inside the commodore 64™

By: Don French

The first machine known to have been invented on planet Earth with the ability to contain a stored and modifiable program to control its operation was invented in 1801 by the French silk weaver, Joseph-Marie Charles Jacquard. It was known as the Jacquard loom and its programs were encoded as a series of punched holes on paper cards, the dominant means of program storage for the next 170 years. The invention enabled the creation of beautiful and intricate patterns in the finest fabric known to man and was the precursor of the modern computer. The French silk weavers gained international acclaim for the unsurpassed quality of their weavings. Their tradition of producing the world's finest fabrics had its beginnings with the loom that could be programmed. French silk is to this day the finest, smoothest, most carefully crafted of fabrics.

Inside The Commodore 64

by
Don French

Published by French Silk
P.O. Box 207
Cannon Falls, MN 55009

Manufactured in the United States of America

Library of Congress number 83-090411

ISBN 0-9612422-0-5

Table of Contents

Introduction

This book has been written both as a general purpose Commodore 64 and 6510 microprocessor tutorial and as a specific complement to and guide for the use of a set of software tools. The tools, along with this book comprise Develop-64, a product of French Silk. While this book is certainly a useful tool in its own right, its value may be magnified considerably when used in conjunction with the other tools. Programmers of all levels of experience should find this guide a valuable resource.

The book may basically be divided into three sections. The first section provides information on the use of the Develop-64, the software. It brings you step-by-step through the mechanics of creating, modifying, running and debugging machine language programs.

The second section gives a detailed look at the architecture of the 6510 microprocessor. Its addressing modes, register set and instruction set are examined. The nature and structure of data is also explored. The introduction to assembly language is presented in this section.

The third section is very Commodore 64-specific. It provides the information which is necessary to utilize the Commodore 64's built-in programs. It provides memory maps of all of the 64's BASIC operating system along with the information as to how to use some of the built-in programs. It describes how to build custom characters, create sprites and how to produce bit-image graphics and to program the music synthesiser.

The appendixes contain additional information on the 6510 and the Commodore 64 along with some useful tables and sample progams. Also included is a tutorial on binary and hexadecimal number bases.

While this product provides tools which may facilitate the copying of legally protected software, it is not the intent of French Silk that you put these tools to this use. We oppose and wish to very strongly discourage the pirating of software. We would also like to point out that there are severe penalties associated with copyright violation including fines up to $10,000 and imprisonment of up to one year (not to mention civil liability). We remind you that programmers have the same sort of financial obligations as everyone else and when you make a copy of a software product to give to a friend you are committing an act of theft against the programmer. In the interest of promoting the continued development of high-quality and low-cost software, the consumer must play his and her roles in helping to eliminate this problem.

Thank you for your cooperation and may all your programs work the first time. Enjoy Develop-64.

DCF

Using Develop-64 - I

This chapter will provide an overview of the software tools which comprise Develop-64. It also will lead you through the first steps of the use of the tools. It is recommended you go through the exercise of entering, modifying, assembling, etc. the given programs in the text as you read. The sample program created in this chapter will be used in subsequent chapters to illustrate the use of other functions and capabilities of Develop-64. It is not necessary to understand the program you enter at this time. It will be explained in more detail later. This and the next few chapters are designed to familiarize you with the mechanics of operating Develop-64. Very detailed explanation of machine language, the architecture of the 6510 and the Commodore 64 will be presented in subsequent chapters. You may find it valuable to return to these first chapters after you have gained more understanding.

Overview

To get started you should have the 64 turned on and the tape or diskette in the appropriate device ready to load. Load Develop-64 much as you would load any other program, i.e. LOAD "DEVELOP-64",8 for disk or LOAD "DEVELOP-64" for tape. If you experience trouble loading Develop-64 be sure you have spelled the name right and that all of your equipment is in proper working condition. If LOAD ERRORs prevent you from loading the program or the program just doesn't seem to be on the media at all and you are sure your machines are in good working order, you should return the media for replacement.

Start the program in the usual fashion, i.e. by typing "RUN" followed by [R] (the symbol used in this book to signify the return key). Please note that we use the quotes (") frequently in this book to bracket the response you are instructed to give to various prompts. The quotes are not a part of what you key into the computer. Key only what is enclosed within them.

The first question you will be asked relates to where in the memory space of the 64 you would like Develop-64 to reside. The default values if you hit [R] will be from $0800 to $BFFF (2048 to 40959). It may be placed anywhere there is 16K available. The

reasons for wanting to determine the starting and ending addresses will be discussed later. For now hit [R] twice.

The message which is now displayed is:

THE MAXIMUM NUMBER OF STATEMENTS YOU MAY HAVE IN A SOURCE SEGMENT IS NOW ....

The value printed is computed from the starting and ending addresses given above. Now the screen will get strange for awhile and then the copyright message will appear followed by the main menu as described below. After the menu will be a question mark (?).

The main menu

When Develop-64 is initially run and upon exiting from any of the sub-programs the main menu of possible sub-programs is displayed:

```
1) EXIT  2) EDIT   3) ASSEM   4) DECODE
5) DEBUG 6) LOAD   7) SAVE    8) NEW
```

Option 1 of the main menu causes the return to BASIC. Every sub-program also has number 1 as the exit option. For the sub-programs this option causes a return to the main menu.

Option 2 causes the editor sub-program to be run. The editor is used to create assembly language programs from scratch or to make modifications to them. With it you can insert, delete, modify and list lines of the source program. A walk-through of the use of the editor to create a small machine language program follows the overview.

Option 3 causes the assembler sub-program to be run. The function of the assembler is to translate programs written in assembly language into machine language. The assembler finds the assembly language program, called the "source program", in memory in the source program area. The program must have been previously created with the editor (option 2) or the decoder (option 4) or loaded by the loader (option 6). The source program is assembled by the assembler and the resulting machine language program is produced. The listing of the

program is directed to either the screen or the printer. The machine language may be directed to either an "object" file on tape or disk or directly POKEed into memory or both. The specifications for the creation of assembly language programs are given in Chapter 10.

Option 4 selects the decoder as the sub-program to be run. The decoder does the opposite of the assembler. You specify the starting and ending addresses in memory which you want to decode and it will produce the assembly language program which corresponds to the machine language program in memory. The decoder lists the assembly language program on the screen and optionally to the printer. It also can place the generated assembly language program in the source program area of memory where you may access it with the editor and/or the assembler.

Option 5 of the main menu selects the debugger. This tool allows you to run any machine language program a single instruction at a time. As each instruction is executed it is decoded and all the internal registers of the 6510 microprocessor are displayed, including the individual status bits of the processor status register. Any memory location may be displayed and any memory location in RAM may be modified as may any of the registers. While single-stepping through a program any instruction may be bypassed.

Option 6 selects the LOAD sub-program. This sub-program can load two types of files. It is used to re-load source files and to load "object files" (machine language programs which were created with the assembler). Files may be loaded from tape or disk.

Option 7 of the main menu selects the SAVE sub-program. SAVE allows you to do two kinds of saves. You may save your source program which was created with the editor and/or the decoder onto tape or disk. Source programs automatically have the suffix ".SRC" appended to the file name you assign. They may be re-loaded into the source program area by the LOAD sub-program (option 6). Also, blocks of memory may be saved as "binary" files which may be re-loaded later with the usual BASIC "LOAD" command. The third kind of saving done by Develop-64 is done as an option of the assembler. It saves "object" files to tape or disk which the LOAD sub-program can then read and poke into memory. These files automatically have the suffix ".OBJ"

appended to their file names.

Option 8 is used to clear the source program area. It's use is required only when you wish to replace the current source program with another.

Using the tools - a tutorial

The following tutorial will bring you step-by-step through the many uses of the tools. The tutorial will not attempt to explain the assembly language program or much about the syntax of the statements created. For an understanding of the rules of proper assembly statement construction, see Chapter 10. For an understanding of the way the various instructions cause the 6510 to behave, read Chapters 6-9. For an understanding of the 64-specific parts of the program read Chapters 11-13. This tutorial is designed to familiarize you with the mechanics of creating a working assembly language program, assembling it, loading it into memory and running it both at normal speed and at single-step speed. You will also learn how to decode a machine language program back into assembly language.

Getting going with the editor

Develop-64 is waiting for a menu selection. We wish to create a source program from scratch so key a "2" to select the editor.

The editor menu should now be on the screen. The editor's options are:

```
1) EXIT   2) LIST   3) INSERT  4) DELETE  5)MODIFY
```

As mentioned before, option 1 will return you to the master menu, i.e. exit the editor. Option 2 will list the source program if there is one to list. We haven't gotten that far yet. What we want to do now is to insert lines of a source program into the source program area. So the option to select is "3". Now the next prompt, "INSERT AFTER?" will appear.

Insert mode

The question the editor is asking is where we want to start

inserting our program lines. Each line or statement of assembly language will be automatically assigned a line number as it is created. You will see that in a moment. At the start of making a program the first line number is a "1". So the first insertion will come after line number "0". That is the correct response at this time. If you ever try to fool the editor by answering with a line number higher than the current high line of the program in memory, it will just repeat the question until you answer something reasonable. Later, when you wish to go back and add lines to your program you will specify where you wish to insert your new program statements by giving the appropriate line number.

Once you have responded to the "INSERT AFTER?" prompt, the screen will clear and a solitary quote mark (") will appear on the second line of the screen with a flashing cursor following. This is the place where lines of assembly language programs are entered. It is called the "edit window".

Now to enter the first line of the sample program. Without hitting [R], key in the statement:

```
;SAMPLE PROGRAM
```

The semi-colon should be keyed in the first position after the prompt. If you made errors in keying the above statement you may move the cursor to the error with the cursor control keys and correct it by typing over the error or by using the [delete] and [insert] keys. Until [R] is hit, changes may be made at will. Each statement entered may be at most 79 characters long. This is two complete screen lines minus the quote.

Hit [R] when you have got the line right. You should see the statement appear a few lines lower on the screen with the line number 1 on the left. This statement is a comment. The fact that it is a comment is signified by the ";" in the first position of the line. You may place comments anywhere in your program and they have no restrictions on their format except for the first character. One restriction does apply to this and all other statements entered with the editor. Never may you key a quote (") character.

Now enter the second statement of the program:

```
SRC EQU251 SOURCE VECTOR
```

Be sure to leave a space between the label, "SRC" and the mnemonic, "EQU". Be sure NOT to leave any space between the "EQU" and the operand, "251". Not keying a space between these two fields may seem unnatural to those who have experience with other assembler/editors but it saves you a keystroke and becomes natural very quickly. The assembly listing and listings produced with the editor will have a space inserted for improved readability.

Another space should be present after the operand and before the comment, "SOURCE VECTOR". This is a typical assembly language statement. It has all four possible fields present. The label and comment fields are not always present and for some instructions the operand field is not present either. The mnemonic is like the verb of the statement and must always be present. It must always have a space before it. If there is a label, there must be only one space separating the two fields. If there is no label, the first character must be a space and the mnemonic must start in the second position after the quote. Comments must always be separated from the preceding part of the statement by a space. Note that there are two kinds of comments, those on lines by themselves and starting with a ";" and those which are on the same line as the assembly language instruction.

Once you have the statement keyed just as shown, hit [R]. It should appear below the previous line with line number "2". Now enter the remainder of the following short assembly language segment, one statement at a time. If you make mistakes while entering the program you may correct them before hitting [R] on the line in question or you may correct the error later with the modify option.

```
; SAMPLE PROGRAM
SRC EQU251 SOURCE VECTOR
DST EQU253 DEST VECTOR
SRCE EQU$D000 ORIG CHAR SET
DSTE EQU$C800 NEW CHAR SET
START EQU$C000
```

Error Messages

It is possible you may get an error message or two while keying the program. Errors are signaled by an audible "beep" and by a

message above the edit window in reverse red letters. The format of editor error messages is ERR 1 or ERR 2. When an error occurs, the line just keyed will remain in the edit window until it is corrected. ERR 1 signifies a mnemonic which is not valid. ERR 2 signifies either a syntax error such as no space before the mnemonic or an extra space between the mnemonic and the operand. ERR 2 will also be given if the addressing mode is not a legal one for the mnemonic used. A list of all the mnemonics and their valid addressing modes may be found in Appendix A.

To exit the insert mode it is necessary only to key a [R] with no further entry in the edit window. Once done, the menu will reappear and you may select other options as you choose.

It would be useful to try the modify mode now even if no mistakes were made in the original entry.

Modify mode

To select the modify option key a "5". You will now be asked where to begin modifications.

"BEGIN AT?" should appear on the screen. Give it the line number where you would like to begin making modifications. Try a "1". Line number 1 should now appear in the edit window waiting for your changes. You may make changes or not as you wish. If no changes are desired hit [R]. If you wish to make changes, move the cursor to the place you wish to change, make the modification and when done, hit [R]. You may use the insert and delete keys if you wish.

Once you have hit [R] the modified line will replace the old line and will appear below. Now the next line will appear in the edit window for your examination and possible modification. You will stay in modify-mode until one of two things happens. Either the last line of the program has been modified or you terminate modify-mode by over-keying a "/ " in the first character position of the line put up in the edit window and hitting [R]. When terminated, the menu will re-appear, and Develop-64 will automatically enter list mode as described below.

List mode

The source program is listed automatically after exiting from insert mode and from modify mode and after deleting lines and after loading a source program from tape or disk. In these cases the listing will be directed to the screen only and the listing will start with line 1. It is possible to command Develop-64 to begin listing the program at any time you are in the edit sub-program by selecting option 2.

Upon selecting the list option, you will then be asked "BEGIN AT?". Give it a line number, such as "1". Next you will be asked whether you want the listing to go to the "PRINTER ?". A "Y" or a "N" is expected. The "N" response is the default, in which case the listing will go on the screen. Whether the list mode was entered automatically or by explicit menu selection, while the listing is proceeding you may pause it at any time by hitting any key. Hitting [R] will cause the immediate return to the menu. You may resume a paused listing by hitting any key except [R]. Hitting [R] will cause a return to the menu.

Insert mode revisited

Insert mode may be entered to insert lines of source in the middle of a program as well as for creating one from scratch. With the sample program created thus far, select option 3 to re-enter insert mode. The "INSERT AFTER?" prompt should be answered with a "1". Now several lines of the program segment will be displayed on the screen and the edit-window will be open again. Key a single ";" character and [R]. This should cause a new line to be entered and inserted after line number 1 and it will appear below with the following statements automatically renumbered. The empty edit-window will re-appear and you may now key another statement to be inserted after the one just entered. If you wish, try inserting other statements. Anything starting with a ";" will be accepted. To leave the insert mode, like before, hit [R] with a blank line. Listing will automatically commence.

Delete Option

If you wish to delete some lines from your program the menu selection is "4". You will be asked the starting and ending line numbers to have deleted. As with insert, upon deleting, the remainder of the program will be renumbered. Listing will commence.

Now you can enter the rest of the program. This program will be used in the following chapters to illustrate the use of the other tools. It is also an example of creating your own custom character set. It is recommended you try it. The effects are unusual.

The following is a list of the complete program. This list is in the format produced by using the LIST option and a printer. It inserts a space between the mnemonic and the operand for readability. It also lines up the comments for readability. When keying the program remember not to key the extra space and only leave one space between the operand and the comment. A little practice and it will become very natural.

```
1       ; SAMPLE PROGRAM
2       ;
3  SRC      EQU 251            SOURCE VECTOR
4  DST      EQU 253            DEST VECTOR
5  SRCE     EQU $D000          ORIG CHAR SET
6  DSTE     EQU $C800          NEW CHAR SET
7       ;
8       ; ENTRY POINT ($C000 = 49152)
9       ;
10 START    EQU $C000
11          LDA #>SRCE         BUILD SOURCE
12          STA <SRC+1           VECTOR
13          LDA #<SRCE
14          STA <SRC
15          LDA #>DSTE
16          STA <DST+1         AND DEST VECTOR
17          LDA #<DSTE
18          STA <DST
19          LDA 56334
20          AND #$FE           INTERRUPTS OFF
21          STA 56334
22          LDA <1
23          AND #$FB           I/O OUT, ROM IN
24          STA <1
25          LDX #8             # CHAR MEM PAGES
26          LDY #0
27 LOOP     LDA (SRC),Y        GET SOURCE BYTE
28          PHA                 AND SAVE IT
29          TYA
30          EOR #7             FLIP CHAR PATTERN
31          TAY
32          PLA                RETRIEVE BYTE
33          STA (DST),Y        AND STORE IT
34          TYA
35          EOR #7             FIX THE Y-REG
36          TAY
37          INY                BUMP THE INDEX
38          BNE LOOP           AND DO IT AGAIN
39          DEX                PAGE COUNTDOWN
40          BEQ DONE
41          INC <SRC+1         INCREMENT PAGES
42          INC <DST+1         OF SRC AND DEST
43          JMP LOOP           AND KEEP GOING
44 DONE     LDA <1
45          ORA #4             ROM OUT, I/O IN
46          STA <1
47          LDA 56334          RE-ENABLE
48          ORA #1               INTERRUPTS
49          STA 56334
50          RTS                RETURN TO BASIC
```

When you have completed the entry of the above program double check it for accuracy. Once you have made any necessary modifications the source program should be saved.

Saving a source program

In the main menu, option 6 is selected to save a source program. Once the SAVE sub-program is entered, you will be asked whether you wish to save a source or a binary file. The default and the correct reply here is source ("S"). Hitting [R] will cause a source file to be saved. You will next be asked whether you want to save your source program to tape or disk. Reply as is appropriate for your system. Finally, you will be prompted for the name you wish to assign to the file. Whatever name you give, Develop-64 will automatically append the suffix of ".SRC" When re-loading the same file in the future you will only be required to specify the base name, not the suffix. Object files, when saved out of the assembler sub-program will be created with the suffix of ".OBJ". Note that either type of save will cause any previous version by the same name and with the same suffix to be deleted and replaced by the file you are now creating.

Using Develop-64 - II

Preparing to use the assembler

The assembler sub-program will process a source program and produce an object program. The object program is the machine language which is the ultimate objective of writing an assembly language program. The source program must be in Develop-64's source program area. A source program may get into the source program area via the editor, decoder and loader subprograms. In the previous chapter the use of the editor to create a sample source program was described. If the source program is still in the source program area you may now assemble it by selecting option 3 of the main menu. If not, you will have to create a program with the editor or load a saved source program with the loader. The procedure for using the loader is given below.

Loading a source program

Option 6 selects the loader sub-program. The first question asked is whether you wish to load a source or object file. The possible responses are "S" and "O". "S" is the default and you need only hit [R]. You will now be given your choice of loading the file from disk or tape (D/T). Respond appropriately for your system. The next question you must answer is what file name Develop-64 is supposed to find and load. When source files are saved with the SAVE sub-program, the suffix, ".SRC" is automatically appended to the file name you gave it. It is not necessary to add that now to the file name to load. The loader will automatically find and load the file with the name you specified plus the appended suffix.

Finally, you will be asked where in the source program area you wish to have the source program loaded. That is, after which line number do you want the file inserted. If you already have a program in the source program area and want this one to replace it you must precede the load process with a "NEW" (option 8). It is possible to merge multiple source files by not NEWing between loads. In this case you must tell Develop-64 where you want each file inserted and as the files are loaded they will be inserted accordingly, with the program already in the source area being automatically renumbered to reflect

the insertion.

In the case of loading a program into an empty source program area, the correct response to the "INSERT AFTER?" question is "0". Once the program is loaded the list mode of the editor sub-program will be automatically entered.

Assembling the source program

To enter the assembler sub-program it is necessary to select option "3" of the main menu. The first question you must answer is "DEC/HEX (D/H)? You are being asked how you wish to see the generated machine language displayed on the assembly listing. The two possible responses are "D" and "H". Addresses and data will both be displayed in the format you select. If you hit [R] the default value of "D" for decimal will be used. The hexadecimal choice causes the assembly to run somewhat slower.

The next prompt will be "POKE ?". If you answer "Y" the generated machine language program will be POKEd directly into the memory of the computer. This can be dangerous if the memory addresses where the program is designated to reside overlap the memory space where Develop-64 itself resides. This is called self-destruction and will not give pleasant results. There are several ways to avoid this problem: 1) Design the machine language program to reside in "sacred RAM" starting at $C000 (49152). 2) Don't select the POKE option. Use the next option which creates an "object file" which can then be loaded with a three statement BASIC program. 3) Set the start and end of Develop-64's address space to addresses which will preclude it from being in the space which will be occupied by the eventual machine language program you are creating. This is done at the very beginning of Develop-64.

All these options are explained in more detail in the Chapter 5. For the sample program it doesn't matter how you respond since the program is set up to reside in sacred RAM.

The next question, "CREATE OBJECT ?", is asking whether Develop-64 should build a file of the machine language output which can then be loaded later with the loader sub-program. This is a convenient way of saving the machine language program. The loader sub-program is a very small routine which could be very easily

incorporated into a BASIC program. This allows you one means of writing a BASIC program which has machine language subroutines. In the next chapter we will present just such a BASIC program which calls the sample program. You should answer "Y" to this question if you wish to follow the sample program through to its completion.

Once done, the next question is "DEVICE (D/T) ?" This is asking you whether you wish to save the generated machine language object file to tape or disk. Answer as is appropriate for your system. Next, Develop-64 wants to know what you want to call the file. Whatever you respond to "FILE NAME ?" the actual name assigned will have the suffix of ".OBJ" automatically appended. You may pick any name you wish but the program given in the next chapter assumes a file name of "SAMPLE".

Finally, Develop-64 wishes to know if you want the assembly listing to go to a printer. If you answer "Y" the listing will go only to the printer. If you answer "N" the listing will be displayed on the screen only.

When this final option is selected, Develop-64 will display the message "NOW ASSEMBLING" and the first pass of the assembler will commence. When completed, the assembly listing will begin to appear on the selected listing device. On the screen, the source line will appear first, followed by the machine language which the assembler created. The line numbers will appear on the second line along with the machine language. The machine language is visually separated from the source by being displayed on the screen in reverse and on the printer to the right of the source statement. Errors are displayed both on the printed listing and in reverse red on the screen, accompanied by a warning "beep".

Error messages

There are four errors which the assembler recognizes. Each is displayed on the line with the generated machine language in the format: "ERR 3 FIELD" where the number after the ERR may be 3, 4, 5 or 6 and FIELD will be the actual data the assembler found in error.

ERR 3 means a label specified in an operand cannot be found anywhere in the source program. ERR 3 can also occur on the first pass of the assembler and signifies that an EQU statement has a label in the operand field which has not yet been encountered in the source

program.

ERR 4 means a relative branch instruction such as a **BNE** or a **BPL** instruction is attempting to specify a branch to an address outside the range of the instruction.

ERR 5 means an invalid character has been encountered in a hexadecimal term of an address expression.

ERR 6 means that an address has been specified which is out of the range of -65536 to 65535.

For detailed descriptions of the specifications for writing valid assembly language programs see Chapter 10.

While the listing is being generated, it is possible to pause it by hitting any Key. If the Key hit is [R], the assembly will terminate and the main menu will reappear. Once it has been paused, you may continue by hitting any Key except [R]. Hitting [R] will cause the main menu to reappear. You may at any time re-assemble by re-selecting option 2 of the main menu. If you have not selected insert, delete or modify options of the editor or have not done a source file load since the last assembly, the first pass will be bypassed, allowing speedier assembly.

If you have actually created the sample program and assembled it, selecting printer output, you should have received a program listing similar to the following listing.

FRENCH SILK DEVELOP-64 ASSEMBLY LISTING

```
1       ; SAMPLE PROGRAM
2       ;
3  SRC     EQU 251          SOURCE VECTOR      251
4  DST     EQU 253          DEST VECTOR        253
5  SRCE    EQU $D000        ORIG CHAR SET      53248
6  DSTE    EQU $C800        NEW CHAR SET       51200
7       ;
8       ; ENTRY POINT ($C000 = 49152)
9       ;
10 START   EQU $C000                           49152
11         LDA #>SRCE       BUILD SOURCE       49152 169 208
12         STA +SRC+1         VECTOR           49154 133 252
13         LDA #<SRCE                          49156 169 0
14         STA +SRC                            49158 133 251
15         LDA #>DSTE                          49160 169 200
16         STA +DST+1       AND DEST VECTOR    49162 133 254
17         LDA #<DSTE                          49164 169 0
18         STA +DST                            49166 133 253
19         LDA 56334                           49168 173 14 220
20         AND #$FE         INTERRUPTS OFF     49171 41 254
21         STA 56334                           49173 141 14 220
22         LDA +1                              49176 165 1
23         AND #$FB         I/O OUT, ROM IN    49178 41 251
24         STA +1                              49180 133 1
25         LDX #8           # CHAR MEM PAGES   49182 162 8
26         LDY #0                              49184 160 0
27 LOOP    LDA (SRC),Y      GET SOURCE BYTE    49186 177 251
28         PHA                AND SAVE IT      49188 72
29         TYA                                 49189 152
30         EOR #7           FLIP CHAR PATTERN  49190 73 7
31         TAY                                 49192 168
32         PLA              RETRIEVE BYTE      49193 104
33         STA (DST),Y      AND STORE IT       49194 145 253
34         TYA                                 49196 152
35         EOR #7           FIX THE Y-REG      49197 73 7
36         TAY                                 49199 168
37         INY              BUMP THE INDEX     49200 200
38         BNE LOOP         AND DO IT AGAIN    49201 208 239
39         DEX              PAGE COUNTDOWN     49203 202
40         BEQ DONE                            49204 240 7
41         INC +SRC+1       INCREMENT PAGES    49206 230 252
42         INC +DST+1       OF SRC AND DEST    49208 230 254
43         JMP LOOP         AND KEEP GOING     49210 76 34 192
44 DONE    LDA +1                              49213 165 1
45         ORA #4           ROM OUT, I/O IN    49215 9 4
46         STA +1                              49217 133 1
47         LDA 56334        RE-ENABLE          49219 173 14 220
48         ORA #1             INTERRUPTS       49222 9 1
49         STA 56334                           49224 141 14 220
50         RTS              RETURN TO BASIC    49227 96
```

If you did not select the printer option, the listing you got on the screen should have been similar, except that the machine language is on a line by itself. If there are any differences, particularly in the machine language portion of the listing, there is something significantly wrong with your source program. If you received errors while assembling you must find the source of your mistake, use the editor to correct it and go back and re-assemble. Once it looks right, proceed to the next chapter.

If you did not select the printer option, the listing you got on the screen should have been similar, except that the machine language is on a line by itself. If there are any differences, particularly in the machine language portion of the listing, there is something significantly wrong with your source program. If you received errors while assembling you must find the source of your mistake, use the editor to correct it and go back and re-assemble. Once it looks right, proceed to the next chapter.

Using Develop-64 - III

Loading and running the machine language program

If you have created and assembled the sample program described in the past two chapters and have created an object file as described you can now use the following BASIC program to load and run the program. Note that the first two lines of the program are written for the disk user but the cassette user can eliminate line 7 and change line 5 to: 5 OPEN 1,1,0,"SAMPLE.OBJ"

```
 5 OPEN1,8,2,"0:SAMPLE.OBJ,S"
 7 CLOSE15:OPEN15,8,15:INPUT#15,A,B$,C,D:IF A THEN
   PRINT A,B$,C,D:CLOSE1:CLOSE15:STOP
10 INPUT#1,N: IFN=-1000 THEN CLOSE1: GOTO30
20 IF N<1 THEN POKE P,-N: P=P+1:GOTO10
25 P=N:GOTO10
30 POKE 648,196: POKE 56578,PEEK(56578) OR 3:
   POKE 56576,PEEK(56576) AND 252: SYS 49152
40 POKE 53272,(PEEK(53272) AND 240) OR 2
50 PRINT "[CLR] MIRROR, MIRROR ON THE CEILING"
```

It is assumed that the name you gave to the file when the assembler sub-program asked for file name was "SAMPLE". If it was something else, substitute that for "SAMPLE" in line 5.

The three POKES in line 30 are set-up preliminaries in preparation for running the machine language program. They could have just as easily been done in the machine language program. Chapter 11 on graphics programming explains in detail the process which is being performed here.

The jump into the machine language program is accomplished by the SYS 49152. Note that the first non-EQU instruction in the assembly language program has an address of 49152 ($C000). This is the first executable statement of the program. It is called the entry point of the program. Its address was determined by the EQU

immediately preceding it. As explained in Chapter 10, the address in RAM where the machine language program gets POKEd is set each time an EQU is encountered in the source program. The assembler writes the address expressed in the EQU to the file as a positive decimal number. It writes the bytes of machine language to be POKEd as negative numbers. The little loader routine thus can identify every EQU and change the address at which to start POKEing subsequent machine language.

The POKE in line 40 of the BASIC program could also have been included in the machine language program. Its function is also explained in Chapter 11 in the section labeled multiple character sets.

If all went as planned, the results of running the above program should be apparent. All screen output from now on will appear upside down. (Try listing the BASIC program, for example). You may escape from this mode only by turning the machine off or by typing SYS 64738 [R] or by doing some POKEs to switch the character set back to its usual state.

If the program did not seem to work as advertised, you will need to back up and try to find the error. Please, before calling the author, do your best to try to find the problem. Look for discrepancies between your source program and that listed in the book. If you got an error message on trying to load the object file, find out why. Go back and re-assemble if other strange things occur. Select the POKE option and then use the decoder sub-program, described below, to decode the machine language to compare the generated assembly language with the original sample program. (Note: the listing of the sample program given in this book came directly from Develop-64 and DOES work. It is also highly unlikely that Commodore has made a change in the design of the 64 which will cause the sample program to be ineffective.)

On the other hand, if you have exhausted all other possibilities and it certainly appears that something is amiss, please send us a copy of your Develop-64, a written description of the problem, and any other relevant information and data files which can assist us in finding the source of the problem.

