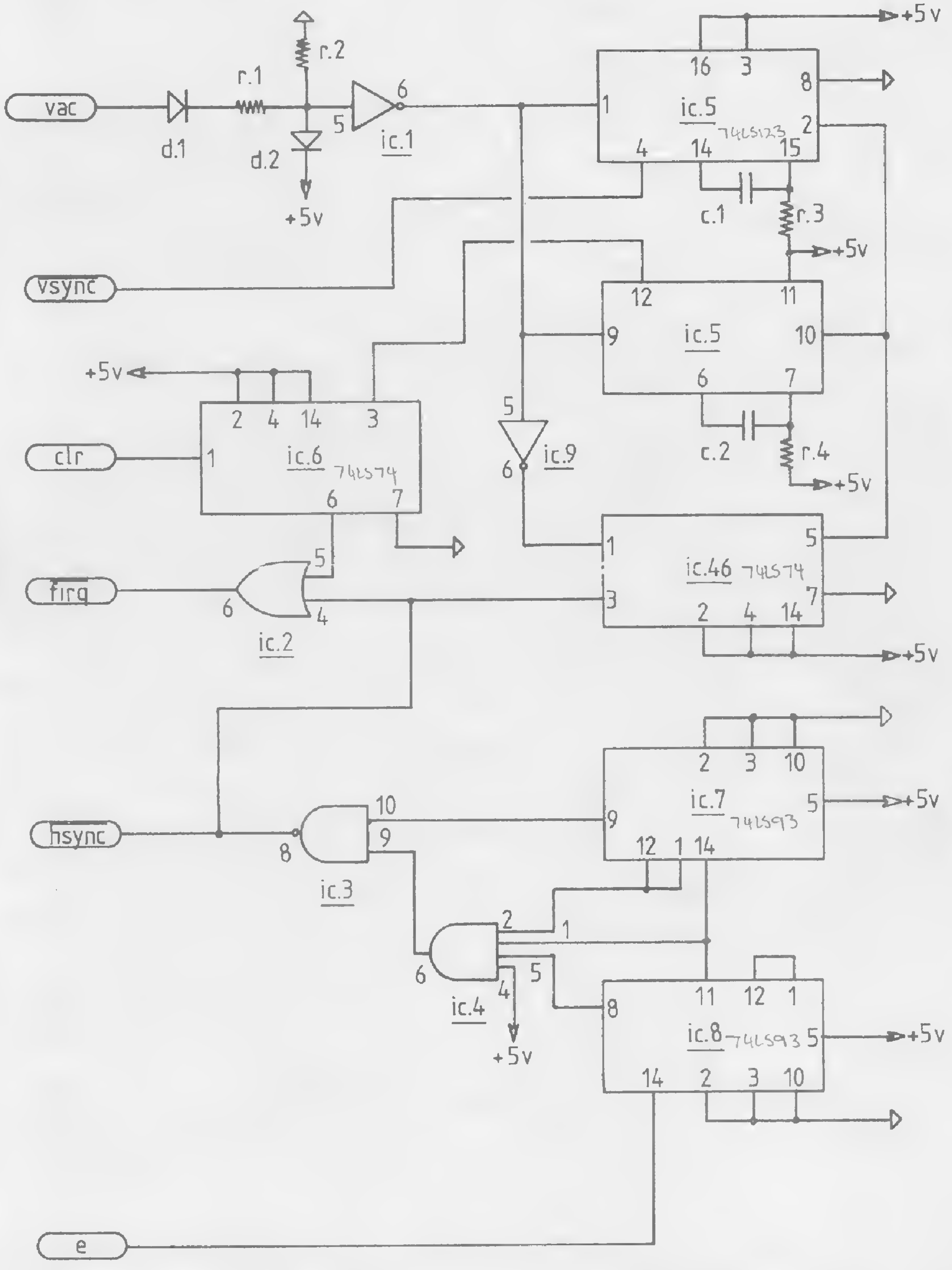
The outputs from the two character generators are selected by the char line from the pia via two 74LS157 multiplexers. The selected output (char rom if char is low, char ram if char is high) is then loaded into a 74LS165 shift register. The clock timing for the shift register is generated by the 74LS04 and the series of RC networks, synchronized to the master clock (e).
The 74LS245 data buffer provides microprocessor access to the programmable character generator ram.

Page.8

The 74LS157 multiplexes the eight columns of the keyboard (each with a pullup resistor) to four lines fed to the key inputs of the system pia. This is controlled by the pia's asc line.
The most significant data bit from the video ram is fed to half the 74LS74 which is clocked by the shift register load pulse. This synchronizes it with each new byte of video data. The output from this flip flop is then used to selectively invert video data via a 74LS86. This provides individual polarity control of the on screen characters.
Another 74LS32 is used to control blanking of video data via the blank signal from the system pia.Video data and sync are combined by a 4066 to produce composite video output.

