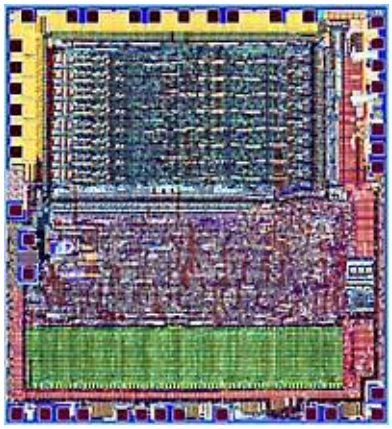
cbA65

User's Guide and Reference Manual

v. 1.00

A MOS Technology/Commodore 65xx Cross-assembler
for Microsoft DOS Platforms

Charles R. Bond
http://www.crbond.com



(6502 image courtesy http://microscopy.fsu.edu)

September 4, 2009

Contents

List of Figures

List of Tables

Part I

Using cbA65

Chapter 1

Introduction

The 65xx family of microprocessors was developed by MOS Technology, which was later acquired by Commodore.[1] The principal designer, Chuck Peddle, had previously designed the 6800 processor for Motorola.

cbA65 is one of a series of programmers tools for general purpose 65xx code development and analysis. Other tools include a macro-assembler, a symbolic disassembler, source generators, debuggers, relocators and emulators.

This assembler[2] is provided as an executable file which runs under Microsoft DOS on Pentium compatible hardware platforms, although other platforms may also provide a suitable operating environment.

In the following chapters the reader will find a detailed description of the **cbA65** software, its features, capabilities and limitations, the source file format, syntax and user options. Special features of the assembler are fully described. The 65xx CPU architecture and instruction set are also covered for convenience. Some examples of programming techniques have been included to illustrate assembler usage concepts.

A summary of the contents of each chapter should help you find the sections of interest.

- Chapter 2 – Assembling a source file.
- Chapter 3 – Assembler directives and syntax.
- Chapter 4 – Source and include files.

[1] Manufacturing licenses were granted to Synertek, Rockwell and Western Design Center among others.

[2] For brevity, in the following chapters, **cbA65** will be referred to as an assembler, rather than a cross-assembler.

- Chapter 5 – Output files and formats.
- Chapter 7 – 65xx processor model and architecture.
- Chapter 8 – Data tables and indexing.
- Chapter 9 – A hashing example.

1.1 Features

cbA65 is a full-featured assembler suitable for program development with all members of the 65xx processor family. Although this version does not support macros, repeat blocks or conditional assembly, it has a number of unique and advanced features which may recommend it to the general 65xx programmer.[3]

A few significant features are:

Performance — Assembles files quickly, accurately and efficiently.

Readability — Listings are formatted for maximum readability.

Listing controls — Supports user control over list file format.

Object files — Offers several optional object file types.

Cycle counting — Provides processor cycle counting option.

Anonymous labels — Advanced support for unnamed target addresses.

Size control — User control over instruction address size.

Instruction usage — Reports instruction usage counts.

Cross reference — Sorted symbol cross-reference table.

1.2 A Note on the Readability of Listings

The highest priority in the development of this assembler was accuracy. Performance, flexibility and robustness were also development objectives, but one of the top priorities was to enhance the readability of the output listings.

[3] A macro assembler, which includes all features of this assembler as well as support for macros, repeat blocks, table generation and conditional assembly will be released as **cbM65**.

With many assemblers, the listings are difficult to read, complicating the debug process. Experience has shown that code listings which are strongly column oriented are easier to read and analyze, partly because the eye can locate sections of interest by scanning vertically in straight lines. Free form assemblers, in contrast, often produce listings which force the programmer to scan left and right as well as up and down to find the lines he wishes to examine.

An example of readability problems can be seen in this excerpt from a code listing produced by an assembler for an Intel processor.[4]

```
1   633 000000C8  adcd0:
1   634 000000C8  75 0D 90 90 90 90      jnz adcd1       ; use binary flags
1   635 000000CE  9C      pushf
1   636 000000CF  80 0D 00000003r 02      or s_reg,ZFLAG
1   637 000000D6  9D      popf
1   638 000000D7  adcd1:
1   639 000000D7  79 0B 90 90 90 90      jns adcd2
1   640 000000DD  80 0D 00000003r 80      or s_reg,NFLAG
1   641 000000E4  adcd2:
1   642 000000E4  A2 00000000r      mov a_reg,al
2   643 000000E9  C3  RET  00000h
1   644 000000EA  adcb0:  ; binary add
1   645 000000EA  80 25 00000003r 3C         and s_reg,NOT (PFLAGS OR ZFLAG)
1   646 000000F1  D0 EE      shr dh,1
1   647 000000F3  12 D0      adc dl,al
1   648 000000F5  71 0D 90 90 90 90      jno adcb1
1   649 000000FB  9C      pushf
1   650 000000FC  80 0D 00000003r 40      or s_reg,VFLAG
1   651 00000103  9D      popf
1   652 00000104  adcb1:
1   653 00000104  80 15 00000003r 00      adc s_reg,0
1   654 0000010B  88 15 00000000r      mov a_reg,dl
1   655 00000111  E9 000000E0      jmp nz_flags
```

Clearly, the programmer will have significant problems sifting through the information in this listing to identify and correct errors. Compare this listing with the following output of **cbA65** for an assembly of the KIMATH subroutine package.

[4] From a TASM32 assembly of a 6502 emulator.

```
000637 FBAC 9D 01 02            sta ra+1,x
000638 FBAF CA                  dex
000639 FBB0 10 F7               bpl rsra0
000640 FBB2 A9 00               lda #0
000641 FBB4 8D 00 02            sta ra
000642 FBB7 60                  rts
000643                  ;
000644                  ;   Clear working storage.
000645                  ;
000646 FBB8 A2 34       clear   ldx #len*3+1
000647 FBBA A9 00               lda #0
000648 FBBC 9D 00 02    az0     sta ra,x
000649 FBBF CA                  dex
000650 FBC0 10 FA               bpl az0
000651 FBC2 60                  rts
000652                  ;
000653                  ;   Convert the contents of cnt
000654                  ;   from bcd to hex and store the
000655                  ;   result in cnt.
000656                  ;
000657 FBC3 F8          dechex  sed
000658 FBC4 A2 00               ldx #0
000659 FBC6 38                  sec
000660 FBC7 A5 03       dhcnv1  lda cnt
000661 FBC9 E9 16               sbc #$16
```

This listing is far easier to read and debug. The strong column orientation facilitates scanning for specific code blocks.