Decoding the program

Now that you have a working copy of the sample program in memory it would be a good time to see what the decoder can do for you. If you just ran the sample BASIC program which loaded and ran the machine language program you can get back to normal characters without losing the machine language program by typing SYS 64738. If the machine language program is not in memory you should put it there by assembling the source program and selecting the POKE option to POKE the output into memory as the program is being assembled.

Load and run Develop-64. Select the decoder option of the main menu, option 4. Your next menu will be:

1) EXIT 2) PRINTER 3) SCREEN 4) INSERT

All three options to decode machine language will cause the generated assembly language to appear on the screen. If option 2 is selected the output will go to your printer as well. If option 3 is selected the lines of assembly language will be inserted into the source program area where they may be modified and/or assembled and/or saved. If insert is selected the "INSERT AFTER ?" prompt will be given and you must tell Develop-64 where you want the source inserted in the current source program. If there is already a program in the source program area and you want this one to replace it, it will be necessary to clear the area with the NEW option (8) of the main menu before proceeding with the decoder sub-program.

In every case, the first prompt will be "DEC/HEX (D/H) ?", asking whether the generated source should have addresses and data in hexadecimal or decimal format. Hex causes the decoder to run about twice as slow as decimal.

The next prompt will be "START, END ?", asking for the addresses in memory between which you would like to decode. You may give your answers in either hex or decimal. Hex values must be preceded by a "$". The end point may be expressed as data value instead of an address. By precedeing the data value with a "#", the decoder will be directed to decode until it recognizes the specified data value in the op-code of an instruction. For example, if you want to decode until the end of a subroutine, you could specify an end address of #96 or #$60 which will cause the decoder to decode until it encounters a RTS instruction (value of 96 or $60).

If you have the sample machine language program in memory you may now decode it. For the purposes of this demonstrateion, select option 4 to INSERT. Answer "0" to the "INSERT AFTER?". Specify a starting address of $C000 and an ending address of #$60, the op-code for the RTS instruction.

The generated output should look very much but not exactly like the original source program which created it. The differences are due to the fact that the decoder can not create comments. Nor does it generate labels.

The decoder sub-program can decode any memory block in the computer, including the operating system. It can also decode cartridges if the cartridge is activated after power-on. The reason for this exception is that if a cartridge is in place when the power is turned on the power-up program in the computer will automatically give control to the program in the cartridge. Since most cartridges will not allow any means of giving control back to another program there is no way for Develop-64 to be run. The most common way to defeat this is to have the cartridge plugged into an expansion chassis which has switches which activate and de-activate the cartridges which are plugged into it. Then the procedure is to power-up with the cartridge de-activated, load and run Develop-64, then activate the cartridge. Since the only time control is passed to the cartridge is at power-up, the cartridge is not now in control, yet it is addressable by the decoder. There are some cartridges which are designed to cause BASIC to be "switched out" when they are activated. These cartridges may not be possible to decode with Develop-64.

As a matter of curiosity, you could now exit the decoder and enter the assembler and re-assemble the decoded program. The machine language which is generated will be exactly the same as the machine language which was generated from the original source program.

Below is a listing of the re-assembled output of the decoder. Note that the generated machine language from the assembler is identical to the generated machine language of the original assembly.

FRENCH SILK DEVELOP-64 ASSEMBLY LISTING

```
1   EQU 49152                                  49152
2   LDA #208                                   49152 169 208
3   STA ←252                                   49154 133 252
4   LDA #0                                     49156 169 0
5   STA ←251                                   49158 133 251
6   LDA #200                                   49160 169 200
7   STA ←254                                   49162 133 254
8   LDA #0                                     49164 169 0
9   STA ←253                                   49166 133 253
10   LDA 56334                                 49168 173 14 220
11   AND #254                                  49171 41 254
12   STA 56334                                 49173 141 14 220
13   LDA ←1                                    49176 165 1
14   AND #251                                  49178 41 251
15   STA ←1                                    49180 133 1
16   LDX #8                                    49182 162 8
17   LDY #0                                    49184 160 0
18   LDA (251),Y                               49186 177 251
19   PHA                                       49188 72
20   TYA                                       49189 152
21   EOR #7                                    49190 73 7
22   TAY                                       49192 168
23   PLA                                       49193 104
24   STA (253),Y                               49194 145 253
25   TYA                                       49196 152
26   EOR #7                                    49197 73 7
27   TAY                                       49199 168
28   INY                                       49200 200
29   BNE @-15          49186                   49201 208 239
30   DEX                                       49203 202
31   BEQ @+9           49213                   49204 240 7
32   INC ←252                                  49206 230 252
33   INC ←254                                  49208 230 254
34   JMP 49186                                 49210 76 34 192
35   LDA ←1                                    49213 165 1
36   ORA #4                                    49215 9 4
37   STA ←1                                    49217 133 1
38   LDA 56334                                 49219 173 14 220
39   ORA #1                                    49222 9 1
40   STA 56334                                 49224 141 14 220
41   RTS                                       49227 96
```

Using Develop-64 - IV

Using the debugger

The debugger sub-program provides the capability of running a machine language program one instruction at a time. As each instruction is executed, the internal registers of the Commodore 64's 6510 microprocessor are displayed, including the individual bits of the status register. The instruction to be executed is also displayed, both in machine language and in assembly language. The capability is provided to bypass the execution of any instruction and to display and modify any memory location while in the process of running the program. The debugger sub-program may be selected by selecting option 5 of the main menu. The menu of debugger options is:

```
1) EXIT  2) START S/S  3) EXECUTE  4) BYPASS
5) MEMORY DISP/MOD
```

For the purpose of demonstrating the features of the debugger you should have loaded into memory the program created, assembled, loaded and run in the previous chapters.

Option 2 is the menu option to set the address of the next instruction to be executed. It will be followed first by the "DEC/HEX (D/H) ? prompt, then the "START ADDR ?" prompt, asking for the starting address. Valid replies are decimal numbers in the range of 0-65535 and hexadecimal numbers in the range of $0000-$FFFF.

Option 3 is the default option and may be selected by hitting [R]. It is not to be used until option 2 has been selected to set the first address to execute. Selecting this option causes the current instruction to be executed. This will be clarified as you single-step through the sample program.

Option 4 will cause the instruction about to be executed to be bypassed. Rather than execute the displayed instruction, the instruction following the displayed one will be displayed.

Option 5 causes Develop-64 to enter the memory display/modify mode. This is also followed by the "START ADDR" prompt, requesting the starting address of displaying/modifying.

The sample program explained

At this time you can see the "slow-motion" execution of the sample program. If you don't know much about machine language or the architecture of the 6510 microprocessor this exercise may not make a lot of sense to you. You may want to skip ahead and read the Chapters 6-9 to gain an understanding of the machine. It could also be helpful to just walk through the following explanation prior to having a fuller understanding just to familiarize yourself with the mechanics of using this tool. Either way, if you are just getting started with machine language, it is recommended you return to this chapter after having studied the following chapters.

Key a "2" if you wish to single step through the sample program. Answer the "START ADDR" prompt with $C000, the address of the entry point of the program. The first instruction of your program should now appear on the screen.

As each instruction is about to be executed, it is displayed in both machine language and in assembly language. The processor status (PS) register is broken into its individual bits (N = negative, V = overflow, B = break mode, D = Decimal mode, Z = Zero, I = interrupts inhibited, C = Carry). The other registers displayed are the A-reg, the X-reg, the Y-reg, and the Stack Pointer (SP). Upon execution of each instruction, the registers are loaded from these save areas in memory:

A - 780 X - 781 Y - 782 PS - 140

If you wish to modify or pre-initialize any of the registers at any time you may do so by entering the Memory-Modify mode and modifying the above locations.

If, while single-stepping through some program, you should execute a RTS or PLA or PLP without first having pushed something onto the stack with a JSR or PHA or PHP, a stack underflow will occur. A TXS instruction setting the SP to some out-of-range value will also cause stack underflow. Overflows are caused by repetitive PHA's PHP's or JSR's without corresponding PLA's PLP'S or RTS's until the maximum stack depth has been exceeded. In the event of underflows and overflows, a "STK ERR" message will be displayed. Execution may

continue if desired but results are likely to be unexpected if your monitored program is expecting to find some significant information in the stack (like a return address).

Certain machine language programs are written to modify the Stack directly by storing data in the high end of page 1. Executing these instructions in debugging mode will not cause the desired stack modification effect. In fact, it is quite likely that Develop-64 will actually crash upon the execution of such instructions. Since it is written partially in BASIC, any instructions which modify the BASIC vectors or other information vital to the running of BASIC programs may also cause unwanted results.

As Develop-64 single steps through a machine language program, it checks each op-code encountered for validity. If an invalid op-code is encountered, the message "OP-CODE = xxx" (where xxx is the encountered op-code) will appear where the mnemonic would otherwise appear. Develop-64 will not try to execute invalid op-codes. Nor will it try to execute BRK or RTI instructions. All of these will be automatically bypassed.

If the sample program is now in memory and you have selected the single-step option and specified address $C000 as the starting address and selected decimal as the display format, you should see on the screen the assembly language statement "LDA #208" followed by the address 49152 and the machine language equivalent of the above assembly language statement: 169 208.

On the next line will be the display of the registers, the A-reg, X-reg, Y-reg and SP(the stack pointer), prior to the execution of the displayed instruction. You will also see the status register displayed broken down into its component bits, the Negative, oVerflow, Break, Decimal, Interrupt disable, Zero and Carry flags. The registers will have no particular significance at this point because they were never initialized. Note, however, the value of the A-reg because after executing the instruction it will probably change. The Zero flag, if it is a one now should also change as a result of the execution of the instruction. The Negative flag is also affected by a LDA instruction.

Note the disassembled statement is not identical to the assembly language statement you originally wrote. It is equivalent but not identical. The original statement was: LDA #>SRCE. This discrepancy occurs because The Monitor can't tell what went into the assembler, only what came out and it does the best it can in

reconstructing a valid assembly language statement from the machine language it has to work with.

To execute the instruction, hit return. If all is well, the next instruction of the sample program will be displayed and the first instruction will have been executed. You may verify that by checking the A-reg. It should be of value 208 now. The zero bit should be a 0 because the result of loading the A-reg is non-zero. Note that the resulting value in the A-reg does have the high-order bit on (i.e. the number in the A-reg is greater than 127 or $7F). Consequently, the Negative flag should now be turned on.

The next instruction which is now up for execution will store the A-reg in location 252. None of the status flags are affected by this instruction so we should see no change when we execute it. Push [R] and see.

This would be a good time to look at the Memory Display/Modify mode. Rather than hit [R] at this time, key "5 [R]".

Upon entering this mode, the address where you left off in single-stepping will be saved and the "START ADDR" prompt will be displayed. You may enter the first address you wish to examine or modify. The address may be in the range 0-65535 or $0000-$FFFF. To look at location 252, key a "252 [R]". The address (252) will be displayed followed by the contents of the specified location. In this case it should be 208 because that is the value we just stored there. The value of the data stored at the requested address will be displayed and the prompt "VAL?" will follow. You may do one of three things. You may eXit the Memory mode by keying "X [R]". You may modify the displayed location by keying a value in the range 0-255 or $00-$FF. Or you may continue viewing the next sequential memory locations by hitting [R].

You may now change the contents of location 252 if you wish by keying some new value. The next location will now be displayed, memory location 253. Note its contents and modify them if you wish.

To just scan through memory, simply continue to hit [R] each time a value is displayed. When you wish to return to the main menu, key "X [R]" for exit. Once this has been done, the instruction you left off at will be redisplayed along with the menu. If you modified location 252 you should now re-enter M-mode by entering the "5" option, and address 252 again. When the value you stuck in 252 is displayed, change it back to 0 and hit [R], and when location 253 is

displayed, hit "X [R]" to get out again.

The next instruction in the program is now displayed. It is a LDA instruction. Note the registers still have the same contents as before the M-mode excursion. The A-reg will be loaded with the value 0 by the instruction to be executed next.

This program is setting up a vector in locations 251 and 252 of zero page to address character ROM at $D000. A loop in the program will sequentially move characters from that area in memory to an area where an alternate character set will be built. Hit [R] and see the next instruction which is a Store of the A-reg to location 251. Hit [R] again. The next four instructions build another vector at 253 and 254. This is the vector which points to the location where the new character set will reside. Hit [R] to execute each of these instructions. You may verify that locations 251-254 contain the addresses of the two vectors by going into memory display mode if you wish.

The next three instructions cause the timer to be turned off. Location 56334 is one of the registers associated with the hardware timer which interrupts the Commodore 64 60 times a second. As explained in more detail later, the character ROM starting at $D000 (53248) shares its address space with input/output (I/O) registers. To read the character ROM, it is necessary to switch the I/O out and switch the ROM in. The only problem with doing this is that the I/O registers are used in the servicing of interrupts. So, while the I/O is switched out to access the character ROM the timer must be turned off so as to discontinue interrupts. One of the things the operating system does when it processes the interrupts every 1/60 th of a second is to poll the keyboard to see if any keys have been pressed. Since we need to have that function intact while running the debugger, we can't really allow the interrupts to be disabled. So it is necessary to bypass the instruction at 49173 which accomplishes the disabling. This is where the bypass option is useful. When that instruction is displayed, about to be executed, press "4" instead of [R].

The switching-in of the ROM is accomplished in the next three instructions. The 6510, as explained at the end of Chapter 7, uses location 1 as an I/O port and the 2-bit controls whether the ROM or the I/O registers are switched in. The AND #251 instruction accomplishes the turning off of that bit. Now, since we had to leave the interrupts enabled, we can't switch out the I/O registers since they are used in processing the interrupts. If you should make the

mistake of executing the instruction at 49180 the computer will hang and there will be no recourse except to turn it off and back on again and start over.

Next, the X and Y registers are set up to count the number of times through the following loop. The instruction which is at 49186 was labeled LOOP in the original source. This instruction will load the A-reg with the byte at the location computed from the sum of the contents of the Y-reg and the address vector in locations 251 and 252. An unfortunate consequence of not being able to switch in the character ROM is that the data being loaded is not the same as it is when the program is run at full speed.

If you take the value stored at location 252 (208), which is the page # of the character ROM) and multiply it by 256 and add the value stored in location 251, (0), you will get the base address to which the Y-reg is added. All vectors work the same way: add the contents of the first byte of the vector to 256 times the second byte to get the address being referenced.

The next instruction, the PHA, saves the retrieved byte onto the stack. Note the value of the SP (stack pointer) before and after executing this instruction.

Next, there is a three instruction trick played with the Y-reg to cause the eventual turning upside-down of the characters. The flipping of each character top-to-bottom requires a knowledge of how character information is stored in memory. It turns out to be a fairly simple process but one which can be better understood by reading Chapter 11. Suffice it here to say that a simple manipulation of the Y-register modifys the sequence in which the character information occurs in the new character set.

The PLA instruction pulls the saved byte of character information back off the stack. You can see the stack pointer being modified again as the PLA instructions is executed. Once the byte is back in the A-reg, it is stored in the new character set in a position determined by the source vector at 253,254 and the value of the Y-reg.

Since the Y-reg was manipulated to cause the flipping over of the character, it must now be fixed back to its original value before the modification. The next three instructions, TYA, EOR #7, and TAY do the trick. This is just the reverse of the operation which modified it in the first place.

Once the Y-reg is restored, its use as a loop counter is

employed. The INY instruction bumps it up by one and the next instruction tests it to see if it has gone past 255, its maximum value. If it has, it will have the value of zero. The BNE instruction tests the zero bit of the status register and as long as it is off (a zero), the branch will be executed and the next instruction to be executed will be at the top of the loop at 49186. Once the Y-reg gets incremented past 255 to 0, the program will "fall through" to the DEX instruction. This will happen after the 256 sequential bytes of data in the first page of the character ROM have been moved to the new character set location.

Stepping through a few cycles of the loop would be instructive for the newcomer to machine language programming. There are eight pages of 256 bytes each of character information which needs to be moved. The Y-reg is used not only to count through the 256 bytes of each page but also to index the address where data is being retrieved from and stored to. If you step through the loop eight times and record the value of the Y-reg prior to executing the instruction at 49186 and prior to executing the instruction at 49194 some insight may be gained into the technique employed. Go through it another eight times and see that the pattern repeats.

You may continue through the rest of the program to see it to completion without going through 2048 (8 times 256) cycles of the loop. Here is another place the bypass option is useful. To get out of the inner loop which terminates at the BNE instruction at 49201 you may select option 4 instead of executing the branch instruction. This will cause the program to "fall through" to the next instruction, the DEX. Following the DEX is the instruction which tests to see if all eight pages have been processed. If not, the vectors for the source and destination LDA and STA instructions are increased and the inner loop is entered again for another 256 iterations. To get past doing this again, the JMP instruction at 49210 must be bypassed.

The last seven instructions in the program switch the I/O back in and re-enable the interrupts and return to the calling program. These may be executed without danger. Executing the last instruction, the RTS, will cause a stack error. This is what should be expected because the return from subroutine was not preceded by a jump to subroutine. The SYS instruction in the BASIC program was the intended means of getting to the machine language program. The RTS is the intended means of returning to the BASIC program.

Making the Final Product

There are several places where machine language (ML) programs may be designed to reside. These include: 1) Inside a BASIC program 2) Before or after the BASIC program. 3) In the cassette buffer, $33C-$3FB (828-1019) 4) In "sacred" RAM at $C000-$CFFF (49152-53247). 5) Anyplace that Develop-64 itself does not reside. 6) Anywhere in memory that a "mini-loader" program can address.

Inside a BASIC program

To get a ML program into a BASIC program so that it may be saved with the BASIC program and reloaded right along with it, a couple of techniques may be used.

The first way is to load a BASIC program with several REM statements which take up space and will be overlaid by the ML. The token for the first REM must not be overlaid but everything after it can be.

To find the address where the REM statement of your BASIC program is stored it is necessary to understand how BASIC programs are structured. Each line of a BASIC program is stored in memory in a condensed fashion. All the "keywords", such as GOTO, FOR, PRINT, etc. are stored as a single byte of data, called a token. preceding the condensed BASIC line is four bytes of system information. The first two bytes are a link address pointing to the next BASIC statement following this one. The second two bytes are the line number of this BASIC statement. At the end of each statement is a single byte with value 0. This is the statement terminator.

The last BASIC statement in the program has a link address of 0. The first statement in the program is pointed to by an address vector stored in locations 43 and 44. Once your program is in memory you may find the address of any given statement by searching for the line number with the following short statement which may be entered in direct mode (does not have to be in a program).

```
I=43:FORJ=1TO10000:I=PEEK(I)+256*PEEK(I+1):IFPEEK(I+2)+
PEEK(I+3)*256<>...THEN NEXT
```

Where the "..." appears you must key the statement number you

are searching for. When the "READY." reappears on the screen, the variable I will contain the address of the first byte of the statement you are searching for. Simply PRINT I. The value printed will be the address of the first byte of the link address which precedes the actual statement in memory. The address of the first byte of the statement will be four greater than the value of I.

The ML program you wish to include with your BASIC program may be of greater length than what may be accomodated by a single BASIC REM statement. The solution is to have multiple REMs. Before overlaying them, it will be necessary to change the link address preceding the first one to point to the statement following the batch of statements to be overlaid. So it is necesary to find the address of the statement following the overlay area using the technique above, and to modify the link at the beginning of the area accordingly.

For example, if the address of the first statement to be overlaid is 5000 and the address of the first statement after those to be overlaid is 6000, the following POKEs would do the job: POKE 5001,6000/256: POKE 5000,6000-PEEK(5001)*256. This would allow you to save a ML program from 5005 (the byte after the REM token), to 5999 (the byte before the terminator of the last REM overlaid).

There are two places within your BASIC program which make sense to use for the overlay area. There are advantages and disadvantages to both.

If you place it anywhere but the beginning, the problem exists of having the machine language program shifting locations every time a modification is made to the BASIC program. This can be a major problem if the ML has non-relocatable code, e.g. a JSR instruction to some fixed address within the program. Non-relocatable programs should therefore overlay the beginning of a BASIC program where modifications to subsequent statements of the BASIC program will not cause any shifting of the position of the ML.

An example of such a program would be:

```
10 GOTO100
20 REM12345678901234567890123456789012345678901...
30 REM12345678901234567890123456789012345678901...
40 REM12345678901234567890123456789012345678901...
50 REM12345678901234567890123456789012345678901...
60 REM12345678901234567890123456789012345678901...
70 REM12345678901234567890123456789012345678901...
80 REM12345678901234567890123456789012345678901...
90 REM12345678901234567890123456789012345678901...
100 REM START OF THE PROGRAM
110 .....
```

The drawbacks to this scheme are potentially serious. The biggest drawback is that nowhere in the ML program may there appear the value of **0**. That could be a real bother to get around in some situations. This limitation is due to the fact that BASIC interprets the **0** as a line terminator and any modifications or even SAVEing the program will cause havoc. The other drawback is that certain bytes will cause the listing of the program to appear strange or not appear at all. In fact, the value of 204 ($CC) will cause BASIC to choke upon listing it. It will give a SYNTAX? error. The program will run OK but will not be listable beyond that point. You may list the remainder of the program by typing "LIST 30-" for example if the syntax error stopped the listing prior to statement 30. This might be considered a sort of protection feature, as it will cause the opposition difficulty in listing your program (at least for awhile).

The other place which is a likely candidate for storing the program within a BASIC program is at the end of the BASIC program. Here, the non-relocatability problem is present but the problem with having a **0** in the ML is solved. If you can be sure you will never modify your BASIC program there is no relocatability problem. It only exists if you add to or delete characters in the BASIC program and the passenger program gets relocated.

The best technique for adding the ML at the end differs from the REM technique explained above. First, find the end of your BASIC program. This is as simple as: PRINT PEEK(45) + PEEK(46) * 256. The value printed will be the end of BASIC and the start of variable storage. To add a passenger at the end you need to change the vector

at 45 and 46 to increase the program size by as much as you need for your ML component. If for example the end of your BASIC program is 10000 and you wish to add a ML program of length 2000, you would key the following: POKE 46,12000/256: POKE 45, 12000-PEEK(45)*256. The next thing to do is to SAVE the program and immediately re-LOAD it. When the machine language is loaded into the reserved area it will not appear on the listing and it may have zeros or any other value. Now, any additions and deletions, SAVEing and re-LOADing of the BASIC program will keep the ML intact. Referencing addresses within it from your BASIC program is most easily accomplished by computing the address as a displacement from the end of the program. For example, if the entry point of the ML program is 500 bytes from the end of the program, your call to it would be SYS PEEK(45)+PEEK(46)*256-500. This way, no matter where the program gets shifted to by changes to the BASIC program, the SYS statement is always correct.

Once you decide where you want your ML program to reside the problem is to get it there. There are a few options. You first need to find the address of the beginning of the machine language program, i.e. the address inside the BASIC program where the ML program will reside. This address must then be the address expressed in the operand of the EQU statement in the ML program which immediately precedes the first actual byte of ML to be generated. (Every EQU encountered resets the address of where succeeding ML gets stored).

There is a loader sub-program which is a part of Develop-64. It is possible to use it to load the object file containing the ML into the area you have reserved in your BASIC program. It is also possible to use the POKE option of the assembler sub-program to directly POKE the ML into the reserved area. A third choice is to write a brief loader in BASIC as illustrated in the sample program in chapter 8. All of these options have one thing in common. They all involve having two BASIC programs in memory at the same time, the BASIC program you are trying to add the ML to and the program which is doing the adding.

This is easily accomplished. First, load the BASIC program to be impregnated. Next, print the values of locations 43, 44, 45 and 46 (e.g. ?PEEK(43), etc.). Write these down. Next, change the value stored at 44 to one greater than what is in 46. Now load the program which will do the loading of the ML and POKEing it into the reserved area of the first BASIC program. This program will now load after the end of your BASIC program. (This technique of changing the load

address of BASIC programs is useful also for placing Develop-64 in a memory location which will not conflict with a program which you wish to decode or debug).

Running the second program, be it Develop-64 or a mini-loader, to load the object file or to assemble the source program and POKE out the ML will cause the ML to be placed into the space you have carefully reserved for it by the preceding operations.

Once done, it is necessary to save the BASIC program with its embryo. To do this, you must restore the values of 43, 44, 45 and 46 (POKE 43,...: POKE 44,...: etc.) with the values you recorded previously. Immediately follow this with a SAVE of the first program. The SAVE function of the Commodore 64 causes whatever is between the addresses pointed to by 43,44 and 45,46 to be saved. When you re-LOAD the BASIC program it should be carrying its ML child, ready to deliver (providing it was well conceived in the first place).

The final way of getting ML into a BASIC program is the hardest and ugliest way available. This is to take the listing of the assembler and create a string of DATA statements with one decimal value for every byte of generated ML and to include a FOR NEXT loop reading every value and POKEing it into the desired memory locations. For programs which need to be self-documenting this may be the right solution. Anyone reading the BASIC program will have all the information needed to get the program to work. This is why so many magazines use this clumsy approach in their articles.

The following approaches all assume that the ML will not be a part of some BASIC program. They all require some means of getting the ML into memory. For each case, the techniques for accomplishing this are the same. They are the same techniques as described above for getting the ML into a reserved space in a BASIC program, i.e. using Develop-64 to POKE the ML or to LOAD it from a created object file or with a mini-loader as in Chapter 8. For stand-alone ML programs, once they have been loaded by one of these means, they may be SAVEd by the "binary" SAVE feature of the SAVE sub-program of Develop-64. This will then allow them to be loaded with the normal LOAD"name",8,1 statement for disk or LOAD"name" for cassette.

If you wish to do a binary save without Develop-64 in memory, the following routine will accomplish it for you.

```
10 TD=1:REM FOR DISK TD = 8
20 POKE 781,TD: POKE780,1: POKE782,2: SYS 65466
30 INPUT"START,END";A,B: INPUT"FILE NAME";A$
40 POKE780,PEEK(202)-11:POKE781,0:POKE782,2:SYS65469
50 POKE780,251: POKE252,A/256:POKE251,A-PEEK(252)*256
60 POKE782,B/256:POKE781,B-PEEK(782)*256:SYS65496
```

This program illustrates the ability to call Kernal routines directly from BASIC. The A-reg, X-reg and Y-reg may be set up in locations 780-782 respectively before doing a SYS and the called routine will be entered with these registers pre-initialized. Upon returning from the called routine, the above memory locations will contain the values of the registers upon exiting the routine. This can be a convenient way to communicate with your machine language program from BASIC.

Before or after Basic

The upper limit of the memory space which BASIC believes it has available for BASIC programs is maintained in an address vector at locations 55,56. At 52,53 is another vector which BASIC uses to set the highest memory address usable for string storage. The beginning address, as mentioned above is pointed to by a vector at 43,44. BASIC starts storing strings at its highest available memory location and works back down. The free memory available in a BASIC program is the space between the high end of variable storage and the bottom of string storage. Strings are continually filling up the free memory gap as they are created by the program. Only when free memory is exhausted will BASIC clean up the string area for future string usage. This is what is known as "garbage collection" and what occasionally causes BASIC programs to pause for a while before continuing. The result of this system of string management is that no memory between the start of BASIC and the "top of memory" is free from possible destruction by the BASIC program.

It is possible to save ML program segments in a space which will not be molested by BASIC programs if you modify the two vector sets at 51,52 and 55,56 so that they point to an address below the ML. Or you could modify the start of BASIC vector before loading the BASIC

program. In either case, BASIC will not even know of the existence of that memory space and will not try to save anything there.

Cassette Buffer $33C - $3FB (828 - 1019)

There is a serious problem with putting the ML in the cassette buffer if you don't have a disk drive. Whatever program you use to load the object file with will itself use the cassette buffer. And so, as it is reading the object file it is destroying the POKEd ML. If you have a diskette based system, there is no problem.

In sacred RAM $C000 - $CFFF (49152 - 53247)

No protection from BASIC need be implemented when the program is up here. BASIC can't get to this area. It therefore makes an ideal place for ML programs. The only disadvantage is that the ML cannot be loaded along with the BASIC program. It must be loaded separately or by the BASIC program.

The 6510 - A Data Processor

Since Develop-64 is focused on the development of software through the creative use of machine language, it will be necessary to first understand the machine before learning its language.

You're probably aware that there is something inside the case of the Commodore 64 which is known as the 6510 microprocessor. This is the heart of the Commodore 64. It is a slightly modifided version of the 6502 which is the heart of the VIC 20, the Apple, the Atari, the PET, KIM, AIM, SYM, OSI and a few other microcomputers. It is a product of MOS Technology, a wholly owned subsidiary of Commodore Business Machines.

The 6510 is an integrated circuit. That is, it is a single chip of silicon which has built into it, sort of like etched onto it, the electronic circuitry which connects thousands of microscopically small electronic components. These components are deposited on the silicon by some marvel of modern technology which is beyond the scope of this text. We won't go into how the electronics are created or how they function electronically. We are interested here in how to use this machine and how it fits into the environment of the Commodore 64 personal computer.

The piece of silicon called the 6510 is packaged in a piece of plastic or ceramic material about one inch by two inches. It has 40 little bug-like legs called pins which connect the internal circuitry to the outside world. These pins plug into a circuit board which has other similar appearing chips of silicon, each with its own set of pins and its own internal characteristics, different from the characteristics of the 6510. Each of the chips has its own specific function and together they are combined, through their connecting pins and the circuit etched on the printed circuit board into which they are plugged, to make a microcomputer.

This will become more and more clear as we describe what each of the component chips are for and how they work and how they communicate with one another. The description of the functional characteristics of the 6510 will completely define the processor from the programmer's standpoint. The electrical or electronic characteristics are of no interest to us as we have no need to understand the machine at that level.

The 6510 is a data processor. It is a machine which performs

simple operations of data manipulation under the control of a stored program. Both the data and the program are stored in memory devices which are integrated circuits electrically connected to the 6510. Programs are a special class of data and we will explain programs after we discuss data in a more general sense.