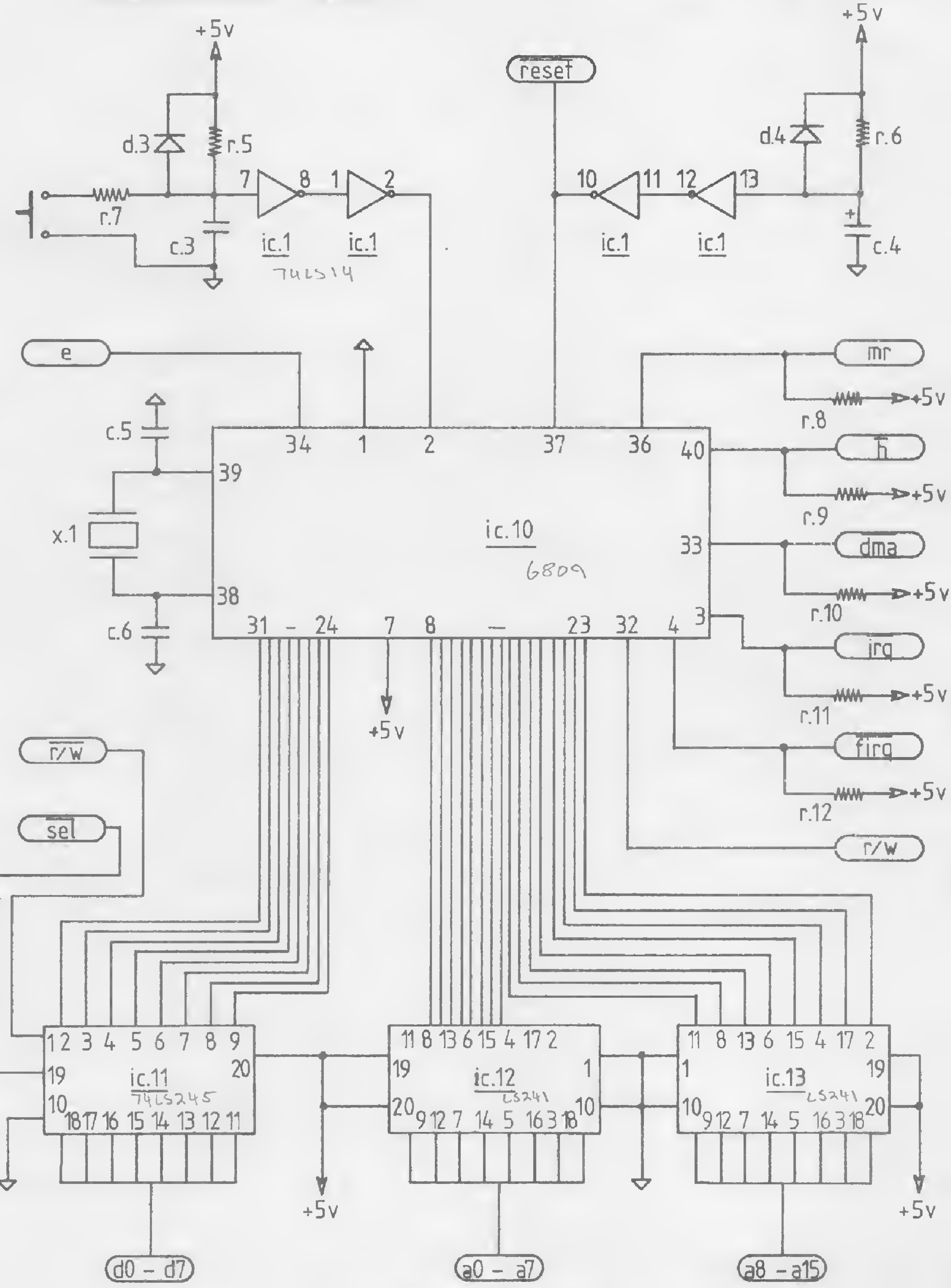
Aamber Pegasus

SJH 1/81 V1.0



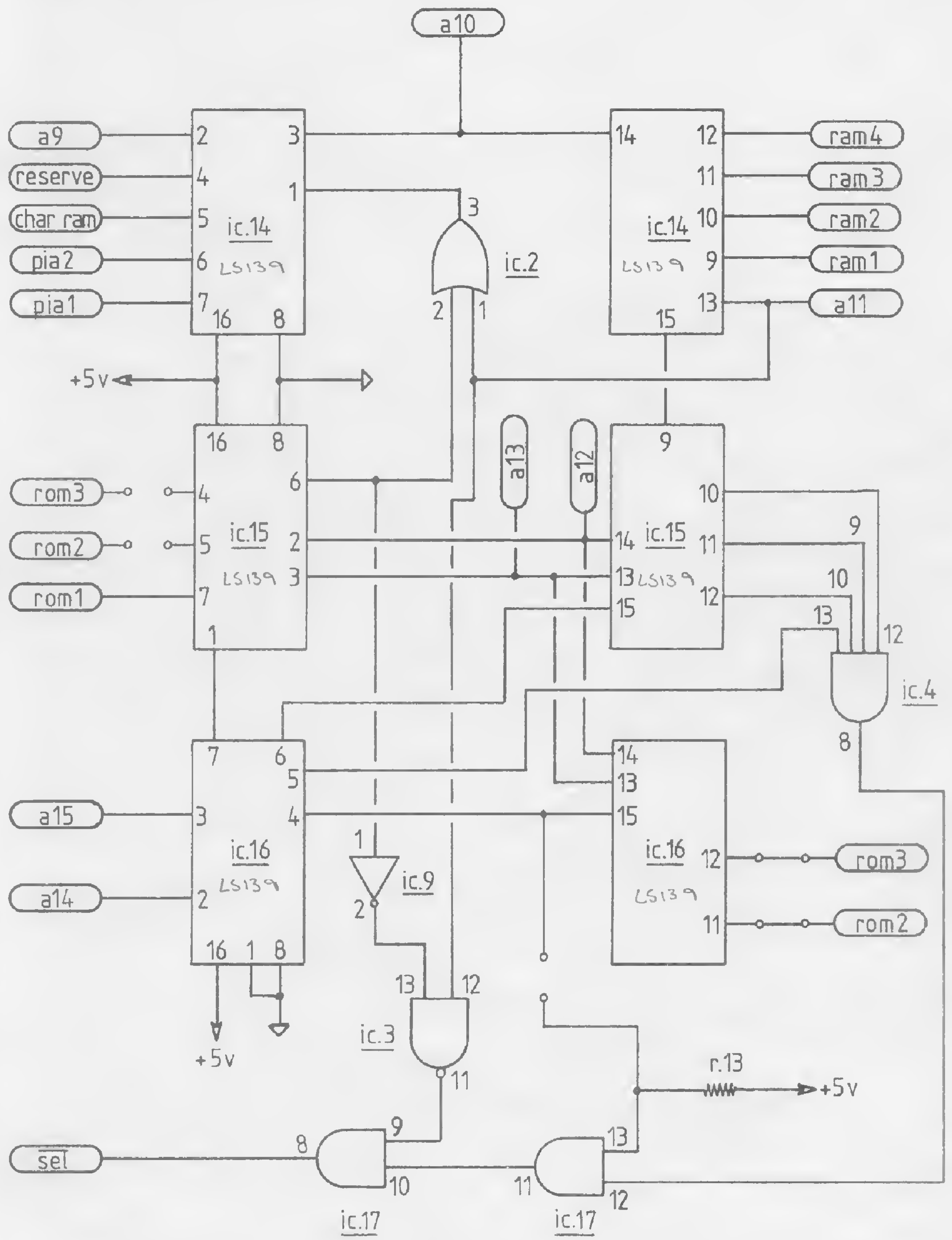
Aamber Pegasus

SJH 1/81 V1.0



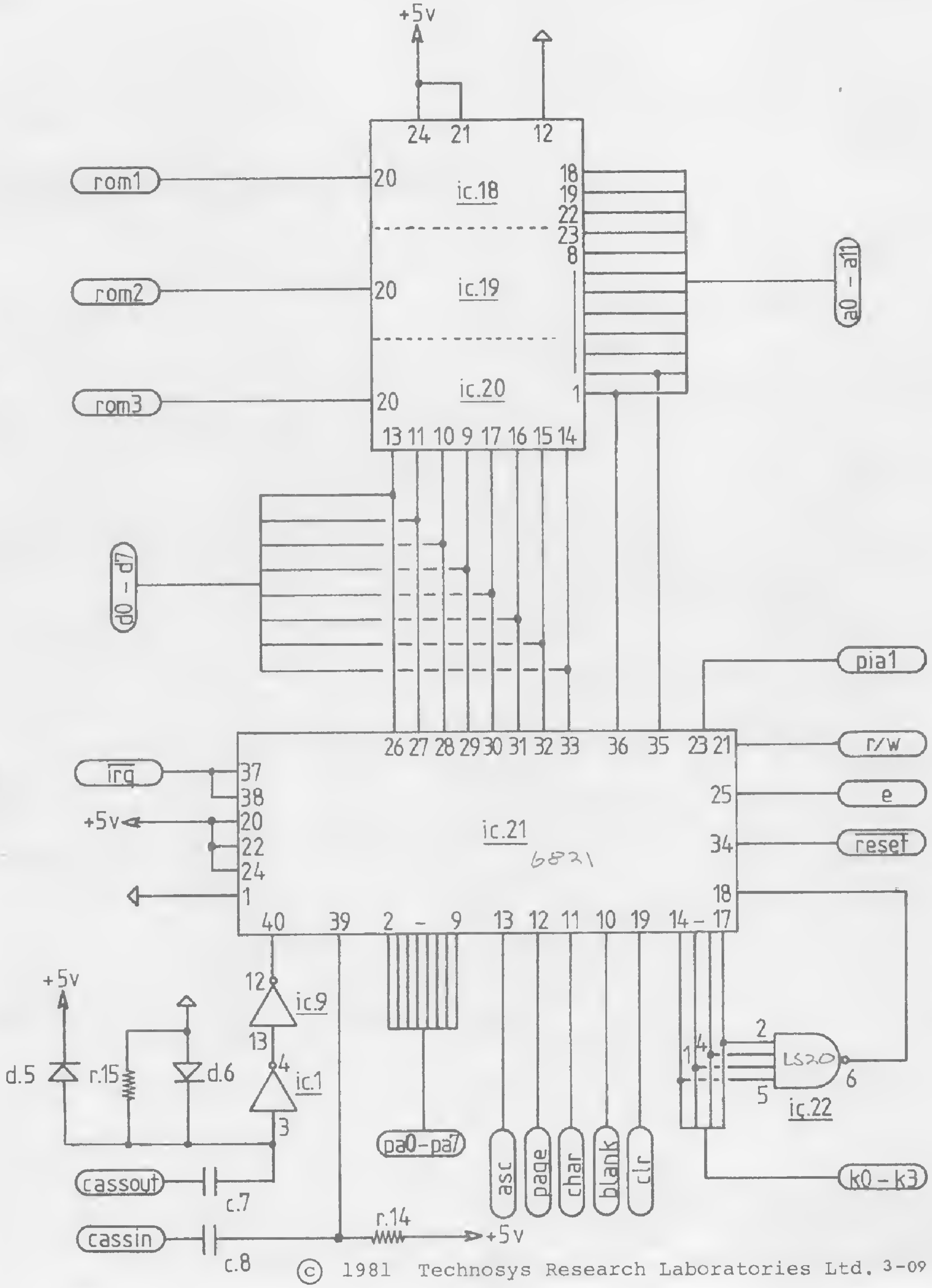
Aamber Pegasus

SJH 1/81 V1.0



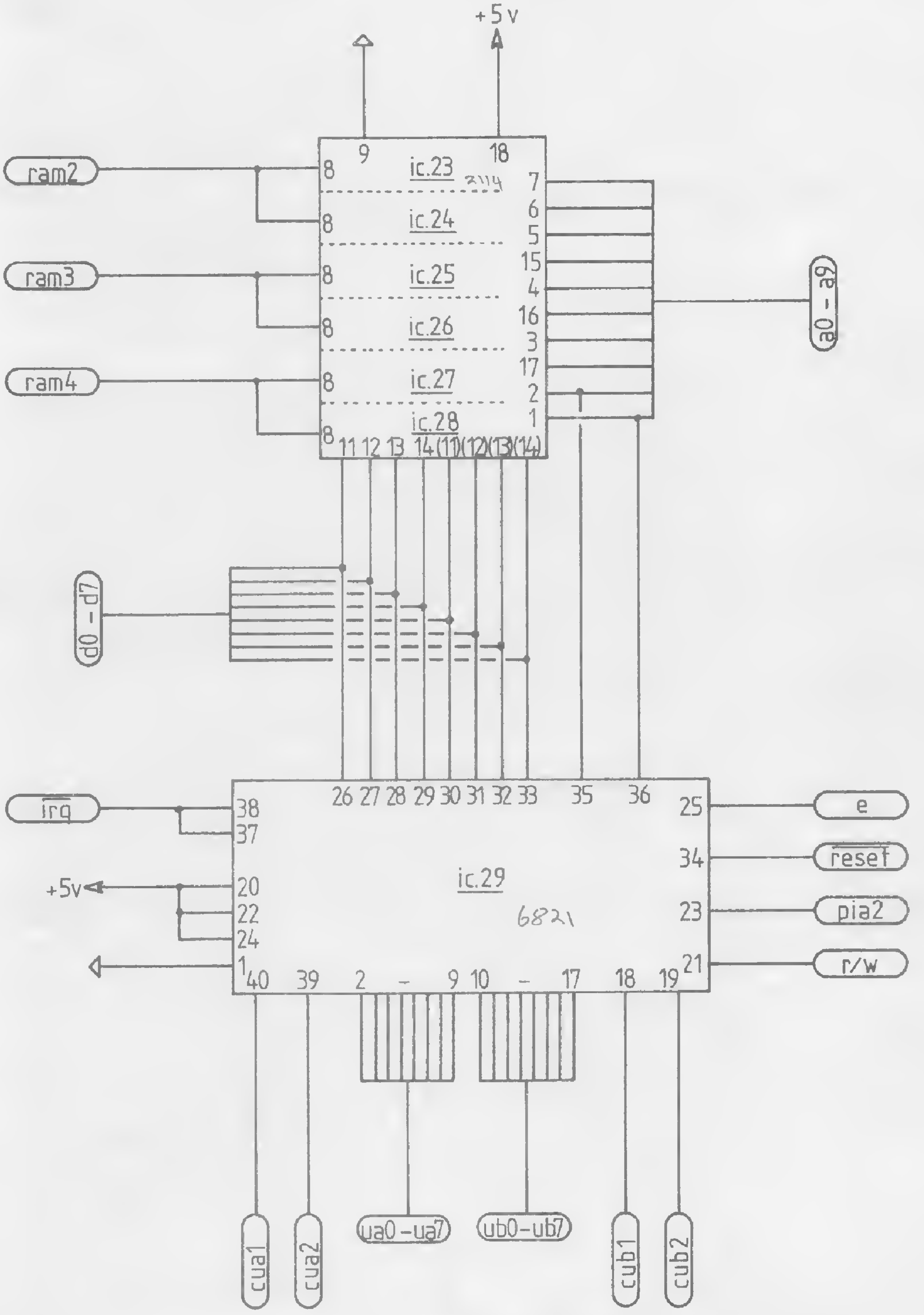
Aamber Pegasus

SJH 1/81 V1.0



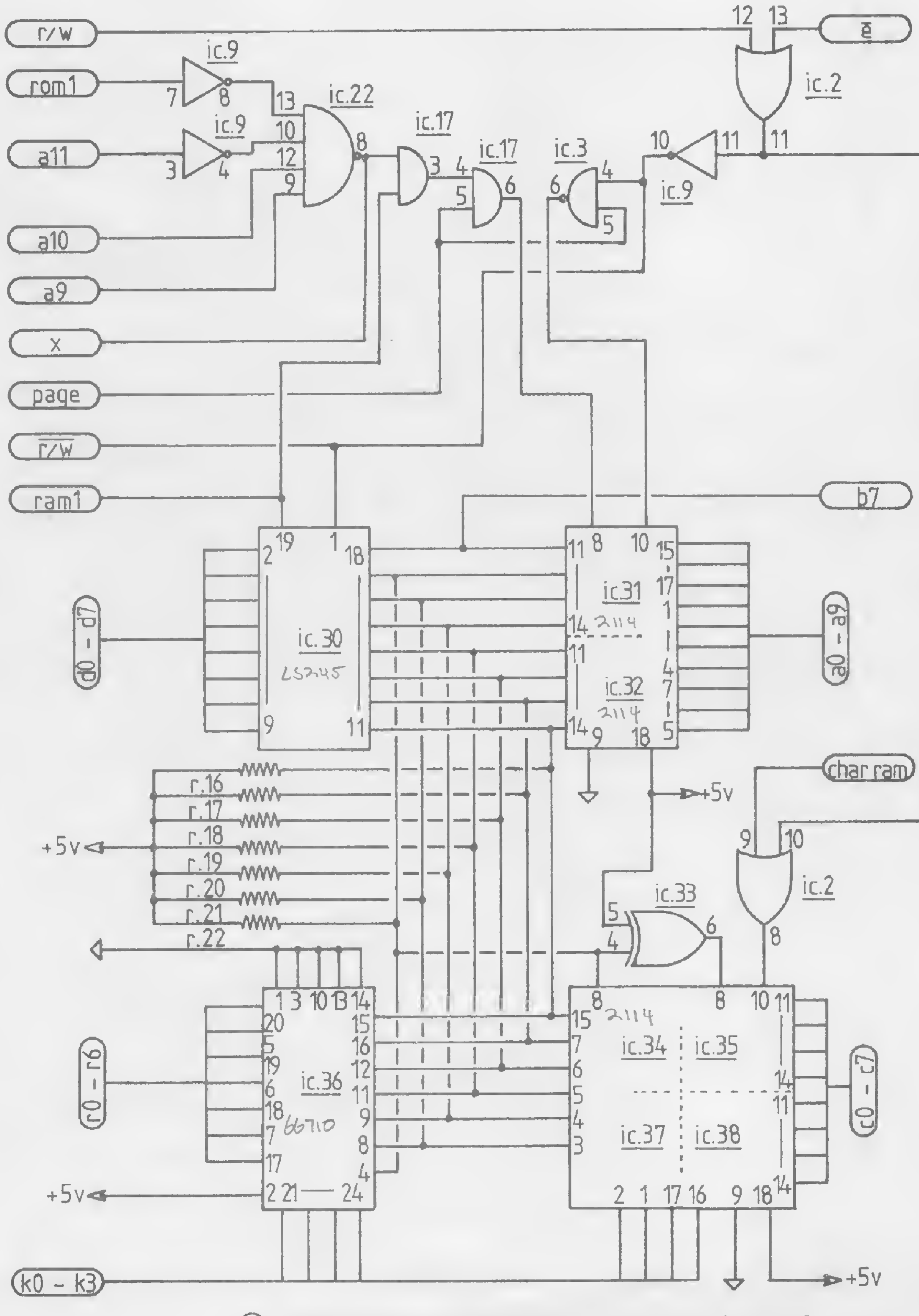
Aamber Pegasus

SJH 1/81 V1.0

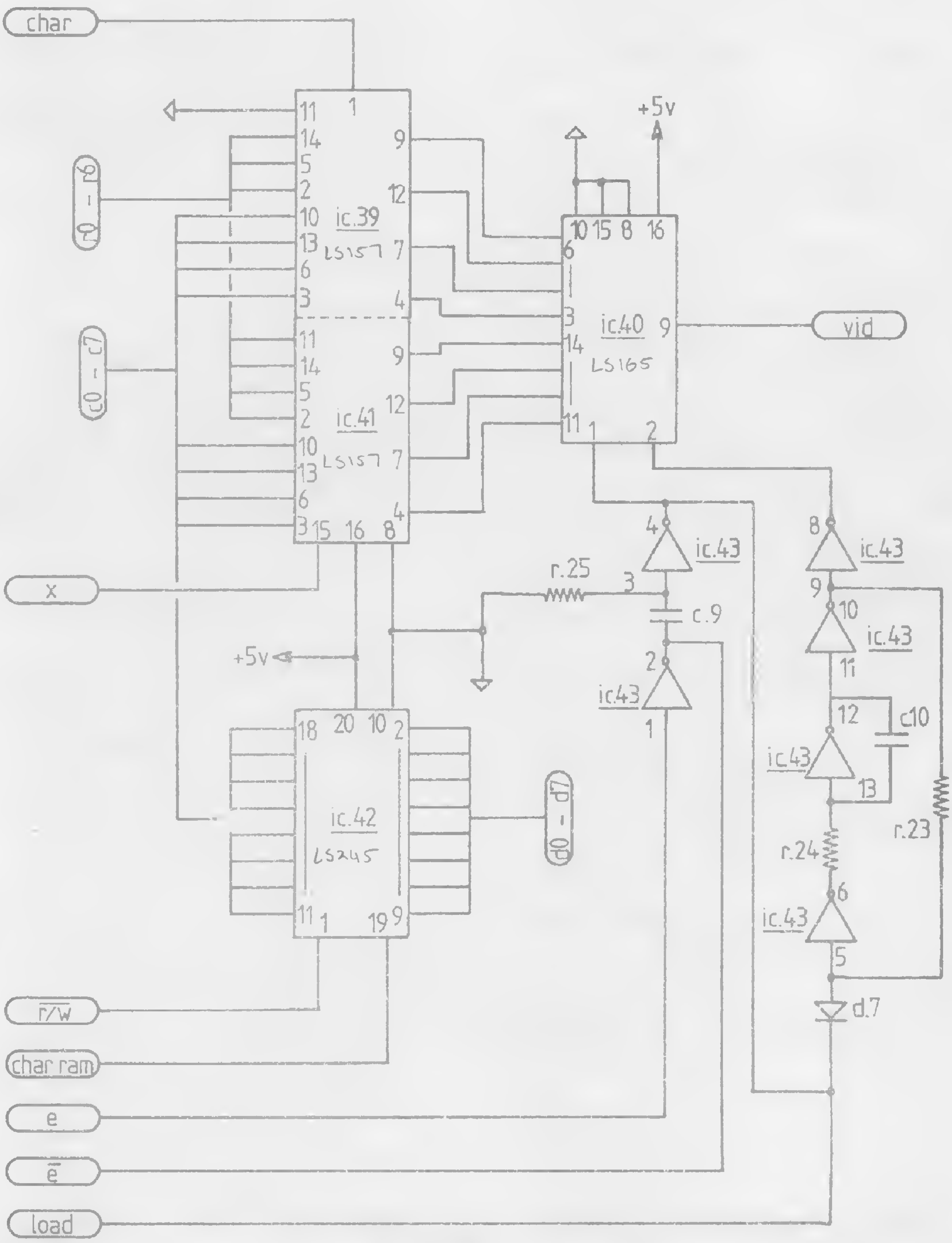


Aamber Pegasus

SJH 1/81 V1.0

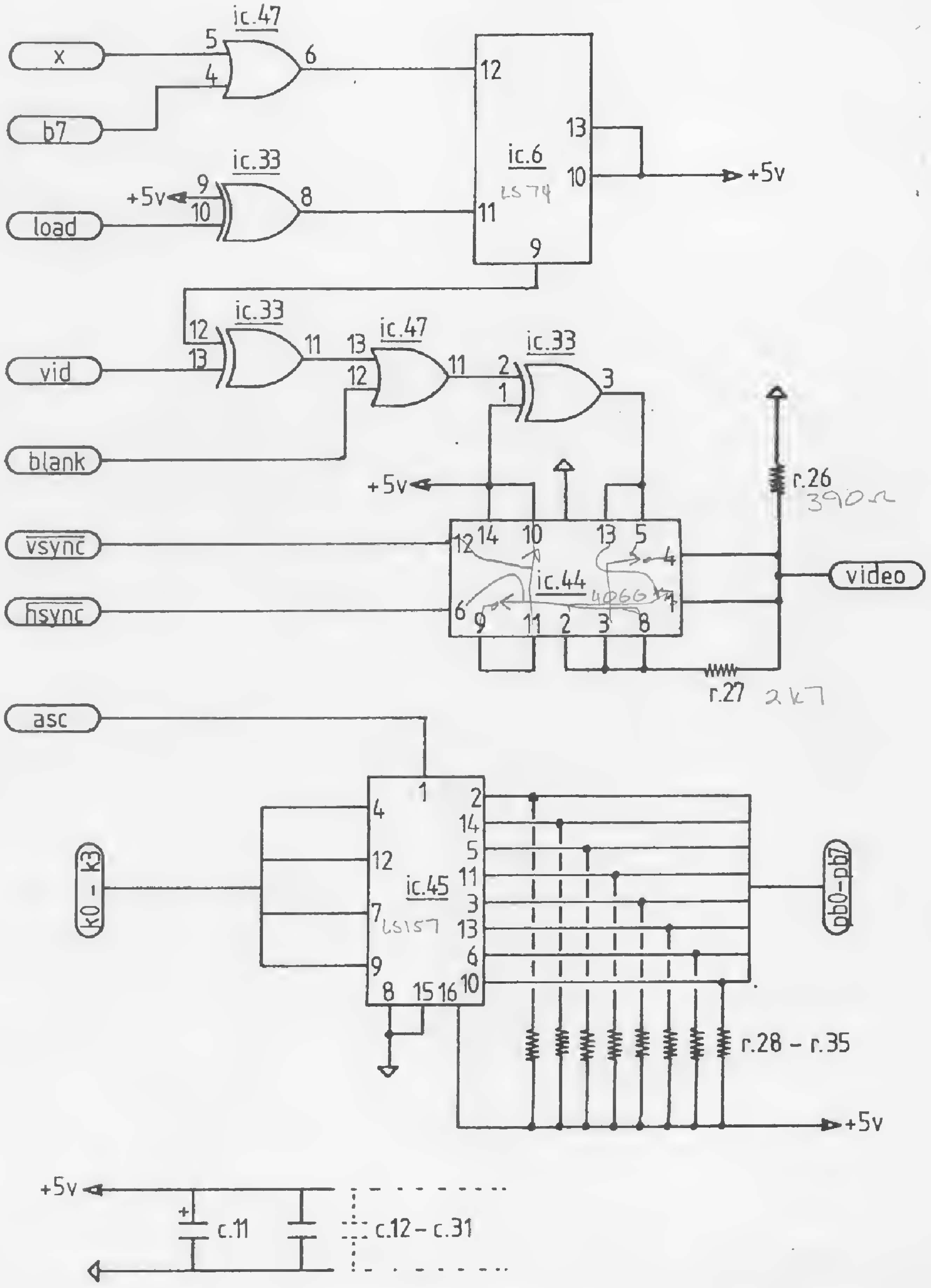


Aamber Pegasus



Aamber Pegasus

SJH 1/81 V1.0



AAMBER PEGASUS

Parts list

IC.1	74LS14		IC.24	MC2114-30	
IC.2	74LS32		IC,25	MC2114-30	
IC.3	74LS00		IC.26	MC2114-30	
IC.4	74LS21		IC.27	MC2114-30	
IC.5	74LS123		IC.28	MC2114-30	
IC.6	74LS74		IC.29	MC6821	*
IC.7	74LS93		IC.30	74LS245	
IC.8	74LS93		IC.31	MC2114-30	
IC.9	74LS04		IC.32	MC2114-30	
IC.10	MC6809		IC.33	74LS86	
IC.11	74LS245	*	IC.34	MC2114-30	*
IC.12	74LS241	*SS-50 bus expansion	IC.35	MC2114-30	*
IC.13	74LS241	*	IC.36	MC66710	
IC.14	74LS139		IC.37	MC2114-30	*
IC.15	74LS139		IC.38	MC2114-30	*
IC.16	74LS139		IC.39	74LS157	
IC.17	74LS08		IC.40	74LS165	
IC.18	TMS2532		IC.41	74LS157	
IC.19	TMS2532	*	IC.42	74LS245	*
IC.20	TMS2532	*	IC.43	74LS04	
IC.21	MC6821		IC.44	MC14066	
IC.22	74LS20		IC.45	74LS157	
IC.23	MC2114-30		IC.46	74LS74	
			IC.47	74LS32	

* denotes optional components

AAMBER PEGASUS

Parts List

R.1	100	R.14	470
R.2	100	R.15	4.7K
R.3	8.2K	R.16 - R.22	1K
R.4	27K	R.23	1K
R.5	100K	R.24	390
R.6	1m	R.25	470
R.7	1K	R.26	390
R.8 - 12	1K	R.27	2.7K
R.13	1K	R.28 -R.36	1K

all resistors ¼w carbon all values in ohms

D.1 - D.7	IN4148
X.1	4.0 MHz Crystal

C.1	150nf mylar	C.7	1 uf 35v tantalum
C.2	330nf mylar	C.8	0.1uf disc ceramic
C.3	o.1 uf disc ceramic	C.9	47pf disc ceramic
C.4	1uf 35v tantalum	C.10	330pf disc ceramic
C.5	27pf disc ceramic	C.11	68uf 16v tantalum
C.6	27pf disc ceramic	C.12 - C.36	0.1uf disc ceramic

SOFTWARE IMCOMPATABILITIES WITH 6800/6802

7. The 6809 right shifts (ASR,LSR,ROR) do not affect V; the 6800/6802 shifts set V = $b_7 + b_6$.

8. The 6809 H-flag is not defined as having any particular state after subtract-like operations (CMP, NEG, SBC, SUB); the 6809/6800 clear the H-flag under these conditions.

9. The 6800/6802 CPX instruction compared MS byte, then LS byte; consequently only the Z-flag was set correctly for branching. The 6809 instruction CMPX set all flags correctly.

10. The 6809 instruction LEA may or may not affect the Z-flag depending upon which register is being loaded; LEAX and LEAY do affect the Z-flag while LEAS abd LEAU do not. Thus, the User Stack does not emulate the index registers in this respect.

FORTH REFERENCES

BYTE	August 1980
Dr. DOBB'S JOURNAL	January 1981
USING FORTH	FORTH, Inc. 2309 Pacific Coast Highway, Hermosa Beach California, 90254
FORTH DIMENSIONS	Forth Interest Group P.O. Box 1105 San Carlos,California 94070

FORTH

The following pages begin an introduction to the language FORTH . This section of the manual should familiarise you with stack operations, simple integer calculations and elementary programming development. For more comprehensive programming, the magazines or books in the reference list should be consulted.

Using FORTH:	1) Switch your computer on
	2) You should see a display including the words FORTH 1.1
	3) press F
	4) The message Pegasus 6809 FORTH should appear.
	5) Now you are ready to begin reading the FORTH manual.
To EXIT FORTH	1) Type MON (press RETURN)
	2) You are now back to the Select mode.
TO REENTER FORTH	1) You should be in the Select mode
	2) Type M . G 0011. (the . should return you to FORTH)
	3) ONLY use this method if you have previously been in FORTH since switch on.
PANIC!	1) If for some reason (usually an illegal loop operation) the computer 'hangs-up' and will not respond, all is not lost. Press the NMI button inside the computer and then reenter FORTH as above.

A Gentle Introduction to Pegasus Forth

Forth as a language is very different from most other computer languages, such as Basic or Pascal. It requires a structured approach, (having no GOTO statement), yet has all the convenience of an interactive interpreter. An expert's definition of Forth might be:

threaded, extensible, interactive, tree-structured, self-implementing, interpretive language.

Stack Concepts

Forth is a stack language. A stack can be defined as a collection of data items, usually byte (8 bit) or word (16 bit) quantities, which are pointed to by a register known as a stack pointer, and are arranged so that the last item placed on to the stack is the first to be removed. Stacks usually grow from high memory addresses down to low memory addresses, and are contiguous.

Any form of data may be placed on a stack, including numbers, characters, or addresses which point to data. If you have used a calculator with Reverse Polish Notation (RPN, e.g. Hewlett Packard), you will be at home with Forth. Such a calculator operates something like Forth in its use of a stack, and post-fix notation (or RPN), to evaluate expressions.

For instance, to evaluate (3*4)+(6*2), where the asterisk '*' is the standard computer symbol for multiplication, we would enter into the computer:

```
3 4 * 6 2 * + . (CR)
```

with a space separating each symbol, or word.
Evaluating from left to right: 3 is stacked, then 4, then the multiplication operator multiplies the two numbers, removing them from the stack, and leaving 12 on the top. Then 6 and 2 are stacked and multiplied, leaving another 12 on top of the 12 already there. The two products are then added, leaving 24 on the stack, which is then removed from the stack and printed with the '.' operator.

After the RETURN key is typed, the computer will respond

```
24 ok
```

It is a convention with Forth that when the top item of the stack is operated upon, then it is destroyed.

As an example, let's write a small program to evaluate the polynomial

$$3x^2 + 4x - 7$$

for x=2 and x=7.

We type in the following:

```
: POLY DUP DUP * 3 * SWAP 4 * + 7 - . ;
```

The colon word ':' means that a definition is to follow - the definition is terminated by the semicolon word ';'.
POLY is the name of the Forth word that we are defining - any name of up to 31 characters is acceptable here, and it can include any character except the space or RETURN, including control codes or special punctuation characters.

DUP is a Forth word that duplicates the top of stack, and leaves the result on the stack. Executed twice, this makes two copies of the number on the top of the S stack.

The * operator is used to multiply these two copies, destroying them and leaving the result on top. The result is then multiplied by 3, leaving a new result, then SWAPped with the original number, x, which is multiplied by 4. Then the two terms are added, 7 is subtracted, and the result printed.
The program is now written, so let's run it:

```
2 POLY (CR) 13 OK
7 POLY (CR) 168 OK
```

It's as simple as that!
POLY is now a Forth word, that will be there until we turn off the Pegasus, or until we tell it to

```
FORGET POLY (CR)
```

in which case all definitions subsequent to and including POLY will be destroyed (unless they are saved on cassette first).

Arithmetic Operations

In normal operation, this version of FORTH uses 16-bit signed arithmetic which allows integers from -32768 to +32767 to be used. You can develop methods to work with numbers outside this range, but only this range will be assumed for now.

CALCULATIONS

Reverse Polish Notation (RPN) is used in calculations. For example:

```
6 + 5 * 2  is entered as   5 2 *  6 +
(Note:  * is the symbol for multiplication)
(       * is done before + )
```

To print the answer, (which is stored on top of the stack) the FORTH word . is used (i.e. a full stop)

```
Type in          5 2 *  6 +  .
```

The answer 16 should be printed in the same line followed by OK,

Exercises: (The answers are on the following page)

Change these operations to RPN; then use FORTH to print the answers.

```
(a)    7 * 2 + 3 + 4
(b)    7 * 2 + 3 * 4
(c)    7 * (2 + 3) * 4
(d)    72 ÷ 9    ( use / on the keyboard for division )
(e)    (64 + 6) ÷ 7
```

ANSWERS:

```
(a)   7 2 *  3 +  4 +  .

(b)   7 2 *  3 4 *  +  .
(c)   2 3 +  7 * 4 *  .        ( + adds the 2 and 3 )
                               ( first * multiplies the answer by 7 )
                               ( second * then multiplies by 4      )
(d)   72 9 /  .
(e)   64 6 +  7 /  .
```

Note: Other answers are possible. For example (a) could be done

```
3 4 7 2 * + + .
```

In this case, all numbers were put on the stack, and then the operations were performed.

```
Similarly:  (c)  7 4 2 3 + * * .
            (e)  7 64 6 + / .
```

FORTH PROGRAMMING

Screen Operations

Although you must understand how the stack operates to do calculations, you can do much more with FORTH than imitate a programmable calculator. This section will show you how to define words to output text, numbers and graphics onto the video screen.

The PEGASUS monitor uses control codes to move the cursor and provide other screen functions. Some of the frequently used codes are given below.

FUNCTION	ASCII number (decimal)
clear screen	12
cursor on	6
cursor off	15
set inverse video	1
clear inverse video	2
set cursor position	11
delete current line	21

Since the functions may be used often in one program, you are advised to define words for these tasks.

The FORTH word EMIT will output the ASCII number at the top of the stack to the screen. If the number is from 32 to 127, EMIT will put a letter, digit or other character onto the screen. Try typing in these commands to see how the word EMIT can be used.

```
65  EMIT
75  EMIT                 (remember to press RETURN key at
49  EMIT                   end of each line)
42  EMIT
```

Look at the table ASCII characters. You will find the characters A, K, 1, * beside the numbers you used above.

Use the word EMIT to put these characters on the screen P, p,), #.

If you use numbers less than 32, one of the screen functions will be activated instead. Try these

```
12  EMIT
15  EMIT
 6  EMIT
 1  EMIT
 2  EMIT
```

You are now ready to define words to do these functions whenever you wish.

ASCII CHARACTER TABLE

The members from 0 to 31 are reserved for special screen function controls.