In the sections on list file options and formats, you will find information which can make the location and identification of errors and code sequences for **cbA65** easy — even in nested include files. Additional examples of output listings generated by **cbA65** will appear later in this document.

1.2.1 Symbolic Labels

Another consideration involved the naming conventions for symbolic labels, opcodes and directives. The strategy adopted for **cbA65** is to require that all symbolic labels for memory addresses or other numeric quantities begin in the first column of the source file. Where spaces are allowed in the source file their number is arbitrary, so this is the only column-specific formatting requirement.

1.2.2 Opcode Mnemonics and Directives

The assembler uses standard MOS Technology/Commodore mnemonics for the opcodes and uses a period `.` as the leading character in every assembler directive. The purpose is not only to simplify the file processing but also to simplify debugging. It is much easier for a programmer to scan a listing when the distinction between directives, opcodes and symbolic labels are obvious.

The burden imposed on the programmer by these conventions are simply stated:

- If a symbol defines an address label or numeric quantity, it must begin in the first column,
- If a symbol is a directive, it begins with a period,
- If a symbol is an opcode, it uses standard 65xx mnemonics, and must NOT begin in the first column,
- Comments are preceded by a semicolon,
- Blank lines may be used to separate code or data sections.

With these rules, and with the appropriate assembler directives, the output listings can be formatted for maximum clarity and ease of analysis. There is never any ambiguity between labels, directives and opcode mnemonics. See Chapter 3 for more on the assembler line format and Section 3.4 for information on the allowable characters and restrictions on symbolic labels.

Chapter 2

Getting Started

2.1 Installing cbA65

The assembler is provided as an executable file called, **cba65.exe** No additional support files, other than the programmer's source and include files, are required to complete an assembly.

To install **cbA65** simply create a folder on your chosen drive with a suitable name. Copy `cba65.exe` to the folder. A good strategy for managing multiple assembler projects is to create a sub-folder below the one containing **cbA65** for each project. Place the appropriate source and include files in the folder and add the location of the assembler to the operating system search path. With this method, simply go to the desired project folder and call the assembler from there. The output files will be written in this folder.

2.2 Assembling a Source File

Source code assembly requires four passes over the source file. These perform the following functions:

Pass 1 — Expand include files and build symbol table, check for syntax errors and duplicate labels.

Pass 2 — Trial assembly, check for size inconsistencies, issue warnings, output `.ECHO` messages.

Pass 3 — Resolve forward references, report range and phase errors.

Pass 4 — Final assembly, resolve phase errors, generate listing, log and object files.

To invoke the assembler for a typical application type:3

```
C:[path]>cba65 mycode.cba
```

where `[path]` is the current path and `mycode.cba` is the name of the user's source file. Including the extender with the filename is optional. Although it is not necessary to type the extender, **cbA65** will only accept source files whose extender matches "`cba`". This restriction minimizes the possibility of mixing files meant for different assemblers.

It is also possible to invoke the assembler by simply typing

```
C:[path]>cba65
```

at the DOS prompt.

In this case, the assembler will prompt the user for the source filename. Again, typing the extender is not required.

See the chapter on output files for a complete description of all files generated by the assembler.

2.3 Command Line Options

Unlike most assemblers, `cbA65` uses the command line solely to identify the name of the source file and to specify the assembler configuration options. No command line options are used to specify the output file types or the listing format. The reason for this option management strategy is to assure that the assembly as at least partially self-documented in the source and listing files. That is, any customization specifications must be made in the source file and will therefore be printed in the listing file. At any time it can be determined exactly how the programmer configured the options. See the section on *directives* for a complete list of the file control directives and options.

The assembler configuration options control the maximum allowed number of symbols and anonymous labels. The following example illustrates their use.

```
C:[path]>cba65 myfile -a=600 -s=2000
```

Here, the file `myfile.cba` will be assembled, with the assembler set to allow 600 (`a`)nonymous labels and 2000 (`s`)ymbolic labels. The default values are 500 and 1000, respectively. Note that the options are separated by spaces but may not contain spaces.

Several variants of the above syntax are allowed:

- Either or both of these options may be omitted.
- The hyphen – may be replaced with a forward slash /.
- The equal sign = may be omitted or replaced with colon :.
- The command line parameters may appear in any order.

The assembler configuration options are only available if the assembler is invoked on the command line. If the assembler is started with no command line parameters, the user will be prompted for the filename but no options will be accepted. The default values will the used in this case.

2.4 About Errors and Warnings

Writing code for any processor is typically an iterative or recursive process. Code is written and assembled. Errors in the source files are identified and corrected. Code is then reassembled. This process is repeated until no further errors are found. Then the code is run or emulated. If logic errors are found, they are corrected and the process begins anew. When everything is working as required, the process terminates.

A key requirement for a viable programming environment is the ability to locate and identify various kinds of errors. **cbA65** provides elementary but critical support for error detection.

The assembler marks its progress by printing the pass number to the screen at each stage of the assembly. Certain warnings may be displayed during the assembly, but if fatal errors occur the assembly will be aborted.[1] In this case an error message will be given which identifies the type of error and the file and line number at which the error occurred. See Chapter 6 for a detailed explanation of the error messages.

cbA65 can also issue advisory or informational messages chosen by the programmer using the `.ECHO` directive at critical points in the program text. This directive is executed during assembler pass 2, and prints the appended message to the screen.

[1] With **cbA65** a *warning* is a rule violation which may not invalidate the object code. An *error* renders the object code invalid.