Data is information. It can be the balance of your checking account or the grade your physics teacher gave you or the position of PAC-MAN on your video screen. Information, as it is stored inside the computer's memory devices is coded by a special set of simple rules. Memory devices are composed of thousands of cells, or storage locations, where the data is kept. Each cell is composed of eight switches which can be turned either on or off. When the letter "A" is pressed on the keyboard of your Commodore 64, some electronic circuitry will automatically create a pattern of eight switch settings which is uniquely identified as the pattern for the letter "A". This is the code for "A". Every character has its own code and it is different from all the other character's codes. There are only 256 different unique codes which can be constructed from eight switch settings. There are therefore only 256 possible different characters which can be represented and stored in the memory chips of the Commodore-64. That's sufficient to handle A-Z, 0-9, all the special characters and the graphics characters.

These switches are usually called "bits". Bit is short for binary digit. A digit may have 10 possible values, 0-9. A bit may have two possible values, 0-1. A bit which is turned on may be thought of as having the value of 1 and if it is off, it is a 0. So characters are represented as a string of eight bits with bit values of either 0 or 1. The actual bit string for the letter "A" is 01000001. The coding scheme used is an international standard called ASCII, which stands for American Standard Code for Information Interchange.

The 6510's data link with the memory devices is called the data bus. This is nothing more than a set of eight lines, or electrical connections, between the memory chips and the 6510. When the 6510, under control of a program, wishes to either transfer data to or from a memory device, it sends an electrical signal on the R/W (Read/Write) line, telling the device which direction the data is to go. Since the memory device can store thousands of characters of data, it is necessary for the 6510 to tell it which storage location it wishes to get data from or send it to. It does this through another set of

electrical connections called the address bus. Every storage cell within the memory device has a unique address associated with it and when the signal comes to do a data transfer, the memory device is designed to use the information sent on the address bus to know which cell is being selected. Each data cell holds eight bits of information. This basic unit of data, called a byte, is transferred all at once along the eight parallel electrical connections known collectively as the data bus.

The address bus is very similar to the data bus except it has sixteen parallel electrical connectors. Like the data bus, the address bus information is coded in binary. That is, each of the sixteen lines may have only one of two possible states, a 0 or a 1, presented to the devices as 0 or +5 volts. The buses are connected directly to the pins which go inside the plastic package and connect to the internal microcircuitry on the silicon chips. The sixteen bit address bus allows for 65536 different addressable memory locations where data may be stored.

Finally, there is a control bus which contains lines which help to control the various chips in the Commodore 64. The R/W line mentioned above is one of the control signals. With the exception of the interrupt lines, discussed later, we don't need to know much about the control bus.

Because the 6510 transfers and processes data eight bits at a time, it is known as an eight bit parallel processor. Appendix goes into much more detail on the format of the data as it is stored in the memory devices. It is strongly recommended you read it.

Memory devices come in two basic varieties as of this writing. ROM is read-only memory. When the power is turned off ROM doesn't lose its contents. ROM is a kind of chip which may be "Read" but not written to. Its contents are "burned-in" at the factory. There is a similar kind of memory device called a PROM which stands for Programmable ROM and it may be modified by a special piece of hardware called a PROM programmer. It must be erased by shining an intense ultraviolet light on its top surface for some prescribed length of time.

Both of these differ from the other main kind of memory device which is called RAM. RAM is badly misnamed. It should be called MOM for MOdifyable Memory or RAW for Read And Write. RAM stands for Random Access Memory, which means you can extract data from it in any sequence you want. The same thing is true of all currently available

types of memory, including ROM.

Anyway, the differences between RAM and ROM is that a program can write to RAM and change its contents and when the power is turned off the contents of RAM are lost, whereas ROM cannot be modified by any program under any circumstances and when the power is turned off, the contents of ROM are kept intact. In the Commodore 64, all of the operating system programs and the BASIC interpreter are in ROM. That is why you can run BASIC programs as soon as you turn the machine on, without having to load anything from tape or disk.

The Processor

The 6510 microprocessor chip is a 6502 microprocessor with a twist. It executes exactly the same instruction set and has the same addressing modes as the 6502. It has some additional features which render it more powerful than the 6502 and we will look at these at the end of this chapter. In the meantime the following discussion will describe the 6502 and it will apply equally to the 6510, the microprocessor which controls the Commodore 64.

The 6502 is a machine. Its moving parts are electrons and the only work it does is with data. It has some internal data storage which is identical in nature to the data storage in the memory devices to which it is attached through its external connectors. The internal storage is known as the machine's registers. The registers are eight bits wide and there are only seven of them. Each register has some special characteristics in the way the 6502 can utilize the data contained in it. The data processed by this machine is stored in the external memory and the registers. Processing consists of manipulating the data in some logical sequence which results in accomplishing some desired goal. The 6502 processor does its processing of data by interpreting and executing "instructions".

Instructions are data. They are stored in the memory devices as eight bit bytes which are precoded (programmed) to make the 6502 do some desired operation. The first byte of each instruction is called the Operation Code. The 6502 has pre-programmed circuitry built in to its microelectronics, etched onto its silicon chip, which can decode operation codes and figure out what it is supposed to do next based on the bit structure of the operation code. This logic comes with the 6502. The bright electronic engineers who designed the 6502 at MOS Technology back in 1975 figured out how to make it interpret bit structures and to take whatever action each operation code was designed to make it do. They preplanned a set of data manipulation operations which they thought would be useful for a microprocessor to be able to do and then set about designing the machine and the operation codes so that those operations could be interpreted and performed. The way the processor is programmed by you the programmer is for you to place in the memory of the computer a sequence of instructions designed to make the 6502 do some presumably

useful task. You choose the instructions carefully from the set of available instructions which the 6502 can perform. You code these instructions in the language which the machine can understand, machine language. Each instruction has a specific bit pattern which is understood by the 6502 to mean perform some operation and use one of the 13 possible addressing modes.

If you have an assembler you can write the 6502 instrucions in an understandable format which makes some sense to humans when they read it. The assembler will then convert the human understandable program into a machine undersandable sequence of bit patterns (bytes). ASM/EDT, which comes with Develop-64 is such a program for the Commodore 64. The next section tells you how to use it. In this section we will explore the 6502's architecture, its registers, its instruction set and its various addressing modes. First, the registers.

The Program Counter

The 16-bit Program Counter Register is actually two 8-bit registers, the PCL and PCH registers. These two registers are always used as a pair. The "PCH" stands for Program Counter High and "PCL" stands for Program Counter Low. Together, they are used by the 6502 to form a sixteen bit address pointer. The 6502 moves the contents of these two registers to the address bus when it wants to fetch an instruction from some memory chip attached to the 6502.

The Program Counter tells the 6502 where the next instruction to be executed is located in memory. When the computer is turned on, an initialization process occurs automatically. This process includes moving the data contained in addresses $FFFC and $FFFD directly into the PCL and PCH respectively. Addresses such as this which are stored in memory and point to the starting point of some other program are called "vectors". This is how the 6502 finds the address of its first instruction to be executed. So, every 6502 must have the address of the beginning of the first program to be executed prestored at $FFFC, $FFFD (65534,65535). This must obviously be in ROM.

This initialization process occurs at power-on time and whenever the RESET button (available on some expansion chassis but not on the standard Commodore 64) is pushed. After each execution of an instruction by the 6502, the Program Counter is incremented to the next instruction, and so the program flow occurs.

The A-reg

The A-reg may be thought of as the Arithmetic register. It is often called the Accumulator. Like all the registers, it is a one-byte (8-bit) register. It is the place where arithmetic operations occur. Instructions like ADC (Add with Carry) and SBC (Subtract with Carry) cause data to be added to or subtracted from the A register. The A reg may be loaded (new value brought into it) with an instruction such as LDA (Load the A-reg). Its contents may be stored out to some memory location with an instruction such as STA (Store the A-reg). The address in memory where data is loaded-from and stored-to is specified by further addressing information provided to the 6502 by the program in a manner discussed in the next chapter.

References to "STA" or "LDA" instructions are referring to the assembler language English-like mnemonic which gets translated into a machine language instruction recognizable by the 6502. There is a one for one correspondence between assembly language statements and machine language instructions.

The X and Y registers

There are two other "working" registers called the X and Y registers. These are also known as the index registers. These are all eight bit storage registers in the 6502 chip itself. The X and Y registers are used mostly in addressing functions as explained next chapter. The X and Y registers may both receive their contents (be loaded) from memory with instructions such as LDX and LDY (Load the X and Load the Y registers). They may be saved (stored) in memory with STX and STY instructions. They may also be incremented, decremented, and compared to data in external memory.

Stack Pointer (SP)

The second 256-byte block of memory ($0100-$01FF) is used by the 6502 in a special way. It is called the Stack. The Stack is a special storage block which is automatically utilized by certain instructions. Its primary reason for existence is to allow subroutine "nesting" and to allow for the smooth handling of interrupts.

Subroutines are program segments which can be executed by many

different programs. They are sub-programs, which are jumped-to and returned-from. They save having to write commonly used program segments over and over again. The stack is the 6502's communication mechanism for remembering where a subroutine was "called" from. The JSR (Jump to Subroutine) instruction is explained in detail in Chapter 5. Suffice it here to say that the JSR causes the program to jump to a subroutine in such a way that the subroutine can return to the instruction after the JSR once it is done doing its processing. The 6502 saves the return address on the stack when a JSR occurs. It pulls it off the stack when the RTS (return from Subroutine) occurs. The position of the next available stack location for recording return addresses is kept in the SP. The SP is initialized by the Commodore 64 start-up program to the value of $FF at power-on and RESET time. The high-order byte of the stack address is always $01. This is fixed inside the 6502. The first time a JSR instruction is executed, the address of where to return to is pushed onto the stack at $01FF and $01FE. The SP is then decremented by two so that the new value of the SP is $01FD. The subroutine called by the JSR may then call additional subroutines and the return addresses will be stored below the initial return address. There may be up to 128 levels of subroutines calling other subroutines. Each subroutine must have as its last instruction a RTS (return from subroutine) instruction which causes the 6502 to load the PC with the saved address from the stack and to increment the SP by two. Thus the return to the address from which the subroutine was called.

Interrupts are caused by an electrical signal to the 6502 from the outside world. There are two interrupt lines attached to the pins of the 6502. One of them, the NMI line, will cause the 6502 to be interrupted regardless of what it is doing. This is called the Non-Maskable Interrupt. This facility is provided so that hardware which has critical timing requirements may cause the 6502 to service them immediately. It is also provided as a means of unconditionally breaking into the processing of the machine if it is suspected that something has gone awry in a program and there is no other way to sieze control of the machine short of turning it off and turning it back on again.

The other kind of interrupt is a maskable interrupt. A means exists to prevent the interrupt from being serviced. An interrupt may be "masked" (made so it can't be seen by the 6502) by the means of an "interrupt disable" bit in the Processor Status register. Maskable

interrupts are those whose electrical connections are to the IRQ pin of the 6502.

Both kinds of interrupts are handled in about the same way by the 6502. The 6502 finishes processing the instruction which was in progress when the interrupt signal was recognized. It then saves the PC on the stack just as if a JSR had been executed. Additionally, it saves the Processor Status Register on the stack and decrements the stack by three. It now loads the PC with the address found in location $FFFA and $FFAB for a NMI interrupt or $FFFE and $FFFF for a non-maskable interrupt. These locations must have been pre-programmed to contain the addresses of the programs which were written to service the interrupts.

The interrupt processing routines must be exited via a RTI instruction (Return from Interrupt). This acts like the RTS instruction except that the Processor Status Register is reloaded from the stack before the PC is pulled from the stack. In this fashion, the interrupted program may continue where it was interrupted and the status of the machine will be as it was at the time of interruption.

The stack is also used by the PHA and PHP instructions to store the A-reg and the P-reg respectively in the stack. This is used to pass information to the subroutine. More information about these instructions may be found in Chapter 5.

The Processor Status Register

The Processor Status Register is a collection of eight bits, sometimes called flags, which reflect and control the operation of the 6502. The bit assignment of the P-reg is:

```
P-reg bit position     76543210
Status-bit label       NV BDIZC
```

The Negative Bit

"N" is the Negative-bit. It is turned on by the processor upon the execution of certain instructions. It reflects whether the result of an addition or subtraction is negative or not. A 1 value in the Negative bit indicates a negative result. This bit is set by load instructions and compare instructions too. See the chapter on the 6502 instruction set for a complete explanation of each instruction

and how each affects the various status bits.

The Overflow Bit

The "V" bit is the overflow bit. It reflects whether or not "two's complement" overflow has resulted from a SBC (Subtract with Carry) instruction. It is also set by the BIT instruction. See the descriptions of those instructions for a more complete explanation.

The Break Bit

The "B" bit is set by the processor when a BRK instruction is executed. The BRK instruction causes an interrupt to occur. It is a software interrupt, used mostly in debugging machine language programs. The BRK interrupt is processed almost like a maskable interrupt. The same vector is used by the 6502 to find the address of the interrupt processing routine. The only way the processing routine can know if the interrupt was a software or hardware interrupt is by examining the "B" bit in the Processor Status Register which has been stored on the stack. The BRK is not maskable, but it uses the maskable interrupt vector. Also, the return address stored on the stack is the address of the BRK instruction plus two.

The Decimal Mode Bit

The "D" flag in the P-reg is a cue to the processor to do all ADC and SBC instructions in Decimal mode. In decimal mode, the data being added or subtracted will be assumed to be composed of two decimal digits per byte. Each digit is coded as a four-bit pattern having the range of values $0 - $9. The range of values which can be contained in a byte is 0-99 decimal. The name of this data type is Binary Coded Decimal (BCD).

If the Decimal mode is clear, the ADC and SBC instructions will treat the data being added and subtracted as eight-bit binary values in the range 0-255.

The "D" bit may be set and cleared by the program with SED and CLD instructions.

The Interrupt Disable bit

The "I" flag is the interrupt-disable flag. It may be set with the SEI instruction and cleared with the CLI instruction. When set, only non-maskable and software interrupts may occur. When Clear, all interrupts are enabled.

The Zero bit

The "Z" flag is set like the "N" flag, by arithmetic and load and compare instructions. If the result of these operations gives a zero result, the Z-flag is set. Otherwise it is cleared.

The Carry bit

The "C" flag is the Carry bit. It is set by addition, subtraction, shift and compare instructions as well as the specific SEC and CLC instructions.

The N, V, Z and C bits may all be tested by conditional branch instructions. A full explanation of this facility is provided in the following chapters.

The 6510 special characteristics

The 6510 has a built-in Input/Output (I/O) port. There are eight pins on the 6510 which may be connected to other pieces of hardware. Through these pins, numbered 0-7, data may be transfered one bit at a time. The connected external pieces of hardware may transfer data directly to and from the memory of the computer. The data must be placed in memory location 1 for it to be transferred to the connected I/O device. This is the same address where the data passed by the external device to the 6510 will be found.

The individual bits of location 1 may be programmed to be either input bits or output bits. This is accomplished by setting the corresponding bits of location 0 to reflect the direction of data transfer. A 1 in any given bit position of location 0 will cause data to be transferred from the corresponding bit position of location 1 to the device attached to the corresponding I/O pin of the 6510. A 0 in any given bit position of location 0 will cause data to be transferred

from the device attached to the corresponding I/O pin of the 6510 to the corresponding bit position of location 1. Input bits are those which are turned on or off by the connected external device as it alters the voltage level of the electrical signal which appears at the I/O pin on the 6510. When the device puts a high voltage (+5 volts) signal on the connecting pin and the data direction register is set as input, the value 1 will appear in the corresponding bit position of memory location 1. A low signal (0 volts) will cause the value 0 to be stored in the corresponding bit position of location 1.

Output bits are those which are stored in location 1 by some program running on the 6510. The value of the bit to be transferred (either a 1 or a 0) gets translated to an electrical signal which appears on the I/O port pin of the 6510. The connected output device must be designed such that it understands a voltage signal of +5 volts to mean the value 1 and a voltage signal of 0 volts to mean the value 0. For data to be sent as desired, location 0 must have a 0 in the bit position of the data to be transferred from location 1 to the output device.

Location 0 is called the data direction register for the 6510's I/O port. Location 1 is called the I/O port. These registers do not exist on the 6502 and are what distinguish the two microprocessors.

The Commodore 64 has a dedicated usage of the I/O port which will be examined in later chapters.

Processing

The instructions within the program which control the processor cause it to either: 1) load data from memory into one of its internal registers or 2) to move data from one register to another or 3) to store data from one of the registers into memory or 4) to modify the data in one of the registers by some arithmetic operation or 5) to cause a change in the flow of the program.

The 6510, as it decodes each instruction after having fetched it from memory, discovers three things about the instruction: 1) the number of bytes the instruction takes in memory; 2) the addressing mode of the instruction; 3) the operation to be performed. The operation code takes only one byte for every instruction but some instructions need to supply the processor additional information beyond the op-code. This is either address information or data. If, for example, an instruction's purpose is to direct the processor to store the data contained in its A register into some location in memory, it needs to provide the 6510 the information as to where to store it. This could take one or two additional bytes depending on the addressing mode.

Each operation code has coded into it the information as to what addressing mode should be used to accomplish the desired operation. The 6510 has thirteen addressing modes.

ABSOLUTE MODE - Absolute adressing requires a three byte instruction. The first is the op-code and the next two are the two bytes of address information.

You recall that it takes two bytes (sixteen bits) to specify an address in the 6510 address space. This is because the address bus is sixteen bits wide. The first byte of the two byte address is called the high-order byte or the most-significant byte of the address. The second byte is the low-order or least significant byte. These are sometimes abbreviated the MSB and the LSB. The 6510, when it executes instructions with absolute addressing, simply fetches the next two bytes after the op-code and puts them on the address bus when it does the memory access operation specified by the op-code. The memory devices, which are attached to the address bus and the data

bus and the control bus, are signaled by the R/W signal and read the address information off the address bus to select the memory location to either store data into or read data out of. If the R/W signal signifies Write, it will take the data from the data bus and store it into the memory location specified by the address on the address bus. If it is a Read signal, it will take the data already stored at the specified location and load it onto the data bus where the 6510 will find it and do whatever the op-code indicated should be done with it. In absolute mode, the address is stored after the op-code with the least significant byte immediately following the op-code and the most significant byte following that. Example:

```
        LDA $4521
20C0    AD 21 45
```

This is an example of both an assembly language statement on the first line and the machine language following it. The format is the same as that which appears on the screen when you run The Assembler. The assembly language mnemonic is LDA. It is the assembly language equivalent of the op-code. It means Load the A register. The Assembler, which converts assembly language into machine language, decodes the mnemonic and the following operand and produces the machine language which appears on the second line. The Assembler also produces a cassette or diskette file containing the machine language which The Loader can then read and store in the appropriate memory locations, where finally it can be executed as a program.

For now, lets just look at the two statements as they appear here. The $4521 is called the operand field. It specifies to the 6510 where the data to be loaded into the A register is to be found. The 20C0 is the address where the instruction is located. It was arbitrarily picked for this example. The second field, $AD, is the hex representation for the op-code. Following the op-code is $21, the second byte of the address specfied in the assembly statement above. It is followed by $45, the first byte of the address. This is the order the 6510 expects to find addresses. The 6510 processes this instruction when its Program Counter has the value of $20C0. It fetches the op-code at that address and decodes it and executes it in the following sequence: 1) It determines from the op-code of $AD that this is an instruction to cause the A register to be loaded from a location whose address it will find immediately after the op-code. 2)

It fetches the next byte after the op-code, $21, and puts it on the least significant byte of the address bus. 3) It fetches the next byte, $45, and puts it on the most significant byte of the address bus. 4) It sets the R/W line to R. 5) It waits for the memory device to get the specified data and put it on the data bus. 6) It reads the data from the data bus and puts it in its A register. 6. It increases the PCH,PCL register pair by 3 so that it now points to the next op-code.

This is a complete instruction cycle.

ZERO PAGE MODE - If you take the sixteen bit address bus and split it in half, the first eight bits could be thought of as a "page number" and the second eight bits could then represent the address (from 0 - 255) within that page. This would mean that there are 256 possible pages, each with 256 memory locations. Zero Page would then represent all memory locations from $0000 to $00FF (0 to 255). Page one would immediately follow, containing the addresses $0100 to $01FF (256 to 511).

These are two pages which have special significance for the 6510. Page zero addresses may be specified with certain machine language instructions which are specifically coded as Zero Page addressing mode. Page one is designated the stack page as explained in the previous chapter. More about that later. The designers of the 6510 decided it would be good to have an addressing mode which allowed the 6510 to execute instructions faster and would consume less memory. The Zero Page addressing mode was part of the solution. The addressing mode is specified as a part of the op-code. In decoding the op-code, the 6510, upon determining that the addressing mode is ZP, then knows that the address to be accessed is in zero page. It therefore has only to load one more byte of addressing data, the low-order or least-significant portion of the address. The high-order half of the address will be forced to $00 by the 6510. Therefore, Zero Page instructions are only two bytes long; the op-code and the address within zero page where the data is to be stored or found. Example:

```
STA B$7C
20C0 85 7C
```

The left-arrow is the code to the assembler that this is a Zero

Page instruction. The address specified by the operand is $007C. This is a Store the A-reg instruction. Op-code $85 causes the contents of the A-reg to be stored into the specified zero page memory location ($007C in this example).

IMMEDIATE MODE - Some instructions may direct the processor to find the needed data immediately after the op-code rather than having to go to some specified address to find it. These are two-byte instructions, one for the op-code, one for the data. They execute even faster than the zero page instructions because the 6510 needn't put anything on the address bus or wait for another fetch cycle to complete before it gets the data it needs. Example:

```
SBC #25
20C0 E9 19
```

Note here that the operand field has a "#" preceeding the data value. This is the code to The Assembler that this is an Immediate Mode instruction. Note also that no "$" preceeds the value 25. The Assembler recognizes four data types, decimal, hexadecimal, symbolic labels and ASCII character. Hex numbers are indicated by a leading "$", ASCII characters by a " ' " decimal numbers by a first character of 0-9, and symbolic labels everything else. In every case, The Assembler will convert the specified data value into binary (or hex if you wish, the shorthand notation for binary) which is all the 6510 can ultimately understand. The second line displays the machine language in the same format as before. The first field is the address in hex where the instruction will reside, followed by the op-code in hex, followed by the data value in hex also. Check for yourself that $19 is the same thing as decimal 25.

The Assembler has an option you may select each time it is run to print the addresses and generated machine language in either decimal or hex. The file which is created for loading by The Loader will always contain hexadecimal.

SBC stands for Subtract with Carry. It is an instruction to Subtract the specified data value from the contents of the A-reg and to store the result back in the A-reg.

One further note: Most instructions may be specified with a variety of addressing modes. The Assembler examines the operand field

to determine which addressing mode is being specified. It then generates the proper machine language op-code to indicate both the operation to be performed and the addressing mode. As an example, "SBC $4FF3" is an absolute mode version of the SBC instruction and the op-code for it is $ED as compared with $E9 as in the above example. Appendix A contains a list of all instructions and their allowable addressing modes.

IMPLIED MODE - The implied mode of addressing is the fastest executing and the shortest instruction length. In implied mode, only the internal registers of the 6510 are addressed. Beyond the op-code, no more information is required, so implied mode only takes one byte. Examples:

```
    TAX
023F AA
    TYA
0240 98
```

TAX causes the contents of the A-reg to be transferred to the X-reg. TYA causes the contents of the Y-reg to be transferred to the A-reg.

A-REG MODE - The A-reg is sometimes called the Accumulator. This is a carry over from more primitive times. In any event, Commodore and MOS Technology choose to refer to the A-reg Mode as the Accumulator Mode. Call it what you like, it is really an implied mode. In A-reg Mode, only the A-reg and the Carry, Negative and Zero bits of the Status register are affected. The A-reg mode instructions are valid only for the "shift" instructions, ASL, ROL, LSR and ROR. For more information on shift instructions, see the next chapter. Examples:

```
    ROR A
021A 6A
    ASL A
021B 0A
```

RELATIVE MODE - There are three classes of instructions which cause the flow of the program to change. The JMP and JSR instructions are explained in the following chapter. All three types of instructions

accomplish program flow changes in the same general way. They cause the Program Counter register to be modified. The PC is the register pair which points to where the next instruction is to be found in memory. It is automatically incremented by the instruction length each time an instruction is executed. The instructions which modify the PC cause the program to "take a branch". That is, the next instruction to be executed will not be the one immediately following. It will be found at an address which is determined by the addressing data supplied by the branching instruction.

The adddressing information supplied by the "relative mode" instructions is a single byte of data which follows the op-code and which is added to the value of the PC to determine the new PC value. The branch is to an address which is the specified number of bytes away from the branch instruction itself. The address information is called the relative displacement.

These instructions only modify the PC sometimes. They are called conditional branch instructions. They test the status of a bit in the Processor Status Register and the 6510 decides at the time the instruction is executed whether to Branch (modify the PC) or not, based upon the value of the bit being tested. Example:

```
        CMP #'A         0300 C9 41
        BNE NOTA        0302 D0 03
        JMP PROCESSA    0305 4C 7D 04
NOTA    CMP #'B         0308 C9 42
        BEQ PROCESSB    030A F0 CC
```

This is a short program segment which first compares the contents of the A-reg with the character "A". The format of the program listing is the same as that which is produced by The Assembler when printer output is employed. Note the Immediate symbol, "#" and the "character" symbol " ' ". The function of the compare instruction is to set the Zero bit in the status register if the compare proves to be a match. Otherwise the Zero bit is cleared.

The instruction after the compare is the conditional branch instruction. It is a Branch Not Equal instruction. If the result of the compare results in the zero flag being set, the branch will not happen and the JMP instruction following the BNE will be executed next as usual. If not, the next instruction to be executed will be the

instruction at $0308.

Here you see one of the great advantages of having an assembler which allows you to use labels to identify program locations. You, the programmer can use meaningful symbols to refer to some address in memory. You write the assembly language program without regard to the actual addresses of each instruction in the program. If you want to cause the program to branch to some instruction somewhere in the program, you put a label in English on the instruction and The Assembler automatically computes the address of the instruction for you. In this case the symbolic label is "NOTA".

Assemblers which permit this capability are called symbolic assemblers. Single-line assemblers such as is incorporated in 64MON are severely limited by their lack of this feature.

The generated machine language is particularly interesting here. The byte following the BNE op-code is a "$03". Relative addresssing means that the value found in the byte after the op-code is the number to be added to the PC to find the address to be Branched to. That is, if the 6510 finds that the status bit being tested by the Branch instruction indicates a Branch should be taken, it then adds the value found in the byte after the op-code to the PC. (It has already incremented the PC by two before testing the status bit). The Assembler automatically computes the difference between the address of the instruction following the BNE instruction and the beginning of the instruction NOTA. In this case the amount of adjustment is three bytes. NOTA occurs three bytes past the address of the JMP instruction. It is possible to specify a conditional branch forward by as much as 127 bytes or backwards as much as 128 bytes. When backward branches are specified, the displacement value in the byte after the op-code must contain a negative number.

We mentioned earlier that a single byte may represent the decimal range of 0-255. For special situations such as the relative branch instructions it is convectenient to allow the 256 possible values of the eight bits to represent a range of numbers from -128 to +127. To accomodate this need, a system was devised to indicate negative numbers. It is called the two's complement system. It seems a little strange at first, but with a little practice it, too, can be mastered.

Negative one is represented as $FF. Negative two is represented as $FE (or 254 in regular decimal). You can convert a number to its negative by subtracting it from 256 then converting the

result to hex. For example, the hex representation of -6 is 256-6=250=$FA. This system is popular with computer designers because it makes arithmetic easy. Note that 256 in hex is $0100. Subtracting six from $0100 gives $FA. But $0100 takes two bytes. The maximum value of one byte is $FF. If we add one to $FF we get $00 with a carry of one. And with $FF representing -1 it is very nice that when we add 1 to -1 we get 0. Likewise when we add 1 to minus 2 ($FE) we get $FF (-1). Another advantage of this system is that all the negative numbers have the high-order bit on (leftmost bit value = 1). The positive numbers are $00 (0000 0000) to $7F (0111 1111) and the negative numbers are $80 (1000 0000) to $FF (1111 1111). It is not coincidental that the N-flag (Negative bit of the Status Register) is set every time any arithmetic operation results in a value with the high-order bit on.

The Assembler will automatically convert relative branch displacements to the proper value for both forward and backward branches. And The Assembler will also allow the expression of negative numbers in the more familiar format of decimal numbers (-34, -122, etc.) and do the conversion to the two's complement value for you. In debugging, however, it sometimes comes in handy to be able to do the conversion yourself and is worth knowing how to do.

INDEXED MODES - The X register and the Y register are sometimes called the index registers. This is because they are used as indexes to data. That is, the relative position of data in a string may be addressed by specifying a base or starting address of the string plus some position index to indicate which data element in the string is being addressed. The X and Y registers can be used with certain instructions to be the position index. The instruction specifies a base address and an index register. The 6510 adds the value of the index register to the base address to get the effective address of the data to be accessed. This is a very useful capability. The following program segment illustrates the use of indexed addressing to move a string of data from one place in memory to another:

```
STRING1 EQU $0400        0400
STRING2 EQU $0480        0480
        EQU $2000        2000
        LDX #10          2000 A2 0A
LOOP    LDA STRING1,X    2002 BD 00 04
        STA STRING2,X    2005 9D 80 04
        DEX              2008 CA
        BPL LOOP         2009 10 F7
```

Several new things are presented in this program segment. The first two statements cause The Assembler to equate a symbolic label with a specific address in memory. For every subsequent reference to the label being EQUated (STRING1 in this example) The Assembler will know that the address being referred to is the one in the operand field of the EQU statement ($0400). Note the generated machine language which follows the statement; LOOP LDA STRING1,X. The address $0400 is automatically generated by The Assembler (in the required low-order-byte-first format).

The third EQUate tells The Assembler what value to assign to the Location Counter. The Location Counter is The Assembler's equivalent of the 6510's Program Counter. The difference is that the Program Counter is an actual hardware register contained on the 6510 chip. The Location Counter is The Assembler's symbolic equivalent of the PC. The address printed on the machine language line after each assembly instruction is the value of the Location Counter for each instruction. It shows us where the generated machine language will be in memory when The Loader finally loads the program into memory. Every EQUate statement sets the Location Counter to the address expressed in the operand field of the statement.