CHARACTER	ASCII (decimal)	CHARACTER	ASCII (decimal)	CHARACTER	ASCII (decimal)
Space	32	@	64		96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(	40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
-	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[	91	{	123
<	60	\	92	|	124
=	61	]	93	}	125
>	62	^	94	~	126
?	63	_	95	■	127

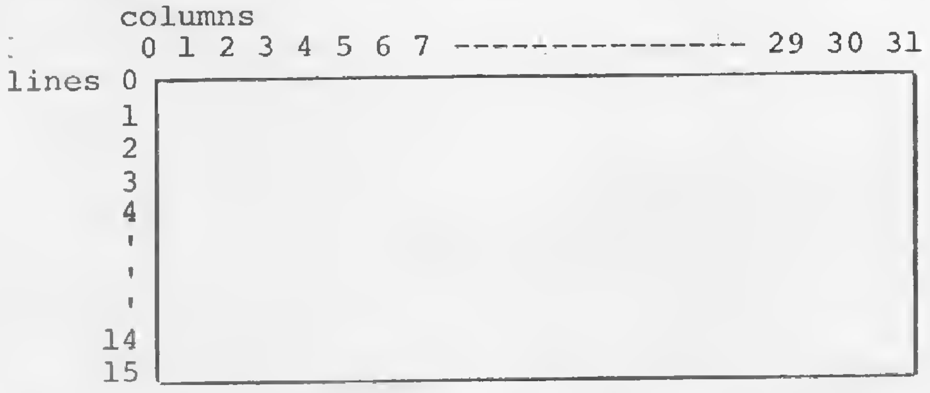
Note the difference between the digit 0 and the letter O

The easy ones are: (leave a space after : and after EMIT)

```
    : CLS 12 EMIT ;                 -clears screen
    : CO 6 EMIT ;                   -turns cursor on
    : CX 15 EMIT ;                  -turns cursor off
    : IO 1 EMIT ;                   -turns inverse video on
    : IX 2 EMIT ;                   -turns inverse video off
```

Type in these 5 definitions, then test each one by typing the word you have defined. (e.g. try CLS then CX then CO etc.)

The word to set the cursor at a certain place on the screen is more difficult to define. The screen has 16 lines, 32 columns wide. The figure below shows how these are numbered.



If you want to put the cursor at column 20 of line 5 you must use EMIT three times.

$\underbrace{\text{11 EMIT}}_{\text{prepares to set position}} \quad \underbrace{\text{20 EMIT}}_{\text{column number}} \quad \underbrace{\text{5 EMIT}}_{\text{line number}}$

All this must be done even before you can put anything there. To put the letter 'A' (ASCII 65) at this position, type in:

$\underbrace{\text{11 EMIT 20 EMIT 5 EMIT}}_{\text{sets position}} \; \underbrace{\text{65 EMIT}}_{\text{prints A}}$

You can define a word CPOS to set the cursor for you. (type this in)

```
    : CPOS 11 EMIT SWAP EMIT EMIT ;
```

Now to print A in column 20, line 5, type:

```
    20 5 CPOS 65 EMIT ( a bit easier?)
```

HOW AND WHY CPOS WORKS:

CPOS expects two numbers to be on the stack, the column number and the line number.

```
          STACK           OPERATION

Step 1    5              —line number entered last
          20             —column number entered first
          x
          x

Step 2    11              CPOS 11 puts 11 on stack
          5
          20
          x

Step 3  ↑ 5      11 ——————→    EMIT  - first EMIT uses 11 to
        | 20                           prepare to set the position
        | x
          x

Step 4    20                   SWAP  - we need column number first
          5
          x
          x

Step 5  ↑ 5      20 ——————→    EMIT  - column number now used
        | x
        | x
          x

Step 6  ↑ x       5 ——————→   EMIT   - line number now used
        | x
        | x
          x
```

The stack has used the two numbers 20 and 5 to set the cursor position and it is now ready for further entries.

Now, use CLS and then CPOS four times to put four x's, one below the other, starting at column 25 on line 4.

You still have to type a lot to just to get one letter onto the screen. If you want several words, or a line of characters, there are other ways instead of using EMIT.

PRINTING TEXT The FORTH word to print text is ." followed by a space. The word " ends the text to be printed.

Example Define a word W1 that will print PEGASUS.

```
: W1 ." PEGASUS" ;
```

Now to print PEGASUS starting at column 20 line 5, type

```
20 5 CPOS W1
```

If you don't want OK printed after PEGASUS, then include CR (carriage return) after W1

```
20 5  CPOS W1 CR
```

Let's define a TASK now that will combine several of our new words.

```
: PRINT CPOS W1 CR ;
```

This new definition will expect two numbers, column number, and line number.

Example 20 5 PRINT

How Print Works

Step 1. 20, then 5, are placed on stack
2. Cursor put at column 20 line 5 (CPOS)
3. PEGASUS is printed (W1)
4. Cursor goes to next line (CR)
5. OK printed and Forth ready for new commands.

Exercises (clear the screen after each exercise)

1. Use IO and IX to print PEGASUS in inverse video
2. Use IO 9 SPACES IX to print 9 white blocks (SPACES is a FORTH word)
3. Define a TASK to print PEGASUS near the middle of the screen in normal video, with a white border, top bottom and sides. (break TASK into several parts, and then combine for final definition - test each part as you define them)

Some of the passible answers for these tasks are given on the following page.

POSSIBLE ANSWERS

```
1.   : PRINT1 CPOS IO W1 IX CR ;     to use type  20 5 PRINT1

2.   : 9SP  CPOS IO 9 SPACES IX CR ; to use type 20 5 9SP
```

3. You should break this task into several parts.

```
     (a)  : TB 10 6 9SP ;          - top border
     (b) Define another word
          : 1SP CPOS IO 1 SPACES IX CR ;
         Use 1SP to provide the left and right borders.
          : LB 10 7 1SP ;          - left border
          : RB 18 7 1SP ;          - right border
     (c)  : PG 11 7 PRINT ;        - prints PEGASUS
     (d)  : BB 10 8 9SP ;          - bottom border
```

Now put all these parts together.

```
          : TASK CLS TB LB PG RB BB ;
```

To use, type TASK

COMBINING MATHS AND TEXT PRINTING

This section will describe a task using defined words for

(a) finding the cube of a number

(b) calculating cubes of 10 numbers

(c) printing headings and putting results in chart form.

(a) If you want to cube a number, you need 2 duplicate numbers to multiply with the original

```
: CUBE DUP DUP * * ;
```

Type. 5 CUBE . (the . is needed to print the result)

Make sure you understand how CUBE works.
(note: The stack can only store numbers from -32768 to 32767
32 CUBE will cause stack to overflow)

(b) We must use a DO ----- LOOP to do a calculation more than once. Let's define a word to print the numbers from 1 to 10.

```
: COUNT 11 1 DO I . CR LOOP ;
```

Type CLS COUNT to test.

How COUNT works

11 1	-	loop starts at 1 but stops at 10 , NOT 11
DO	-	Start of Loop
I .	-	prints this number
CR	-	begins new line
LOOP	-	end of loop

When the list was printed, the 0 of 10 was not in the one's column. Redefine COUNT using 6 .R instead of . and try COUNT now. (6 .R - sets up a block of 6 spaces with the one's column on right.

Exercise

If we want to print a list of cubes, we can use a loop similar to COUNT.

```
; 10CUBES 11 1 DO I CUBE 6 .R CR LOOP ;
```

The only difference is that I is cubed before it is printed.

Next combine these ideas into one loop that will print the number and its cube.

```
: CUBECHART 11 1 DO 4 SPACES  I 6 .R 10 SPACES I CUBE 6 .R
  CR LOOP ;
```

4 SPACES provides spaces on the left margin of the chart;

10 SPACES gives a gap between the numbers and their cubes.

(c) Finally, we need a word to print a heading for each column. You can now do this yourself.

Exercise Use what you learned about text printing and CUBECHART to do the following.

1. Clear the screen
2. Print on line 0 CUBES FROM 1 TO 10
3. Print on line 2 in inverse video

        NUMBER   CUBE

4. Print columns for number and its cube.

POSSIBLE ANSWERS

```
: L0 ." CUBES  FROM 1 TO 10" ;
: W1 ." NUMBER" ;
: W2 ." CUBE" ;
: L2 3 SPACES IO W1 IX 13 SPACES IO W2 IX ;
: CHART CLS CX L0 CR CR L2 CR CUBECHART CO ;
```

Your answer may have different methods of printing line 0 and 2. If it works, it's right!

FORTH DICTIONARY WORDS.

DEFINING WORDS

(a) New words are defined with a COLON DEFINITION.

:	begins the definition
;	ends the definition

Example: `: CLS 12 EMIT ;`

(b) Variables may be stored for later use.

0 VARIABLE NUM	defines NUM with an initial value 0
25 NUM !	changes value to 25 (! is pronounced "store")
NUM @	puts the value of NUM onto the stack (@ is pronounced "fetch")
NUM ?	prints the value of NUM

(NOTE- ? is the same as @ .)

(c) Constants may be defined for later use.

50 CONSTANT N	defines N with the initial value 50

The value of N cannot be changed once it is defined.

N	puts the value of N onto the stack
N .	prints the value of N

NOTE: Observe the differences in putting the values of variables and constants onto the stack and for printing their values.

	VARIABLE	CONSTANT
Definition	0 VARIABLE NUM	50 CONSTANT N
Change value	25 NUM !	fixed value
value onto stack	NUM @	N
print value	NUM @ .	N .
	or NUM ?	

PROGRAMMING ADVICE

* A list of FORTH WORDS is given in the FORTH HANDY REFERENCE pages.

* A new WORD may be defined using previously defined words by using

```
   :  to begin a definition (leave a space after
                                  the colon)
   ;  to end the definition
```

```
EXAMPLE:    : CLS 12 EMIT ;      to clear the screen
            CLS is the new word to be defined. To use the
            word, type  CLS
```

* A WORD can be any string of up to 31 characters bounded by spaces.
* Keep WORDS simple in operation. Test each new word after you define it. Then build further words from these simpler words until the full TASK is developed.
* All FORTH WORDS must be bounded by spaces.

```
For numbers    5 2 6   enters three numbers
               5 26    enters two numbers ( 5 and 26 )
```

SOME FORTH MESSAGE STATEMENTS

WORD NOT FOUND	You have not defined a word used or you have misspelt the word
REDEF: ISN'T UNIQUE	Just a warning that a word has been defined before.You may continue with the new definition.
CONDITIONALS NOT PAIRED	You forgot one of the words that are used to end DO , IF, or BEGIN structures.

NUMBER BASES

Calculations may be done in any number base.

DECIMAL	sets all operations to base 10
HEX	base 16
n BASE !	base n, where n is 2, 3, 4, etc.

Number Base Conversions:

To change 20 (decimal) to other bases, try this;

```
0 VARIABLE NUM        define NUM with an initial value 0
DECIMAL 20 NUM !      stores number in base 10
HEX NUM ?             prints the value in base 16
2 BASE ! NUM ?        prints the value in base 2
```

To change 1E (hex) to other bases;

```
0 VARIABLE NUM
HEX 1E NUM !
DECIMAL NUM ?
8 BASE ! NUM ?
2 BASE ! NUM ?
```

MEMORY UTILISATION WORDS

Each address location in the computer can store 1 byte of 8 bits. One byte can represent numbers from 0 to 255; thus ASCII characters may be stored as 1 byte.

1 cell is 2 bytes for a total of 16 bits.

One cell can represent numbers from 0 to 65535 or
numbers from -32768 to 32767

FORTH uses this last range of numbers.

IN MOST CASES:	1 byte	is used for ASCII characters
	1 cell	is used for numbers

(a) @ (pronounced "fetch")

@ replaces a memory address with the number in the cell starting at this address.

C@ replaces an address with the contents in the byte at this address.

Examples: `NUM @`

NUM was defined as a location for a variable.
NUM puts the address on the stack.
@ replaces this address with the value of the variable.

`HEX BF00 C@`

C@ replaces the hex address BF00 with the value in this address. This value is now on the stack.

(b) ! (pronounced "store")

! and C! work in the same way as @ and C@ but contents are stored into cells or bytes.

Examples: `25 NUM !`

`HEX 41 BF00 C!` (puts 41 (hex) into BF00)

(c) CMOVE is used to move bytes from one address to another.

The screen addresses are from:

```
            BE00 to BFFF     in HEX

           48640 to 49151    in DECIMAL

    HEX BE00 BF00 20 CMOVE    moves 20 (hex) bytes starting
                              at BE00 to 20 (hex) bytes
                              starting at BF00 .
```

In DECIMAL the operation becomes:

```
    DECIMAL 48640 48896 32 CMOVE
```

(e) FILL ERASE BLANKS

A comment is required about ASCII characters shown on the screen. The ASCII code for A is 65 (decimal)
If 65 is stored into the screen memory address, an INVERSE A will appear.
If 193 (i.e. 65 + 128) is stored into this screen memory address,then a normal A will appear

In HEX the comparable values are

```
        41   inverse A
        C1   normal A      ( C1 = 41 + 80)
```

FILL This word is used to fill an area of memory with a single character of your choice. For example

```
      HEX BF00 1 41 FILL     1 location at BF00 is filled
                             with an inverse A
      HEX BF00 0D C1 FILL    0D (hex) locations are filled
                             with a normal A
```

In DECIMAL, these operations become:

```
      DECIMAL 48896 1 65 FILL
      DECIMAL 48896 13 193 FILL
```

ERASE and BLANKS These words fill an area of memory with nulls (i.e. ASCII 0) or blanks (ASCII 32 in decimal, 20 in hex)

In HEX

```
      BF00 40 ERASE  is the same as   BF00 40 0 FILL

      BF00 40 BLANKS is the same as   BF00 40 20 FILL
```

FORTH HANDY REFERENCE

Stack inputs and outputs are shown; top of stack on right.
This card follows usage of the Forth Interest Group (S.F. Bay Area); usage aligned with the *Forth 78* International Standard.
For more info: Forth Interest Group
P.O. Box 1105
San Carlos, CA 94070.

Operand key:

n, n1, . . .	16-bit signed numbers
d, d1, . . .	32-bit signed numbers
u	16-bit unsigned number
addr	address
b	8-bit byte
c	7-bit ascii character value
f	boolean flag

STACK MANIPULATION

DUP	(n → n n)	Duplicate top of stack.
DROP	(n →)	Throw away top of stack.
SWAP	(n1 n2 → n2 n1)	Reverse top two stack items.
OVER	(n1 n2 → n1 n2 n1)	Make copy of second item on top.
ROT	(n1 n2 n3 → n2 n3 n1)	Rotate third item to top.
-DUP	(n → n ?)	Duplicate only if non-zero.
>R	(n →)	Move top item to "return stack" for temporary storage (use caution).
R>	(→ n)	Retrieve item from return stack.
R	(→ n)	Copy top of return stack onto stack.