Chapter 3

Assembler Syntax

cbA65 is a line oriented assembler with formatting rules which facilitate ease of programming and readability of the output listings. The assembler is not case-sensitive, so you may write your files using conventions which you prefer. The output listings will retain the case you specify in your code, but don't forget that the assembler ignores case when identifying or manipulating symbolic values.

The source file line format generally follows one of the following schemes:

LABEL OPCODE [OPERANDS] [COMMENT]

or

DIRECTIVE [OPTIONS[ARGUMENTS]] [COMMENT]

A number of variations are possible, such as:

LABEL

LABEL `.EQU` **[EXPRESSION]**

OPCODE [OPERAND]

COMMENT

DIRECTIVE

BLANK

where **COMMENT** consists of ASCII characters preceded by a semicolon. Bracketed terms are optional in each line format.

OPCODE must *not* appear in the first column, as explained earlier.

For variable assignments, `cbA65` uses the `.EQU` directive. Each label must be unique, or a fatal error will be reported. The value assigned to the label may be any valid expression which may include other symbols.

The maximum line length supported by **cbA65** is controlled by the `COLS` option for the `.PRINT` directive. See Section 3.6 for details.

3.1 Source File Format

A minimal source file might be as simple as the following:

```
        .org $200
        brk
        .end
```

This file will assemble without errors and produce all default output files. Here is an example of a somewhat more useful source file:

```
;
;   upper.cba -- convert null terminated string in include file to upper case
;
        .org $200
        ldx #0
start   lda mystr,x         ; get next character
        beq stop
        and #$20            ; convert to upper case...
        sta mystr,x         ; ...and save it
        inx
        bne start
stop    brk
        .org $300
mystr .include message.inc
        .end
```

No matter how complex the source file, the simple line formatting rules will apply to each line.

Every source program must contain an `.ORG` directive before any instructions which generate code or data. Otherwise an error message will be issued. See Section 3.7 for accessing the location counter to specify program addresses.

3.2 Constants

Constants consist of numeric quantities or ASCII characters and may be declared in several formats. For the 65xx family numeric constants are either 8-bit or 16-bit quantities. Symbols assigned to represent constants will be interpreted as the numbers they represent.

3.2.1 Numeric Constants

The default radix for numeric constants is decimal. Ordinary decimal integers may appear anywhere a numeric value is needed. Floating point numbers are not supported.

Hexadecimal numbers are distinguished by a leading $ character, as

in

```
lda #$45
jmp $8000
```

Valid hexadecimal digits include the decimal digits and the characters `A` through `F` or `a-f`.

Binary numbers are indicated by a leading % character.

```
lomask  .equ %00001111
himask  .equ %11110000
```

Only the digits **0** and **1** are recognized in binary constants. In this example, the numeric values of the constants have been assigned to symbolic labels.

3.2.2 Character Constants

An ASCII character may be enclosed in single quotes to reference its numeric value. For example

```
lda num,x       ; get next ASCII digit
sec
sbc #'0'        ; convert to numeric value
```

which gets the 8-bit numeric value of a digit from its ASCII representation. The complete ASCII character set is printed in Appendix C

Generally, the trailing quote is not really needed to identify a character constant and may be omitted.

3.3 Character Strings

Strings consist of ASCII characters and are most often used to represent readable text. Strings may be placed in the output file with the `.TEXT` directive and are specified by surrounding the text with double quote or single quote symbols. Some examples are:

```
greet   .text "Hello, universe!"
warn    .text 'Caution! Memory almost full.'
```

If it is necessary to place a single or double quote within the string, you may use the other one as the delimiter.

```
mystr   .text "Don't stop."
```

Although it is possible to handle strings which contain both delimiters by splitting the string into pieces, **cbA65** provides a better solution. In fact, you may choose your own delimiter to enclose strings. The first non-space character encountered after the `.TEXT` directive is taken as the string delimiter, and any other characters may be entered until the chosen delimiter appears again.[1]

Here is an example which uses the *at* sign `@` as a delimiter:

[1] An exception is that you cannot use a semicolon as a delimiter.

```
teststr .text @"Paddy O'Furniture"@
```

In this case, the double quotes are included in the resulting string.

Multiple comma-separated strings may be placed on the same line as follows:

```
monst   .text "It's"," alive!"
```

but note that the strings will simply be concatenated. Unterminated strings on consecutive lines will also be effectively concatenated, since they are simply placed in consecutive memory locations.

The directive `.STR` is an alternate to `.TEXT` and has the same syntax. There are also two variants `.STRA` and `.STRZ` which automatically append terminators to the text. See Section 3.6.3 for more on terminated strings.

3.4 Symbols

The symbols used by `cbA65` are taken from the standard ASCII character set. Some of these symbols, *e.g.* the alphabetic characters, are normally used for symbolic labels, CPU mnemonics, directives and comments. The numeric and some of the punctuation characters may have special meanings in particular contexts. These are explained in the next sections.

3.4.1 Labels

Labels are symbolic objects which stand for numeric values, program addresses or other values. They must begin with an alphabetic character or an underscore _. The allowable characters after the first include alphabetic characters, numeric digits, the underscore, the `@` sign , the question mark, the colon and the period in any order.

```
A-Z a-z 0-9 _ @ ? :
```

Here are a few *valid* labels.

- Port_A
- query???

- Two.by.two
- _AUTO

Here are a few *invalid* labels.

- ?HELP
- 2PI
- N*F*G
- on/off

Labels are limited in length to 12 characters. Labels which are longer than this will be truncated with an assembler warning. Assembly will not be terminated as a consequence of label truncation, but there is a possibility of incorrect treatment of long labels which are similar in the first twelve characters.

3.4.2 Operators

Operators are used in arithmetic or logical expressions to indicate which operations are to be performed on the associated operand(s). `cbA65` uses accepted operator precedence rules to correctly evaluate math expressions, regardless of complexity. Parentheses are supported so that ambiguous or conflicting precedence requirements can be resolved.

For example, unary minus is supported to allow the manipulation of negative numbers as in

```
NCOUNT  .equ -COUNT
```

The unary minus is tightly bound so the following two expressions involving the unary minus produce different results.

```
COUNT   .equ  63
NUM1    .equ  -COUNT-1
NUM2    .equ  -(COUNT-1)
```

In the above example, `NUM1` evaluates to -64, and `NUM2` evaluates to -62.