The LDX instruction loads the X-reg with the value 10. The next two instructions illustrate the indexed addressing mode. The first instruction loads the A-reg with a byte of data found at address $400A. The address specified in the LDA instruction is $0400. The contents of the X-reg are added to the specified address by the 6510 before putting the address on the address bus. Since the X-reg has just been loaded with the value 10, the address where the data will come from to be loaded into the A-reg is $0400 + $000A or $040A.

The next instruction turns right around and stores that same data back in memory at the address $048A. So we now have three copies

of the same byte of data. One copy in $040A, one in $048A and one in the A-reg. The next instruction, DEX, causes the X-reg to be DEcreased or decremented by 1. Since it had the value of 10 coming into this instruction, after the instruction is executed, it will have the value of 9. The next instruction, BPL LOOP, will conditionally branch to the instruction which has the label LOOP. Note this is a backward branch of 9 bytes. The generated machine language value of how far to branch is $F7. Remember about negative branch displacements? 256 - 9 = 247. 247 / 16 = 15 with a remainder of 7. Hence the hex value of -9 = $F7. Conveniently, The Assembler made that calculation for us.

The BPL instruction tests the Negative bit of the Status Register. BPL stands for Branch if PLus. The Negative bit is affected by every execution of a DEX instruction (and many other instructions as well). If the X-reg was zero before the execution of the DEX, the result of the DEX would be a negative number in the X-reg. The Negative bit would be set to one. The BPL tests the negative bit. If it is not a one (not negative) the X-reg must still be positive or zero after having been decremented by the DEX.

The branch will be taken back up to LOOP. The A-reg will now be loaded from address $0409 and stored into address $0489. The X-reg will be decremented again and tested to see if it went negative yet. Once again, it is positive (it has the value 8 now). The branch will be taken back up to LOOP, the same process will occur once again, this time moving a byte from $0408 to $0488. Once again, the X-reg will be decremented and tested and found not-negative. The loop will be executed a total of 11 times, with the X-reg varying from 10 to 0, the address of data being loaded into the A-reg varying from $040A to $0400, the address of where data is moved to varying from $048A to $0480.

This is the process by which a wide variety of repetitive operations are performed upon data with the 6510. This is a very standard loop. If it still seems mysterious to you read it over again and when we get to the section on actually using The Assembler / Editor, The Loader, The Monitor, and The Decoder, we will create an actual program which does just this process. We will execute the program one step at a time with The Monitor, watching how everything works and seeing the registers and the memory locations changing as we go through the program.

There are four modes of addressing which are called indexed. Both the X-reg and the Y-reg may be used in indexed instructions and both may be used in combination with absoulte and Zero Page modes. Like the absolute and ZP addressing modes, the absolute,X and ZP,X instructions take three and two bytes respectively. The Y-reg and the X-reg function identically in their respective modes. The four indexed modes then are: abs,X ; ZP,X ; abs,Y ; ZP,Y .

INDIRECT MODE - There is only one instruction which uses the simple indirect mode of addressing. This is the JMP (addr) instruction. The parenthesis around the absolute address signifies indirect. What happens with the indirect jump is the following: 1) The 6510 gets an address from the two bytes immediately following the op-code. Rather than load the PC with this value directly, it fetches an address from the specified memory location and the memory location immediately following it. This is the address which it loads into the PC. Thus a change in the program flow is caused.

To state this another way, for the JMP indirect to work as desired, there must be an address pre-stored somewhere in memory. The JMP instruction must tell the 6510 where that address is located in memory. The 6510 will then load its PC with the address stored therein. Such a prestored address is called a vector. BASIC has several vectors saved in the first few pages of memory which point to various processing programs.

Vectors are convenient ways of allowing the flexible design of operating systems such that new versions and updates to the operating system can be compatible with the old versions. Programs which need to use the various routines pointed to by vectors will not need to be changed because of a different location of the routine in the new version. The vector will be the only thing which will have to be modified to allow compatibility.

(INDIRECT),Y - This mode is somewhat similar to the indirect mode discussed above. The above instruction was applicable only to the JMP instruction. This mode is applicable to various data access and manipulation instructions. These instructions are two byte instructions. The second byte of the instruction specifies an address in zero page. Like with the previous mode, there must be an address stored at the specified location. The big difference here is that the Y-reg is added to the address found in zero page to give the 6510 the

eventual address of where the data should come from or go to. The (Indir),Y mode is actually used exactly like the addr,Y mode. It is useful for doing loops. The only difference is that the base address of the loop is stored in Zero Page rather than specified directly by the instruction. The instruction then specifies the Zero Page location of the base address. This is a very handy way to program a subroutine which is used at different times and called from different places in the mainline program to do the same general task but with differing sets of data. The base address of the data to be manipulated by the subroutine must be appropriately set up in Zero Page each time just before the subroutine is called. The subroutine itself never has to change anything. It uses the zero page vector to get the data it's been called to use. Example:

```
SUBRA   LDY B$40
LOOPA   LDA ($7C),Y
        STA ($80),Y
        DEY
        BPL LOOPA
        RTS
```

Here, the subroutine, SUBRA, is designed to do a general purpose move of data from some location in memory to some other location. The number of bytes to be moved is found in location $0040. The location of where to move the data from is found in $007C and $007D. The address of where to move it to is found in locations $0080 and $0081. The calling program must set those locations up with the appropriate values for the subroutine to function as desired. Note that no left arrow is required for the (indexed),Y mode even though the specified address is always in Zero Page.

(INDIRECT,X) - This is the last and probably the least useful addressing mode provided with the 6510. You may have guessed that it is similar to the previous mode. It would be a lot more useful if it were identical except for the register used. Unfortunately, it isn't. You should notice that the X is inside the parenthesis, whereas in the previous mode, the Y is outside the parenthesis. This is reflective of the important distinction. The parenthesis indicate the "indirection". In the previous example, the Y-reg was added to the address found in Zero Page, and the result of that addition provided

the address of the desired data. Here, the X-reg is added to the specified address to find where in Zero Page the vector resides. The X-reg is therefore an index to a table of vectors stored in Zero Page. This mode is not useful, therefore, in the same way that the other one is. It might find some use in unusual situations where there is a need to have a list of addresses of data bytes and a routine is needed to process the various data bytes. Such a routine would step through the list, an address (two bytes) at a time, using the X-reg to bump thru the list and to point to the appropriate address at which the data will ultimately be found. Perhaps you can find some better use of this mode.

These last four chapters have been fairly packed with information on the workings of the 6510. It will be necessary to write some programs and to examine the programs of others to get a comfortable feeling about the use of the various instructions and addressing modes. The next chapter will present every available instruction on the 6510. After completing it, you will be able to start to take control of your computer by writing powerful machine language programs.

6510 - Instruction Set

There are 56 separate instructions which the 6510 has been designed to execute. There are also two "pseudo-op instructions" which are not a part of the 6510's instruction set but which The Assembler understands.

There are 13 addressing modes. Some instructions are limited to a single addressing mode, others are capable of utilizing up to eight. Most of the instructions are quite simple functionally, and require only a sentence or two to describe their characteristics. We will cover these first.

Register-only instructions

TAX - Transfer the contents of the A-reg to the X-reg. Only the receiving register is modified. The Zero bit of the Status Register is set (made to have the value 1) if the value transferred is zero, otherwise it is cleared (made to have the value 0). The Negative bit of the Processor Status Register is set if the high-order bit of the value transferred is on (value 1), else it is cleared.

TAY - Transfer A-reg to Y-reg. The A-reg is stored into the Y-reg. The same notes apply as with TAX.

TYA - Transfer Y-reg to A-reg. The Y-reg is stored into the A-reg. The same notes apply as with TAX.

TXA - Transfer X-reg to A-reg. The X-reg is stored into the A-reg. The same notes apply as with TAX.

TXS - Transfer X-reg to Stack Pointer. The X-reg is stored into the Stack Pointer. No status bits are affected. This is the only way of initializing the Stack Pointer.

TSX - Transfer Stack Pointer to X-reg. The Stack Pointer is stored into the X-reg. The same notes apply as with TAX.

SEC - SEt the Carry bit. The Carry bit is set to value 1.

CLC - CLear the Carry bit. The Carry bit is cleared.

SED - SEt the Decimal Mode bit. The Decimal Mode bit is set to value 1. The Decimal mode of arithmetic is enabled.

CLD - CLear the Decimal Mode bit. The Decimal mode of operation is disabled. The Decimal Mode bit is cleared.

CLV - CLear the oVerflow bit. The overflow bit in the Processor Status Register is cleared.

CLI - CLear the Interrupt disable bit. The Interrupt disable bit in the Processor Status Register is cleared, allowing interrupts to occur.

SEI - SEt the Interrupt Disable bit. The Interrupt disable bit in the Processor Status register is set to 1, causing all interrupts to be masked (disabled) until the Interrupt disable bit is cleared with a CLI.

DEX - DEcrement the X-reg. The value contained in the X-reg is decreased by 1. If the resulting value in the X-reg is zero, the Zero bit is set in the P. If it is not zero, the zero bit is cleared. If the resulting value in the X-reg has the high-order bit on, the Negative bit in the Processor Status Register is set. Otherwise the Negative bit will be cleared. Note that register values of 0-127 will have a clear high-order bit and values of 128-255 will have the bit set.

DEY - DEcrement the Y-reg. The value of the Y-reg is decreased by 1. The Zero and Negative bits are affected as with the DEX instruction.

INX - INcrement the X-reg. The value of the X-reg is increased by 1. The Zero and Negative bits are affected as with the DEX instruction.

INY - INcrement the Y-reg. The value of the Y-reg is increased by 1. The Zero and Negative bits are affected as with the DEX instruction.

Memory accessing instructions

INC - INCrement memory. The value of a byte located in memory is increased by 1. The Zero and Negative bits are affected as with the DEX instruction. Only ZP; ZP,X; ABS; ABS,X modes are valid.

DEC - DECrement memory. The value of a byte located in memory is decreased by 1. The same comments as apply for INC.

LDA - LoaD the A-reg. The contents of a memory location are transferred to the A-reg. The Zero and Negative bits are affected as with the TAX instruction. Valid addressing modes are: Immediate; ZP; ZP,X; Absolute; Absolute,X; Absolute,Y; (Indirect,X); (Indirect,Y)

LDX - LoaD the X-reg. The contents of a memory location are transferred to the X-reg. The Zero and Negative bits are affected as with the TAX instruction. Abs, ZP and ZP,Y addressing modes are valid.

LDY - LoaD the Y-reg. The contents of a memory location are transferred to the Y-reg. The Zero and Negative bits are affected as with the TAX instruction. Abs, ZP and ZP,X addressing modes are valid.

STA - STore the A-reg. The contents of the A-reg are transferred to a memory location. No registers or status bits are affected. Valid addressing modes are: ZP; ZP,X; Absolute; Absolute,X; Absolute,Y; (Indirect,X); (Indirect,Y)

STX - STore the X-reg. The contents of the X-reg are transferred to a memory location. No status bits or registers are affected. Abs, ZP and ZP,Y addressing modes are valid.

STY - STore the Y-reg. The contents of the Y-reg are transferred to a memory location. No status bits or registers are affected. Abs, ZP and ZP,X addressing modes are valid.

Conditional Branch Instructions

BCC - Branch Carry Clear. The program counter will be modified by the amount of the specified displacement if and only if the Carry bit is clear (0). Forward branches may occur up to 128 bytes from the address of the first byte following the branch instruction. Backward branches may occur up to 127 bytes from the same address. No other registers other than the PC are affected. Only Relative addressing mode is valid.

BCS - Branch Carry Set. The program counter will be modified by the amount of the specified displacement if and only if the Carry bit is set (1). Same comments as for the BCC instruction.

BEQ - Branch EQual. The program counter will be modified by the amount of the specified displacement if and only if the Zero bit is set (1). Same comments as for the BCC instruction.

BNE - Branch Not Equal. The program counter will be modified by the amount of the specified displacement if and only if the Zero bit is clear (0). Same comments as for the BCC instruction.

BMI - Branch MInus. The program counter will be modified by the amount of the specified displacement if and only if the Minus bit is set (1). Same comments as for the BCC instruction.

BPL - Branch PLus. The program counter will be modified by the amount of the specified displacement if and only if the Minus bit is clear (0). Same comments as for the BCC instruction.

BVC - Branch oVerflow Clear. The program counter will be modified by the amount of the specified displacement if and only if the Overflow bit is clear (0). Same comments as for the BCC instruction.

BVS - Branch oVerflow Set. The program counter will be modified by the amount of the specified displacement if and only if the Overflow bit is set (1). Same comments as for the BCC instruction.

Jump instructions

JMP - JuMP. The PC is loaded with the specified address, causing a change in the flow of the program. Both Absolute and Indirect addressing modes are valid.

JSR - Jump to SubRoutine. The PC is first incremented by 2. The new PC is then stored on the stack. The PCH is stored in the stack location addressed by the Stack Pointer. The Stack Pointer is decremented and the PCL is then stored in the new stack location as addressed by the SP. The SP is decremented a second time and the address specified by the JSR instruction is loaded into the PC, causing a jump to the specified subroutine location. Note that the address stored on the stack is not what might be expected. The address is of the third byte of the JSR instruction, not the address of the next sequential instruction after the JSR. The RTS instruction, which causes a return from the subroutine, compensates for this anomaly. Generally, the casual programmer needn't worry about the mechanics of stack operations as long as she always has a RTS for every JSR. However, advanced machine language programmers are fond of direct stack manipulation techniques, especially for passing arguments to subroutines.

RTS - ReTurn from Subroutine. The Stack Pointer is first incremented. The PCL is loaded from the stack address pointed to by the Stack Pointer. The SP is then incremented again and the PCH is loaded from the stack. The PC is incremented to compensate for the JSR operation of putting the address of the third byte of the JSR on the stack instead of the address of the next instruction. Now the PC has the address of the instruction immediately following the most recent JSR instruction. The program flow is thus returned to the mainline program from the subroutine.

RTI - ReTurn from Interrupt. This instruction reverses the process which occurs when an interrupt occurs. The Processor Status Register is retrieved from the stack where the interrupt caused it to be stored. The PCH and PCL are then reloaded from the stack where they too were stored as a part of the 6510's interrupt processing. The return to the point of interruption is thus complete, with the

Processor Status Register having the same value it had at the time of interruption.

BRK - BReaK. Interrupt processing is caused to occur. The address of the next byte following the BRK instruction is saved on the stack. The Break bit is turned on in the Processor Status Register which is then saved on the stack. The PC is loaded with the address found at memory location $FFFE and $FFFF.

Stack Push & Pull Instructions

PHA - PusH the A-reg. The A-reg is stored on the stack at the address pointed to by the SP. The SP is then decremented. All PHA's should generally be matched with a following PLA.

PLA - PuLl the A-reg. The SP is incremented, then the A-reg is loaded from the stack location pointed to by the new value of the SP.

PHP - PusH the P-reg. The Processor Status register is pushed onto the stack in the same fashion the A-reg is with the PHA.

PLP - PuLl the P-reg. The Processor Status register is pulled off the stack in the same fashion the A-reg is with the PLA instruction.

Pseudo-op Instructions

BYT This is not really an instruction in the instruction set of the 6510. It is an instruction which The Assembler recognizes and interprets to mean generate machine language data. The operand field of the BYT instruction can express several types of data which The Assembler will understand.

If the first character of the BYT is a " $ ", the following characters must be hex characters, i.e. 0-9, A-F. The Assembler will handle a string of hex characters up to 75 characters in length. It will generate a data string with two nybbles (a half byte - 4 bits) per byte, inserting a $0 in the high-order nybble of the high-order byte if there are an odd number of characters specified.

If the first character is a " ' ", The Assembler will create a data string with as many bytes as there are characters following the " ' ". The values of the generated bytes will be the ASCII values of the corresponding characters.

If the first character of the operand is a ">" or "<" the remainder of the operand field will be interpreted as an address expression and the generated byte will be either the high-order byte of the address or the low-order byte depending on whether the first character is s ">" or a "<". Any other first character will cause the operand field to be interpreted as an address expression and the assembler will compute a two-byte address in the low-order-byte-first format. Address expressions are covered fully in the chapter on writing assembly language programs.

EQU - EQUate. This pseudo-op does not generate any data which gets stored into memory by The Loader. It is an instruction to The Assembler to set the Location Counter and to cause the label in the label field to be assigned to the address which is expressed in the operand field.

Shift Instructions

ASL - Arithmetic Shift Left. The contents of the A-reg or of a memory location are shifted one bit position to the left. The low-order bit position is forced to value zero. The high-order bit is shifted into the Carry bit. The Negative bit is set if the bit shifted into the high-order bit position is a 1. It is cleared if it is a zero. The Zero bit is set if the resulting value of the shifted byte is zero, cleared if it is not.

As an example, if the A-reg has the value $CC (1100 1100), after the "ASL A" instruction has been executed, it will have the value $98 (1001 1000) and the carry bit will be set. If the A-reg has the value of $5F (0101 1111) the "ASL A" will cause it to become $BE (1011 1110) and the carry bit will be clear. Note that each left shift causes the A-reg to double in value as long as the high order bit is not a one before the shift. Shifting left is a convenient way of multiplying a value by two. Using ASL in combination with ROL, a multiple precision shift may be effected. See the description of the

ROL instruction. Valid addressing modes are: Accumulator; ZP ; ZP,X ; ABS ; ABS,X.

ROL - ROtate Left. The A-reg or a byte in memory may be shifted one bit to the left. The high-order bit gets shifted into the Carry bit. The low-order bit recieves the previous contents of the Carry bit. The same addressing modes apply as for the ASL instruction.

A multiple precision bit shift is one where a string of bytes is treated like one long bit pattern and the shift causes the bits which come out of the high-order positions of one byte get shifted into the low-order positon of the next byte in the sequence. For example:

```
        LDY #3      SHIFT 4 BITS
BITSH   LDX #4      THRU 5 BYTES
        ASL STR,X   RIGHT-MOST BYTE
        DEX
ROLIT   ROL STR,X
        DEX
        BPL ROLIT   ALL BYTES ?
        DEY
        BPL BITSH   ALL BITS ?
```

This routine would cause the string of five bytes at STR to be left shifted four bits. Each execution of the ROL shifts the contents of the Carry bit into the low order bit of the byte being shifted. The Carry bit will contain the bit which was shifted out of the high-order bit position of the previous byte. The ASL is used as the first shift instruction to force zero bits into the low order positions of the low-order bytes.

LSR - Logical Shift Right. The contents of the A-reg or memory location specified is shifted one bit position to the right. The low-order bit gets shifted into the Carry bit. The high-order bit position is forced to zero. The Zero bit is set based upon the resulting value of the shifted byte. The Negative bit is forced to zero. The same addressing modes apply as for the ASL instruction.

ROR - ROtate Right. All bits in the rotated byte are shifted one bit

position to the right. The Carry bit is shifted into the high-order bit position and the low-order bit position is shifted into the Carry bit. The same addressing modes apply as for the ASL instruction. The LSR and ROR instructions are the same as the ASL and ROL instructions except they shift bits in the opposite directions. Note that a one-bit shift right results in an effective division by two.

Boolean arithmetic instructions

AND - Logical AND. The A-reg is logical ANDed with the specified data byte. The boolean AND operation is performed between corresponding bits of the two bytes. Each bit position of the pair of bytes is operated on individually, the result of the operation replacing the corresponding bit in the A-reg. The rules of the AND operation are: if the two bits being ANDed are value 1, the result is a 1; if either bit is a 0 the result is a 0. That is, [0 AND 1] = 0 ; [0 AND 0] = 0 ; [1 AND 1] = 1 ; [1 AND 0] = 0. Example:

```
      11001010  Memory
AND   10101100  A-reg
      --------
      10001000  new A-reg
```

Only the bit positions which had a 1 in both bytes ended up with a 1 in the result. The AND instruction is frequently used to selectively clear individual bits while maintaining the status of the other bits in the byte. This is done by creating a "mask-byte" which has a 0 in every bit position which needs to be cleared (set to 0), and a 1 in all the other bit positions. The mask byte may be in either the A-reg or the specified memory location but the result of the operation always replaces the A-reg.

This process works because a 0 ANDed with either a 0 or a 1 gives a 0 result while a 1 ANDed with a 0 gives a 0 and ANDed with a 1 gives a 1.

ORA - Logical OR. The A-reg and the specified memory location are logical ORed together, the result replacing the A-reg. The boolean OR operation, like the AND operation, is a bit by bit operation. Each bit

of the A-reg is ORed with the corresponding bit of the byte in memory by the following rules: If either bit is a 1, the result is a 1. If both bits are 0, the result is 0. That is, [1 OR 1] = 1 ; [1 OR 0] = 1 ; [0 OR 1] = 1 ; 0 OR 0 = 0. Example:

```
          11001010  Memory
ORA       10101100  A-reg
          --------
          11101110  A-reg
```

The ORA is frequently used to selectively turn bits on (set to 1). Like with the AND, a mask must be created which indicates the desired bits to set and the bits to be unaffected. The OR mask must have a bit on in the bit positions to be set and off in the positions which need to be maintained. This is opposite of the AND mask. There, 1-bits maintained the status quo. Here, 0-bits have that responsibility. A 0 in the mask byte when ORed with a 1 gives a 1 and when ORed with a 0 give a 0. And a 1 in the mask byte always results in a 1 result. [1 OR 1] = 1; [1 or 0] = 1. Valid addressing modes are the same as for the AND instruction.

EOR - Exclusive OR. The contents of the specified memory byte are EORed with the contents of the A-reg, replacing the A-reg with the result. Like the AND and OR instructions, this is a bit oriented instruction. The rules of Exclusive-Oring are: The result will have a 1 in any bit position for which only one of the two bytes being EORed have a one. All other bit positions of the result will have a 0. (i.e. [1 EOR 0] = 1; [1 EOR 1] = 0; [0 EOR 0] = 0; [0 EOR 1] = 1).

Example:

```
          11001010  Memory
EOR       10101100  A-reg
          --------
          01100110   New A-reg
```

The EOR instruction is useful for inverting bits. A 1 in any mask bit position will cause the corresponding bit in the result to

have the opposite value as that of the corresponding bit in the object byte. A 1 before EORing will result in a 0 after and vice versa. A zero in the mask byte will cause the corresponding bit in the object byte to go unmolested.

BIT - BIT test. The A-reg is logically ANDed with the contents of the specified memory location. The Zero flag is set if the result of the operation gives a zero result. The Negative bit is set if the high order bit of the memory location is set. The Overflow bit is set if the second-highest-order bit (the 6-bit) of the memory location is set. The Negative and Overflow bits are cleared if the 7-bit and 6-bit of the memory location are clear. The A-reg is not affected by the execution of this instruction.

This instruction is very useful for testing individual bits of bytes in memory. The A-reg is loaded with a mask which has ones in the bit positions to be tested and zeros in the rest. If any of the memory byte's bits in the tested positions are on the result of the ANDing operation will be non-zero. Note that the mask may be either in memory or in the A-reg for the test to work. The Negative and Overflow bits are set based only upon the bit configuration of the memory byte however. Example:

```
MASK    BYT $06
        LDA MASK
        BIT VAL1
        BMI BIT7
        BVS BIT6
        BNE BIT1OR2
OK      EQU @
```

The value of the byte assigned to label MASK is $06. The 1-bit and the 2-bit are on and all others are off. The BIT instruction ANDs the contents of the A-reg, $06, with the byte at VAL1 and the result will be non-zero only if either the 1-bit or the 2-bit of VAL1 is a one. If VAL1 has its high-order bit on, the program will branch to BIT7. If the 6-bit is on in VAL1 the program will branch to BIT6. If either bit-2 or bit-1 are on, the branch to BIT1OR2 will be taken.

The " @ " is used in the last statement of the program. It has a special significance to The Assembler. Used in the operand field of an instruction it is like a

symbolic label except it references the current value of the Location Counter. Here, it performs the function of assigning the value of the Location Counter to the label "OK".

Arithmetic instructions

ADC - ADd with Carry. Addition is performed with the A-reg, the specified memory location and the Carry bit. (A-reg) = (A-reg) + (addr) + (carry). The (...) is used here to mean "the contents of". The result replaces the A-reg. The mode of arithmetic is determined by the status of the Decimal mode bit at the time the instruction is executed. If set, the mode of addition is the Decimal mode. If clear, the mode is binary. Valid addressing modes are the same as for the LDA instruction.

Binary addition is quite simple. It is just like decimal addition except the highest number you have to worry about is 1. The rules of addition are:

```
0 + 0 = 0
0 + 1 = 1
1 + 1 = 10  (2 in decimal)
```

The third example is the only one which is different from decimal addition. When we add a pair of binary numbers with more than one bit apiece, we proceed from right to left just like decimal addition. We add the two bits together and if the result is greater than 1 we have to carry 1. So, 11 + 11 = 110. Doing this addition one step at a time, taking the rightmost bits, 1 and 1, and adding them by the above rules, we see that the answer is 10, or 0 with a carry of 1. Next we add the next pair of bits plus the carry. 1 + 1 + 1 = (1 + 1) + 1 = 10 + 1 = 11. Thats a 1 with a carry of 1. Note that 1 + 1 + 1 = 3 in decimal and 1 + 1 + 1 = 11 in binary which is the binary equivalent of 3 decimal. The most you have to remember in binary addition of two numbers is 0 + 0 = 0 ; 0 + 1 = 1 ; 1 + 1 = 10 (0 carry 1) ; 1 + 1 + 1 = 11 (1 carry 1). The ADC instruction does this binary addition work for you, so you might not need to know it. On the other hand, you probably will when you go to debug your Galactic Gobbler Game. So you might just as well learn it now.

When in Decimal mode, the 6510 expects the data it is adding to be in "binary coded decimal" format. This is yet another data format. In BCD, the eight bits of a byte are interpreted to be two four bit decimal digits. Each four bit digit may have the hexadecimal values of 0-9. If there is some other bit configuration in the range of A-F in either half of the bytes being added, the results will be unmeaningful. When the ADC is executed in Decimal mode, the two low-order digits are added and any decimal carry is added together with the two high-order digits. The carry bit will be set if there is a decimal carry from the addition of the high-order digits.

It is standard procedure to clear the carry bit before using the ADC instruction because the Carry bit is added into the result. This is a nice feature when you are doing multiple precision addition such as adding two 32 bit numbers together. The carry bit is the needed communication between the successive bytes of the addition.

Example of 32 bit multiple precision addition:

```
        LDX #3      4 BYTE ADDITION
        CLC
NEXT    LDA VAL1,X
        ADC VAL2,X
        STA VAL3,X
        DEX
        BPL NEXT
```

SBC - SuBtract with Carry. The specified byte in memory and the inverse of the Carry bit are subtracted from the A-reg, replacing the A-reg with the result. That is, (A-reg) = (A-reg) - (addr) - [1 - (Carry)]. The Carry bit is set if the contents of the A-reg are greater than or equal to the value being subtracted from it. The Carry bit is cleared if the value subtracted is less than the contents of the A-reg. In deciding whether the contents of the A-reg and the memory location are greater than or less than one another, the 6510 interprets the values as unsigned integers in the range of 0-255.

The carry bit is like an inverted borrow. If a borrow is required, the Carry bit is 0. If no borrow is required, the Carry is 1. The Negative and Zero bits are set based on the result of the subtraction. The Overflow bit is set if "two's complement overflow" occurred. Valid addressing modes are the same as for the LDA

instruction.

The normal subtraction procedure is to set the Carry bit before the SBC is executed. Using the SBC to compare two values requires testing the Minus bit after the subtraction is complete.

If the desire is to branch to PRGA if the contents of VAL1 are less than the contents of VAL2 the following program would be used:

```
        SEC
        LDA VAL1
        SBC VAL2
        BMI PRGA
OK      EQU @
```

The above routine works for unsigned numbers. If a comparison is being made of numbers which may have negative values, a more complex routine must be used. The possibility of "negative overflow" exists when subtracting numbers which are intended to represent values in the range of 0 to 127, -1 to -128. If we try to subtract 10 from -124, the result would be -134. This is out of the range of negative numbers. The 6510 lets us know that if this was a signed operation, the result had "negative overflow". This is done by setting the Overflow bit in the Processor Status Register. The value which ends up in the A-reg is 122 (256-134) in the above example. The unsigned equivalent of the -124 is 132 (256-124). And 132 - 10 = 122. This positive result gives a false indication of the relative magnitude of the two signed numbers. When working with signed numbers the following technique is necessary to make accurate comparisons of magnitude:

```
        SEC
        LDA VAL1
        SBC VAL2
        BVS CHKMI
        BMI PRGA
        BPL OK
CHKMI   BPL PRGA
OK      EQU @
```

Positive overflow can occur the same way. Suppose VAL1 has

the value of 126 and VAL2 has the value of -4. The intention of the above program is to branch to PRGA if VAL1 is less than VAL2. 126 - (-4) = 130. 130 is out of the range of signed numbers (0-127). Positive overflow occured. The 6510 tells us this in the same way, by setting the Overflow bit. Does the program work for this case? 126 is not less than -4. The overflow bit was set by the SBC. The branch is taken to CHKMI where the Negative bit is tested. It is on because the value in the A-reg is greater than 127. It is 130. So the branch is not taken to PRGA. It works!

Compare Instructions.

CMP - CoMPare. The magnitude of the specified memory location is compared with the magnitude of the A-reg. The Zero and Negative bits are set as though a SBC had occurred. The A-reg is not modified by this instruction. The carry bit need not be pre-set or cleared before executing the instruction. The carry bit is set if the compare finds that the A-reg is greater than the value of the contents of the memory location. The Overflow bit is not affected by this instruction. It is therefore not possible to use the CMP to make magnitude comparisons of signed numbers. The description of the SBC instruction illustrates the technique for accomplishing this. Valid addressing modes are the same as for the LDA instruction.