NUMBER BASES

DECIMAL	(→)	Set decimal base.
HEX	(→)	Set hexadecimal base.
BASE	(→ addr)	System variable containing number base.

ARITHMETIC AND LOGICAL

+	(n1 n2 → sum)	Add.
D+	(d1 d2 → sum)	Add double-precision numbers.
−	(n1 n2 → diff)	Subtract (n1−n2).
*	(n1 n2 → prod)	Multiply.
/	(n1 n2 → quot)	Divide (n1/n2).
MOD	(n1 n2 → rem)	Modulo (*i.e.* remainder from division).
/MOD	(n1 n2 → rem quot)	Divide, giving remainder and quotient.
*/MOD	(n1 n2 n3 → rem quot)	Multiply, then divide (n1*n2/n3), with double-precision intermediate.
*/	(n1 n2 n3 → quot)	Like */MOD, but give quotient only.
MAX	(n1 n2 → max)	Maximum.
MIN	(n1 n2 → min)	Minimum.
ABS	(n → absolute)	Absolute value.
DABS	(d → absolute)	Absolute value of double-precision number.
MINUS	(n → -n)	Change sign.
DMINUS	(d → -d)	Change sign of double-precision number.
AND	(n1 n2 → and)	Logical AND (bitwise).
OR	(n1 n2 → or)	Logical OR (bitwise).
XOR	(n1 n2 → xor)	Logical exclusive OR (bitwise).

COMPARISON

<	(n1 n2 → f)	True if n1 less than n2.
>	(n1 n2 → f)	True if n1 greater than n2.
=	(n1 n2 → f)	True if top two numbers are equal.
0<	(n → f)	True if top number negative.
0=	(n → f)	True if top number zero (*i.e.*, reverses truth value).

MEMORY

@	(addr → n)	Replace word address by contents.
!	(n addr →)	Store second word at address on top.
C@	(addr → b)	Fetch one byte only.
C!	(b addr →)	Store one byte only.
?	(addr →)	Print contents of address.
+!	(n addr →)	Add second number on stack to contents of address on top.
CMOVE	(from to u →)	Move u bytes in memory.
FILL	(addr u b →)	Fill u bytes in memory with b, beginning at address.
ERASE	(addr u →)	Fill u bytes in memory with zeroes, beginning at address.
BLANKS	(addr u →)	Fill u bytes in memory with blanks, beginning at address.

CONTROL STRUCTURES

DO . . . LOOP	do: (end+1 start →)	Set up loop, given index range.
I	(→ index)	Place current index value on stack.
LEAVE	(→)	Terminate loop at next LOOP or +LOOP.
DO . . . +LOOP	do: (end+1 start →) +loop: (n →)	Like DO . . . LOOP, but adds stack value (instead of always '1') to index.
IF . . . (true) . . . ENDIF	if: (f →)	If top of stack true (non-zero), execute. [Note: *Forth 78* uses IF . . . THEN.]
IF . . . (true) . . . ELSE . . . (false) . . . ENDIF	if: (f →)	Same, but if false, execute ELSE clause. [Note: *Forth 78* uses IF . . . ELSE . . . THEN.]
BEGIN . . . UNTIL	until: (f →)	Loop back to BEGIN until true at UNTIL. [Note: *Forth 78* uses BEGIN . . . END.]
BEGIN . . . WHILE . . . REPEAT	while: (f →)	Loop while true at WHILE; REPEAT loops unconditionally to BEGIN. [Note: *Forth 78* uses BEGIN . . . IF . . . AGAIN.]

TERMINAL INPUT-OUTPUT

.	(n →)	Print number.
.R	(n fieldwidth →)	Print number, right-justified in field
D.	(d →)	Print double-precision number.
D.R	(d fieldwidth →)	Print double-precision number, right-justified in field.
CR	(→)	Do a carriage return.
SPACE	(→)	Type one space.
SPACES	(n →)	Type n spaces.
."	(→)	Print message (terminated by ").
DUMP	(addr u →)	Dump u words starting at address.
TYPE	(addr u →)	Type string of u characters starting at address.
COUNT	(addr → addr+1 u)	Change length-byte string to TYPE form.
?TERMINAL	(→ f)	True if terminal break request present.
KEY	(→ c)	Read key, put ascii value on stack.
EMIT	(c →)	Type ascii value from stack.
EXPECT	(addr n →)	Read n characters (or until carriage return) from input to address.
WORD	(c →)	Read one word from input stream, using given character (usually blank) as delimiter.

INPUT-OUTPUT FORMATTING

NUMBER	(addr → d)	Convert string at address to double-precision number.
<#	(→)	Start output string.
#	(d → d)	Convert next digit of double-precision number and add character to output string
#S	(d → 0 0)	Convert all significant digits of double-precision number to output string.
SIGN	(n d → d)	Insert sign of n into output string.
#>	(d → addr u)	Terminate output string (ready for TYPE).
HOLD	(c →)	Insert ascii character into output string.

DISK HANDLING (To be available later)

LIST	(screen →)	List a disk screen.
LOAD	(screen →)	Load disk screen (compile or execute).
BLOCK	(block → addr)	Read disk block to memory address.
B/BUF	(→ n)	System constant giving disk block size in bytes.
BLK	(→ addr)	System variable containing current block number.
SCR	(→ addr)	System variable containing current screen number.
UPDATE	(→)	Mark last buffer accessed as updated.
FLUSH	(→)	Write all updated buffers to disk.
EMPTY-BUFFERS	(→)	Erase all buffers.

DEFINING WORDS

: xxx	(→)	Begin colon definition of xxx.
;	(→)	End colon definition.
VARIABLE xxx	(n →) xxx: (→ addr)	Create a variable named xxx with initial value n; returns address when executed.
CONSTANT xxx	(n →) xxx: (→ n)	Create a constant named xxx with value n; returns value when executed.
CODE xxx	(→)	Begin definition of assembly-language primitive operation named xxx.
;CODE	(→)	Used to create a new defining word, with execution-time "code routine" for this data type in assembly.
<BUILDS ... DOES>	does: (→ addr)	Used to create a new defining word, with execution-time routine for this data type in higher-level Forth.

VOCABULARIES

CONTEXT	(→ addr)	Returns address of pointer to context vocabulary (searched first).
CURRENT	(→ addr)	Returns address of pointer to current vocabulary (where new definitions are put).
FORTH	(→)	Main Forth vocabulary (execution of FORTH sets CONTEXT vocabulary).
EDITOR	(→)	Editor vocabulary; sets CONTEXT.
ASSEMBLER	(→)	Assembler vocabulary; sets CONTEXT.
DEFINITIONS	(→)	Sets CURRENT vocabulary to CONTEXT.
VOCABULARY xxx	(→)	Create new vocabulary named xxx.
VLIST	(→)	Print names of all words in CONTEXT vocabulary.

MISCELLANEOUS AND SYSTEM

(	(→)	Begin comment, terminated by right paren on same line; space after (.
FORGET xxx	(→)	Forget all definitions back to and including xxx.
ABORT	(→)	Error termination of operation.
' xxx	(→ addr)	Find the address of xxx in the dictionary; if used in definition, compile address.
HERE	(→ addr)	Returns address of next unused byte in the dictionary.
PAD	(→ addr)	Returns address of scratch area (usually 68 bytes beyond HERE).
IN	(→ addr)	System variable containing offset into input buffer, used, e.g., by WORD.
SP@	(→ addr)	Returns address of top stack item.
ALLOT	(n →)	Leave a gap of n bytes in the dictionary.
,	(n →)	Compile a number into the dictionary.

Forth Interest Group, P.O. Box 1105, San Carlos, CA 94070

A BEGINNERS GUIDE TO PEGASUS TINY BASIC

INTRODUCTION

This book will teach you how to communicate with your Pegasus Computer. You will learn how to speak its language so that by giving it meaningful instructions you can make it do what you want it to do. That's all programming is, by the way.

There are many computer languages. Your Pegasus understands a language called Tiny BASIC which is a simplified form of BASIC (BASIC stands for Beginner's All-purpose Symbolic Instruction Code).

Tiny is perhaps the best language for the beginning microcomputer programmer. It is easily learned (as you will soon see) and programs may be developed quickly. For the more experienced programmer Tiny can form the basis of a system whose sophistication may be indefinitely extended.

So lets get started. Get to know your Pegasus. It can do an infinite number of things for you.

CHAPTER 1

GETTING STARTED

In this chapter we will introduce you to your Pegasus. You will learn how to use your keyboard and how to control the output display on your T.V screen.

Connect your computer by referring to the appropriate section in your Computer Operation Manual.

Switch it on and you will be greeted by the following heading on your television screen:

```
AAMBER Pegasus 6809
Technosys Research Laboratories
Tiny Basic 1.0
Monitor    1.0
Select one of the above:
```

Press T and your Pegasus will be 'ready' to go.

Do you see the flashing light? This is called the cursor and it indicates to you where on the screen the characters you type in on the keyboard will be displayed.

Try it. Type the following exactly as shown below:

```
PRINT  "HI, I'M YOUR PEGASUS COMPUTER"
```

When you reach the end of the line on the screen, keep on typing. The last part of the message will appear on the next line automatically. (Notice that the screen can display a maximum of 32 characters).

Now check your line. Is it alright?

If you made a mistake, no problem. Simply press the BACK SPACE key and you will observe the last character you typed will disappear. Press again, and the next will disappear, and so on...

This is what you should see on the screen;

```
Ready
PRINT "HI, I'M YOUR PEGASUS COMP
UTER"
```

Now press RETURN. This key tells the computer that you have finished the line. The computer then proceeds to execute it.

Your screen will then display:

```
Ready
PRINT "HI,I'M YOUR PEGASUS COMP
UTER"
HI, I'M YOUR PEGASUS COMPUTER
Ready
```

As you can see, the computer has obeyed your command and is 'Ready' for more.

Now type:

```
PRINT "2   2"
```

and press return. The computer obeys and prints your message:

```
2   2
```

How about some answers! Alright, try it without the quotation marks:

```
PRINT 2   2 (RETURN)
```

This time the computer prints something different - the answer to the expression 2   2.

Experiment further by typing the following:

```
PRINT   3+4 (RETURN)
PRINT   "3+4" (RETURN)
PRINT   "3+4 EQUALS",3+4 (RETURN)
PRINT   8/2, "IS 8/2" (RETURN)
PRINT   "6/2" (RETURN)
PRINT   6/2 (RETURN)
```

This demonstrates that the computer sees everything you type as either strings or numbers. If it is in quotation marks it is a string. If it is not in quotes it is a number. The computer sees it exactly as it is. The number might be in the form of a numerical expression (e.g. 3+4) in which case the computer reduces it to a single value.

By now it is likely that the computer has printed some unknown messages on your screen. If it hasn't, type the following, deliberately mispelling the word PRINT:PRIINT "HI" (RETURN)

The computer prints:

```
ERROR #4
```

This indicates that the computer has detected a syntax error. You will have to type the line again properly.

There are other types of errors too. Try:

```
PRINT 5/0 (RETURN)
```

The computer prints:

```
ERROR #8
```

This indicates an impossible division by zero command.

So whenever, Tiny BASIC detects an error while executing a line it generates an error message. A listing of error numbers and their corresponding meanings is given in Appendix 1.

CHAPTER 2

NUMBERS, VARIABLES, AND EXPRESSIONS

Before we go on it is important that we understand the meanings of numbers, variables and expressions.

NUMBERS

Pegasus Tiny BASIC is an integer BASIC, which means that all numbers in it have no fractional part e.g. 3.7, 4.02 and 3.1415926 are not integers.

Besides this there are two operating modes - signed and unsigned. When you first switch the machine on the computer automatically goes into the signed mode. In this mode integers in the range -32768 to 32767 are only allowed.

You can change to the unsigned mode by using the USIG statement.

Type:

USIG (RETURN)

In this mode integers in the range 0 to 65535 are only allowed.

To return to the signed mode use the SIG statement. Type:

SIG (RETURN)

If you input a number outside the allowed range, or the intermediate or final result to a calculation is outside the allowed range, then an error message will be returned.

VARIABLES

In Tiny BASIC a variable is represented by a single capital letter (A to Z) which directly corresponds to a location in the computer memory - we call this the name of the variable. The value of the variable is the number stored there.

For example, assign the variable A the value of 13 and the variable B the value of 7 by typing:

```
LET A = 13 (RETURN)
LET B =  7 (RETURN)
```

As LET is used very often in computer programs, the computer will understand you if you leave out the keyword LET altogether. From now on that's what we'll do.

OK. Now have the computer print out your numbers:

```
PRINT A,",",B
```

Notice the use of commas in that print statement.

Your computer will remember your assigned values for A and B as long as it is switched on, or until you decide to change them. Do this by typing:

```
A = 15 (RETURN)
```

Then, when you ask it to print A it will print 15.

EXPRESSIONS An expression is a combination of one or more numbers, variables or functions joined by operators.
You are probably most familiar with the mathematical operators which are:

+ addition

- subtraction

* multiplication

/ division

Let's say we want to divide the sum of 9 and 6 by 3. You might write this as:

9 + 6 / 3

Now, try it on your computer.

Type:

```
PRINT  9 + 6 / 3    (RETURN)
```

Is this the right answer to your problem? No, it isn't! This is because your computer has first worked out 6 divided by 3 (that's 2) and added this to 9 to give 11.

This demonstrates the way the computer works out arithmetic problems. The computer looks at the expression and does multiplication and division first. Then it does addition and subtraction.

So, to get the computer to solve the problem differently, you'll have to use parentheses. Type it as:

```
PRINT (9 + 6) /3 (RETURN)
```

That's better. The computer solves the expression in parentheses first before doing anything else.

What will your computer print as the answers to the following problems:

```
PRINT  12 - (6 - 4) /2
PRINT  12 - 6 - 4 /2
PRINT  (12 - 6 - 4) /2
PRINT  (12 - 6) - 4 /2
PRINT  12 - (6 - 4 / 2)
```

Check by typing them out.