A summary of the precedence rules for all arithmetic, logical and relational operators can be found in Section 3.4.3.

Arithmetic Operators

The math (arithmetic) operators recognized by `cbA65` are `+ - * / %` which represent *add*, *subtract*, *multiply*, *divide* and *mod* operations, respectively. These are all binary operators which retain their ordinary meaning when applied to integers.[2] Note that *mod* returns the remainder after an integer division, and that an integer divide discards the remainder.

There are also a few unary math operators which can precede an argument or expression. These are `+ - ~` where `~` returns the one's complement of the expression which follows.

Bitwise Operators

Bitwise operators are binary operators which operate on each bit in an integer individually. They are `& | ^ SHL SHR` which represent *and*, *or*, *exclusive or (xor)*, *shift left* and *shift right*, respectively. Note that the math operator `~` may also be considered a unary bitwise operator, since it also operates on the individual bits of the argument which follows.

Logical Operators

Logical operators operate on boolean quantities which only take on two values. These two values are variously symbolized as *(1,0)*, *(yes,no)*, *(true,false)*, *(on,off)* depending on context. These operators are useful in decision and control logic and differ from the bitwise operators.

The binary logical operators are `AND OR XOR` . Truth tables defining their behavior are shown in Fig. 3.1. Although the logical operators are particularly useful in assemblers

AND

A	B	A · B
0	0	0
0	1	0
1	0	0
1	1	1

OR

A	B	A + B
0	0	0
0	1	1
1	0	1
1	1	1

XOR

A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

Figure 3.1: Logical Operators

which support conditional assembly they may also be used to communicate compile time details to the program. See next section for an example.

[2] An integer, in this context, is an 8-bit byte or 16-bit word.

Relational Operators

In addition to the math and logical operators there are operators which are referred to as *relational* operators. These operators return the result of comparing two arguments or expressions for a specific relation. The result returned is either 1 or 0 depending on whether the relation was or was not satisfied. There are no unary relational operators.

The relational operators are `LT LE EQ NEQ GE GT` , which test for the quantity to the left of it is *less than*, *less or equal*, *equal*, *not equal*,*greater or equal* and *greater than* the quantity to the right.

Consider the following:

```
lda #(tptr lt 128)
```

which will load the accumulator with '1' if the location of `tptr` is in the lower half of page zero, and '0' otherwise.

Special Unary Operators

cbA65 supports two special unary operators which are useful in an 8-bit processor environment with 16-bit addressing: < and >. These unary operators return the *low* or *high* bytes of a 16-bit argument, respectively. This is extremely useful when constructing pointers to addresses in zero page or elsewhere.

3.4.3 Precedence Rules

The operator precedence used by the `cbA65` parser

```
highest     ( )
            + - ~ < > NOT   (unary operators)
            * / %
            + -
            SHL SHR
            LT LE GE GT
            EQ NEQ
            &
            ^
            |
            AND
lowest      OR XOR
```

Figure 3.2: Operator Precedence Chart

3.4.4 Special Characters

Certain characters perform unique roles in determining the values of constants in expressions or opcodes.

The `#` character, often called a *hash mark* or *pound sign* is used to signify 'immediate' addressing mode. This is a disambiguating operator which tells the assembler that the numeric value following is an immediate mode operand and not a variable representing a memory location.[3]

The `<` and `>` operators signify that the *low* byte or *high* byte of the following symbol is to be taken and is used as an immediate operand. Since these are also a disambiguating operators, the `#` is not necessary and may be omitted.

As explained elsewhere, the `$` sign signifies a hexadecimal number, as opposed to an ordinary decimal number. Binary numbers are specified using the `%` prefix. Octal numbers are not supported.

The grave accent ` tells the assembler to use 16-bit(absolute) addressing modes instead of

[3]The default is to interpret any number or symbol representing a number as a memory location.

the 8-bit (zero page) modes.[4] This is used to override the default addressing mode when conditions require it. See Section 3.5.

3.5 Expressions

In this section, the types of expressions which the assembler handles are explained and some examples are given.

The simplest expressions involve the arithmetic operators. For example:

```
        lda numtable+8
```

will load the accumulator with the eighth byte of `numtable`.

To create a variable indicating the length of a string, simple place a label at the location following the string and write an appropriate expression.

```
strlen  .equ str2 - str1            ; get length of string 1
        .
        .
        .
str1    .str "This is a test string"
str2    .str "Another string."
        .
        .
```

Complex expressions may require parentheses to parse correctly.

```
nument  .equ (tblend-tblstart)/sz   ; get number of entries in table
magic   .equ (mnum ^ $ff) & $f      ; compute magic number
```

Numbers may be mixed with ASCII characters.

```
        lda #'A'+128                ; set high bit
```

[4] This symbol is usually located on the same key as the tilde ~ key, at the left of the keyboard.

Overriding Zero Page Opcodes

Many of the 65xx instructions exist in two versions, depending on whether the address in the operand can be specified with an 8-bit value or must be specified with a 16-bit value. The assembler determines the size of the operand and, if the quantity can be stored in 1 byte, will code the zero page version of these opcodes. The zero page versions are faster and shorter, so this strategy is generally optimal.

However, there are times when it is desirable to force the 16-bit version of the instruction, even though the operand does not require it. For example, in sections of self-modifying code it may be necessary to allow for 16-bit addresses even though the address may be initially in page zero. In other cases there may a cycle counting requirement which is better served with the 16-bit version, even though the address remains in page zero.

To allow for these special cases **cbA65** permits the programmer to force the 16-bit version by flagging the operand with a leading *grave* accent ‘.

```
000016 0213 A5 80            lda $80      ; allow zero page addressing
000017 0215 AD 80 00         lda ‘$80     ; force 16-bit address
```

This feature may also be useful when an address operand is specified using a pure number. For example,

```
000354 0433 A5 FE            lda -2       ; zero page default
000355 0435 AD FE FF         lda ‘-2      ; force absolute address
```

3.6 Assembler Directives

`.ALIGN` – Moves program counter to next power-of-two boundary and, optionally, fills space with desired character.