CPX - ComPare the X-reg. The X-reg is compared to the contents of a specified memory location. This instruction functions exactly like the CMP instruction except the register being compared is the X-reg. The valid addressing modes are Absolute, Zero Page and Immediate.

CPY - ComPare the Y-reg. The Y-reg is compared with the contents of the specified memory location. This instruction functions exactly like the CMP and CPX instructions. Addressing modes are the same as CPX.

Impotent Instructions

NOP - NO OPeration. The amazing NOP instruction does absolutely nothing. It causes the 6510 to spin its electronic wheels for a few microseconds. It takes up one byte of memory.

Specifications for Assembly Language

General

This chapter gives a compact description of the capabilities and restrictions of the assembly language of the Develop-64. For a tutorial walk-through of the operational use of the tools read Chapters 1-4.

The assembly language statement may be a maximum of 79 characters in length. This is exactly two lines on the screen minus the prompt character ("). There are two kinds of statements: comments and regular assembly language statements. Comments must begin with a " ; " in the first position. There are no other restrictions on comments other than the disallowance of the quote (") character. This restriction applies equally well to all statements.

Labels

The label field, if used, is the first field of the assembly statement. It is not a required field. If used, it must start with a letter (A-Z) and may be of any length. The single character "A" should not be used, as it will be confused with the "A" of the Accumulator addressing mode when appearing in an operand field. Other characters which should be avoided are the seven algebraic operators and the quote ("). Labels of more than seven characters will cause the assembly listing to be somewhat less neat appearing but work just fine.

Mnemonics

Standard mnemonics

The mnemonic is the English-like code which gets translated by Develop-64 into the machine language op-code. A discussion of the mnemonics is found in chapter 9. See appendix A for a complete list of all valid mnemonics and their legal addressing modes.

The EQU pseudo-op

Develop-64 comes equipped with two special mnemonics, EQU and BYT. These do not generate processor-executable instructions as do the standard mnemonics. The BYT pseudo-op is covered in the next paragraph. The EQU mnemonic is an instruction to Develop-64 rather than an instruction to the machine. It has two functions. The first is to set the location counter. This is a function sometimes left to a separate pseudo-op such as ORG in other assemblers. The second function is to equate a label with an address. The format of the EQU instruction is:

```
LABEL EQU addr-expression
```

The label field is optional. There is one exception to the general rules for address expressions for the EQU instruction. The expression may not reference labels which do not precede the EQU instruction in the segment. This is true only for the EQU instruction and is the only limitation on it. Any program references to the label on the EQU instruction will refer to the address expressed in the operand field.

The address expression must always be present. All the rules and capabilities of address expression as defined below apply.

EQUate statements are used to allow one to use meaningful English words instead of numeric values for addresses and other numbers. The EQU is set up once and all references to the assigned label will be interpreted by the assembler as the value associated with the label. The EQU may also be used to reserve a block of memory and assign it a symbolic label. The following two lines will assign the label, DATA to the 200-byte block of data starting at $C000:

```
DATA EQU $C000
     EQU @+200
```

The "@" (at sign) symbolizes the location counter, explained in more detail later in the chapter.

The BYT pseudo-op

The BYT instruction, unlike the EQU instruction, does cause machine language code to be generated by Develop-64. It is not generally used to generate executable code like the standard mnemonics. Its function is to provide a means of causing data to be stored in memory. The data generated by the BYT instruction may be specified in several ways. Depending on the first character of the operand field, the BYT instruction may specify hexadecimal strings or ASCII strings or address constants or single byte values of hex, decimal or ASCII.

Hexadecimal strings

If the first character of the operand field is "$" then a hex string will be generated. All characters following the "$" should be hex digits, 0-9 A-F. There may be any number of such digits, up to the maximum line length constraint. Develop-64 will create one byte of data for each pair of digits. If there are an odd number of digits, Develop-64 will append a "0" on the left of the string.

Literal text strings

If the first character of the operand field is a " ' ", all following characters will be translated to the Commodore ASCII value of the characters. Each text character in the operand will generate one byte of data. All the characters which may be entered from the keyboard with the exception of the quote (") are legal. The BYT literal instruction is the only instruction which cannot have a comment field. Comments would be interpreted as part of the literal text. The length of the literal string is limited only by the maximum line length limitation.

Data constants

A "<" in the first position of the operand field causes the expression which follows to be evaluated by the rules of address expression evaluation. The single byte which is generated is the low-order byte of the resultant two-byte evaluation.

A ">" in the first position causes the generation of a single byte whose value is the high-order (or page number) of the address expression which follows.

Address constants

If the first character of the operand does not meet any of the above criteria then the operand is evaluated as an address expression and the two bytes of data which are generated are in the 6510 address format, with the low-order byte preceding the high-order byte. It should be noted that a BYT instruction such as:

```
BYT $3CC
```

will not generate an address constant in the low-high format. It is a hexstring and will generate 03 CC. To get an address constant it would be required to write an instruction such as:

```
BYT 0+$3CC
```

which would generate CC 03, as expected.

The following will also generate the same address expression:

```
MSG EQU $3CC
BYT MSG
```

The operand field

The operand field follows immediately after the mnemonic field and specifies to Develop-64 the address of the data to be accessed and the mode by which it will be addressed. There are actually 13 distinct addressing modes by which the location of data is specified. The format of the operand field determines which mode will be used and therefore exactly which op-code will be generated and how many bytes of address data will be generated. Not all instructions may use the same set of addressing modes. Appendix A specifies the valid addressing modes for each instruction or mnemonic.

Address expressions

One of the features of this assembler is the ability to create address expressions of great complexity (or simplicity) with ease and flexibility. The term "address expression" is meant to include the "immediate" character and BYT data constants as well as actual memory locations. Address expressions are algebraic combinations of one or more terms.

Terms

There are five different kinds of terms which may be algebraically combined in an address expression. Each has its own distinguishing format. The result of the evaluation of the expression must not be out of the range of -65536 to 65535 but the individual terms have no magnitude restrictions. Address expressions which exceed the magnitude restrictions will cause an ERR 6.

Decimal format

Any term in an address expression which begins with 0-9 or a "." will be interpreted as a decimal term. Decimal terms may be integers or may contain a decimal point and a fractional component. Floating point notation may also be used (e.g. 3E2 will be interpreted as 300).

Hexadecimal format

Hexadecimal terms are those which have a "$" as the first character. All following characters, up till the next operator or the end of the expression, must be 0-9, A-F. Hex terms which have characters other than these will cause an ERR 5 when the program is assembled.

Literal format

Any term in an address expression which begins with the character " ' " will be interpreted as a literal. That is, the value

assigned to that term will be the ASCII value of the first character to the right of the " ' " . Any characters following the first character following the quote will be ignored.

Symbolic label

Any term which does not start with a "$" or a "'" or a "@" or a "." or a number (0-9) will be interpreted as a symbolic label. Develop-64 will search the entire program looking for a match on the label field. If none is found, ERR 3 will be generated. If a match is found, the term will be assigned the value of the address associated with the label. Labels may be of any length.

Location counter

The location counter is Develop-64's equivalent of the program counter. The location counter is assigned the value of the address assigned to the first byte of the instruction in which it appears. It is the address at which the instruction will reside once The Loader POKES the load segment into memory. This assembler uses the "@" symbol to signify the location counter. Most other assemblers use the "*" symbol for this function. However, the "*" is interpreted by Develop-64 as a multiplication operator. The "@" symbol seems a logical choice, being the "at" sign and signifying where we are "at" in memory. Any term which has "@" as its first character will have the value of the location counter. Any following characters within the term, should they exist, will be ignored.

Algebraic operators

The various terms of the address expression may be combined algebraically by the following operators: + - * / ^ & % . As the evaluation proceeds from left to right, each term is added to, subtracted from, multiplied by, etc. the result of the evaluation of that portion of the expression to the left of the operator, giving a new current evaluation. The fractional portion of the result of any operation will be carried into the next operation. The final evaluation of the expression will truncate any fractional components and will convert negative numbers into sixteen bit two's complement values.

Addition " + "

The addition operator causes the term immediately following the "+" to be added to the result of the evaluation of the portion of the expression to the left of the "+".

Subtraction " - "

The subtraction operator causes the term to the right of the minus sign to be subtracted from the expression to the left of the minus.

Multiplication " * "

The multiplication operator causes the expression to the left of the " * " to be multiplied by the term to the right of the " * " .

Division " / "

The division operator causes the expression to the left of the " / " to be divided by the term to the right of the " / ".

Exponentiation " ^ "

The exponentiation operator causes the expression to the left of the " ^ " to be raised to the power of the term to the right of the " ^ ". Fractional powers may be employed with decimal terms. It is therefore possible to do such things as take square roots with expressions such as:

X ^ .5

Logical AND " & "

The logical AND operator causes the result of the evaluation of the expression to the left of the "&" to be logically ANDed with the term to the right of the "&" . The logical AND operation compares the two terms of the operation bit by bit, giving a result with a bit set on in every bit position where both terms have a bit on. Its main use is to force bits off in certain desired bit positions, while

retaining the status quo in all other positions.

Logical OR " % "

The logical OR operator causes the result of the evaluation of the expression to the left of the "%" to be logically OR'ed with the term to the right of the "%" . The logical OR operation compares the two terms of the operation bit by bit, giving a result with a bit set on in every bit position where either term has a bit on. Its main use is to force a bit on in certain desired bit positions, while retaining the status quo in all other positions.

Expression evaluation

As has been indicated in previous sections, several terms may be combined into an algebraic expression, the eventual evaluation of which will result in the address specification. There are several features of the evaluation algorithm which must be understood for proper use of the algebraic capability. First, the order of evaluation is not like BASIC. Here, the expression is evaluated from left to right, regardless of what operators appear in the expression. For example, the expression:

LA&1+GH%$C/IJ

would be evaluated in the following way: The address of LA would be AND'ed with the value 1. The result would be added to the address of GH. That result would be OR'ed with $0C. The result of that operation would be divided by the address of IJ. The final massage of the expression converts negative expression values to a two's complement value by the addition of 65536 to the negative value. By way of example:

-1 = $FFFF, -2 = $FFFE, -256 = $FF00, and -257 = $FEFF.

">" The high-order symbol

If the ">" symbol is the first character of the address expression, the expression will be evaluated as usual but the final

operation will be to divide the expression's value by 256. This results in the high-order byte of a two-byte value. This is a convenient way of expressing the page-number of an address. Example:

```
SCREEN EQU $0400
       LDA #>SCREEN
```

will cause the value $04 to be loaded into the A-reg.

"<" The low-order symbol

This symbol operates just like the high-order symbol except the final operation is to produce only the low-order byte of the address expression evaluation.

Complex equations

For those who wish to use more complex equations than can be handled by expressions which are evaluated strictly left to right, it is possible to accommodate them by a series of EQU's which themselves are expressions. For example, to represent an equation such as:

```
( B + C - ( D / E )) * ( F & G )
```

you could write the following code:

```
DE  EQU D/E
FE  EQU F&G
AB  EQU B+C-DE*FE
```

Much more complex expressions may be represented in a similar fashion.

The comment field

Comments are entered on the statement line by skipping at least one space after the operand field before keying the comment. Comments are not allowed on BYT instructions which define literal strings.

Explicit zero page addressing convention

When zero page mode of addressing is desired it is necessary to explicitly indicate such desire by preceeding the address expression with the "left-arrow" (←). This is the only way the two-byte zero page addresssing mode will be selected. Omitting it will result in the long form of the instruction. This provides the capability of addressing zero page with a long instruction if desired. Note that this requirement is for ZP, ZP,X and ZP,Y modes only and not for (indir,X) and (indir),Y modes.

Graphics on the Commodore 64

These next chapters will attempt to expand upon the information found in the Commodore 64 Programmer's Reference Guide (PRG). The PRG should be considered an indispensible reference tool. In it you will find complete descriptions of the various special function integrated circuits which make the 64 the powerful computer it is. Also included is information on the 64's memory organization and the "Kernal" routines and how to use them.

This chapter will provide the machine language programmer's perspective on graphics generation. Joy stick and paddle usage and sound generation will be covered in the following chapter. The last chapter will cover some of the internal programs contained in the 64's ROM and the means by which you can make use of them.

The Video Interface Chip

The Video Interface Chip (the VIC-II) is the electronic machine, the integrated circuit within the 64 which, among other things, causes patterns to be displayed on your video screen. It is also known as the 6567. The VIC-II is connected to the 6510 and the RAM and ROM of the 64 via the address, data and control busses.

The VIC-II runs continuously from the time the power is turned on untill it is turned off. It is under control of its own internal program which is built into the electronics of the chip. It has 47 8-bit registers which may be addressed by the 6510 and therefore any program running on the 6510. The registers are the communication medium by which we direct the operation of the VIC-II. They are wired directly to the address bus in such a way that we can change their contents by storing data into addresses $D000 through $D02E (53248-53295). POKEs from BASIC and STAs, STXs and STYs from machine language into these locations will modify the internal registers of the VIC-II. We will look at the specification of these registers and how they cause the various graphics capabilities to be activated and controlled.

Bank Switching

But first, it is necessary to understand the bank switching capability incorporated into another chip, the 6526 Complex interface

adapter chip #2 (CIA#2). Bank switching is a term which refers to the capability of disconnecting one bank of memory and connecting another bank in its place. When the VIC-II is doing its thing, it "looks at" a bank of memory to find the patterns to send out of the video port and onto your video screen. The VIC-II is designed to "see" any one of four 16K banks of memory. Which bank it sees at any one time is determined by the "bank-select bits" of the CIA#2. To switch between different banks the following machine language program will do the trick:

```
BANK    EQU 3       COULD BE 0,1,2 OR 3
        LDA $DD02   DATA DIR REGISTER
        ORA #3      BITS 0,1 SET TO OUTPUT
        STA $DD02
        LDA $DD00   OUTPUT PORT A CIA#2
        AND #$FC    FORCE BITS 0,1 OFF
        ORA #BANK   SELECT BANK
        STA $DD00
```

When the 64 is powered up bank 0 is selected automatically. The address range of the various banks and the corresponding value which must be specified in the BANK EQUate follow:

BANK EQU	BANK NUMBER	BANK ADDRESS RANGE
3	0	$0000-$3FFF (0-16383)
2	1	$4000-$7FFF (16384-32767)
1	2	$8000-$BFFF (32768-49151)
0	3	$C000-$FFFF (49152-65535)

All data used by the VIC-II in its creation of video images will come from the bank of memory which is currently selected via the CIA#2. The map of what data will be found where in each block will depend on the values stored in the various VIC-II registers. But before getting into that lets look at how the VIC-II puts characters on the screen.

Multiple character sets

You have no doubt noticed that each character which is displayed on your screen is a composite of up to 64 dots arranged in an 8 by 8 block. The information which describes the characteristic dot pattern of each character is stored in the 64's ROM.

A certain section of RAM is reserved for use by the 64 as "screen memory". There is one byte of screen memory reserved for every character position on the screen. Since there are 40 columns and 25 rows there are 1000 bytes of screen memory. The data stored in each position of screen memory may have the value of 0-255, the range of values of one byte of information. This may be considered the character "code". Each code stands for one unique pattern of dots. There are therefore 256 possible dot patterns which may be displayed at any one time. The VIC-II scans continuously through screen memory and translates each code for all 1000 screen positions. The translation process consists of using the value of the code (0-255) to compute the displacement (or number of bytes) into a character pattern table. Each character pattern has 64 bits (8 bytes) of "dot" information. So to find the right pattern, the VIC-II multiplies the value of the code by 8 and adds the result to the starting address of the character pattern table. There, the VIC-II finds the 8-byte pattern of dots to send to the video screen. The first eight bits of a character's pattern are the top row of dots which appear on the screen. The second eight bits are the second row and so on untill the

eighth eight bits for the bottom row. The VIC-II does this scanning of screen memory and translating the codes found there using the pattern table and sending the dot information out to the video port all automatically and continuously.

The BASIC "PRINT" statement causes "screen codes" to be stored in screen memory. It is possible also to POKE screen codes into screen memory. Machine language programs may store screen codes into screen memory. The VIC-II automatically and continuously scans the screen memory, getting one screen code per screen location and translates the screen code into an eight by eight dot pattern which it then sends to the video output port to be displayed. In appendix B of the PRG there is a table of screen codes and the corresponding characters which get displayed. These are the standard characters which are burned into the 64's character-table ROM at $D000-$D1FF (53248-55295).

You are probably aware that when you hold the Commodore and the Shift Keys down simultaneously, the characters which appear on the screen are from a second character set. One character set contains upper and lower case characters and the other contains upper case and the graphics characters. There are two sets of 8 x 8 patterns stored in ROM and the VIC-II can be "switched" between the sets. In fact, the VIC-II can be switched between several different tables of 8 x 8 patterns. One of the VIC-II's registers determines where in the bank the character table is to be found. Since each bank which the VIC-II can see is 16K bytes long and since 256 characters take 8 * 256 or 2048 (2K) bytes, there can be eight different character tables accessible to the VIC-II in any given bank. Since there are 4 banks available, there may be up to 32 separate character sets accessible to the VIC-II.

The register which controls which of the eight blocks within the current bank to use for the character pattern table is found at 53272 ($D018). It takes three bits to specify the pattern location. These three bits are bits 3, 2 and 1 of the register. The four high-order bits of this same register are used for another purpose and must not be modified when switching between character sets. Bit-0 is not significant. The following routine illustrates the selecting of the desired character-pattern table.

```
TABLE  EQU 0        OR 2 OR 4, 6, 8, 10, 12 OR 14
       LDA 53272    GET THE REGISTER
       AND #$F0     TURN BITS 3-0 OFF
       ORA #TABLE   SET THE TABLE-SELECT BITS
       STA 53272    RESET THE REGISTER
```

The value of the TABLE EQU selects the location of the character pattern table within the selected bank. The following table gives the position within the bank for the various possible values of TABLE:

TABLE	LOCATION WITHIN BANK
0	$0000-$07FF (0-2047)
2	$0800-$0FFF (2048-4095)
4	$1000-$17FF (4096-6143)
6	$1800-$1FFF (6144-8191)
8	$2000-$27FF (8192-10239)
10	$2800-$2FFF (10240-12287)
12	$3000-$37FF (12288-14335)
14	$3800-$3FFF (14336-16385)

Since the location specified above is the relative address within the currently selected bank, to arrive at the actual address of the character pattern table it is necessary to add the starting address of the selected bank to the addresses above. Selecting an alternate table address and creating your own pattern table is the technique which you must use to create your own custom characters and to do high-resolution bit-image graphics. We'll discuss those capabilities more later.

The Commodore 64 comes pre-programmed with only two character pattern tables, the upper-case/graphics set and the upper/lower case set. These are in ROM at $D000-D1FF and $D800-$DFFF. This also happens to be where the VIC-II's registers reside, which seems mighty confusing at first. There is an explanation. The 6510 has a very interesting feature which allows both RAM, ROM and I/O to all occupy the same address space. Not at the same time of course. The bank switching concept is used here. We discussed at the end of Chapter 7 the special characteristics relating to location 0 and 1 in the 6510's memory space. We mentioned at that time that the 64 uses the I/O port

at 0,1 for its own special purposes. This port controls a bank-switching device which switches RAM, ROM and I/O in and out of the memory space. This is the way the 64 can have a full 64K of RAM, 20K of ROM and 4K of I/O all addressable on a 16-bit address bus which can only have 64K possible unique addresses.

If the programmer wishes to copy a part or all of the character ROM into some part of RAM, it will be necessary to switch the I/O at $D000 out and switch the ROM in. Because interrupt processing on the 64 utilizes the I/O which you want to switch out, it is necessary to disable interrupts before switching in the ROM. The following routine will turn off interrupts, switch in the ROM, move the character table, switch the I/O back in and turn interrupts back on.

```
        LDA 56334
        AND #$FE     SET BIT 0 OFF
        STA 56334    TURNS INTERRUPTS OFF
        LDA B1
        AND #$FB     BIT 2 OFF
        STA B1       SWITCHES ROM IN
        LDX #8       COUNT OF PAGES TO MOVE
        LDY #0       BYTE COUNTER
LOOP    LDA (SRCE),Y
        STA (DEST),Y
        INY
        BNE LOOP
        DEX          DECR PAGES BY 1
        BEQ DONE
        INC BSRCE+1  NEXT PAGE OF DATA TO MOVE
        INC BDEST+1  NEXT PAGE OF WHERE TO PUT IT
        JMP LOOP     DO NOTHER PAGE
DONE    LDA B1
        ORA #4
        STA B1       SWITCH I/O BACK IN, ROM OUT
        LDA 56334
        ORA #1       TURN INTERRUPTS BACK ON
        STA 56334
```

This routine assumes SRCE and DEST are labels defined elsewhere in the program which address two-byte zero-page vectors which have been pre-set to address the character pattern ROM and the place you

wish to move the character table to.

When the system is powered up bank 0 and TABLE 4 are automatically selected by the power-up program. The 64 has another feature which can be either convenient or a bother depending on your needs. When bank 0 or bank 2 is selected and the TABLE value is 4 or 6 the VIC-II will get the character patterns from the standard ROM pattern table rather than from RAM as you might expect. Thus, the VIC-II sees the ROM character patterns at $1000-$1FFF and at $9000-$9FFF even though there is really RAM there. It's a very tricky chip, that VIC-II. Note that this is a VIC-II related phenomenon only. The memory at these addresses is really RAM and you may treat it as such for all purposes. Programs and data stored there will not be in any way molested.

Now, to create your own custom characters you must build a pattern table which describes the dot patterns of the characters you wish to build. The pattern tables consist of up to 256 patterns. Each pattern has eight bytes. A bit turned on in any of the bytes will cause the corresponding dot on the screen to be illuminated. (The screen dots are also called pixels which comes from "picture elements".)

In the standard pattern table the screen code "0" refers to the first 8-byte pattern, that of the character "@". This pattern may be found beginning at 53248 ($D000) when the ROM has been switched in. ree the above routine for switching the ROM in machine language. To do it in BASIC the following routine could be used.

```
POKE 56334,PEEK(56334)AND254: POKE 1,PEEK(1)AND251
```

The second character in the standard character ROM starts at 53256 and is the pattern for the letter "A" (screen code 1). Every eight bytes, another pattern is stored. This goes on for 256 8-byte patterns. The screen codes 128-255 are the "reverse" of the first 128 patterns. That is, where a bit is on in one, it is off in the other.

The first byte of screen memory holds the character code which corresponds to a 8 by 8 pattern in the table to be displayed in the first column of the first row of the screen. The first 40 positions of screen memory correspond to the first row of the screen. The second 40 characters correspond to the second row and so forth.

At power-on time the 64's bank address is initialized to $0000 and the screen memory location is set to $0400 (1024). You may create

and switch between several screens very rapidly to create the effect of a foreground/background or for other interesting effects.

If you POKE1024,1 the screen code "1" will be placed in the screen memory position corresponding to the upper left character position of the screen. Do it and you should see the letter "A" appear there. The VIC-II has translated the value 1 into the character pattern for the letter "A" by computing 1(the character code) * 8 (the number of bytes per pattern) and adding that to the start of the character pattern table which starts at 53248 (ROM location which can only be PEEKed by switching the I/O out and the ROM in as explained above). It starts at that address (53256) building the dot pattern to send to the video screen. It finds the following eight bytes of data starting at 53256:

addr	dec	hex	binary
53256	24	$18	00011000
53257	60	$3C	00111100
53258	102	$66	01100110
53259	126	$7E	01111110
53260	102	$66	01100110
53261	102	$66	01100110
53262	102	$66	01100110
53263	0	$00	00000000

If you look at the above pattern of ones and zeros, you will be able to see the shape of the character "A" formed by the ones on the backgraound of zeros. All of the characters' shapes are formed in the same fashion. The characters you form in your program must follow the same rules.

To create the new characters it is necessary to build a set of 8-byte patterns in the same way the original set is constructed. It is not necessary to reserve a complete 2048 byte block of memory for a complete 256 characters if you only have a few characters you wish to ever see on the screen. However, you may not use both the standard sets and any special programmed sets simultaneously. Once the bank address and the VIC-II's register at 53272 ($D018) has been set, the entire screen will be generated using the character patterns found at the specified pattern table area.

Multiple Screens

The location of screen memory is determined by both the 16K bank selected as described above and the high-order 4 bits of the register found at 53272 ($D018). The location of screen memory within the bank may be selected with the following simple routine:

```
SCRN EQU $00 (OR $10,20,30,40,50,...E0,F0)
     LDA $D018     GET REGISTER
     AND #$0F      TURN OFF HIGH 4 BITS, SAVE LOW 4
     ORA #SCRN     TURN ON THE SCREEN SELECT
     STA $D018     SAVE NEW REGISTER
```

There are 16 possible locations within the current bank at which screen memory can begin. Screen memory is 1000 bytes long (25 x 40). The following table defines the displacement within the bank for the beginning of each of the possible screen locations and the value of SCRN which will select each.

SCRN	SCREEN MEMORY STARTING ADDRESS WITHIN BANK
$00	$0000 (0)
$10	$0400 (1024)
$20	$0800 (2048)
$30	$0C00 (3072)
$40	$1000 (4096)
$50	$1400 (5120)
$60	$1800 (6144)
$70	$1C00 (7168)
$80	$2000 (8192)
$90	$2400 (9216)
$A0	$2800 (10240)
$B0	$2C00 (11264)
$C0	$3000 (12288)
$D0	$3400 (13312)
$E0	$3800 (14336)
$F0	$3C00 (15360)

Color controls

The border around the area where characters are displayed is called the border or exterior area. The color of that area may be set independently of all other colors. The register at 53280 controls the border color. To set it the following program sequence may be used:

```
BORDER EQU 53280
       LDA #COLOR (MAY BE IN RANGE OF 0-15)
       STA BORDER
```

COLOR must have been defined in the label file of some EQU with the operand being a number having the value of 0-15. The value chosen will determine the actual color of the border area:

```
0 - Black     8  - Orange
1 - White     9  - Brown
2 - Red       10 - Light Red
3 - Cyan      11 - Gray 1
4 - Purple    12 - Gray 2
5 - Green     13 - Light Green
6 - Blue      14 - Light Blue
7 - Yellow    15 - Gray 3
```

The color of the background of the screen may be selected with the same variety of choices. The register which controls background color is at $D021 (53281). A similar few statements may be used to modify the screen color.

```
SCRNCLR EQU 53281
        LDA #COLOR
        STA SCRNCLR
```

Character colors

The individual characters displayed on the screen may have their colors set independent of any other character. There is a block of RAM reserved for just this purpose. Unfortunately, it is one area of

control information which is not selectable like the screen or character pattern table. That is, there is only one color memory area and it defines the color of each of the 1000 character positions on the screen regardless of which screen memory or character set you are using. The 1000 bytes of color memory starts at 55296 ($D800) and runs through 56295 ($DBE7). The first 40 bytes define the color of the top row of characters and so on. The 16 possible colors and the vlaues which correspond to them are those listed above for the background and border.

Alternate display modes

There are two control bits which further determine the way the VIC-II interprets character shape data from the character memory. The 4-bit in the register at 53270 sets the multi-color mode. The 5-bit of the register at 53265 sets the high-res bit map mode. Either or both of these bits may be set and reset by the programmer by typical bit-manipulation code:

```
HIRES EQU 53265
MULTI EQU 53270
SETHI LDA HIRES  GET THE HIRES REGISTER
      ORA #32    OR ON THE 5-BIT
      STA HIRES  STORE THE NEW HIRES REGISTER

SETMC LDA MULTI  GET THE MULTI-COLOR REGISTER
      ORA #16    OR ON THE 4-BIT
      STA MULTI  STORE THE NEW MULTI-COLOR REG

CLRHI LDA HIRES
      AND #$DF   AND OFF THE 5-BIT
      STA HIRES

CLRMC LDA MULTI
      AND #$EF   AND OFF THE 4-BIT
      STA MULTI
```

The way in which the video display is created depends upon the value of these two bits and sometimes on the value of the individual

screen color memory bytes. The following table summarizes the possibilities:

Multi-color	Hi-res	Bit-3 of color mem	Mode
0	0	any	Std Char (entire screen)
1	0	0	Std Char (this character)
1	0	1	Multi-color (this char)
1	1	any	Multi-color bit map (screen)
0	1	any	Bit mapped hi-res (screen)

Multi-color Characters

If multi-color is selected and hi-res is not, the characters in the character pattern table will all be displayed in what is called multi-color mode. In multi-color mode of display, the character table which describes the shape of the patterns of each of the characters will be interpreted in a different fashion by the VIC-II. In the standard character display mode each of the 64 bits of the character pattern represents a background/foreground choice (bit on = display the pixel in the color set in color memory for the character in question ; bit off = pixel is set to background color i.e. invisible). Here, in multi-color mode, the characters are of lower resolution. Each byte of the character pattern describes only four pieces of display information instead of eight. The bits of the pattern information are grouped in twos and the characters displayed will have four double-wide dots in eight vertical rows. The pairs of bits in each byte will determine the color of the double-wide dots in the following way: If the pair is a 00 the color of that dot will be the same as the background, or a non-display dot. If the value is a 01 the color of the double-wide dot will be that of the "background color #1" which is set in the register at 53282 ($D022). If the value of the bit pair is a 10, the color will be that set in the register at 53283 ($D023) (the background color #2). If the bit-pair has the value of 11 the color of the displayed fat dot will be the color which is found in the color memory associated with the character position on the screen. Since the 3-bit of the color memory byte must be on to activate multi-color character mode, only the second eight colors (colors 8-15). may be used for the "character color" in multi-

color mode.

If bit-3 of the character color byte for any given screen position is not on, the usual standard resolution character is generated. Here, the only colors which may be used are 0-7.

Bit mapped Graphics

In this mode each pixel on the screen may be individually turned on and off. It is useful for plotting geometric shapes and for graphing functions and creating complex graphics.