Now, what happens if you type in:

```
RINT  (12 - (6 - 4)) / 2 (RETURN)
```

If the computer sees a problem with more than one set of parentheses it solves the inside parentheses first and then moves to the outside parentheses.

In other words, it does this,

$$\begin{tikzcd}
\text{(12 - (6 - 4) /2} \arrow[dr] & \\
& \text{6 - 4 = 2} \\
\text{(12 - 2) /2} \arrow[dr] & \\
& \text{12 - 2 = 10} \\
\text{10/2} \arrow[dr] & \\
& \text{10/2 = 5}
\end{tikzcd}$$

Can you imagine any problem with interger division ? What is 13/5 ?

Try it:

```
PRINT 13/5 (RETURN)
```

It gives 2 which is the whole number part of the result. If you want the remainder use the MOD operation.

Try it:

```
PRINT 13 MOD 5
```

It gives 3. You can use MOD just like you use *,/,+, and -.

There are other classes of operators available in Pegasus Tiny BASIC besides the mathematical operators - we'll look into these in later chapters.

As stated in the definition you can also include variablesin expressions.

Try it by typing:

```
PRINT A/3 + B (RETURN)
```

(Remember A was 15 and B was 7)

This feature is particularly useful in programs as we will soon see.

CHAPTER 3

INTRODUCTION TO PROGRAMMING

Type:

```
NEW (RETURN)
```

This is just to erase anything that might be in the Computer memory.

Now type this line (don't forget the line number, 10):

```
10 PRINT "HI, I'M YOUR PEGASUS COMPUTER"
```

Press RETURN. Nothing happened, did it?

What you have just done is to type your first program. Next, type:

```
RUN (RETURN)
```

And now you have just run it. Type RUN again - and yes, it runs again.

Add another two lines to the program.

Type:

```
20 PRINT "GIVE ME A NUMBER"
30 PRINT "AND I WILL DOUBLE IT"
```

Then Type:

```
LIST (RETURN)
```

Your computer obeys by listing your program. Your screen should look like this:

```
10 PRINT "HI, I'M YOUR PEGASUS CO
MPUTER
20 PRINT "GIVE ME A NUMBER"
30 PRINT "AND I WILL DOUBLE IT"
```

Don't attempt to type in a number because the computer isn't ready for it. Add the line:

```
40 INPUT T (RETURN)
```

Add one more line:

```
50 PRINT "2 TIMES ",T," IS ",2*T
```

Now list again, and your program should look like this:

```
10 PRINT "HI, I'M YOUR PEGASUS C

OMPUTER

20 PRINT "GIVE ME A NUMBER"

30 PRINT "AND I WILL DOUBLE IT"

40 INPUT T

50 PRINT "2 TIMES ",T," IS ",2*T
```

Now run it. The input statement prompts you with a question mark. Type in a number (integers only, remember, which the computer will label T) and then (RETURN)

Didn't you do well! This is what you should have got (it depends on your number of course);

```
HI I'M YOUR PEGASUS COMPUTER

GIVE ME A NUMBER

AND I WILL DOUBLE IT

?  9

2 TIME 9 IS 18
```

Run the program a few more times, inputting different numbers.

OK. Add another line. Type:

```
60 GOTO  10 (RETURN)
```

And run it.... the program runs over and over again without stopping.

That last GOTO statement tells the computer to go back to line 10:

```
10 PRINT "HI, I'M YOUR PEGASUS COMPUTER"

20 PRINT "GIVE ME A NUMBER"

30 PRINT " AND I WILL DOUBLE IT"

40 INPUT T
```

```
50 PRINT "2 TIMES ",T," IS ",2*T
60 GOTO 10
```

This is called a loop, and in this program it will cause it to run perpetually. However, you can get out of it by pressing the BREAK key, then any number and RETURN

Change line 60 so that it goes to another line number. How do we change a program line? Simply by re-typing it, using the same line number. Type:

```
60 GOTO 50
```

Your program listing should then look like:

```
10 PRINT "HI, I'M YOUR PEGASUS C
OMPUTER
20 PRINT "GIVE ME A NUMBER "
30 PRINT "AND I WILL DOUBLE IT"
40 INPUT  T
50 PRINT "2 TIMES ",T," IS ",2*T
60 GOTO 50
```

Run it.... OK, press the BREAK key when you have seen enough.

There is a more desirable way of getting out of the loop. Why not get the Computer to politely ask you if you want to end it?

Change line 60 to the following:

```
60 PRINT " DO YOU WANT IT DONE AGAIN?"
```

And add these lines:

```
70 R = INKEY : IF R = O GOTO 70
80 IF R = 89 GOTO 20
```

Then run the program....type your number .... then type Y and the program loops back again. If you type anything else(e.g."N") the program stops.

This is what the program looks like:

```
10 PRINT "HI, I'M YOUR PEGASUS C
OMPUTER"
20 PRINT "GIVE ME A NUMBER"
30 PRINT "AND I WILL DOUBLE IT"
40 INPUT T
50 PRINT "2 TIMES ",T," IS ",2*T
60 PRINT "DO YOU WANT IT DONE AGAIN?"
70 R = INKEY: IF R =0 GOTO 70
80 IF R = 89 GOTO 20
```

What are these new lines?

Line 60 simply printed a question.

Line 70 is infact 2 lines, the two statements being separated by the colon ":". The first part assigns the ASCII equivalent of the key depressed on the keyboard to the varia e R. (ASCII is the Standard Code for Information Intercha If no key is pressed then R is assigned 0. The second part of the line tests for this condition and loops back to INKEY if it is true. However, as soon as a key is depressed it gets out of the loop and proceeds to the next line...

Line 80 tells the computer to go to line 20 IF (and only IF) the Y key (THAT's ASCII 89) has been depressed. If not, the program ends as there are no more lines after this.

This chapter has covered a lot of important concepts of Pegasus Tiny BASIC. Don't worry if some things are not absolutely clear. Experiment with your computer and above all, enjoy it.

CHAPTER 4

MORE PROGRAMMING

In this chapter we will practise using functions and statements in Pegasus Tiny BASIC.

Type this:

```
10 FOR x = 1 TO 10
20 PRINT "X =", X
30 NEXT X
40 PRINT "FINISHED"
```

Run the program.

See how it has printed X for X = 1 to 10.

Now replace line 10 with the following:

```
10 FOR X = 5 TO 8
```

And run again.

Lets look at the program listing:

```
10 FOR X = 5 TO 8
20 PRINT "X=", X
30 NEXT X
40 PRINT "FINISHED"
```

It's clear that line 10 determines the starting and ending values of the variable X. Line 30 tells the computer to get the next number - the NEXT X - and to jump back to the line following the FOR ... TO... line (i.e. line 20) until it reaches the last number. At this stage it goes straight on to execute the final statement.

We can further investigate the path of program execution by using the TRON statement. Try it. Type:

```
TRON
```

and press RETURN.

Now run the program again. This statement has turned on a trace, which provides a line number listing for statements as they are executed. The trace should look like this:

```
<10> <20> X = 5
<30> <20> X = 6
<30> <20> X = 7
<30> <20> X = 8
<30> <40> FINISHED
```

See how the program keeps jumping from line 30 to line 20 until it eventually goes from line 30 to line 40 and stops.

To turn the trace off, type:

TROFF

and RETURN

If you like, run your program again to see if the trace has gone.

An extra feature of the FOR... TO... statement is that you can specify the actual STEP size. Change line 10 to:

```
10 FOR X = 2 TO 10 STEP 2
```

And run the program. See how X goes from 2 to 10 in steps of 2. Before, when we didn't specify the step size it assumed STEP 1. What will happen if line 10 is replaced with:

```
10 FOR X = 3 TO 10 STEP 3
```

Try it ... and see that it loops back only for $X \leq 10$.

How about:

```
10 FOR X = 10 TO 1 STEP -1
```

Yes, it counts backwards too.

Now try a new program - that's right, type NEW and RETURN - then type:

```
10 FOR    X = 1 TO  3
20 PRINT  "X = ", X
30 FOR    Y = 1 TO 2
40 PRINT  "Y = " ,Y
50 NEXT   Y
60 NEXT   X
```

Run it.... This is what you should get:

```
X = 1
Y = 1
Y = 2
X = 2
Y = 1
Y = 2
X = 3
Y = 1
Y = 2
```

Notice how it loops within another loop.

Programmers call this a "nested loop ".

Now for something completely different.

Type in this new program:

```
10 S = RND /26
20 PRINT "GUESS THE NIMBER"
30 INPUT G
40 IF G = 5 THEN GOTO 70
50 PRINT "NO, TRY AGAIN"
60 GOTO 30
70 PRINT "YES, THAT'S IT"
```

And run it... guess numbers between 0 and 9 inclusive (the division by 26 in line 10 gives us this range).

The new statement type encountered here is the IF...THEN conditional statement. The statement tests the expression G = 5 and IF false will skip immediately to the next line; but IF that statement is true THEN it executes the next statement GOTO 70.

The condition is often the result of a relational operation. In Tiny BASIC these are:

=	Equal to
<>	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

These are often combined with logical operators, AND, OR, NOT to perform quite complex tests, here's an example:

```
600 IF A = 0 OR (C< 127 AND D<>0) GOTO 100
```

This will cause a branch to line 100 if A is equal to 0 or if both C is less than 127 and D is not equal to zero.

This type of expression essentially evaluates to 0 for false and -1 for true.
Besides being used for true/false evaluation, logical operators can operate on binary numbers. For example, type:

PRINT 6 AND 7 (RETURN)

This gives decimal 6 which is 0110 ANDed with 0111.

So far we have been looking at relatively short programmes. Before long, no doubt, you will be so proficient with your Pegasus that you will be writing quite long and complex programs.

We'll now look at some expressions which will help us to keep things in order. Type and RUN the following:

```
10 PRINT "EXECUTING THE MAIN PROGRAM"
20 GOSUB 400
30 PRINT "NOW, BACK IN MAIN PROGRAM"
40 END
400 PRINT "EXECUTING THE SUBROUTINE"
410 RETURN
```

Line 20 tells the Computer to go the the Subroutine beginning at line 400. RETURN tells the Computer to continue execution with the line following the GOSUB expression. The END expression is necessary to separate the main program from the subroutine.

Subroutines are written for operations that are frequently required. They result in economy of effort when it comes to writing programs.

One final point - you can use the REM statement to place remarks and comments thoughout your program. Anything following the REM statement is ignored. These remarks are often placed at different points in a program, particularly at the beginning of subroutines to explain how uncleat or complicated sections of the program work.

Here is a final program that illustrates these points.
Try it.

```
10 REM THIS PROGRAM RAISES A
20 REM NUMBER TO AN EXPONENT
30 INPUT "NUMBER"N
40 INPUT "EXPONENT"E
50 GOSUB 1000
60 PRINT:PRINT N," EXPONENT ",E," IS ",A
70 END
80 REM   ---------//---------
1000 REM THIS SUBROUTINE DOES
1010 REM THE ACTUAL EXPONENTIATION
1015 IF E=0 THEN A=1:RETURN
1020 A=1
1030 FOR X=1 TO E
1040 A=A*N
1050 NEXT X
1070 RETURN
```

By now you should feel to be in complete control of your Pegasus.
Try writing some programs of your own.
Good luck, and have fun!

If YOU HAVE READ 'a beginners guide to pegasus tiny basic'

THEN YOU WILL LOVE

"A Gentle Introduction to Pegasus Tiny Basic"

now available in your Aamber Pegasus Manual!!

A Gentle Introduction to Pegasus Tiny Basic

The BASIC Language

BASIC is the most common computer language in the world today. The word BASIC is an acronym, that stands for:

Beginner's All-purpose Symbolic Instruction Code.

BASIC is a computer program that was originally developed at Dartmouth College in the U.S. as a means of teaching students the principles of computer fundamentals, as well as making it easier to write more computer programs. BASIC itself is usually written in machine code assembler, although higher-level languages have been used.

Bells and Whistles

Hundreds of BASIC interpreters (i.e. programs that will accept and interpret a program written in BASIC) have been written since the first version, and each one is usually unique in its features and limitations. Theoretically anyone with enough knowledge and time can write a BASIC interpreter, although not many people do.

When they do, however, each likes to add their personal touch, in the form of special features, and this is known as adding Bells and Whistles. (We have not stinted in this tradition.) Thus, although BASIC is so common, there are many different dialects.

Where Do I Start?

At the beginning, of course! We'll look at the idea that a computer program is like a recipe. Let's make a milkshake, for example:

```
Fetch container.
Fetch milk.
Pour milk into container.
Fetch flavoured powder.
Add powder to milk in container.
Pick up container.
Shake!
Oops!
Put down container.
Clean up mess.
```

Put lid on container.
Shake!
Take lid off.
Drink milkshake.
End of recipe.

A trivial, yet useless example. Each line, or statement, is a command, or instruction (apart from 'Oops', which is a comment, or perhaps invective). The statements were executed sequentially, starting from the top. Note that each statement leaves out a very large amount of detail - like what sort of container is used, where the milk came from, what flavour powder was used - even whether it was enjoyed or not!

Computer programs are quite like this in their lack of detail - a great deal is implicit or assumed. Computer programs are much simpler, however, in the actions that they describe, in that the tasks a computer performs are (usually) logical and straightforward (unlike the 'real' world of gravity and spilt milk.)

Using Numbers

BASIC, like many other computer languages, is designed to work with numbers. Usual operations in BASIC are addition, subtraction, multiplication and division (+,-,*,/). There are two ways that numbers are used in BASIC - constants and variables.

A constant has a value which it keeps for as long as the program runs. Typical constants are 7, 24, 0, -32768, 2000. Variables are symbols for memory cells that may contain numbers. In Pegasus Tiny BASIC, we use the letters A through to Z to represent these variables. Thus, we can refer to a variable in a computer program without having to know its value. When a computer program is first RUN, all the variables A to Z will have a value of zero.

Number Size

Pegasus Tiny BASIC is an <u>integer</u> BASIC, which means that all numbers in it have no fractional part. E.g. 3.7, 4.02 and 3.1415926 are not integers. Further, the Pegasus has 16 bit signed two's complement and 16 bit unsigned numbers, which means that for signed numbers you are limited to -32768 to 32767, while unsigned integers have a range of 0 to 65535. Any outside this range will cause an error.