`.BYTE` – Places a list of 8-bit values at given address.

`.DS` – Reserves storage for specified number of bytes.

`.ECHO` – Prints text to terminal during Pass 2.

`.END` – Designates the end of the program.

`.EQU` – Assigns a numeric value to a symbolic label,

`.FILES` – Allows specification of output file options.

`.FILL` – Fills memory with a given number of given character.

`.INCLUDE` – Specifies include file name and location.

`.LIST` – Enables generation of list file (default: TRUE).

`.NOLIST` – Cancels `.LIST` option.

`.ORG` – Sets the location of the program counter.

`.PAGE` – Eject current page.

`.PRINT` – Allows specification of list printing options.

`.SKIPB` – Place opcode $24 (BIT Z.P.) in output file to cause CPU to skip next byte.

`.SKIPW` – Place opcode $2C (BIT ABSOLUTE) in output file to cause CPU to skip next word.

`.STRA` – Places a string at given address and adds carriage return/line feed terminator.

`,STRX` – Sets the high bit of the last character in the string.

`.STRZ` – Places a string at given address and adds NULL terminator.

`.TEXT` – Places an ASCII string at a given address.

`.TITLE` – Adds title strings to header of output listings.

`.WORD` – Places a list of 16-bit values at given address.

In addition to these directives, several synonyms are recognized. These are:

`.BY` – Same as `.BYTE`.

`.DB` – Same as `.BYTE`.

`.DW` – Same as `.WORD`.

`.STR` – Same as `.TEXT`.

`.WO` – Same as `.WORD`.

3.6.1 File Control

Every assembly generates at least one output file. There will always be a *filename.log* file associated with each run. The contents of this file are described in a Section 5.1. Several options are available for specifying output files with special file formats. The *list* file is important for aiding the programmer in developing, debugging and optimizing his code. Other file formats are needed to permit execution of the code in different environments. Common file formats are those used by hardware ROM and PROM programmers, or for loading images of the software into computers or emulators.

cbA65 supports the following output file formats:

`BIN` – A raw binary dump intended for ROM applications.

`H65` – A simple, text hex file format suitable for hand editing.

`H6X` – A CPU instruction oriented hex file format.

`INTHEX` – The Intel hex file format.

`MOSHEX` – MOS Technology hex file format.

`MOTSREC` – Motorola 'S' file format.

`XREF` – Program symbol cross reference file.

The list file is turned *on* or *off* with the `.LIST` or `.NOLIST` directive, respectively. These directives may be placed as desired in the source file. The default is to generate a list file, so unless the `.NOLIST` directive is encountered the full list file will written.

The other output file types are specified with the `.FILES` directive. Multiple selections are comma separated. Any or all of the file types can be output during an assembly. Details of each file format are in Chapter 5.

The syntax for the `BIN` file format is: BIN=ROMSIZE, where ROMSIZE is a numeric quantity or expression equal to some power of 2. **cbA65** will output a binary object file containing the code and data generated by the program. If the binary object file exceeds the specified ROM size, an error message will be issued. Otherwise the additional ROM space, if any, will be padded with zeros and the ROM file written.

An example is:

```
.files h65,mot,bin=2048
```

3.6.2 Listing Options

The basic control over the list file is managed by the `.LIST` and `.NOLIST` directives. These turn the listing output *on* or *off*, respectively.

Print Options

Listing format controls and options are specified following a `.PRINT` directive. The print options which pertain to the list file are:

`CHKSUM` – Print checksum over all object code at end of listing.

`COLS` – Specify maximum number of columns in listing. (Default is 78)

`CSORT` – Sort opcode statistics alphabetically or by frequency.

`CYCLES` – Print CPU cycle count at each instruction.

`ROWS` – Specify maximum number of rows per page in listing. (Default is 60)

`SRCLINES` – Print line numbers of source code lines or list lines. (Default is to print line number of source files)

`SSORT` – Sort symbols alphabetically or by line number.

`STATS` – Print CPU statistics to log file.

`XREF` – Generate cross reference file.

Here is an example of a directive used to adjust the page format.

```
.print cols 118,rows 66
```

Note that the `.PRINT` directive supports options to specify the number of columns and number of rows in each page of the listing. The `COLS` and `ROWS` options each take a single parameter, which may follow an ASCII space ␣ or equal sign =.

The basic output listing format attempts to provide an enhanced version of the original source code with the object code and critical symbol values printed. A typical listing format follows this scheme:

```
 LINE       CURRENT  CODE        SOURCE
 NUMBER     ADDRESS  (DATA)      LINE.............................

 00000345      0204  A9 45       wstrt  lda #$45  ; do a warm start
```

The 'LINE NUMBER' refers the to current line in the active source file. 'CURRENT ADDRESS' indicates the value in the program counter, 'CODE (DATA)' shows the opcode and operands or other data associated with the source line. 'SOURCE LINE' is simply a copy of the (possibly truncated) line in the source file.

A variation on the above format provides support for include files. For example,

```
 LINE       CURRENT  CODE        SOURCE
 NUMBER     ADDRESS  (DATA)      LINE.............................

 000411        0204  =(516)           .include regtbl.inc
i100001        0204  00 00 00 00 r1  .byte 0,0,0,0   ; temporary storage
i100002        0208  00 00 00 00 r2  .byte 0,0,0,0
i100003        020C  00 00 00 00 r3  .byte 0,0,0,0
 000412        0210  4C 00 08        jmp init_regs
```

This strategy aids in identifying the exact line under assembly by the currently active file. The format uses a lower case `i` in the leftmost column to flag the processing of an included file. The ID number next to the `i` is the nesting level for this file (1-9). The remaining numbers identify the actual source line number of the current line in the included file. When the file has been read, the line numbers revert back to those in the calling file. Notice that this format makes it easy to debug from the listing, because there is no ambiguity in the location of any line.

Include files may be nested up to nine levels deep without disturbing the listing format. When the assembler detects errors, it can report their location by filename and line number. See Section 4.1.