Since there are 1000 screen positions and each position has 64 pixels, it must take 64,000 bits or 8000 bytes to store all the screen information in bit-mapped mode. Bit-mapped mode is activated by setting bit-5 of the register at 53265 ($D011) as indicated above. Once the mode has been selected, the VIC-II now uses screen memory to indicate the color combination of the bits which are turned on and off in the eight by eight square of the screen. The high-order four bits tell it the color of the bits which are set to 1. The low-order four bits tell it the color of the bits which are set to 0.

But where are the bits stored which indicate pixels to illuminated in the two possible colors? In standard character mode recall that the screen memory bytes told the VIC-II where to go to find the character bit pattern to display on the screen. In bit-mapped mode the character memory is still where the information is stored. But now the first eight bytes of character memory always corresponds to the 64 pixels in the upper left corner of the screen. The first byte contains the pixel information for the top row of eight dots. The second byte contains the information for the row of eight dots immediately below the first row and so forth untill the eigth row eight dots in the upper left corner of the screen. The second eight bytes contains the dot information of the second 8 by 8 square on the first row of the screen. The 40th group of eight bytes corresponds to the upper right corner and the 1000th eight bytes contains the information for the lower right 64-bit square. The program in the appendix illustrates in detail how the machine language programmer can plot points on the screen by computing the proper byte and bit to turn on or off given a pair of numbers which can vary from 0-319 and 0-199 (the x and y axis of the plot).

Multi-color bit-mapped mode

By setting both the hi-res and the multi-color bits you can bit map using three colors in addition to the background screen color. This is similar to multi-color mode in that the dots are fat in the horizontal direction and the bits which indicate what is on and off on the screen are grouped in pairs. The same amount of memory is required to multi-color bit map the screen, 8000 bytes. It is, like in standard bit-map mode, found where the character pattern table is set to be. The four two-bit combinations in each byte tell the VIC-II which colors to make the fat dots on the screen. A "00" combination says make it the background color. A "01" says make it the color indicated by the high-order four bits of the associated screen memory byte. A "10" combination says make it the color indicated by the low-order four bits of the screen memory byte. Finally, the "11" combination says make it the color set in the color memory byte associated with the screen location in question.

Extended background color mode

Unbelievably, there are still more modes of display available through the combined electronic intelligence of the 6510 and the VIC-II chip. If you have somehow made it this far, you might as well wade through this one too because sprites are still to come.

This mode is selected by turning on the 6-bit of the register at 53265 ($D011). What it does is to let you vary the background color from character to character if you are not satisfied with having the same background on the entire screen. The way it's done is: The two high-order bits of the screen memory bytes are interpreted as a background color selector. The first thing this means is that in this mode the character set is limited to 64 characters (the remaining 6 bits can only have 64 possible unique values). The two selector bits indicate the register for the VIC-II to use to make the background color. The following table gives the possibilities:

Bit-7	Bit-6	Background color register
0	0	53281 ($D021)
0	1	53282 ($D022)
1	0	53283 ($D023)
1	1	53284 ($D024)

You may set the registers with the usual 16 possible color codes as identified above.

Sprite Graphics

The subject we've all been waiting for. As you probably know, sprites are special movable figures which may be designed to have any shape and color combination and will coexist with all other modes of display simultaneously. They are nice game programmers' tools. It is recommended you read the description of sprites and their capabilities in the PRG, as it is quite good. Once you have a comfortable feeling about machine language you will be able to translate the BASIC statements given in the PRG easily to assembler.

Eight sprites may be defined at one time. The method of sprite definition is identical to that of custom character definition. In fact, there are two modes of sprite definition, standard and multicolor, just like characters. The multi-color mode mimics the other multi-color standards.

The standard sprite is 24 dots wide by 21 high. The multi-color sprite takes just as much space on the screen but each definable dot in the horizontal direction is twice as fat as a standard dot. Therefore, there are only 12 definable horizontal positions in the multi-color sprite. Also, the bits which define the shape of the sprite are grouped in pairs for multi-color, again just like multi-color characters. The standard sprite has higher resolution but may be of only one color.

The memory location where sprites are "drawn" must be in the same 16 K block as the screen memory and character memory as explained at the beginning of this chapter. At the end of the 1 K block of memory which holds the screen memory the VIC-II expects to find the pointers to the sprite definitions. Each pointer is a single byte and may have a value in the range of 0-255. The pointer value, when

multiplied by 64 and added to the base address of the bank, will point the VIC-II to the sprite definition block. The VIC-II does all that arithmetic itself. You needn't worry about it any more than to set it up once correctly. That is, you must pick the place in memory where you would like to build a sprite (it must be on an even 64-byte boundary), compute how far that is from the beginning of the block, divide by 64 and store that value in the sprite pointer location at the end of the screen memory 1K block. Got that?

The sprites are numbered 0-7 and the corresponding sprite pointers are in locations 1016-1023 of the screen memory block. The sprite number also specifies the sprite intersection priority. The lower the sprite number, the higher the priority. This means that when two moving sprites pass each other on the screen, the one with the lower sprite number will pass in front of the higher numbered sprite. Transparent areas in the front-passing sprite will enable the lower-priority sprite to be seen through the "window".

There is a "sprite-enable" register at $D015 (53269) which is the way sprites are triggered once they have been built and pointed-to. The VIC-II, when it detects a sprite has been enabled, will find its shape definition via the pointer, its desired location on the screen via another set of registers. It will proceed to then build the video signal to reflect the description you have provided it. The sprite-enable register has an enable bit for each sprite. Logically enough, bit-0 enables sprite-0 and bit-1 enables sprite-1, etc.

Each standard sprite may have any of the 16 possible colors. The registers which determine sprite color are at 53287-53294 ($D027-$D02E) for sprites 0-7 respectively. All dots which are "on" as flagged by a bit on in the 63 byte sprite descriptor area will have the color indicated in the appropriate register. The other space as defined by bits turned off (set to 0) will take the color of whatever background is behind the sprite at any given time. As mentioned before, the sprites may have more than one color if defined as a multi-color sprite. This is accomplished by setting the corresponding bit (0-7) in the sprite multi-color register at 53276 ($D01C). When a sprite is so selected, the VIC-II will interpret the bit pairs of the sprite shape description in the following way: A "00" bit pair will cause transparency. That is, the background will shine through. A bit pair of "01" will cause the double-wide dots having that definition to be displayed in the color set in the sprite multi-color register #0 at 53285 ($D025). The bit pair "10" will cause the

aforementioned sprite color register appropriate for the sprite in question (at 53287-53294) to appear. Finally, a bit pair of "11" will cause the color set in sprite multi-color register #1 at 53286 ($D026) to shade all parts of the sprite which are set up with the "11" definition.

Sprites may be expanded in both the horizontal and the vertical directions. Registers at 53277 ($D01D) and 53271 ($D017) are the sprite-expand registers for the horizontal and the vertical directions respectively. The bit-positions in each register correspond to the sprite to expand. When the bit is on the sprite will be automatically expanded to twice its size in the axis selected. The direction of expansion is to the left and toward the bottom of the screen.

To position a sprite on the screen, it is necessary to tell the VIC-II where you want it to go. As you might have guessed, there are some registers which do just that. The sprites may be positioned on or off the screen. There is a sort-of sprite overflow area on all borders of the screen. Sprites may be made to drift smoothly off the screen by properly defining the position of the sprite.

The position which you must give to the VIC-II is that of the upper left-most dot of the sprite, even if that dot is defined as an invisible dot. The position you define is a horizontal pixel position and a vertical pixel position of that upper-left corner of the sprite. Horizontal positions may vary from 0 to 511 and vertical from 0 to 255. Since there are only 320 pixels by 200 in the actual viewing area this leaves the necessary space above, below and to the right and left of the screen for off-screen sprite movement. The left of the screen is horizontal pixel position 24 and the right side of the viewable screen is pixel position 343. So, a sprite would be visible at least partially if the specified horizontal position were between 1 and 343 (assuming that there are non-transparent dots in the rightmost and leftmost columns of dot positions in the sprite definition). Sprites expanded in the horizontal direction will be at least partially visible if the specified horizontal position is less than 343 or greater than 488. The second condition is due to the wrap-around nature of the pixel addressing scheme used by the VIC-II. Position 511 is equivalent to -1 on the left side of the screen.

The top of the screen may be considered to be vertical pixel position 50 (from the top of the off-screen area which starts at 0). The bottom of the screen is pixel position is 249. Normal sprites will be completely off the top of the viewing screen if the vertical

sprite position is set at 29 or less. It will be off the bottom if the vertical position is 250 or greater. Expanded sprites will be off screen if less than 9 and if greater than 249.

The registers to set position information are at 53248 through 532634. The first 16 registers are the horizontal and vertical position registers for sprites 0-7 respectively. The horizontal position needs more information than can be contained in a single byte, however. Up to 512 unique pixel positions may be specified for the horizontal direction. Therefore another bit is required to completely define the position. That bit is found in the Most-significant-bit register at 53248. Each sprite, 0-7 has its horizontal most-significant-bit of position information stored in the corresponding (0-7) bit positions of the MSB register. Smooth horizontal sprite movements will take some extra care from the programmer to keep track of the position and set that extra bit correctly.

Finally, collision detection must be covered. Sprites may collide with other sprites and with the background. It can be nice to know when a collision has occurred. The versatile little VIC-II watches over its video domain and reports all such happenings to you. All you have to do is read the register which the VIC-II maintains for that purpose.

Collisions are defined as a non-transparent portion of a sprite overlaps a non-transparent portion of another sprite or background characters. For the purpose of collision detection, multi-color sprite dots defined with the "01" bit pair are considered transparent. A little quirk there.

The sprite-sprite collision register is at 53278 ($D01E) and each bit in the register stands for a sprite. Any time a sprite is involved in a collision with another sprite both sprite bits are turned on in the collision register. So, if you care if collisions occur, it will be necessary to check the register after every movement of the sprites you are concerned about. Reading the register will cause it to be cleared automatically. You can not prevent that from happening, so if you want the information for future reference you must store it someplace where you can get at it.

Sprites can also collide with the background (text, etc). Like sprite-sprite collision, sprite-data collisions are kept in a register (53279 $D01F). Each sprite has its own bit (0-7) and indicates that that sprite has been involved in a collision. This register is also

cleared by a read of the register (LDA, LDX, LDY instruction), so save it if you don't want to lose it.

Once again, its very strongly recommended you study the PRG for all areas of graphic programming. The insights which may be gained from reading and understanding the BASIC programmer's perspective on these subjects will greatly increase your understanding and ability to move into assembly language.

The 6581 — A Sound synthesizer

The Commodore 64 comes equipped with a very respectable single-chip sound synthesizer. It has three voices, each with independent attack-decay-sustain-release (ASDR) envelopes, four kinds of filters, a resonance control and a master volume control. The programmer can make the computer generate a wide variety of musical and other sounds, simulating various instruments solo and in concert. The sound output may be played through the speaker on your TV or directed to your stereo for high-fidelity output.

The music and associated wave shape theory which is required to accomplish these ends is considerable. This chapter will present you with the necessary technical information on the requirements for programming the synthesizer. It will not go deeply into the theory of synthesized sound. The PRG is a source of more detailed information on the subject.

Register Assignment

The synthesizer chip is called the 6581 or "SID" (sound interface device) chip. The chip should be initialized before attempting any sound generation. It has a set of 29 registers which are directly addressable by the 6510 and therefore any program running on it. Initializing may be accomplished by setting the registers to zero:

```
SID   EQU $D400
      LDX #$18
      LDA #0
LOOP  STA SID,X
      DEX
      BPL LOOP
```

The register set is a contiguous string of bytes starting at $D400 (54272) and running through $D41C (54230). This block of registers is in the I/O block at $D000-$DFFF which also contains the VIC-II chip and which may be switched with the character ROM as explained in the previous chapter.

Tone Generation

The SID chip contains three tone generators, their associated envelope generators and the filtering and volume controls. There are seven registers for each of the three voices. The first voice's registers are at $D400-D406 (54272-54278). The second voice is at $D407-$D40D (54279-54285) and the third voice is at $D40E-D414 (54286-54292).

The individual voices may have frequencies ranging from 0-3894 hz. To cause any given frequency to be generated, it is necessary to store a 16-bit number in the first two registers for the voice desired. The value of the number to store is 16.777 times the frequency desired. In Appendix O of the PRG there is a description of the design criteria for a routine to select any of the 12 semi-tones of any of eight octaves. The following program segment will accomplish those ends:

```
LOW     EQU $D400   VOICE 1 FREQ LOW
HI      EQU $D401           FREQ HI
NOTES   EQU $C000
C7      BYT 34334
C#7     BYT 36376
D7      BYT 38539
D#7     BYT 40830
E7      BYT 43258
F7      BYT 45830
F#7     BYT 48556
G7      BYT 51443
G#7     BYT 54502
A7      BYT 57743
A#7     BYT 61176
B7      BYT 64814
;
SELCT   EQU @
; REG A MUST HAVE NOTE IN LOW 4 BITS (0-11)
;       OCTAVE IN HIGH 4 BITS    (0-7)
:
        TAX       SAVE FOR OCTAVE
        AND #$0F  JUST THE NOTE
        ASL A     NOTE * 2
```

```
        TAY       INDEX INTO NOTE TABLE
        LDA NOTE+1,Y
        STA HI
        LDA NOTE,Y
        STA LO
        TXA       RETRIEVE NOTE/OCT
        AND #$F0  JUST THE OCTAVE
        TAX
BUMP    ADC #$10
        BMI OUT
SHIFT   ROR HI      DIVIDE FREQ BY TWO
        ROR LOW     FOR EACH OCTAVE
        CLC
        BCC BUMP
OUT     RTS           ALL DONE
```

Wave Shape regulation

Each voice may have its wave shape independently set. The wave shape affects the timbre of the generated tone. The four possible waveforms are triangle (flute type sound), sawtooth (brass type sound), variable pulse (wide variety of possibilities), and white noise. The shape information is set in bits 7-4 of the fourth byte of each voice's register set. More than one wave shape may be selected per voice but the resulting waveform will not be generally useful. The following routine will set the wave shape.

```
SID   EQU $D400
VOICE EQU 0           (OR 1 OR 2)
      SEC
      LDA #0
      ROR A           ONCE FOR NOISE
      ROR A           TWICE FOR PULSE
      ROR A           THRICE FOR SAWTOOTH
      ROR A           FOUR TIMES FOR TRIANGLE
      STA VOICE*7+4+SID
```

Pulse width must be specified if pulse waveform is selected. The pulse width is set by specifying the percentage of time the pulse is on versus the time it is off. Fifty percent creates a square wave. 0% and 100% cause a constant signal output which is therefore not an audio tone. The width is set as a 12-bit value which is related to percent on-time by the following formula: WIDTH = TIME% * 40.95 The low-order eight bits are set in the third register of the voice's set. The high-order 4 bits of WIDTH are set in the low four bits of the fourth register.

```
WIDTH  EQU PCTON*40.95
       LDA #WIDTH/256
       STA VOICE*7+SID+3
       LDA #WIDTH&255
       STA VOICE+7+SID+2
```

Each voice has a programmable envelope generator which allows you to program the volume of the output signal in several phases. The attack phase is when the note begins and increases in volume to its maximum. The decay phase follows the reaching of maximum volume and continues as the volume decreases till it reaches the sustain phase. Here the note maintains a constant volume until the release phase, when the volume decreases back to zero. The envelope generator allows you to program the rate of volume increase in the attack phase, the rate of decrease in the decay phase, the level at which it sustains and the rate of final decrease during the release phase.

The note is started by turning on the start bit (called the gate signal) in bit-0 of the fifth register of the voice:

```
       LDA #1
       ORA VOICE*7+SID+4
       STA VOICE*7+SID+4
```

The attack rate is set in the sixth register, the high-order four bits (7-3). The value of the four-bit field corresponds to the time it takes for the tone to reach maximum volume as indicated in the following table.

The decay rate is set in the low-order four bits of the same register as the attack. The corresponding table values (in seconds) are the times for the volume to decrease to the sustain level.

The sustain level is a four-bit value in the high-order nybble of the seventh register of each voice. The sustain value (0-15) controls the relative volume of the generated tone at which decay stops and the note is held until released. The volume at which sustain occurs is in ratio to the peak volume as the sustain register is to 15. Note that the peak volume is the same for all voices and is under control of the master volume control. The release rate is set in the low-order nybble of the seventh register. It specifies the time it will take the note to complete its decay once sustain has been terminated.

The note is turned on (gated) by the gate bit which is bit-0, the low-order bit of the sixth register. When this bit is 1 the note will commence. It will go through the attack, decay and sustain phase and hold there until the bit is set to 0.

The following table gives the value of the various four-bit registers which control the envelope generators and the time in seconds between commencement and cessation of each phase.

VALUE	ATTACK	DECAY/REL	VALUE	ATTACK	DECAY/REL
0	.002	.006	8	.1	.3
1	.008	.024	9	.25	.75
2	.016	.048	10	.5	1.5
3	.024	.072	11	.8	2.4
4	.038	.114	12	1.0	3.0
5	.056	.168	13	3.0	9.0
6	.068	.204	14	5.0	15.0
7	.08	.24	15	8.0	24.0

Filtering

There are four kinds of signal filtering plus resonance control which may be employed. Individual voices may be independently connected and disconnected to the filter. The filter has an 11-bit register which is the frequency cut-off point and ranges from 30 to 12Khz. The low-pass filter will reject all frequencies above the cut-off frequency. The high-pass will reject all frequencies below the cut-off. The band pass filter will allow only the selected frequency and the notch-reject will pass all frequencies except the specified

frequency. Finally, the resonance control has a range of 16 values (0-15) which regulate how much resonance will be present on the output signal (0 = no resonance, 15 = maximum resonance).

The resonance control register is in bits 4-7 of location $D017 (54296). Bits 0-3 control which voices (1-3 plus external input) will be routed through the filter. Bits 3-0 of $D418 controls the peak volume. Bits 6-4 control which filter effect is active (bit 6 = high-pass, bit 5 = band-pass, bit 4 = low-pass, bits 6+4 = notch reject).

The following routine could be used to select the filtering:

```
        LDA SID+24
        AND #$8F
        ORA #TYPE    (16=LOW, 64=HIGH, 32=BAND, 80=NOTCH-REJ)
        STA SID+24
```

The following routine might be used to set the filter frequency:

```
FREQ    EQU 567      ANY VALUE BETWEEN 0 AND 2047
        LDA #FREQ/8
        STA SID+22
        LDA #FREQ&7
        STA SID+21
```

The relationship between FREQ and the actual frequency of the filter is supposedly linear from 30hz to 12000 hz. Experimentation has not born this out, however, and you are encouraged to do your own testing of the filter controls.

Mixing

It is possible to mix an external signal with the generated output signal. One of the pins on the audio output port may be used for mike or other instrument signal input to the 64. See the PRG for details.

A machine language approach to paddle and joystick programming is presented in the PRG and so we will not attempt to duplicate that information here.

64 Internals

The Commodore 64 comes with 16K of ROM in which the internal operating programs reside. These are the programs which interpret BASIC programs and which control the input and output devices which come with the 64 and those which can be added. This block of memory is broken into two segments. The first, the BASIC interpreter, extends from $A000 TO $BFFF. The second, called the "Kernal", extends from $E000 to $FFFF. Some of the Kernal programs have been documented by Commodore in their Programmer's Reference Guide. They deal mainly with input/output (I/O) processing on the 64. This text will not attempt to duplicate the information provided in the PRG. It is once again strongly recommended that you obtain a copy of that reference work.

There are many other subroutines included in the 64's ROM which are not covered in the PRG. We will attempt to provide information on using the more useful of those. We will also present a list of the entry points of the remainder. Those which are not discussed in detail may be decoded with the Decoder for your inspection and understanding.

Floating Point numbers

As BASIC processes your arithmetic expresions, it uses a variety of machine language subroutines to do addition, subtraction, exponentiation, trig functions, etc. These subroutines are available to the machine language program for accomplishing the same functions. They are all fairly similiar in the conventions of their use, i.e. the means of passing parameters, getting the results, etc.

Numbers in BASIC may be expressed as either integers or as "floating point" numbers. Integers have no fractional component. They are sixteen bit signed numbers which may have the range of - 32767 to 32768. Negative numbers are expressed in two's complement notation as discussed in chapter four.

BASIC does all its computations in floating point mode. Floating point numbers have fractional components. They are composed of three portions, the exponent, the mantissa and the sign. The exponent occupies one byte and its binary value is 128 greater than the exponent being expressed. The value of the expressed exponent is the number of bits which the mantissa needs to be shifted. Since the

exponent expression is stored in "excess 128", the range of actual exponents is -128 to 127 (stored as 0 to 255). Negative exponents mean the mantissa (as stored) needs to be shifted to the right and positive exponents means the mantissa needs to be shifted to the left.

The mantissa is a four byte binary value. It is the shifted value of the number to be expressed. This is like scientific notation as used in physics and chemistry. The decimal system of scientific notation would express the number 12876548.765 as 1.2876548765 * 10 raised to the 7th power. In BASIC you would see this number printed as 1.2876548765 E07. A decimal exponent of 7 in scientific notation means the decimal point needs to be shifted 7 places to the right. Or that the mantissa needs to be multiplied by 10 raised to the seventh power.

It works the same way with floating point numbers except the mantissa is in binary and it needs to be multiplied by 2 raised to the power of the exponent. Multiplying a binary number by two is the same as shifting it one bit position. e.g. 6 = 0000 0110 and 12 = 0000 1100.

CBM BASIC always normalizes the mantissa before saving it in the floating point format. This means that it shifts it so the leftmost bit is always a one bit. The number 6 (binary value = 0000 0110) would be shifted so that the normalized mantissa would be 1100 0000. What goes into the exponent field is 128 plus the number of significant bit positions in the original number. 0000 0110 has three significant bit positions, so the exponent would be 131.

A few examples will be helpful. The binary representation of the floating point storage of the number 6 is:

```
1000 0011 1100 0000   0000 0000  0000 0000   0000 0000
   131       192         0           0           0
   $83       $C0        $00         $00         $00
---------  ------------------------------------------
exponent                    mantissa
```

The exponent of 131 represents an actual exponent of 3 (131 - 128). You may consider the mantissa as a fraction with the radix point (decimal point or, rather, binary point) just to its left. The amount it must be shifted to get back to its actual value is three bit positions. In other words, the radix point must be shifted from

the left of the binary number three places to the right.

Other examples:

The number 1 (0000 0001)

```
1000 0001 1000 0000   0000 0000  0000 0000   0000 0000
   129        128          0           0           0
   $81        $80         $00         $00         $00
---------  ------------------------------------------
exponent                    mantissa
```

The number 2 (0000 0010)

```
1000 0010 1000 0000   0000 0000  0000 0000   0000 0000
   129        128          0           0           0
   $82        $80         $00         $00         $00
---------  ------------------------------------------
exponent                    mantissa
```

The number 3 (0000 0011)

```
1000 0010 1100 0000   0000 0000  0000 0000   0000 0000
   129        192          0           0           0
   $82        $C0         $00         $00         $00
---------  ------------------------------------------
exponent                    mantissa
```

The number 65 (0100 0001)

```
1000 0111 1000 0010   0000 0000  0000 0000   0000 0000
   135        130          0           0           0
   $87        $82         $00         $00         $00
---------  ------------------------------------------
exponent                    mantissa
```

The number 15 (0000 1111)

1000 0100	1111 0000	0000 0000	0000 0000	0000 0000
132	240	0	0	0
$84	$F0	$00	$00	$00
exponent	mantissa			

The sign of the number is carried in the high-order bit of the byte following the mantissa. If the sign bit is on, the number is negative. If off it is positive.

There are two floating point accumulators maintained by BASIC, FAC1 and FAC2. FAC1 is located at $61-$66 (97-103) and FAC2 is at $69-$6E (105-111). These two accumulators are used for all mathematical operations. Following the FACs, at $6F (111), is a sign comparison flag. The high-order bit, if on, signifies the two FAC's are of differing signs.

Arithmetic routines

The following routines perform mathematical operations using FAC1, FAC2, and values stored in other memory locations. Each routine may be executed by a JSR instruction to the indicated entry point. References to memory locations are frequently communicated to various routines by an address contained in the A-reg (LSB) and the Y-reg (MSB). We will refer to this format as format-1.

Most of the following routines use the A-reg, X-reg and Y-reg for communication. It is an interesting fact that when a SYS is done from BASIC, these three registers are loaded from memory locations 780, 781 and 782 respectively. Additionally, the Processor Status register is passed in 783. All of the registers are stored back in these same locations upon returning to BASIC so it is therefore possible to easily pass information back and forth between BASIC and machine language programs. You can call the following routines and those in the Kernal from BASIC by SYSing to them after setting up the three register storage bytes.

Integer to FAC1 - $B391 (45969)

A two-byte integer value in format-1 is converted to a floating point number stored in FAC1.

FAC1 to Integer $B1AA (45482)

The FAC1 is converted to a two-byte integer which is saved into locations $64,$65 (100,101). The FAC1 is destroyed.

Memory to FAC1 $BBA2 (48034)

A five-byte floating point number anywhere in memory is loaded into FAC1. The address of the starting memory location is in format-1. The sign flag of FAC1 is set on if the high-order bit of the mantissa is a one, else it is set off. The exponent is returned in the A-reg.

ASCII to FAC1 $B7B5 (47029)

An ASCII string is converted to floating point format and saved in the FAC1. The string may be anywhere in memory and the address of the starting location must be pointed to by the utility string pointer at $22,$23 (34,35). The length of the string must be loaded into the A-reg.

FAC1 to ASCII $BDDD (48605)

The ASCII representation of the value in FAC1 will be saved starting at $0100 (256) and continuing until a $00 is encountered.

Memory to FAC2 $BA8C (47756)

Same as above except using FAC2 and the sign comparison flag is set. The exponent of FAC1 is returned in the A-reg.

FAC1 to Memory $BBD7 (48087)

The FAC1 is stored into any five byte memory location. The MSB of the address of the start of the memory location is passed in the X-reg. The LSB is in the Y-reg. The high-order bit of the mantissa field is forced to the FAC1 sign flag.

FAC2 to FAC1 $BBFC (48124)

A simple move is performed from FAC2 to FAC1. FAC2 is not affected.

FAC1 to FAC2 $BC0F (48143)

A simple move is performed from FAC1 to FAC2. FAC1 is not affected.

Logical AND of FAC1 and FAC2 $AFE9 (45033)

FAC1 and FAC2 are logically ANDed together, the result ending up in FAC1

Logical OR of FAC1 and FAC2 $AFE6 (45030)

FAC1 and FAC2 are logically ORed together, the result ending up in FAC1

FAC1 = FAC1 - FAC2 $B853 (47187)

FAC2 is subtracted from FAC1, the result replacing FAC1. FAC2 is not affected.

FAC1 = FAC1 + FAC2 $B86A (47210)

FAC1 is replaced by the sum of FAC1 and FAC2. It is necessary to set the sign compare flag prior to calling this routine. This is done by EORing locations $66 and $6E (102 and 110) and storing the result in $6F. It is also necessary to load the A-reg with the value found in $61 (97). Note that both of these functions are done by the Mem to FAC2 routine.

FAC1 = FAC1 * FAC2 $BA30 (47664)

FAC1 is replaced by the product of FAC1 AND FAC2. The same notes apply as for the above routine. An alternate entry point for this routine is $BA28 (47656). This entry point will execute the memory to FAC2 routine before doing the multiplication.

FAC1 = LOG (FAC1) $B9EA (47954)

FAC1 is replaced by the LOG of FAC1.

FAC1 = FAC2 / FAC1 $BB12 (47890)

FAC1 is replaced by the qoutient of FAC2 and FAC1. The same notes apply as for addition. However by JSRing to $BB0F (47887) instead, the loading of the FAC2 from memory will be accomplished prior to doing the division.

FAC1 = FAC2 ^ FAC1 $BF7B (49019)

FAC1 is replaced with FAC2 raised to the power of FAC1. Same comments as for addition. By using the $BF78 (49016) entry point, the routine to load FAC1 from memory may be executed prior to the exponentiation routine. The Memory to FAC1 routine does not properly set the sign compare flag however. Also note that when using these alternate entry points, the same setup of the A-reg and Y-reg must be performed as per Mem-to-FAC routines before calling the desired arithmetic routine.

FAC1 = FAC1 / 10 $BAFE (47870)

FAC1 is replaced by FAC1 / 10.

Compare FAC1 and Memory $BC5B (48219)

The A-reg is set depending on the result of the compare between FAC1 and some floating point number in a specified memory location. If they are equal, the result is 0; if they are not equal the result is $FF (255). The address of the start of the memory location is in format-1.

FAC1 = ABS (FAC1) $BC58 (48216)

The FAC1 is replaced by the absolute value of FAC1.

FAC1 = INT (FAC1) $BCCC (48332)

The FAC1 is replaced by the integer portion of FAC1.

FAC1 = SGN (FAC1) $BC39 (48185)

The FAC1 is replaced by the value 0 if it was a zero, by 1 if it was greater than zero and by -1 if it was less than zero.

FAC1 = SQR (FAC1) $BF71 (49009)

The FAC1 is replaced by the square root of FAC1.

FAC1 = EXP (FAC1) $BFED (49133)

The FAC1 is replaced by the value computed by raising e of natural logarithm to the power of FAC1.

FAC1 = COS (FAC1) $E264 (57956)

The FAC1 is replaced by the Cosine of FAC1 expressed in radians.

FAC1 = SIN (FAC1) $E26B (57963)
The FAC1 is replaced by the Sine of FAC1 expressed in radians.

FAC1 = TAN (FAC1) $E2B7 (58039)
The FAC1 is replaced by the tangent of FAC1 expressed in radians.

FAC1 = ATN (FAC1) $E30D (58125)
The FAC1 is replaced by the arctangent of FAC1 expressed in radians.

Input/Output routines

Most of the I/O routines are presented in the PRG but there are two more presented here which do not appear there.