Number Representation

Numbers are stored internally in binary, but to make it easier for people to handle them, we have provided two forms of integer format: numbers may be output (printed) in decimal or hexadecimal (base 16). For instance, if variable A contains 19, then we can print the two forms thus:

```
        PRINT A," ",HEX(A)
```

which will print out

```
        19 13
```

For inputting numbers, they must always be in decimal, but may be signed or unsigned. Hexadecimal numbers may be used directly in a program by preceding them with a dollar sign ($), e.g.:

```
        PRINT $13
```

will print

```
        19
```

on your television screen. Both signed and unsigned numbers may be used, and may be selected with two statements,

SIG and USIG , which stand for SIGned and UnSIGned. Signed numbers have a range of -32768 to +32767, while unsigned are in the range of 0 to 65535. Note that an unsigned number greater than 32767 will be printed as a negative number if the program switches back to signed mode.

Arithmetic

In Tiny BASIC, arithmetic may be done with 'expressions'. An expression is a group of tokens, each of which has a definite value associated with it, that is built up using a set of possible operators, and is solved as an algebraeic expression that returns a single numeric value. Now that we've confused you, let's clear it up with some examples:

```
        A*3+7*R
        (3+Q)-(21/L+(8*I))       note that parentheses must match
        2+2
        1                        yes, a number is an expression too
        $4F OR 51                note the Boolean operator
        ABS(-R)                  functions are expressions too
```

A variable or constant by itself may also be considered an expression, and expressions may consist of other expressions, as long as they are logically organised, and the number of left and right parentheses match correctly. Unlike some BASICs, nearly any complexity of expression may be used.

Operators

Constants, functions, variables and expressions may be related by operators to form a new expression. All the operators work with 16 bit integers, and return 16 bit integers as results.

+	Simple addition
-	subtraction
*	multiplication
/	division
MOD	modulus, same as taking remainder after a division instead of the quotient. E.g. 7 MOD 6 yields 1.
+	unary plus, e.g. +7 by itself
-	unary minus, e.g. -12
NOT	returns one's complement, e.g. NOT $F012 returns $0FED
AND	logical AND, may also be used as Boolean connector
OR	logical OR, similar to AND

Note that expressions are no good unless you do something with them, using one of the statements available. The simplest statement to use is the assignment statement, LET. This is used for assigning values to variables, e.g.

```
        LET Q=I+9              The '=' means 'is assigned'
        L=17*(8+T) MOD 15      The LET is optional
        I=I+1
```

The last statement is of particular interest since it illustrates how a variable is fetched, incremented, and then stored back in to the same memory cell again. The '=' sign does not mean 'equals', but means 'is assigned the value of'. Note that

3=A or 3=7 are illegal, and will give an error message.

Spaces may be used freely in expressions, however they may not be imbedded inside function or statement names.

A quick way of using your Pegasus for math is to use the PRINT statement in conjunction with an expression. Remember that the question mark, '?', is shorthand for PRINT. For instance, ? 7*8 gives 56. When expressions are evaluated, they are executed in an order defined by the OPERATOR PRECEDENCE. This means that values that are conjoined by certain operators will be executed before others in an expression. The precedence order is:

1st	constants, variables
2nd	functions (includes special @ function)
3rd	unary - or +, NOT

4th operators * / MOD AND

5th operators + - OR

The order of evaluation may be changed by using parentheses. Some examples are given for your enjoyment:

```
3+4 * 2+5  resolves to 16
(3+4) * (2+5) evaluates to 49
```

A special class of operator, the relational operator, is covered in the section on Booleans.

6th relational operators (lowest precedence).

<u>Booleans</u>

A Boolean expression is similar to an arithmetic expression, apart from the use of the relational operators. Any relation evaluates to 0 for FALSE and non-zero for TRUE. The most usual non-zero value found will be -1 (hex $FFFF). The relational operators are:

=	Equality
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to

Boolean expressions may be mixed with arithmetic expressions, leading to results like:

```
A=B=C+1
```

Boolean expressions may be used with the IF statement, e.g.

```
IF Q=7 THEN END
IF T THEN GOTO L
```

Here, L is treated as an unsigned line number that the program will GOTO if T is non-zero.

Statements and the Editor

Program lines in BASIC are usually organised in a strictly sequential manner, using line numbers in the range of 1 to 65535. A program will consist of a series of lines, where each line consists of one or more statements (separated by the colon ':'), and is executed sequentially, except where a special statement will change the flow of program logic. Here is a sample program that will print out the integers from between 1 and 10.

```
10 I=0 : REM I is assigned a value of zero
20 I=I+1 : REM I is incremented
30 PRINT I : REM Print out the value contained in I
40 IF I=10 THEN STOP : REM Stop when I reaches 10
50 GOTO 20
```

Follow the program through by hand, or better still, try it on your Pegasus! When typing the program in, terminate each line with the RETURN key, and correct typing mistakes by using the BACK SPACE key. If you notice a mistake on a line that you have already typed in, simply re-type the correct version (with the same line number), and the old line will be automatically replaced. To remove a line entirely, just type the line number by itself, followed by the RETURN key.

Experiment with your own programs to print out different sorts of number sequences, until you are fully satisfied with the material covered so far. If you have trouble stopping a program once you have started it, tap the BREAK key.

Summary of Statements

PRINT expressions, string constants

This statement will evaluate and print results of expressions, as well as printing string constants. A string constant is a collection of characters delimited by double quotes, e.g.

"FRED NURKE WAS HERE"

"THAT'S all FOLKS"

Expressions will be evaluated, ahd the results printed, with no leading or trailing spaces. String constants and expressions MUST be separated by commas. Upon completion of the print statement, the cursor will move to the beginning of the next line, unless the PRINT statement is terminated with a comma. The cursor may be positioned to anywhere on the screen at any stage in the PRINT by using the form [x,y]. For example,

PRINT [10,2],"HELLO",

will move the cursor to column 10, line 2, and print "HELLO", leaving the cursor immediately after the 'O'. The vertical position 'y' is optional, but if it is included then it must be separated from the 'x' column position by a comma. There are three functions that may only be used with the PRINT statement, since all of them produce some sort of output, without returning a value. These output functions are detailed below:

CHR(expression)

This will output the ASCII character that is represented by the result of the expression. The result is forced into the range of 0 to 255 (decimal), or $00 to $FF (hex). If the number is greater than 127, then the character will be inverted. Note that characters in the range of 0 to 31 and 128 to 159 will not print, but will cause one of the control functions to be executed.

HEX(expression)

The expression is evaluated, range 0 to 255, and the appropriate hex number is output, range $00 to $FF.

RAW(expression)

This function is very similar to CHR, except that values in the range of 0 to 31 and 128 to 159 will have a special character output, without executing the appropriate control function.

Note that all functions that require an expression in brackets, must not have a space before the left parenthesis. (The RAW function is associated with the RAWON and RAWOFF statements, covered later in this document.) Examples of their use are given below, for you to try on your Pegasus.

```
PRINT "Print a hex number:",HEX(19),CHR(10),RAW(0)
PRINT CHR($46),CHR($52),RAW($45),CHR($44)
PRINT "There are ",Q," beans in the box."
PRINT CHR(12) : REM Clear screen
PRINT RAW(12) : REM Output Greek letter 'nu'
```

LIST   starting line, ending line

Program lines may be listed out, either as individual lines, subranges of lines, or the entire program. The expressions are both optional, and are unsigned numbers always. If a line is specified that is not in the program, then the nearest one to it will be used. This is the only case in which such leniency is tolerated. If you try to force Tiny Basic to use the 'nearest' line number in other statements, then a small quantity of plastic explosives attached to your Pegasus will be detonated, removing yourtyping fingers.
YOU HAVE BEEN WARNED.
Note that the starting and ending lines may be expressions, and the LIST statement may be part of a BASIC program.

RUN   expression

The RUN statement will initialize all variables, then start program execution at the line number specified. If no number is specified, the program will start at the beginning.

<u>INPUT</u>	"string constant" input list This statement, unlike many others, can only be executed with a line number as part of a program. Its purpose is to request numbers from the user for input to the program. The string constant (if specified) will be printed out as a prompt to the user before input is requested, and must <u>not</u> be followed by a comma. When each input expression (yes, expressions can be input) is typed, it must be terminated with a RETURN key. Only one string constant may be specified, and if used it must be immediately after the INPUT.
<u>FOR</u>	variable = start value TO end value STEP step-size This is the standard BASIC looping statement. This will cause all statements between the FOR and its appropriate NEXT to be executed repeatedly until the variable's value reaches or exceeds the end value. Note that the step size may be positive or negative. If the step is not given, it will default to one.
<u>NEXT</u>	variable name Terminating statement for FOR loops.
<u>GOTO</u>	expression The expression will be evaluated to an unsigned 16 bit integer, and if a line is found with a matching line number, then that line will be executed next.
<u>GOSUB</u>	expression The expression will be evaluated, and the subroutine which starts with the matching line number will be called, returning to after the GOSUB statement when it reaches and executes the RETURN statement. GOSUBs may be nested to any depth, depending upon free ram space for the stack.
<u>RETURN</u>	This statement indicates the logical end of a BASIC subroutine.
<u>EXIT</u>	The EXIT statement will return you back to the Pegasus Menu selection mode.

NEW

This statement will zero all variables, as well as deleting all program lines.

STOP

The STOP statement will cause program execution to terminate, returning to the line edit mode. Execution may be continued with the CONT statement, as long as the program has not been changed. Any other immediate mode statement may be executed however.

END

The END is similar to the STOP statement, except that the CONT statement will not continue program execution after an END.

CONT

The CONT will cause program execution to continue, as defined by the STOP and END statements.

REM

Any user remarks may appear after this statement, since they will be ignored by the BASIC interpreter. The REMark is terminated by the end of line or a colon.

LET variable name = expression

The assignment operation assigns the value of an expression to the named variable. Only variables and the special function '@' may be used on the left side of the '=' sign.

IF expression THEN statement or expression

The IF statement will evaluate the first expression, and if it is zero, then the remainder of the statement will be skipped, going to the next line. Upon a true state, then the part after the THEN will be executed if it is a statement, or if it is an expression, then it will be evaluated, and a GOTO will be executed.

TRON

This statement will bring the trace mode into effect, whereby each line number will be printed out as the line is executed, following the flow of program execution as the RUN proceeds.

TROFF

This statement will turn the trace mode off.

SIG

This forces the system to accept and print only signed numbers, in the range of -32768 to +32767. A point to note here is this example:
PRINT HEX($B010/256) will yield B1, instead of the expected value of B0. This is because although the hex number is unsigned, SIGned mode is in effect, and must be disabled using USIG before the correct result may be achieved.

USIG

The system can accept unsigned integers, in the range of 0 to 65535, for input, output, and arithmetic expressions. Note that this mode is checked when determining whether the result of an expression is outside its range.

SAVE

BASIC programs are saved on cassette tape, with a filename that you may specify (8 characters only). The BLUE tagged lead goes into the MIC jack, while the YELLOW lead goes into the EAR jack.

LOAD

Previously SAVEd programs may be loaded from cassette tape. The filename and load area will be printed.

POKE expression , expression
The first value resolves as an unsigned 16 bit address, which gives the location to poke the second value into.

<u>LINES</u> expression

This statement controls the number of lines displayed on the screen. The expression must resolve to a number in the range of 1 to 16, or an error will stop execution of the program. Reducing the number of lines displayed has the result of speeding up program execution proportionally.

<u>RAWON</u>

This statement will turn on the RAWMODE flag. This means that any control code that is echoed to the screen through the normal PRINT routine, or through typing in lines, will cause a special character from the character generator ROM to be printed, without executing the control function.

<u>RAWOFF</u>

Turns RAWMODE off.

<u>CLS</u>

This statement, when executed, will clear the video display screen, and move the cursor to the top left hand corner.

BASIC Functions

Pegasus Tiny BASIC has a number of functions, each of which may be used in expressions, (apart from the ones specified in the PRINT statement description). Note that there must be no spaces between the function name and the left parenthesis.

ABS(expression)

Takes absolute value of the argument.

PEEK(expression)

Returns byte at address given by expression.

FREE

Returns number of free bytes available in system RAM.

RND

Returns pseudo-random number in the range 0 to 255.

USR(expression)

Calls a machine language routine subroutine in memory at the address specified by the expression - the function value returned reflects the state of the X register, and may be data or an address pointing at more data.

@(expression)

Special function that implements a one dimensional integer array that utilises all available RAM space. The function may be used anywhere that a variable name is used, including in assignment statements. Unlike variables, the @ array is not cleared by the NEW statement. The array index is unsigned, starts at zero, and its size is FREE/2-1.

INKEY

This function will scan the keyboard to see if a key has been pressed - if one has, then its ASCII value in the range of 1 to 127 will be returned, else if no key has been pressed then zero will be returned.

Information for Experts

The variables, since they are in fixed locations in RAM, (in the 4K system only), can be accessed by machine code subroutines by referencing directly their addresses. There are 26 variables, and they start at $B03C.

Nearly all tokens in Tiny BASIC have a shorthand form, for instance, '?' means PRINT, and 'e' means PEEK(. Try finding out what the rest are - this information will be published in the newletter.

When a program is executing on the Pegasus, it may be stopped in its tracks by using the BREAK key on the keyboard. This is functionally equivalent to the program encountering a STOP statement.

Output that is being sent to the screen may be paused by use of the ESCAPE key, on the upper left of the keyboard. Tapping once will stop, tapping again will start.

If inverse video characters are required inside strings, they may be effected by tapping the blank key on the extreme lower right of the keyboard. Tapping the key again will remove the inverted state. Note that only characters inside double quotes will remain inverted. Inverted characters may also be generated by setting the most significant bit of the byte for the ASCII character (ASCII equivalent greater than 128).