`SRCLINES`

This option controls whether the line number printed on the left side of the listing are listing line numbers or source line numbers. The following two examples illustrate the difference. In the first example, the lines are numbered by listing line.

```
 lines.cba   cbA65 v.1.00a, Feb 27 2008  -- Wed Feb 27 18:02:42 2008
```

```
000001                  ;
000002                  ;   lines.cba -- test for srclines listing option
000003                  ;
000004                          .print srclines=n
000005 0400                      .org $400
000006                  ;
000007                  ;   string storage
000008                  ;
000009 0400 43 6F 63 6B msg1    .text "Cockpit error."
000010 0404 70 69 74 20
000011 0408 65 72 72 6F
000012 040C 72 2E
000013 040E 42 61 62 79 msg2    .text "Baby on board."
000014 0412 20 6F 6E 20
000015 0416 62 6F 61 72
000016 041A 64 2E
000017                          .end

NUMBER OF SYMBOLS: 2
```

In the next example, the lines are listed by source file line.

```
 lines.cba   cbA65 v.1.00a, Feb 27 2008  -- Wed Feb 27 18:01:29 2008


000001                  ;
000002                  ;   lines.cba -- test for srclines listing option
000003                  ;
000004                          .print srclines=y
000005 0400                      .org $400
000006                  ;
000007                  ;   string storage
000008                  ;
000009 0400 43 6F 63 6B msg1    .text "Cockpit error."
       0404 70 69 74 20
       0408 65 72 72 6F
       040C 72 2E
000010 040E 42 61 62 79 msg2    .text "Baby on board."
       0412 20 6F 6E 20
       0416 62 6F 61 72
       041A 64 2E
```

```
 000011                             .end

NUMBER OF SYMBOLS: 2
```

Either one of these formats may used, depending on the preference of the programmer. The default is to print source line numbers.

STATS

This print option directs the assembler to print CPU statistics to the log file. These statistics consists of opcode instruction counts, sorted alphabetically or by frequency. An example of a log file with statistics include is shown below.

```
 kimath.cba   cbA65 v.1.00a, Feb 12 2008  -- Tue Feb 26 12:56:55 2008

Pass 1: OK
Pass 2: OK
Pass 3: OK
Pass 4: OK

TITLE:  KIMATH ROUTINES, MOS TECHNOLOGY

ROMSIZE = 2048, LOROM = 63488, HIROM = 65518
NUMBER OF SOURCE LINES: 1252
NUMBER OF LISTING LINES: 1289
NUMBER OF SYMBOLS: 238
NUMBER OF UNREFERENCED LABELS: 34
NUMBER OF REFERENCES: 586
NUMBER OF ANONYMOUS LABELS: 0
NUMBER OF ASSEMBLER WARNINGS: 0
MAX. INCLUDE FILE NESTING LEVEL: 0
NUMBER OF CPU INSTRUCTIONS: 824

OPCODE USAGE SUMMARY (SORTED BY OPCODE COUNT):

MNEMONIC OPCODE COUNT      MNEMONIC OPCODE COUNT      MNEMONIC OPCODE COUNT
---------------------      ---------------------      ---------------------
JSR nn     20    0114      LDA #n     A9    0065      RTS        60    0048
STA nn     8D    0042      STA n      85    0041      LDA nn     AD    0036
BNE n      D0    0032      DEX        CA    0030      JMP nn     4C    0028
STA nn,X   9D    0027      BPL n      10    0026      LDA n      A5    0025
BEQ n      F0    0024      LDA nn,X   BD    0023      LSR        4A    0018
```

```
LDX n       A6    0018      LDX #n      A2    0013      AND #n      29    0012
BCC n       90    0011      INY         C8    0010      SEC         38    0010
LDA (n),Y   B1    0009      ADC #n      69    0009      INX         E8    0009
BIT nn      2C    0009      ASL         0A    0008      STA (n),Y   91    0007
BVC n       50    0006      BVS n       70    0006      DEC n       C6    0006
BCS n       B0    0006      SED         F8    0006      CLC         18    0005
SBC #n      E9    0005      BMI n       30    0005      INC n       E6    0005
LDY #n      A0    0004      ASL n       06    0004      PHA         48    0004
PLA         68    0004      SBC nn      ED    0004      ORA #n      09    0004
BIT n       24    0004      CPY n       C4    0003      EOR #n      49    0003
ORA nn      0D    0003      CMP #n      C9    0003      ADC nn      6D    0002
LDY n       A4    0002      CPX n       E4    0002      AND nn      2D    0002
STY n       84    0002      CMP n       C5    0002      EOR nn      4D    0002
TAX         AA    0002      STX n       86    0002      SBC nn,X    FD    0002
LDY nn,X    BC    0001      CLD         D8    0001      STA nn,Y    99    0001
ORA n       05    0001      ADC nn,X    7D    0001      ORA nn,X    1D    0001
ADC n       65    0001      TYA         98    0001      LDA nn,Y    B9    0001
CMP nn      CD    0001
```

`CSORT`

The chart of the opcode usage was sorted by frequency of use. It is also possible to sort the list alphabetically using the `CSORT` option of the `.PRINT` directive. `CSORT` takes a parameter 'A' or 'F' which specifies to sort alphabetically or by frequency. The default is to sort by frequency. Syntax for the `CSORT` option of the `.PRINT` directive is `CSORT=A` for alphabetical sort.

`CHKSUM`

This option for the `.PRINT` directive causes a checksum to be appended to the listing file. The checksum consists of the least significant 4 hex digits of the simple sum of all object code generated during the assembly.

3.6.3 Data Storage

8-Bit Numeric or Character Constants

There are several directives which can be used to allocate storage. Data types which may be placed at desired locations in the program space include 8-bit bytes, 16-bit words, 8-bit ASCII characters and strings.