Input into BASIC buffer $A560 (42336)
The 88 byte BASIC input buffer starting at $0200 (512) is filled with characters from the Keyboard. A [Return] terminates the input and a $00 signifies the end of the message in the buffer.

Output string to screen $AB1E (43806)
The starting address of a string of ASCII characters to be printed on the screen is set in format-1. The string must be terminated by a $00.

Appendix A

Mode	1	2	3	4	5	6	7	8	9	10	11	12	13
ADC		x	x	x		x	x	x			x	x	
AND		x	x	x		x	x	x			x	x	
ASL	x		x	x		x	x						
BCC										x			
BCS										x			
BEQ										x			
BIT			x			x							
BMI										x			
BNE										x			
BPL										x			
BRK									x				
BVC										x			
BVS										x			
CLC									x				
CLD									x				
CLI									x				
CLV									x				
CMP		x	x	x		x	x	x			x	x	
CPX		x	x			x							
CPY		x	x			x							
DEC			x	x		x	x						
DEX									x				
DEY									x				
EOR		x	x	x		x	x	x			x	x	
INC			x	x		x	x						
INX									x				
INY									x				
JMP						x							x
JSR						x							
LDA		x	x	x		x	x	x			x	x	
LDX		x	x		x	x		x					
LDY		x	x	x		x	x						
LSR	x		x	x		x	x						
NOP										x			

Mode	1	2	3	4	5	6	7	8	9	10	11	12	13
ORA		x	x	x		x	x	x			x	x	
PHA									x				
PHP									x				
PLA									x				
PLP									x				
ROL	x		x	x		x	x						
ROR	x		x	x		x	x						
RTI									x				
RTS									x				
SBC		x	x	x		x	x	x			x	x	
SEC									x				
SED									x				
SEI									x				
STA			x	x		x	x	x			x	x	
STX			x		x	x							
STY			x	x		x							
TAX									x				
TAY									x				
TSX									x				
TXA									x				
TXS									x				
TYA									x				

Modes:

1 - Accumulator
2 - Immediate
3 - Zero page
4 - Zero page,X
5 - Zero page,Y
6 - Absolute
7 - Absolute,X
8 - Absolute,Y
9 - Implied
10 - Relative
11 - (Indirect,X)
12 - (Indirect),Y
13 - (Indirect)

Appendix B

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL	BINARY	HEX
0	00		@	end-line	BRK	0	0000 0000	00
1	01		A		ORA(I,X)	1	0000 0001	01
2	02		B			2	0000 0010	02
3	03		C			3	0000 0011	03
4	04		D			4	0000 0100	04
5	05		E		ORA Z	5	0000 0101	05
6	06		F		ASL Z	6	0000 0110	06
7	07		G			7	0000 0111	07
8	08		H		PHP	8	0000 1000	08
9	09		I		ORA #	9	0000 1001	09
10	0A		J		ASL A	10	0000 1010	0A
11	0B		K			11	0000 1011	0B
12	0C		L			12	0000 1100	0C
13	0D	car ret	M		ORA	13	0000 1101	0D
14	0E		N		ASL	14	0000 1110	0E
15	0F		O			15	0000 1111	0F
16	10		P		BPL	16	0001 0000	10
17	11	cur down	Q		ORA(I),Y	17	0001 0001	11
18	12	reverse	R			18	0001 0010	12
19	13	cur home	S			19	0001 0011	13
20	14	delete	T			20	0001 0100	14
21	15		U		ORA Z,X	21	0001 0101	15
22	16		V		ASL Z,X	22	0001 0110	16
23	17		W			23	0001 0111	17
24	18		X		CLC	24	0001 1000	18
25	19		Y		ORA Y	25	0001 1001	19
26	1A		Z			26	0001 1010	1A
27	1B		[			27	0001 1011	1B
28	1C		\			28	0001 1100	1C
29	1D	cur right	]		ORA X	29	0001 1101	1D
30	1E		↑		ASL X	30	0001 1110	1E
31	1F		←			31	0001 1111	1F

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL	BINARY	HEX
32	20	space	space	space	JSR	32	0010 0000	20
33	21	!	!	!	AND(I,X)	33	0010 0001	21
34	22	"	"	"		34	0010 0010	22
35	23	#	#	#		35	0010 0011	23
36	24	$	$	$	BIT Z	36	0010 0100	24
37	25	%	%	%	AND Z	37	0010 0101	25
38	26	&	&	&	ROL Z	38	0010 0110	26
39	27	'	'	'		39	0010 0111	27
40	28	(	(	(	PLP	40	0010 1000	28
41	29	)	)	)	AND #	41	0010 1001	29
42	2A	*	*	*	ROL A	42	0010 1010	2A
43	2B	+	+	+		43	0010 1011	2B
44	2C	,	,	,	BIT	44	0010 1100	2C
45	2D	-	-	-	AND	45	0010 1101	2D
46	2E	.	.	.	ROL	46	0010 1110	2E
47	2F	/	/	/		47	0010 1111	2F
48	30	0	0	0	BMI	48	0011 0000	30
49	31	1	1	1	AND(I),Y	49	0011 0001	31
50	32	2	2	2		50	0011 0010	32
51	33	3	3	3		51	0011 0011	33
52	34	4	4	4		52	0011 0100	34
53	35	5	5	5	AND Z,X	53	0011 0101	35
54	36	6	6	6	ROL Z,X	54	0011 0110	36
55	37	7	7	7		55	0011 0111	37
56	38	8	8	8	SEC	56	0011 1000	38
57	39	9	9	9	AND Y	57	0011 1001	39
58	3A	:	:	:	CLI	58	0011 1010	3A
59	3B	;	;	;		59	0011 1011	3B
60	3C	<	<	<		60	0011 1100	3C
61	3D	=	=	=	AND X	61	0011 1101	3D
62	3E	>	>	>	ROL X	62	0011 1110	3E
63	3F	?	?	?		63	0011 1111	3F

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL	BINARY	HEX
64	40				RTI	64	0100 0000	40
65	41	A		A	EOR(I,X)	65	0100 0001	41
66	42	B		B		66	0100 0010	42
67	43	C		C		67	0100 0011	43
68	44	D		D		68	0100 0100	44
69	45	E		E	EOR Z	69	0100 0101	45
70	46	F		F	LSR Z	70	0100 0110	46
71	47	G		G		71	0100 0111	47
72	48	H		H	PHA	72	0100 1000	48
73	49	I		I	EOR #	73	0100 1001	49
74	4A	J		J	LSR A	74	0100 1010	4A
75	4B	K		K		75	0100 1011	4B
76	4C	L		L	JMP	76	0100 1100	4C
77	4D	M		M	EOR	77	0100 1101	4D
78	4E	N		N	LSR	78	0100 1110	4E
79	4F	O		O		79	0100 1111	4F
80	50	P		P	BVC	80	0101 0000	50
81	51	Q		Q	EOR(I),Y	81	0101 0001	51
82	52	R		R		82	0101 0010	52
83	53	S		S		83	0101 0011	53
84	54	T		T		84	0101 0100	54
85	55	U		U	EOR Z,X	85	0101 0101	55
86	56	V		V	LSR Z,X	86	0101 0110	56
87	57	W		W		87	0101 0111	57
88	58	X		X	CLI	88	0101 1000	58
89	59	Y		Y	EOR Y	89	0101 1001	59
90	5A	Z		Z		90	0101 1010	5A
91	5B					91	0101 1011	5B
92	5C					92	0101 1100	5C
93	5D				EOR X	93	0101 1101	5D
94	5E				LSR X	94	0101 1110	5E
95	5F					95	0101 1111	5F

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL	BINARY	HEX
96	60				RTS	96	0110 0000	60
97	61				ADC(I,X)	97	0110 0001	61
98	62					98	0110 0010	62
99	63					99	0110 0011	63
100	64					100	0110 0100	64
101	65				ADC Z	101	0110 0101	65
102	66				RCR Z	102	0110 0110	66
103	67					103	0110 0111	67
104	68				PLA	104	0110 1000	68
105	69				ADC #	105	0110 1001	69
106	6A				RCR A	106	0110 1010	6A
107	6B					107	0110 1011	6B
108	6C				JMP(I)	108	0110 1100	6C
109	6D				ADC	109	0110 1101	6D
110	6E				ROR	110	0110 1110	6E
111	6F					111	0110 1111	6F
112	70				BVS	112	0111 0000	70
113	71				ADC(I),Y	113	0111 0001	71
114	72					114	0111 0010	72
115	73					115	0111 0011	73
116	74					116	0111 0100	74
117	75				ADC Z,X	117	0111 0101	75
118	76				ROR Z,X	118	0111 0110	76
119	77					119	0111 0111	77
120	78				SEI	120	0111 1000	78
121	79				ADC Y	121	0111 1001	79
122	7A					122	0111 1010	7A
123	7B					123	0111 1011	7B
124	7C					124	0111 1100	7C
125	7D				ADC X	125	0111 1101	7D
126	7E				ROR X	126	0111 1110	7E
127	7F					127	0111 1111	7F

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL	BINARY	HEX
128	80		r-O	END		128	1000 0000	80
129	81		r-A	FOR	STA(I,X)	129	1000 0001	81
130	82		r-B	NEXT		130	1000 0010	82
131	83		r-C	DATA		131	1000 0011	83
132	84		r-D	INPUT #	STY Z	132	1000 0100	84
133	85		r-E	INPUT	STA Z	133	1000 0101	85
134	86		r-F	DIM	STX Z	134	1000 0110	86
135	87		r-G	READ		135	1000 0111	87
136	88		r-H	LET	DEY	136	1000 1000	88
137	89		r-I	GOTO		137	1000 1001	89
138	8A		r-J	RUN	TXA	138	1000 1010	8A
139	8B		r-K	IF		139	1000 1011	8B
140	8C		r-L	RESTORE	STY	140	1000 1100	8C
141	8D		r-M	GOSUB	STA	141	1000 1101	8D
142	8E		r-N	RETURN	STX	142	1000 1110	8E
143	8F		r-O	REM		143	1000 1111	8F
144	90		r-P	STOP	BCC	144	1001 0000	90
145	91	cur up	r-Q	ON	STA (I),Y	145	1001 0001	91
146	92	rvs off	r-R	WAIT		146	1001 0010	92
147	93	clear	r-S	LOAD		147	1001 0011	93
148	94	insert	r-T	SAVE	STY Z,X	148	1001 0100	94
149	95		r-U	VERIFY	STA Z,X	149	1001 0101	95
150	96		r-V	DEF	STX Z,Y	150	1001 0110	96
151	97		r-W	POKE		151	1001 0111	97
152	98		r-X	PRINT #	TYA	152	1001 1000	98
153	99		r-Y	PRINT	STA Y	153	1001 1001	99
154	9A		r-Z	CONT	TXS	154	1001 1010	9A
155	9B		r-[	LIST		155	1001 1011	9B
156	9C		r-\	CLR		156	1001 1100	9C
157	9D	cur left	r-]	CMD	STA X	157	1001 1101	9D
158	9E		r-↑	SYS		158	1001 1110	9E
159	9F		r-←	OPEN		159	1001 1111	9F

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL	BINARY	HEX
160	A0			CLOSE	LDY #	160	1010 0000	A0
161	A1		r-!	GET	LDA(I,X)	161	1010 0001	A1
162	A2		r-"	NEW	LDX #	162	1010 0010	A2
163	A3		r-#	TAB(		163	1010 0011	A3
164	A4		r-$	TO	LDY Z	164	1010 0100	A4
165	A5		r-%	FN	LDA Z	165	1010 0101	A5
166	A6		r-&	SPC(	LDX Z	166	1010 0110	A6
167	A7		r-'	THEN		167	1010 0111	A7
168	A8		r-(	NOT	TAY	168	1010 1000	A8
169	A9		r-)	STEP	LDA #	169	1010 1001	A9
170	AA		r-*	+	TAX	170	1010 1010	AA
171	AB		r-+	-		171	1010 1011	AB
172	AC		r-,	*	LDY	172	1010 1100	AC
173	AD		r--	/	LDA	173	1010 1101	AD
174	AE		r-.		LDX	174	1010 1110	AE
175	AF		r-/	AND		175	1010 1111	AF
176	B0		r-0	OR	BCS	176	1011 0000	B0
177	B1		r-1		LDA(I),Y	177	1011 0001	B1
178	B2		r-2	=		178	1011 0010	B2
179	B3		r-3			179	1011 0011	B3
180	B4		r-4	SGN	LDY Z,X	180	1011 0100	B4
181	B5		r-5	INT	LDA Z,X	181	1011 0101	B5
182	B6		r-6	ABS	LDX Z,Y	182	1011 0110	B6
183	B7		r-7	USR		183	1011 0111	B7
184	B8		r-8	FRE	CLV	184	1011 1000	B8
185	B9		r-9	POS	LDA Y	185	1011 1001	B9
186	BA		r-:	SQR	TSX	186	1011 1010	BA
187	BB		r-;	RND		187	1011 1011	BB
188	BC		r-	LOG	LDY X	188	1011 1100	BC
189	BD		r-=	EXP	LDA X	189	1011 1101	BD
190	BE		r-	COS	LDX Y	190	1011 1110	BE
191	BF		r-?	SIN		191	1011 1111	BF

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL	BINARY	HEX
192	C0			TAN	CPY #	192	1100 0000	C0
193	C1			ATN	CMP (I,X)	193	1100 0001	C1
194	C2			PEEK		194	1100 0010	C2
195	C3			LEN		195	1100 0011	C3
196	C4			STR$	CPY Z	196	1100 0100	C4
197	C5			VAL	CMP Z	197	1100 0101	C5
198	C6			ASC	DEC Z	198	1100 0110	C6
199	C7			CHR$		199	1100 0111	C7
200	C8			LEFT$	INY	200	1100 1000	C8
201	C9			RIGHT$	CMP #	201	1100 1001	C9
202	CA			MID$	DEX	202	1100 1010	CA
203	CB					203	1100 1011	CB
204	CC				CYP	204	1100 1100	CC
205	CD				CMP	205	1100 1101	CD
206	CE				DEC	206	1100 1110	CE
207	CF					207	1100 1111	CF
208	D0				BNE	208	1101 0000	D0
209	D1				CMP(I),Y	209	1101 0001	D1
210	D2					210	1101 0010	D2
211	D3					211	1101 0011	D3
212	D4					212	1101 0100	D4
213	D5				CMP Z,X	213	1101 0101	D5
214	D6				DEC Z,X	214	1101 0110	D6
215	D7					215	1101 0111	D7
216	D8				CLD	216	1101 1000	D8
217	D9				CMP Y	217	1101 1001	D9
218	DA					218	1101 1010	DA
219	DB					219	1101 1011	DB
220	DC					220	1101 1100	DC
221	DD					221	1101 1101	DD
222	DE				DEC X	222	1101 1110	DE
223	DF					223	1101 1111	DF

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL	BINARY	HEX
224	E0				CPX #	224	1110 0000	E0
225	E1				SBC (I,X)	225	1110 0001	E1
226	E2					226	1110 0010	E2
227	E3					227	1110 0011	E3
228	E4				CPX Z	228	1110 0100	E4
229	E5				SBC Z	229	1110 0101	E5
230	E6				INC Z	230	1110 0110	E6
231	E7					231	1110 0111	E7
232	E8				INX	232	1110 1000	E8
233	E9				SBC #	233	1110 1001	E9
234	EA				NOP	234	1110 1010	EA
235	EB					235	1110 1011	EB
236	EC				CPX	236	1110 1100	EC
237	ED				SBC	237	1110 1101	ED
238	EE				INC	238	1110 1110	EE
239	EF					239	1110 1111	EF
240	F0				BEQ	240	1111 0000	F0
241	F1				SBC (I),Y	241	1111 0001	F1
242	F2					242	1111 0010	F2
243	F3					243	1111 0011	F3
244	F4					244	1111 0100	F4
245	F5				SBC Z,X	245	1111 0101	F5
246	F6				INC Z,X	246	1111 0110	F6
247	F7					247	1111 0111	F7
248	F8				SED	248	1111 1000	F8
249	F9				SBC Y	249	1111 1001	F9
250	FA					250	1111 1010	FA
251	FB					251	1111 1011	FB
252	FC					252	1111 1100	FC
253	FD				SBC X	253	1111 1101	FD
254	FE				INC X	254	1111 1110	FE
255	FF					255	1111 1111	FF

Auto-Start Cartridges

The Commodore 64 has a feature which allows a program in ROM starting at $8000 (32768) to sieze control of the machine at power-up and RESET times without any further intervention required on the part of the operator.

At power-up time one of the very first things the operating system does is check for a five character sequence starting at location $8004. If it finds $C3,$C2,$CD,$38,$30 it will automatically jump to the address it found at $8000,$8001. A second address is stored in $8002,$8003. This is the address of the [RESTORE] key processing routine.

To create your own cartridges you must encode the first 9 bytes as described above. It will be necessary to burn a PROM (programmable read only memory) and install it on an appropriate card for complete auto-start capability. The Programmer's Reference Guide has complete definition of the card connector specifications. Prom programmers are available from a variety of sources. Check the magazine ads.

Commodore 64 Memory Map

Hex	Decimal	Function
0000	0	On-chip data-direction register
0001	1	On-chip I/O Port
0002	2	Unused
0003	3-4	Float->integer
0005	5-6	Integer->float
0007	7	Search char ":" or endline
0008	8	Scan btwn quotes flag - 00 as delimeter
0009	9	Column pos of last Tab
000A	10	Verify flag (0=Load/1=Verify)
000B	11	Basic input buffer pointer/#subscripts
000C	12	DIM flag
000D	13	Variable flag - type:FF=string - 00=numeric
000E	14	Integer flag :80=integer - 00=floating pt
000F	15	DATA scan flag/LIST quote flag/memory flag
0010	16	Subscript flag;FNx flag
0011	17	input / read (0=input - 64=get - 152=read
0012	18	ATN sign flag:comparison evaluation flag
0013	19	Current I/O device for prompt suppress
0014	20-21	Basic integer adr.(for SYS - GOTO etc)
0016	22	Temporary string descriptor stack pointer
0017	23-24	Last temporary string vector
0019	25-33	Stack of descriptors for temporary strings
0022	34-35	Pointer for number transfer
0024	36-37	Misc number pointer
0026	38-42	Prodct area for mult
002B	43-44	Pointer to start of Basic
002D	45-46	Pointer to end of prog.start of variables
002F	47-48	Pointer to end of variables start of arrays
0031	49-50	Pointer to end of arrays
0033	51-52	Pointer to bottom of strng spce(coming dwn)
0035	53-54	Pointer to top of active strings
0037	55-56	Pointer to end of memory
0039	57-58	Current Basic line number
003B	59-60	Prev BASIC line num
003D	61-62	Previous BASIC statement (for CONT)

Hex	Decimal	Function
003F	63-64	Line number - current DATA line
0041	65-66	Pointer to current DATA item
0043	67-68	Input vector
0045	69-70	Current variable name
0047	71-72	Current variable address
0049	73-74	Variable pointer for FOR/NEXT
004B	75-76	Y save/new op save/curr op pointer
004D	77	Special mask for curr oprtr;Comparison symbol
004E	78-79	Misc. work area;function def pointer hi-lo
0050	80-81	Work area;pointer to strng descrptn
0052	82	Length of above string
0053	83	Constant used by garbage collect - 3 or 7
0054	84-86	Jump vector for functions
0057	87-96	Misc. numerical storage area
0061	97-102	FAC#1
0067	103	Series evaluation constant pointer
0068	104	FAC#1 high ord propogation
0069	105-110	Accumulator #2
006F	111	Sign comparison - FAC1 vs FAC2
0070	112	Low order rounding byte for Acc#1
0071	113-114	Cassette buffer length/series pointer
0073	115-138	Subrtn:Get Basic char;7A - 7B=pointer(CHARGOT)
008B	139-143	RND storage and work area
0090	144	Status ST
0091	145	Stop/RVS key flag
0092	146	Timing constant for tape
0093	147	Load or verify flag L=0/V=1
0094	148	Serial output/deferred char flag
0095	149	Serial deferred character
0096	150	Tape EOT recvd
0097	151	Register save area
0098	152	# open files
0099	153	Input device# - normally 0
009A	154	Output CMD device - normally 3
009B	155	Tape character parity
009C	156	Cassette dipole switch
009D	157	OS message flag - direct=$80 - run=0
009E	158	Cassette error pass 1

Hex	Decimal	Function
009F	159	Cassette error pass 2
00A0	160-162	Jiffy clock (HML)
00A3	163	Serial bit count
00A4	164	Cycle counter for serial I/O
00A5	165	Cntdwn for tape write
00A6	166	Cassette buffer pointer
00A7	167	RS-232 input bit storage/Tape shrtcnt
00A8	168	RS-232 bit cnt in/ Tape read error
00A9	169	RS-232 flag start bit ck/Tape rd bit err
00AA	170	RS-232 byte buffer/Tape rd mode
00AB	171	RS-232 parity storage/Tape chksum
00AC	172-173	Tape start addr/tape buffer / scrolling
00AE	174-175	Tape end addr/end of current program
00B0	176-177	Tape timing constants
00B2	178-179	Addr of tape buffer
00B4	180	RS-232 transmitter bit cnt out
00B5	181	RS-232 transmitter nxt bit to be sent
00B6	182	RS-232 transmitter byte buffer
00B7	183	Length of current file name string
00B8	184	Current logical file number
00B9	185	Curr secondary addr - or R/W command
00BA	186	Curr device number
00BB	187-188	Addr of curr file name string
00BD	189	RS-232 write shift word/Receive input char
00BE	190	#blocks remaining to read/write
00BF	191	Serial word buffer
00C0	192	Cass motor interlock
00C1	193-194	Tape start addr(load)
00C3	195-196	KERNAL setup pointer
00C5	197	Matrix co-ordinates of key pressed
00C6	198	#of characters in keybrd buffer
00C7	199	Reverse mode flag - 0=off - 18=on
00C8	200	End of line for input pointer
00C9	201-202	Cursor log(row - column)
00CB	203	Print shifted Chars flag
00CC	204	Cursor blink enabled flag - 0=on - 1=off
00CD	205	Delay before cursor blinks
00CE	206	Character under cursor

Hex	Decimal	Function
00CF	207	Cursor on/off blink flag
00D0	208	Input from screen/keybrd
00D1	209-210	Screen addr(row)pointer(screen memory)
00D3	211	Position of cursor on curr line
00D4	212	Quote mode flag (0=off / 1=on)
00D5	213	Line length for screen
00D6	214	Current screen line number
00D7	215	ASCII value of last key press
00D8	216	# of inserts outstanding
00D9	217-242	Screen line link table
00F2	242	Screen row marker
00F3	243-244	Screen color ptr
00F5	245-246	Keyscan table indirect
00F7	247-248	Pointer to RS-232 receive buffer addr
00F9	249-250	Pointer to RS-232 transmitter buffer addr
00FB	251-254	Free zero page locations
00FF	255	BASIC storage
0100	256-266	Floating to ASCII work area
0100	256-318	Tape error log
0100	256-511	Processor stack area
0200	512-600	Basic input buffer
0259	601-610	Logical file number table
0263	611-620	Device number table
026D	621-630	Secondary addr of R/W cmd - table
0277	631-640	Keyboard buffer
0281	641-642	Start of memory
0283	643-644	Top of memory
0285	645	Serial timeout flag
0286	646	Active color code
0287	647	Background color under cursor
0288	648	Top of Screen page
0289	649	Keyboard buffer max length
028A	650	Repeat flag - 0=cursor only - 128=all keys
028B	651	Delay before repeat occurs
028C	652	Delay btwn repeats
028D	653	Shift flag byte
028E	654	Last shift pattern
028F	655-656	Indirect for keyboard table setup

0291	657	Shift mode switch - 0=enabled - 128=locked
0292	658	Auto scroll dwn flag(0=on - <>0=off)
0293	659	6551 RS232 control register image
0294	660	6551 RS232 Command register image
0295	661-662	Non standard (bittime/2-100)
0297	663	6551 RS-232 status register image
0298	664	Number of bits to send
0299	665-666	Baud rate full bit time
029B	667	RS-232 end of receiver pointer
029C	668	RS-232 start receive buffer
029D	669	RS-232 start transmit output buf
029E	670	RS-232 end of transmit buffer
029F	671-672	Holds IRQ during tape operation
02A1	673-767	Program indirects
0300	768-769	Indirect error routine
0302	770-771	Indirect warm start
0304	772-773	Indirect tokenize BASIC
0306	774-775	Indirect token print
0308	776-777	Indirect new token
030A	778-779	Indirect symbol evaluation
030C	780	Temporary storage during SYS of A-reg
030D	781	Temporary storage during SYS of X-reg
030E	782	Temporary storage during SYS of Y-reg
030F	783	Temporary storage during SYS of P-reg
0314	788-789	IRQ vector
0316	790-791	BRK vector
0318	792-793	NMI vector
031A	794-795	Open logical file vector (OPEN)
031C	796-797	Close logical file vector (CLOSE)
031E	798-799	Set input device vector (CHKIN)
0320	800-801	Set output device vector (CHKOUT)
0322	802-803	Reset default I/O (CLRCHN)
0324	804-805	Input from device (CHRIN)
0326	806-807	Output to device vector (CHROUT)
0328	808-809	Test STOP key vector (STOP)
032A	810-811	Get from keyboard vector (GETIN)
032C	812-813	Close all files vector (CLALL)
032E	814-815	User defined vector
0330	816-817	Load from device vector (LOAD)
0332	818-819	Save to device vector (SAVE)

Hex	Decimal	Function
0334	820-827	Unused
033C	828-1019	Cassette buffer
03FC	1020-1023	Unused
0400	1024-2047	1024 byte screen memory area
0400	1024-2023	25 lines by 40 columns video matrix
07F8	2040-2047	Sprite data pointers
0800	2048-40959	Normal user Basic area
A000	40960-49151	BASIC ROM or 8K of RAM
A000	40960	Keyword action addresses
A046	41030	Function action addresses
A074	41076	Operator action addresses
A092	41106	Keyword Table
A193	41363	Error messages
A38A	41866	FOR - GOSUB search stack
A3B8	41912	Open memory space
A3FB	41979	Test stack depth
A408	41992	Check available memory
A435	42037	Send error message
A474	42100	Print READY.
A483	42110	New BASIC line processing
A533	42291	BASIC line chaining
A560	42336	Receive line from keyboard
A57C	42364	Tokenize BASIC line
A613	42515	Search for line number
A642	42562	Perform NEW
A660	42592	Perform CLR
A68E	42638	Reset BASIC execution to start-of-program
A69C	42652	Perform LIST
A742	42818	Perform FOR
A7ED	42989	Execute BASIC statement
A81D	43037	Perform RESTORE
A82C	43052	Perform STOP and END
A857	43095	Perform CONT
A871	43121	Perform RUN
A883	43139	Perform GOSUB
A8A0	43168	Perform GOTO
A8D2	43218	Perform RETURN
A8EB	43243	Perform DATA

Hex	Decimal	Function
A906	43270	Scan for next statement
A909	43273	Scan for next line
A928	43304	Perform IF
A93B	43323	Perform REM
A94B	43339	Perform ON
A96B	43371	Get integer from text
A9A5	43429	Perform LET
AA80	43648	Perform PRINT#
AA86	43654	Perform CMD
AA9A	43674	Perform PRINT
AB1E	43806	Print string from any memory
AB3B	43835	Print format character
AB4D	43853	Process bad input
AB7B	43899	Perform GET
ABA5	43941	Perform INPUT#
ABBF	43967	Perform INPUT
ABF9	44025	Prompt & input
AC06	44038	Perform READ
ACFC	44284	Input error messages
AD1E	44318	Perform NEXT
AD78	44408	Type match check
AD9E	44446	Evaluate expression
AEA8	44712	PI in floating point
AEF1	44785	Evaluate within parenthesis
AEF7	44791	Check for ")"
AEFA	44794	Check for "("
AEFD	44797	Check for " - "
AF08	44808	Syntax error
AF14	44820	Search for variable name
AFA7	44967	Set up FN references
AFE6	45030	Perform OR
AFE9	45033	Perform AND
B016	45078	Comparison routine
B07E	45182	Perform DIM
B08B	45195	Locate variable
B113	45343	Check for alpha ASCII
B11D	45341	Create new variable
B194	45460	Array pointer routine

Hex	Decimal	Function
B1A5	45477	32768 in floating point
B1BF	45503	FAC1 to integer
B1D1	45521	Find or Create Array
B34C	45900	Compute subscript size
B37D	45949	Perform FRE
B391	45969	Integer to FAC1
B39E	45982	Perform POS
B3A6	45990	Check for DIRECT mode
B3B3	46003	Perform DEF
B3E1	46049	Check FN syntax
B3F4	46068	Evaluate FN
B465	46181	Perform STR$
B475	46197	Calculate string vector
B487	46215	Set up string
B4F4	46324	Build string vector
B526	46374	Collect garbage (make room for string)
B5BD	46525	Check string collection eligibility
B606	46598	Collect string
B63D	46653	Concatenate string
B67A	46714	Build string to memory
B6A3	46755	Discard unwanted string
B6DB	46811	Clean the descriptor stack
B6EC	46828	Perform CHR$
B700	46848	Perform LEFT$
B72C	46892	Perform RIGHT$
B737	46903	Perform MID$
B761	46945	Pull string parameters from stack
B77C	46972	Perform LEN
B782	46978	Exit string mode
B78B	46987	Perform ASC
B79B	47003	Input byte parameter
B7AD	47021	Perform VAL
B7EB	47083	Get POKE/WAIT parameters
B7F7	47095	FAC1 to integer
B80D	47117	Perform PEEK
B824	47140	Perform POKE
B82D	47149	Perform WAIT
B849	47177	Add 0.5 to FAC1