RAWMODE may be set and reset by tapping the blank key on lower right of keyboard, second one in. When in effect, any control characters typed will appear as a special printing character, without the appropriate control function being executed. Note that the RETURN key, being a control code, will not work as it should until the RAWMODE flag is turned off by tapping the second blank key again. This feature allows you to insert control codes into strings, and then the output of those control codes as characters or functions may be governed by use of the BASIC statements RAWON and RAWOFF.

BASIC may be re-entered from the monitor after using the PANIC button by jumping to $0D offset from its start.

Basic Error Numbers and Messages

Whenever a syntax or execution error occurs, then an error number will be printed out, each number matching to one of the error messages given below:

(1) Out of Memory
This means that there is insufficient RAM space between the program end and the stack to perform the last operation.

(2) Invalid Line Number
A line number was specified that either does not exist, or is illegal.

(3) Next Without For
A NEXT statement was found without the appropriate FOR statement.

(4) Syntax Error
This is a general error that occurs whenever there is incorrect syntax in a program line.

(5) Return Without Gosub
A RETURN statement was executed, but the system did not find a GOSUB to return to.

(6) Immediate Mode Illegal
A statement was executed in immediate mode that is illegal for that mode.

(7) Overflow Error
The results of an arithmetic operation exceed the current range specified.

(8) Divide by Zero
An attempt was made to divide a number by zero.

(9) Screen length Error
The LINES statement must have an argument in the range of 1 to 16 only.

APPENDIX I

ASCII CHARACTER CODES

1968 ASCII American Standard Code for Information Interchange. Standard No. X3.4 -1968 of the American National Standards Institue

	b6	0	0	0	0	1	1	1	1
	b5	0	0	1	1	0	0	1	1
	b4	0	1	0	1	0	1	0	1
$b_3 b_2 b_1 b_0$		0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	A	LF	SUB	*	:	J	Z	j	z
1 0 1 1	B	VT	ESC	+	;	K	[	k	{
1 1 0 0	C	FF	FS	,	<	L	\	l	¦
1 1 0 1	D	CR	GS	-	=	M	]	m	}
1 1 1 0	E	SO	RS	.	>	N	^	n	~
1 1 1 1	F	SI	US	/	?	O	_	o	DEL

GLOSSARY

ACIA Abbreviation for Asynchronous Communications Interface Adapter. An INTEGRATED CIRCUIT used in computers to transfer data to and from the computer in a serial fashion.

ACCUMULATOR Register within the CPU that is used to perform arithmetic and logical operations on data. The Aamber 6809 has two 8-bit accumulators.

ADDRESS Each accessible location in memory has an address associated with it which is used when reading data from, or writing to, a memory location. The maximum number of addressable locations is referred to as the address space.

ASCII Abbreviation for American Standard Code for Information Interchange; this assigns a distinct number in BINARY to every alphabetical character, as well as a number of punctuation and special codes. It is a seven bit code. See appendix for a full description.

ASSEMBLER Program used to assemble SOURCE CODE into OBJECT CODE. This is usually a process of one to one translation.

ASYNCHRONOUS Data sent by this method is sent only when transmitting device is ready, i.e., at no specific times. This differs from SYNCHRONOUS in that timing is not important.

BACKUP Copy; duplicate. Backup of programs is important because machine media is not always reliable.

BAUD A measure of transmission speed in BITS per second. For example, 300 baud equals a speed of 300 BITS per second. Most transmitted characters contain a total of 10 bits so 300 baud equals 30 characters per second.

BCD Abbreviation for Binary Coded Decimal; uses 4 BINARY BITS to represent the decimal digits 0 - 9.

BIDIRECTIONAL An electrical signal path that is capable of both transmitting and receiving on the same path.

BINARY A number base used in digital computers, only two are allowed states:
1 = high = on = true = +5V and 0 = low = off = false = 0V

BIT The smallest unit of storage capacity in a computer. A bit can be equal to either a 0 or a 1. A bit is a shortened form of Binary digIT.

BOOT (bootstrap) A technique or device designed to bring itself into a desired state by means of its own action. For example a machine routine whose first few instructions are sufficient to bring the rest of itself into the computer from an INPUT device.

BUFFER A device used to compensate for a difference in signal level or drive capability when transmitting from one drive to another. The term is also used to describe a temporary storage area in a computer.

BUS A series of electrical signals, usually available on a connector(s), which contains all of the electrical information necessary to connect an external device to the main parts of a computer or device.

BYTE A group of BINARY digits treated as a logically connected entity. Most common size is 8 bits.

CAPACITOR An electrical component capable of storing charge.

CARRIAGE The device in a printer or typewriter that travels along the PLATEN to print characters. Carriage Return is the command that causes the carriage to return to the left-hand side.

CMOS Abbreviation for Complementary Metal Oxide Semiconductor. These devices are fast switching gates,and have very low current consumption. They are very susceptible to static electricity and the pins should not be handled. For this reason the pins are placed in a conductive material e.g. conductive foam or aluminium foil.

CONDITION CODE Single BIT used as a FLAG in the CPU. These are often set or reset according to the results of a logical or arithmetic operation.

CPU Abbreviation for Central Processor Unit. It is the portion inside a computer that performs all logic and arithmetic operations.

CRT Cathode Ray Tube. The glass screen (picture tube) of a computer terminal.

DB-25 CONNECTORS The standard 25 pin connectors used to connect external serial and parallel devices to the computer. The connectors are 'D' shaped and are of two varieties: DB-25P (plug type) and DB-25S (socket type).

DIODE Component that allows current to flow through it in one direction only.

DISKETTE A flat circular plate of Mylar with a magnetic surface on which data can be stored and retrieved.

DYNAMIC MEMORY In dynamic memory a bit is remembered by a charge on capacitance. This charge leaks away and so the memory needs extra circuitry to perform a refresh regularly. This memory is cheaper and smaller as each chip does not require as many TRANSISTORS as, say, static memory.

EPROM An abbreviation for Erasable Programmable Read Only Memory. A non-volatile memory INTEGRATED CIRCUIT which can be programmed by the user and can only be erased by prolonged ultraviolet light.

FIFO Abbreviation for first in, first out. Some data structures operate on this principle.

FILES A collection of related records, program material or text, treated as a unit.

FIRQ Abbreviation for Fast Interrupt Request, similar to IRQ, but only a subset of the machine state is saved on the stack.

FIRMWARE This is a cross between hardware and software e.g. hardware that can be programmed such as EPROMS.

FLAG A single bit that the CPU sets and clears, to remember if an action or condition has occurred.

FLOPPY DISK The storage medium (diskette) for a disk unit which is flexible and removable. Common sizes of floppy disks are eight inches and five inches in diameter.

FULL DUPLEX Method of transmission where each end can simultaneously transmit and receive. Data is echoed back to the transmitting device after going through the receiving device.

GROUND Or Earth; an electrical term meaning at a potential of zero volts.

HANDSHAKE Sequence of events requiring mutual consent of conditions on two or more ends, prior to change. The process where digital signals are transferred by means of a sequence of status and control signals.

HARD DISK The storage medium for a disk which is not flexible. Hard disks are very expensive but are capable of storing much more data, and accessing it at a much faster rate.

HARD SECTORED A method of disk formatting in which the hardware of the disk drive sends out a signal for each encountered sector on the disk. A hard sectored FLOPPY DISKETTE has one sensing hole in the disk for each series of sectors.

HARDWARE The INTEGRATED CIRCUITS, TRANSISTORS, connectors, plugs, circuits and other mechanical and electrical parts that make up a computer.

HEATSINK Piece of alloy, or other metal, connected to certain components to aid in the dissipation of heat, preventing over-heating.

HEXADECIMAL Representation of numbers in base 16. The digits used are 0 - 9 and letters A - F. The A - F correspond to base 10 numbers 10 - 15, with 10 in hex corresponding to 16 in decimal:
e.g. 34 HEX = 52 DECIMAL

HALF DUPLEX This occurs when a TERMINAL is on-line to a computer, and must echo characters typed locally.

INDEXING A method of addressing using data contained in one of the CPU's pointer registers and an offset contained in memory, the offset is added to the registers contents (base address) to give the effective address.

INDUCTOR Device for storing electrical energy in the form of a magnetic field.

INPUT The Process of getting data into the computer for processing, common input devices are tapes, disks and keyboards.

INTERFACE Something that connects one device to another. An interface board in a computer is used to connect an external device (peripheral) to the main BUS of the computer.

INTERGRATED CIRCUIT Collection of logic gates and circuit elements that constitute a functional block, contained in a single package.

INTEGER All positive and negative numbers that do not have a fractional part. The usual range for integers when stored as 16 BIT quantities is -32768 to 32767.

INTERPRETER Program written in machine language that will read and execute a program written in a high-level language, usually line at a time.

INTERRUPT To stop a process in such a way that it can be resumed. The need for interrupts is due to external devices demanding processor attention. These demands are ASYNCHRONOUS so the processer needs to know when they occur so that it can service the interrupt. There are other reasons, such as power failure etc.

IRQ Abbreviation for Interrupt Request.

LATCH An electrical device which can be accessed by the computer and used for control purposes - a latch has the ability to remember or store the data written to it.

LIFO Abbreviation for Last In, First Out; used for STACKS.

LSB Abbreviation for least significant BIT in a BINARY word or BYTE. It is the right hand BIT, usually numbered zero.

MACHINE CODE Code used as instructions for the microprocessor. (Synonym for Object Code.)

MICROPROCESSOR The main INTEGRATED CIRCUIT of a computer which executes and processes the actual program instructions.

MONITOR Software or hardware that observes, supervises, controls, or verifies the operations of a system. The MONITOR is the control program that is stored in ROM and executed by the computer when it is first powered up.

MOS Abbreviation for Metal Oxide Semiconductor. These devices are similar to CMOS and the same precautions apply.

MOTHERBOARD Board within the computer with very little circuitry. Contains connectors for all other boards to plug into.

MSB Abbreviation for the Most Significant Bit in a BINARY word or BYTE. It is the left hand BIT.

NIBBLE Four BITs or half a BYTE.

NMI Abbreviation for Non-Maskable Interrupt. This interrupt cannot be masked out by the CPU but must be serviced immediately. This is used for such occurances as power failure.

OBJECT CODE The computer executes all programs in a special machine-specific code that is usually stored in machine-readable form - this is designated object code.

OCTAL Representation of numbers using Base 8. Octal uses digits 0 - 7 where 10 in Octal stands for 8 in Base 10.

ON LINE Equipment or devices which are connected to, and are under the direct control of, the computer.

OP CODE Mnemonic symbol that directly represents a machine language instruction or operation. These codes are used by the ASSEMBLER.

OUTPUT The process of obtaining data from a computer after it has been processed; common output devices are printers, on-line terminals,disks and tapes.

PARALLEL INTERFACE An interface from the computer to the outside world in which each piece of data is carried on a separate electrical line.

PARITY Method used to detect errors after the transmission of data. The ASCII code uses 7 BITs in a BYTE, the eighth BIT being a parity BIT.

PERIPHERALS Any device which is connected to the computer (disk drive, printer, terminal etc.)

PIA Abbreviation for Peripheral Interface Adaptor. An INTEGRATED CIRCUIT used in 6800/6809 computers to transfer data to and from the computer in a parallel fashion.

PLATEN Long cylindrical part of a typewriter or printer which is used for a backstop for the head, typeball or typewheel when printing onto paper.

PRINTED CIRCUIT BOARD (PCB) A board with copper connectors following set design and organised on the board via a printing process.

PROM Abbreviation for Programmable Read-Only Memory. A non-VOLATILE memory INTEGRATED CIRCUIT which once programmed by the user cannot be erased.

RAM Abbreviation for Random Access Memory. This is the main memory of the computer which will lose data when power is removed. Data can be accessed from any addressable location at any time.

REGISTER Device within the CPU used for temporary storage of data while arithmetic or logical operations are performed on it.

RESISTOR Component usually used for changing voltages.

RESET To return a machine or device to its default state.

RIBBON CABLE A flat group of wires physically fastened together side by side.

ROM Abbreviation for Read Only Memory. A non-volatile memory INTEGRATED CIRCUIT which cannot be erased.

RS232 One of many so-called 'standard' serial communications interfaces that uses changes in voltage levels and a clock reference to transmit and receive information. Typical voltage levels are -12 to +12 volts.

SERIAL INTERFACE An interface from the computer to the outside world in which all data BITs are transferred sequentially over one electrical line.

SOFT SECTORED A method of disk formatting in which the HARDWARE of the disk drive sends out only one signal for the beginning of each track. A soft sectored FLOPPY DISKETTE has one sensing hole in the disk.

SOURCE CODE This consists of a human-readable grouping or file of language instructions used as the input for an ASSEMBLER, INTERPRETER or COMPILER.

SPIKE Most power supplies suffer from very fast fluctuations in voltage caused by such things as washing machines and hair-driers, which can interfere with the correct functioning of a computer. Such an event is called a 'spike'.

STACK A stack consists of a collection of memory cells pointed to by a stack pointer, and arranged so that the last element added to it will usually be the first removed.

STATIC MEMORY Memory that uses many TRANSISTORised gates for storing data, unlike static memory no refresh is required, but it is expensive. A higher level of technology is required to create the IC's containing many thousands of TRANSISTORS.

STRING A string is simply a grouping of ASCII characters arranged for convenience of use.

STROBE A strobe is an electronic method of signalling that consists of a single change in voltage levels, usually between GROUND and +5 volts.

SYNCHRONOUS Describes a situation where timing is important, for instance industrial machine control.

TERMINAL The device, similar to a typewriter, that is used to INTERFACE to the operator in computer systems. A TERMINAL contains a keyboard and a display unit such as a typewriter mechanism or a CRT.

TOP OF FORM When paper is positioned in a printer such that the print head or printing [illegible]chanism is said to be at the top of form.

TRACTORS The small oval shaped mechanisms on a printer that are used to pull the paper, by the pins on the tractor, thru' the printer.

TRANSISTOR Electrical switch that is switched by a change of voltage.

TERMINATE To indicate the end of an event or series of events.

TRACE To follow the flow of a logically organised network, for instance an electrical circuit or a computer program.

TTL Abbreviation for Transistor-Transistor Logic, which is based upon ground being equivalent to 0, and +5 volts meaning 1.

VOLATILE Usually describes the volatile memory syndrome - "It aint there when you turn it on again!"