The basic directive which applies to 8-bit numeric quantities is written `.BYTE`. It may also be specified using `.BY` or `.DB`. An example is:

```
mul3tbl     .byte 0,3,6,9,12,15,18,21,24        ; multiples of 3
            .byte 27,30,33,36
```

Bytes may be specified using hex notation or ASCII characters, as well.

```
hex2asc     .by '0','1','2','3','4','5','6','7'
            .by '8','9','A','B','C','D','E','F'
lohi        .by $0,$10,$20,$30,$40.$50,$60,$70
```

16-Bit Values

For 16-bit numeric quantities, use the `.WORD` directive. The assembler will also accept `.WO` and `.DW` for 16-bit quantities.

```
jmptbl      .dw add,sub,mul,div
lval        .dw 'A','B','C',0,1,2,3
magic       .dw $1234
```

Note that program addresses may be declared and will be stored in low, high order. If 8 bit quantities, such as ASCII characters, are declared, they will be expanded to 16 bits.

ASCII Strings

The directives `.TEXT` and `.STR` are used to embed ASCII character strings in the program. Upper case and lower case values are retained in the resulting code.

For some applications, it is useful to terminate the strings with a more-or-less standard terminator. A common termination consists of a carriage return (ASCII value: 13) followed by a line feed (ASCII value: 10). This termination can be automatically appended to the string by invoking the `.STRA` directive followed by the desired text enclosed in quotes.

Another common termination strategy uses a simple ASCII NULL (ASCII value: 0) following the text. This can be automatically appended by invoking the `.STRZ` directive.

String lists of program keywords are sometimes coded with the last character in each word altered by setting the high bit to '1'. For example, the character 'A', which is 65 ($41) in the ASCII symbol set becomes 65 + 128 = 193 ($C1). This technique facilitates counting or tokenizing the keywords. **cbA65** automates this method with a special string directive `.STRX`.

Other termination schemes can be implemented by the programmer by simply outputting the raw ASCII string and appending the desired terminator(s) using the `.BYTE` directive.

Fill Directive

The `.FILL` directive is used to fill a block of memory with a user specified byte value or ASCII character. This directive takes two parameters: first, a count of the number of bytes to fill and, second, the value of the fill byte. Either of these quantites may be defined by expressions, as in this example.

```
        .org $700
        .fill 16,$ea
thisloc .fill thatloc-thisloc,'A'
        .align 64
thatloc brk
```

The result of using these directives is shown below. Note that the `srclines=y` printing option was in effect.

```
 000069 0700 EA EA EA EA         .fill 16,$ea
        0704 EA EA EA EA
        0708 EA EA EA EA
        070C EA EA EA EA
 000070 0710 41 41 41 41 thisloc .fill thatloc-thisloc,'A'
        0714 41 41 41 41
        0718 41 41 41 41
        071C 41 41 41 41
        0720 41 41 41 41
        0724 41 41 41 41
        0728 41 41 41 41
        072C 41 41 41 41
        0730 41 41 41 41
        0734 41 41 41 41
        0738 41 41 41 41
```

```
       073C 41 41 41 41
000071 0740                    .align 64
000072 0740 00          thatloc brk
```

3.6.4 Alignment

It is often necessary to force data blocks into alignment with specific address boundaries. The simplest example for the 65xx processors is to require alignment to *even* addresses for indirect jump tables. This strategy assures that no two-byte address will cross a page boundary.

The `.ALIGN` directive takes a single parameter which forcesthe program counter to advance to the next aligned value. This parameter must be a power of two. For the previous example, `.ALIGN 2` will force the address to the next even address. If the current address is already even, no action is taken.

Another example occurs when a table of values should begin on a 256-byte page boundary so that the index does not cross pages. We see this most often when access to tables must take place under cycle count control. Use `.ALIGN 256` to apply this directive.

`.ALIGN` also allows for a second parameter which is used to specify a fill byte. If no second parameter is entered, the program counter is simply moved to the new, aligned location. If it is necessary to fill the space between location of the current program counter to the new location, the following syntax is used:

```
.align 256,$ea
```

3.6.5 Cycle Counts

Cycle counts for each instruction can be included in the listing file when it is desired by the programmer. It is an option for the `.PRINT` directive, as in the following:

```
.print cycles
```

A sample listing with cycle counts is shown in the following code snippet.

```
000680                    .print cycles
000681                    ;
```

```
000682                        ;   Right shift rb cnt times.
000683                        ;
000684 FBEE A5 03       ~3  rsbcnt  lda cnt
000685 FBF0 F0 29       ~2=         beq rbofe
000686 FBF2 A6 00       ~3          ldx n
000687 FBF4 BD 12 02    ~4, rsbc    lda rb,x
000688 FBF7 9D 13 02    ~5          sta rb+1,x
000689 FBFA CA          ~2          dex
000690 FBFB 10 F7       ~2-         bpl rsbc
000691 FBFD A9 00       ~2          lda #0
000692 FBFF 8D 12 02    ~4          sta rb
000693 FC02 C6 03       ~5          dec cnt
000694 FC04 D0 E8       ~2=         bne rsbcnt
000695 FC06 60          ~6          rts
```

The cycle counts are identified by a preceding tilde ~ symbol. Following the base count value may be a symbol representing additional cycles which are incurred by indexing or branching across a page boundary.

Branching Information Symbols

Although the assembler has no way of knowing whether a branch will always take place, it can tell whether a page crossing will be involved. Hence, the symbol following the base cycle value for a branch instruction will consist of a hyphen - or equal sign = . These indicate whether there will be an additional one cycle or two cycle penalty for taking the branch. All cycle counts for branch instructions will be followed by a hyphen or equal sign. Use the base count when no branch occurs and add additional one or two counts as appropriate when calculating the total cycle count for the branch.

Indexing Information Symbols

There is also a penalty for indexing across a page boundary for some instructions.[5]

The assembler cannot determine the run-time index values so it cannot tell whether the page-crossing penalty will always be invoked. However, it uses a heuristic argument to

[5]The indexed `ASL, DEC, INC, LSR, ROL, ROR, STA, STX, STY` instructions do not incur page crossing penalties.

signal to the programmer that a page crossing is likely or unlikely when the base address is known, as in the absolute indexed operations.