Hex	Decimal	Function
B850	47184	Perform subtraction
B86A	47210	Perform addition
B947	47431	Complement FAC1
B97E	47486	Overflow
B983	47491	Single byte multiply
B9BC	47548	Floating point constants
B9EA	47594	Perform LOG
BA28	47656	Multiply FAC1 * memory
BA30	47664	Multiply FAC2 * FAC1
BA59	47705	Multiply a bit
BA8C	47756	Memory to FAC2
BAB7	47799	Adjust FAC1/FAC2
BAD4	47828	Underflow/overflow
BAE2	47842	Multiply FAC1 by 10
BAF9	47865	Constant 10
BAFE	47870	Divide by 10
BB07	47879	Divide FAC2 / memory
BB0F	47887	Divide memory / FAC1
BB12	47890	Divide FAC2 / FAC1
BBA2	48034	Memory to FAC1
BBD7	48087	FAC1 to memory
BBFC	48124	FAC2 to FAC1
BC0F	48143	FAC1 to FAC2
BC1B	48155	Round off FAC1
BC2B	48171	Get sign
BC39	48185	Perform SGN
BC58	48216	Perform ABS
BC5B	48219	Compare FAC1 to memory
BC9B	48283	FAC1 to integer
BCCC	48332	Perform INT
BCF3	48371	ASCII to FAC1
BD7E	48510	Get new ASCII digit
BDB3	48563	Constants
BDDD	48605	FAC1 to ASCII
BF11	48913	More constants
BF71	49009	Perform SQR
BF78	49016	Perform exponentiation
BFB4	49076	Perform negation

Hex	Decimal	Function
BFBF	49087	More constants yet
BFED	49133	Perform EXP
C000	49152-53247	RAM available for machine language progs
D000	53248-57343	I/O & Color RAM/Char Gnratr ROM/4K RAM
D000	53248	Sprite 0 X Pos
D001	53249	Sprite 0 Y Pos
D002	53250	Sprite 1 X Pos
D003	53251	Sprite 1 Y Pos
D004	53252	Sprite 2 X Pos
D005	53253	Sprite 2 Y Pos
D006	53254	Sprite 3 X Pos
D007	53255	Sprite 3 Y Pos
D008	53256	Sprite 4 X Pos
D009	53257	Sprite 4 Y Pos
D00A	53258	Sprite 5 X Pos
D00B	53259	Sprite 5 Y Pos
D00C	53260	Sprite 6 X Pos
D00D	53261	Sprite 6 Y Pos
D00E	53262	Sprite 7 X Pos
D00F	53263	Sprite 7 Y Pos
D010	53264	Sprites 0-7 Pos (msb of X coord.)
D011	53265	VIC Control Register
D012	53266	Read/Write Raster Value for Compare IRQ
D013	53267	Light-Pen Latch X Pos
D014	53268	Light-Pen Latch Y Pos
D015	53269	Sprite Display Enable:1=Enable
D016	53270	VIC Control Register
D017	53271	Sprites 0-7 Expand 2X Vertical (Y)
D018	53272	VIC Memory Control Register
D019	53273	VIC Interrupt Flag Reg.(Bit=1: IRQ Occurred)
D01A	53274	IRQ Mask Reg.1=Interrupt Enabled
D01B	53275	Sprite to Bkgrnd Display Priority: 1= Sprite
D01C	53276	Sprites 0-7 Multi-Color Mode Select
D01D	53277	Sprites 0-7 Expand 2X Horizontal
D01E	53278	Sprite to Sprite Collision Detect
D01F	53279	Sprite to Background Collision Detect
D020	53280	Border Color
D021	53281	Background Color 0

Hex	Decimal	Function
D022	53282	Background Color 1
D023	53283	Background Color 2
D024	53284	Background Color 3
D025	53285	Sprite Multi-Color Register 0
D026	53286	Sprite Multi-Color Register 1
D027	53287	Sprite 0 Color
D028	53288	Sprite 1 Color
D029	53289	Sprite 2 Color
D02A	53290	Sprite 3 Color
D02B	53291	Sprite 4 Color
D02C	53292	Sprite 5 Color
D02D	53293	Sprite 6 Color
D02E	53294	Sprite 7 Color
D400	54272	Voice 1:Frequency Control-Low-Byte
D401	54273	Voice 1:Frequency Control-High-Byte
D402	54274	Voice 1:Pulse Waveform Width-Low-Byte
D403	54275	Unused
D404	54276	Voice 1:Ctrl Reg. Random Noise 1=On
D405	54277	Envelope Gnrtr 1: Attack/Decay Cycle Control
D406	54278	Envelope Gnrtr 1:Sust/rel Cycle Control
D407	54279	Voice 2: Frequency Control-Low Byte
D408	54280	Voice 2: Frequency Control-High-Byte
D409	54281	Voice 2: Pulse Waveform Width-Low-Byte
D40A	54282	Unused
D40B	54283	Voice 2: Control Register
D40C	54284	Envelope Gnrtr 2: Attack/Decay Cycle Control
D40D	54285	Envelope Gnrtr 2:Sust/Rel Cycle Control
D40E	54286	Voice 3: Frequency Control-Low-Byte
D40F	54287	Voice 3: Frequency Control-High-Byte
D410	54288	Voice 3: Pulse Waveform Width-Low-Byte
D411	54289	Unused
D412	54290	Voice 3:Ctrl Reg Random Noise: 1=On
D413	54291	Envelope Gnrtr 3:Attack/Decay Cycle Ctrl
D414	54292	Envelope Gnrtr 3:Sust/Rel Cycle Control
D415	54293	Filter Cutoff Freq Low-Nybble (Bits 2-0)
D416	54294	Filter Cutoff Freq High-Byte
D417	54295	Filter Resonance Ctrl/ Input Control
D418	54296	Select Filter Mode and Volume

Hex	Decimal	Function
D419	54297	Analog/Digital Cvtr:Game Paddle 1 (0-255)
D41A	54298	Analog/Digital Cvtr:Game Paddle 2 (0-255)
D41B	54299	Oscillator 3 Random Number Generator
D41C	54230	Envelope Generator 3 Output
DC00	56320	Data Port A (Kybd,Joy,Paddles,Light-Pen)
DC01	56321	Data Port B (Kybd,Joy,Paddles,Game Port 1
DC02	56322	Data Direction Register-Port A (56320)
DC03	56323	Data Direction Register-Port B (56321)
DC04	56324	Timer A: Low-Byte
DC05	56325	Timer A: High-Byte
DC06	56326	Timer B: Low-Byte
DC07	56327	Timer B: High-Byte
DC08	56328	Time-of-Day Clock: 1/10 Seconds
DC09	56329	Time-of-Day Clock: Seconds
DC0A	56330	Time-of-Day Clock: Minutes
DC0B	56331	Time-of-Day Clock: Hours + AM/PM Flag (Bit 7)
DC0C	56332	Synchronous Serial I/0 Data Buffer
DC0D	56333	CIA Intrpt Ctrl Reg (Read IRQs/Write Mask)
DC0E	56334	CIA Control Register A
DC0F	56335	CIA Ctrl Reg B 1= alarm, 0=Clock
DD00	56576	Data Port A (Serial Bus,RS-232,Mem Ctrl)
DD01	56577	Data Port B (User Port,RS-232)
DD02	56578	Data Direction Register-Port A
DD03	56579	Data Direction Register-Port B
DD04	56580	Timer A: Low-Byte
DD05	56581	Timer A: High-Byte
DD06	56582	Timer B: Low-Byte
DD07	56583	Timer B: High-Byte
DD08	56584	Time-of-Day Clock: 1/10 Seconds
DD09	56585	Time-of-Day Clock: Seconds
DD0A	56586	Time-of-day Clock: Minutes
DD0B	56587	Time-of-Day Clock: Hours + AM/PM Flag (Bit 7)
DD0C	56588	Synchronous Serial I/0 Data Buffer
DD0C	56589	CIA Intrp Ctrl Reg (Read NMIs/Write Mask)
DD0E	56590	CIA Ctrl Reg.A TOD Clock Freq 1=50Hz,0=60Hz
DD0F	56591	CIA Ctrl Reg B 1=Alarm,0=Clock
E000	57344-65535	KERNAL ROM
E043	57411	Series evaluation

Hex	Decimal	Function
E08D	574	RND constants
E097	57495	Perform RND
E264	57956	Perform COS
E26B	57963	Perform SIN
E2B7	58039	Perform TAN
E2E0	58080	Constants for trig functions
E30D	58125	Perform ATN
E33E	58174	Constants for ATN
E394	58260	Initialize RAM vectors
E3A2	58274	CHRGET for zero page
E3BF	58303	Initialize BASIC
E45F	58463	Messages
E4AD	58541	Program patch area
E505	58629	Set screen limits
E50A	58634	Track cursor location
E518	58648	Initialize I/O
E531	58673	Normalize screen
E544	58692	Clear screen
E566	58726	Home cursor
E56C	58732	Set screen pointers
E5A0	58784	Set I/O defaults
E5A8	58792	Set vic chip defaults
E5B4	58804	Input from keyboard
E632	58930	Input from screen
E684	59012	Quote mark test
E691	59025	Set up screen print
E6B6	59062	Advance cursor
E6F7	59127	Retreat cursor
E701	59137	Back into previous line
E716	59159	Output to screen
E87C	59516	Go to next line
E891	59537	Do 'RETURN'
E8A1	59553	Check line decrement
E8B3	59571	Check line increment
E8CB	59595	Set colour code
E8DA	59610	Colour code table
E8E2	59618	Code conversion
E8E7	59624	Scroll screen

Hex	Decimal	Function
E965	59749	Open space on screen
E9C8	59848	Move screen line
E9E0	59872	Synch colour transfer
E9F0	59888	Set start-of-line
E9FF	59903	Clear screen line
EA13	59923	Print to screen
EA1C	59932	Store on screen
EA24	59940	Synch colour to char
EA31	59953	Interrupt (IRQ)
EA87	60039	Check keyboard
EB59	60249	Set text mode
EB79	60281	Keyboard vectors
EB91	60305	Keyboard maps
EC44	60484	Graphics/text control
EC4F	60495	Set graphics mode
ED09	60681	Send 'talk'
ED0C	60684	Send 'listen'
ED11	60689	Send control char
ED36	60726	Send to serial bus
EDB0	60848	Timeout on serial
EDB9	60857	Send listen SA
EDBE	60862	Clear ATN
EDC7	60871	Send talk SA
EDDD	60893	Send serial deferred
EDEF	60911	Send 'untalk'
EDFE	60926	Send 'unlisten'
EE13	60947	Receive from serial bus
EE85	61061	Clock line on
EE8E	61070	Clock line off
EEB3	61107	Delay 1 ms
EEBB	61115	RS232 send (NMI)
EF06	61190	New RS232 byte send
EF2E	61230	Error or quit
EF4A	61258	Compute bit count
EF59	61273	RS232 receive (NMI)
EF7E	61310	Setup to receive
EFC5	61381	Receive parity error
EFCA	61386	Receive overrun error

Hex	Decimal	Function
EFCD	61389	Receive break error
EFD0	61392	Receive frame error
EFE1	61409	File to RS232
F017	61463	Send to RS232 buffer
F04D	61517	Input from RS232 buffer
F086	61574	Get from RS232 buffer
F0A4	61604	Check serial bus idle
F0BD	61629	Messages
F12B	61739	Print if direct
F13E	61758	Get..
F14E	61774	..from RS232
F157	61783	Input
F199	61849	Get..tape/serial/RS232
F1CB	61899	Output..
F1DD	61917	..to tape
F20E	61966	Set input device
F250	62032	Set output device
F291	62097	Close
F30F	62223	Find file
F31F	62239	Set file values
F32F	62255	Abort all files
F333	62259	Restore default I/O
F34A	62282	Do file opening
F3D5	62421	Send SA
F409	62473	Open RS232
F49E	62622	Load program
F5AF	62895	'SEARCHING'
F5B8	62904	Print file name
F5D2	62930	'LOADING/VERIFYING'
F5DD	62941	Save program
F68F	63119	'SAVING'
F69B	63131	Bump clock
F6DD	63197	Get time
F6E4	63204	Set time
F6ED	63227	Action stop key
F6FB	63227	File Error Messages
F72C	63276	Find any tape header
F76A	63338	Write tape header

Hex	Decimal	Function
F7D0	63440	Get buffer address
F7D7	63447	Set buffer start - end pointers
F7EA	63466	Find specific header
F80D	63501	Bump tape pointer
F817	63511	'PRESS PLAY'
F82E	63534	Check cassette status
F838	63544	'PRESS RECORD'
F841	63553	Initiate tape read
F864	63588	Initiate tape write
F875	63605	Common tape read/write
F8D0	63696	Check tape stop
F8E2	63714	Set timing
F92C	63788	Read bits (IRQ)
FA60	64096	Store characters
FD8E	64398	Reset pointer
FB97	64407	New tape character setup
FBA6	64422	Toggle tape
FBC8	64456	Data write
FBCD	64461	Tape write (IRQ)
FC57	64599	Leader write (IRQ)
FC93	64659	Restore vectors
FCB8	64696	Set vector
FCCA	64714	Kill motor
FCD1	64721	Check read/write pointer
FCDB	64731	Bump read/write pointer
FCE2	64738	Powerup entry
FD02	64770	Check A-rom
FD13	64787	Set Kernal
FD52	64850	Initialize system constants
FD9B	64923	IRQ vectors
FDA2	64930	Initialize I/O regs
FDF9	65017	Save data name
FE00	65024	Save file details
FE07	65031	Get status
FE18	65048	Flag ST
FE21	65057	Set timeout
FE25	65061	Read/set top memory
FE34	65076	Read/set bottom of memory

Hex	Decimal	Function
FE43	65091	NMI interrupt entry
FE66	65126	RESET/STOP warm start
FEBC	65212	Restore & exit
FEC2	65218	RS232 timing table
FF48	65352	Main IRQ entry
FF81	65409	Jumbo jump table
FFFA	65530	Hardware vectors

Sample Bit-Mapped plotting

The following programs are an example of a machine language subroutine, callable by either BASIC or machine language, and a BASIC program which uses the routine. The machine language program is designed to turn on individual pixels based on a x and y bit-position passed from the calling routine. The routine assumes that the x coordinate of the point to be plotted is stored in 253,254. Location 253 must contain the high-order bit of the number and 254 the low-order eight bits. The x value may range from 0 to 319. The y-coordinate must be stored in location 255. The y value may range from 0 to 199. The upper left corner of the screen is considered bit position 0,0 and the lower right is position 319,199. The carry bit is used as a mode switch. It is passed to the ML program from BASIC by setting the low-order bit of location 783. This is the location of the processor status register when communicating to machine language programs via a SYS. If the carry bit is set the ML program will set the specified bit on the screen to the foreground color. If clear, the color of the bit will be set to the background color. The BASIC program which calls the plot routine must first call the initialization routine which will set the screen location, clear the screen and the color memory. This simple BASIC program plots a sinusoidal pattern.
The BASIC program:

```
10 SYS 49323
20 FOR X = 0 TO 1000 STEP.05:Y = 100 + 40 * SIN (X)
30 XP = X * 30: POKE 254,XP AND 255: POKE 253, XP/256
40 POKE 255,Y: POKE 783,1: SYS 49231: NEXT
```

The machine language program:

```
1       ;
2       ; PLOT SUBROUTINES
3       ;     BY TOM COURT
4       ;
5  LWADR   EQU 251          ADDRESS POINTER
6  HIADR   EQU 252
7  CURHX   EQU 253          X COORDINATE OF PLOT
8  CURLX   EQU 254           RANGE (0-319)
9  CURY    EQU 255          Y COORDINATE (0-199)
10 TEMP     EQU 2
11 CHARMEM EQU $E000          BIT MAP OF SCREEN
12 SCREEN  EQU $C400
13 IRQCTRL EQU 56334
14 ROMSWCH EQU 1
15 BANK    EQU 56576
16 BITMODE EQU 53265
17 SCRNPOS EQU 53272
18      ;
19      ;  * STRAD ROUTINE *
20      ;
21      ; CONVERT X,Y COORD OF PLOT TO
22      ;  ADDRESS OF BYTE TO MODIFY IN
23      ; BIT-MAPPED CHAR MEMORY
24      ;
25      ; X-COORD MUST BE STORED IN CURX
26      ;  (9-BIT VALUE GOES IN TWO BYTES)
27      ;
28      ; Y-COORD MUST BE IN CURY
29      ;
30      ; ON RETURN X-REG CONTAINS 0-7
31      ;  (BYTE WITHIN 8-BYTE BLOCK)
32      ;
33      ; Y-REG CONTAINS BIT POSITION
34      ; WITHIN THE BYTE TO MODIFY
35      ;
36 STRADR  EQU $C000
37         LDA #0
38         STA <TEMP
39         STA <HIADR
40         LDA <CURY
```

```
41          AND #$F8          Y=8*INT(Y/8)
42      ;
43      ; TO MULT BY 40:
44      ;  FIRST MIULT BY 32 (5-BIT SHIFT LEFT)
45      ;  THEN MULT IT BY 8 (3-BIT LEFT)
46      ;  THEN ADD THE TWO TOGETHER
47      ;
48          ASL A
49          ROL <HIADR
50          ASL A
51          ROL <HIADR
52          ASL A
53          ROL <HIADR
54          ASL A
55          ROL <HIADR
56          ASL A
57          ROL <HIADR
58          STA <LWADR
59          LDA <CURY
60          AND #$F8
61          ASL A
62          ROL <TEMP
63          ASL A
64          ROL <TEMP
65          ASL A
66          ROL <TEMP
67          CLC
68          ADC <LWADR
69          STA <LWADR
70          LDA <TEMP
71          ADC <HIADR
72          STA <HIADR
73          LDA <CURLX
74          AND #$F8          X=8*INT(X/8)
75      ;
76      ;POSITION=40*(8*INT(Y/8)*8+INT(X/8))
77      ;
78          CLC
79          ADC <LWADR
80          STA <LWADR
```

```
81          LDA ←CURHX
82          ADC ←HIADR
83      ;
84      ; ADDR = POSITION IN MAP + STRT OF MAP
85      ;
86          ORA #>CHARMEM
87          STA ←HIADR
88          LDA ←CURY
89          AND #7              POS IN BLOCK = (Y AND 7)
90          TAY
91          LDA ←CURLX          BIT POS = (X AND 7)
92          AND #7
93          TAX
94          RTS                 RETURN TO DOT
95      ;
96      ;  DRAW ENTRY POINT
97      ;
98      ; CARRY SET TO:
99      ; PLOT (SET) UNPLOT (CLEAR)
100      ;
101      ; CURX,CURY MUST CONTAIN X,Y COORD
102      ;
103 DOT      PHP
104          JSR STRADR          GET BIT-MAP ADDR TO MOD
105          LDA IRQCTRL         TURN OFF IRQ'S
106          AND #$FE
107          STA IRQCTRL
108          LDA ←ROMSWCH
109          AND #$FD            SWITCH OUT THE ROM
110          STA ←ROMSWCH
111          PLP
112          BCC UNDRAW           TEST CARRY
113          LDA (LWADR),Y       GET BYTE TO MOD
114          ORA TABL,X          TURN ON SELECTED DOT
115          STA (LWADR),Y
116          JMP SKPUND
117 UNDRAW   LDA TABL,X          TURN DOT OFF
118          EOR #$FF            REVERSE ALL BITS
119          AND (LWADR),Y
120          STA (LWADR),Y
```

```
121 SKPUND   LDA <ROMSWCH      BRING BACK ROM
122          ORA #2
123          STA <ROMSWCH
124          LDA IRQCTRL       ENABLE THE IRQ'S
125          ORA #1
126          STA IRQCTRL
127          RTS
128      ;
129      ;COLOR SET ROUTINE
130      ;
131      ; SETS SCREEN COLORS
132      ;
133      ; A-AREG MUST HAVE FGRND COLOR
134      ;  IN HIGH 4-BITS.
135      ; BKGRND IN LOW-4 BITS.
136      ;
137 COLOR    LDX #250          SET COUNT
138 LOPCOL   DEX
139          STA SCREEN,X
140          STA SCREEN+250,X
141          STA SCREEN+500,X
142          STA SCREEN+750,X
143          BNE LOPCOL
144          RTS
145      ;
146      ; CLEARS ENTIRE HIRES SCREEN
147      ;
148 CLS      LDA #0
149          STA <LWADR
150          LDX #>CHARMEM     GET STARTING PAGE
151 LOPOUT   STX <HIADR
152          LDY #0
153 LOPIN    DEY
154          STA (LWADR),Y      STORE A 0
155          BNE LOPIN
156          INX
157          BNE LOPOUT        CONTINUE TILL DONE
158          RTS
159      ;
160      ; INIT ROUTINE
```

```
161       ;
162       ; TURNS ON THE GRAPHICS MODE,
163       ;  CLEARS THE SCREEN
164       ;  SETS FOREGROUUND = WHITE
165       ;  AND BKGRND = BLACK
166       ;
167            LDA BANK+2        SET BANK PORT TO OUTPUT
168            ORA #3
169            STA BANK+2
170            LDA BANK          AND THE VIDEO BANK TO 3
171            AND #$FC
172            STA BANK
173            LDA BITMODE
174            ORA #32
175            STA BITMODE
176            LDA #$18          SET CHARMEM TO E000-FFFF
177            STA SCRNPOS       AND SCREEN TO $C000
178            LDA #$10          SET COLORS
179            JSR COLOR
180            JMP CLS           CLEAR SCREEN AND RETURN
181 TABL       BYT $8040201008040201
```

Explanation:

The screen has 25 rows of 40 characters each. Each character has eight rows of eight dots each. The character memory is where the hi-res bit map of the screen resides. There are 8000 bytes of screen information contained there. This breaks down to 40 columns * 25 rows * 8 bytes per character. The first eight bytes of char mem contain the bit map of the upper left square of 64 dots arranged in an eight by eight array. There are therefore 40 * 8 or 320 bytes to map the first eight rows of dots. The second 320 bytes in character memory reference the second row of characters (8 rows of actual dots). So, block #0 is the pattern for the upper left eight by eight block of dots and block #1 is the block to the right of that and block #40 is the pattern of dots for the block below it and so forth till block #1000 which describes which pixels will be illuminated in the bottom right corner of the screen.

The above machine language subroutine computes which block corresponds to any given X and Y dot coordinate. It also computes which byte within the eight-byte block holds the dot in question and which bit position within that byte to turn on or off.

The Y-coordinate represents how far from the top of the screen the dot is located. If it is in the range of 0-7, the dot is in the first row of characters. If is is 8-15, it is in the second row, etc. So to find the row Y must be divided by 8, discarding any remainder. Each row of characters contains 40 blocks across the screen and each block contains eight bytes of data. So, for every row, the position within the table increases by 8 times 40 or 200.

The address of the appropriate byte of the table to modify is built in the two-byte field labeled LWADR and HIADR. STADR is the routine which computes the address of the block to modify. It also returns with the relative byte within the block in the X-reg and the bit within that byte to modify in the Y-reg. The only thing remaining is to get the actual bit within the byte and either turn it on or off as indicated by the mode switch in location 0. This is done in the mainline of the DOT routine.

Appendix F - Data

Bits. Binary digits. Its actually a contradiction in terms. Binary means it can have two possible values. Digit implies ten.

The bit is the most basic unit of information. It is the foundation of all other more complex information formats. It is the only kind of information which may be stored in a "digital" computer. A bit may have the value of either one or zero. A byte is a grouping of eight bits. The 6510 is called an eight-bit microprocessor. This is because it processes and stores data eight bits at a time. It is more convenient for the purpose of understanding the nature of computer data to break the eight bits into two four bit sections, called nybbles.

There are exactly sixteen unique four-bit combinations of ones and zeros such that no two arrangements are alike. There is a common shorthand notation for identifying these sixteen patterns. It goes like this:

8421	8421	8421	8421
0000=0	0100=4	1000=8	1100=C
0001=1	0101=5	1001=9	1101=D
0010=2	0110=6	1010=A	1110=E
0011=3	0111=7	1011=B	1111=F

It's a convenient shorthand system because it is easy to remember. It simply numbers the patterns from 0 to 15, except, in keeping with the idea of using a one character code for each pattern, the numbers 10-15 are called A-F. So, pattern 12 is called "C" and 15 is "F", etc. It's got another advantage too. The shorthand label system has an order which makes it easy to remember. If the left-most bit position may be considered to have the value of 8; and the next position, the value 4; and the next, 2; and the last, one: then the bit patterns may be converted to their labels by adding the bit values of the individual bit positions. For example: 1010 has a 1 in the 8 position, a 0 in the 4 position, a 1 in the 2 position, and a 0 in the 1 position. So its label is "A" because 8 + 2 = 10 and "A" is the code for 10. Likewise, 0110 is labeled "6" because it has no 8, one

4, one 2 and no 1.

The leftmost bit position is called the "most-significant" bit because it has the greatest bit value. It is also called the "high-order" bit. Likewise, the rightmost bit is the "least significant" or "low-order bit".

Note that all the odd numbers have a one in the rightmost bit position. All the even numbers, a zero.

You can now see that the shorthand notation for identifying a four bit data field is very logical and it is not hard to convert back and forth between the bit patterns and the pattern code. This is important because as machine language programmers, we have to end up working a lot with bits. This is because the 6510 and all of the memory devices store and work with all data coded as bits.

The addresses of where data is located within a memory device is likewise represented and transferred on the address bus in bits.

Now, since data is stored and retrieved to and from the memory devices eight bits at a time (called a byte of data), it takes two shorthand codes to describe the bit content of the data. A byte of data which has a bit structure of 1100 0100 would be identified as having the two-character code of "C4". The bit structure of the code for the letter "A" would be "41" because the bit pattern assigned to "A" in the ASCII coding scheme is 0100 0001. It's helpful to sit with a pencil and paper and write out bit patterns and figure out what their codes are. You should also start with the codes 0-F and convert back to bit patterns.

That there are 256 ways to represent eight bits is obvious from the fact that there are sixteen possible first characters of the two character code which identifies an eight-bit pattern and sixteen possible second characters. And 16 times 16 = 256. If we wanted a byte of data to represent a numeric value instead of an ASCII coded character, we can see that it could represent any value between 0 and 255. The 6510 does, in fact, sometimes treat data as if it is numeric instead of character data. Using the same two-character code to identify an 8 bit pattern, we can convert between the numeric value and the two-character code quite easily. For example, to convert from "A6" to its numeric value, we would multiply 10 times 16 ("A" = 10) and add 6 to give 166. Likewise, to convert "4F" to numeric we would multiply 4 times 16 and add 15 ("F" = 15) to give 79. It's easy to see that the maximum value of "FF" = 15 times 16, (240) plus 15 to give 255. Of course, the minimum would be "00" which is (0 times 16)

plus 0.

How about going the other way? If we want to create the bit pattern for some number in the range of 0 - 255, we only have to divide the number by 16 to get the first character of the code (remember to translate 10 thru 15 to A thru F). Then the remainder of the division is the second character. Now to get the bit pattern we just look it up in the little bit pattern table. Put the first four-bit pattern together with the second and you have an eight bit pattern which represents the numeric value. Not hard at all. Lets try it with a few numbers. Take the number 169. How many times does 16 go into 169? Right. And how is that represented in our single character code? As the letter "A". The remainder from dividing 169 by 16 is 9. So our character-code representation of 169 is "A9" and if we go to the bit pattern table, we will see that the bit patterns are 1010 1001.

It is a valuable exercise to practice converting bit patterns to character-codes and from character-codes to decimal and from decimal to character-codes and character-codes to bit patterns. You will find a table of all 256 bit patterns and the corresponding character-codes and corresponding decimal values in Appendix B. You may check your success with the table.

The character-code system for identifying the various bit patterns is, as you might know or have figured out, what in computer circles is called hexadecimal. Numbers, when represented in hexadecimal, are usually preceeded by a "$". $41 is the hex representation for the ASCII code for the letter "A". This removes any question as to whether the number is decimal or hexadecimal. From now on we will follow that convention in this book.

There is another convention which is widely used to identify the individual bits in a byte. The bits are numbered 0-7 from right to left. The high-order bit is the 7-bit and the low-order bit is the 0-bit.

We discussed the way to represent the decimal numbers from 0 to 255. The same general technique may be used to represent numbers from 0 to 65535. Instead of two hex characters representing one byte of data, we need four hex characters representing sixteen bits or two bytes. Now the first pair of hex characters may be followed by any of 256 pairs of hex characters ($00 - $FF). So the total number of possible combinations of four hex characters is 256 times 256 or 65536. The address bus is sixteen bits wide. Which is why there are

exactly 65536 unique addresses possible. Now, it is frequently useful to be able to convert decimal addresses into hexadecimal. We saw how to do it with a single byte of two hex characters (8 bits). We divided the number by 16 to get the first character and the remainder was the second character. The process is similar for going from a number larger than 255.

We divide the number by 256 to get the first half of the answer. This will be a number between 1 and 255. The remainder will be the second half of the answer. Both of these decimal numbers can then be converted to their hexadecimal counterparts by the dividing-by-16 technique. These hex digits can be then easily converted to bits (binary) by looking up the table or retrieving it from our biological memory device.

Let's do an example: Say we want to get the binary value (bit configuration) of the decimal number 47892. The first thing we do is see how many times 256 will go into 47892. The answer is 187. The remainder is 20. A line of BASIC code to do this computation would be:

```
HA = INT ( NUM / 256 ): HB = NUM - HA * 256
```

HA is the first half of the answer in decimal. HB is the second half. We still have to take 187 and 20 and break them into their two hex components. We do this by dividing by 16. 187 / 16 = 11 with a remainder of 11. So the first two hex digits are $BB. 20 / 16 = 1 with a remainder of 4. So the complete answer is $BB14. The binary equivalent of $BB14 is 1011 1011 0001 0100. To double check our answer, we go back the other direction and convert $BB14 to decimal. $14 = (1 * 16) + 4 = 20. $BB = (11 * 16) + 11 = 176 + 11 = 187. (187 * 256) + 20 = 47872 + 20 = 47892. And that's the number we started with. This process should be practiced. It is very helpful in solidifying the understanding of hex and binary and their relationships to decimal numbers.

P.O. Box 207, Cannon Falls, MN 55009
ISBN 0-9612422-0-5