Since the index is always regarded as a positive number by the CPU, page crossings will always be in the forward (upward in memory) directions. Note that if the base address is on a page boundary, *e.g.*, `$0800` no page crossing can ever occur. In fact, the likelihood of incurring a page crossing penalty is increased as the base address moves to a higher level within a page. The assembler resolves base addresses during the assembly and can provide an indicator to the programmer when the base address provides a large or small headroom for indexing. `cbA65` does this by placing a comma , or apostrophe ' after the base cycle count. The comma indicates at least a half a page of headroom, and the apostrophe indicates less than half a page of headroom. Hence, a comma means that a page crossing penalty is unlikely, and an apostrophe means it is likely.

This indicator should not be taken too literally, as individual cases will determine actual performance. However, you'll find that quick scan of a listing can alert you to indexing situations that may benefit from data relocation or realignment of tables. In any case, the strategy fails for *indirect* indexing because the base address is not known at compile time. For these instructions no penalty assessment is attempted.

Summary of Symbols Used in Cycle and Branch Counting

The following table shows all special symbols used to support cycle counting and their meanings:

Symbols	Meaning
~	Base cycle count follows (one digit)
-	Branch requires 1 additional cycle
=	Branch across page boundary requires 2 additional cycles
'	Indexing over a page boundary is likely requiring an additional cycle
,	Indexing over a page boundary is unlikely

3.6.6 Miscellaneous Directive Examples

`.TITLE`

The `.TITLE` directive provides the programmer with a means to specify the header printed on each page of the listing. An example of its use is

```
    .title "Floating Point Routines by J. Slick, (C) 2002"
```

where all characters, including the quotes will be printed in the header of each listing page.

If no quotes are used delimit the title string, it will be converted to uppercase in each header.

`.EQU`

This assignment directive is used to associate labels with numeric values. The directive may be followed by any valid expression including numbers operators or other symbolic labels. A few examples are:

```
portA   .equ $d840
ptr1    .equ $88
maxbyte .equ 255
tmplo   .equ $420
tmphi   .equ tmplo+1
```

When these lines are written to the list file, `cbA65` will print both the hex and decimal values of the labels.[6]

`.ECHO`

The `.ECHO` directive provides a means to trace the progress of an assembly. During pass 2, and only pass 2, the `.ECHO` directives are processed. Any text which follows this directive will be printed to the screen and in the log file at this time. Hence, it may be used to place visible progress markers in the file.

Example:

```
        .echo "start of tables"
```

`.PAGE`

`.PAGE` causes the listing file to terminate the current page, print the footer and begin a new page. This formatting directive may be useful to separate code or data sections in the printout.

[6] Any label which is defined on a line with no output code will be displayed in the listing file with both hex and decimal values.

`.SKIPB` and `.SKIPW`

These directives use a common trick to provide multiple entry points to a routine with different parameters preselected. The idea is based on the observation that the 65xx `BIT` instructions set flags but otherwise do not modify registers. They exist in zero page and absolute versions. This example of a program listing shows how the directives might be used to print a carriage routine or line feed to a printer.

```
000058 0233 A9 0D       prtcr   lda #cr
000059 0235 2C                  .skipw
000060 0236 A9 0A       prtlf   lda #lf
000061 0238 20 41 07            jsr prtchr
```

In this example entry at `prtlf` will output a line feed. Entry at `prtcr` will output a carriage return, but will not output a line feed. This is because the `.SKIPW` directive has placed a `BIT` absolute instruction which interprets the next two bytes as an address to read, effectively skipping the `lda #lf` instruction. This method can be cascaded to provide multiple entry points.

Caution: this technique is not entirely free of questionable side effects. It is possible that act of reading an arbitrary address could be interpreted by a hardware location as part of a handshaking protocol. The programmer should assume responsibility of examining the listings to ensure that no improper memory accesses will occur.

The possibility of problems is extremely low, and the caution cited should not discourage the use of this directive.

`.DS`

This directive reserves storage by advancing the program the given number of locations. It takes a single parameter specifying the number of bytes to reserve. The following example shows how a group of 8-byte registers might be allocated.

```
; reserve three working registers for floating point ops
wr1     .ds 8
wr2     .ds 8
wr3     .ds 8
```

3.7 Location Counter (Program Counter)

The location or program counter can be accessed at any point in a program with the symbol `*`. This symbol allows the counter to be modified or assigned to a label. It is possible to use the `*` symbol in place of an `.ORG` directive as shown in the following snippet.

```
        * = $C0
ptr1    * = *+2
ptr2    * = *+2
        * = $200
        lda #0
        rts
```

Note that the symbol for the location counter, which specifies the current address, is the same as the symbol for multiplication. No ambiguity exists, however, because multiplication only occurs within an expression, and the location counter is only referenced immediately before or after an equal sign.

3.8 Branch Targets

Branch targets may be specified with a symbolic memory reference or label. Provided the target is within the branch range of the CPU you may specify any labeled address. An error will be reported if the target is out of range.

You may also specify the target address with an immediate value giving the offset of the address from the *next* CPU instruction. This requires prefixing a `#`symbol before the numeric value. A numeric value without the immediate mode prefix is an error.

Recall that with relative branching, a value of zero `(0)` refers to the address of the next instruction. An infinite loop will result if the branch is ever taken in the following code, because it branches back to same instruction:

```
        bne #$fe        ; trap!!!
```

3.8.1 Anonymous Labels

Anonymous labels are also supported. These labels are identified by using the *at* sign `@` as a label. They may be placed at any address containing an executable CPU opcode.

Anonymous labels are referenced by using an `@F`, for a forward branch, or `@B` for a backward branch as an operand for the branch instructions.

Anonymous labels provide a means to identify memory addresses without the need to invent a symbolic name for each location. This capability is useful for code that involves many compare and branch loops which force the programmer to resort to using arbitrary numbers in labels to distinguish them from each other.

The special character, `@`, is used to flag the current memory location to the assembler. These labels are accessed by tracking their proximity to a nearby branch instruction reference. For example, `bne @F` will access the first anonymous label, `@`, following it. Similarly, `bcs @B` will access the first anonymous label preceding it.

An extension to the use of anonymous labels allows skipping a number of labels in either direction by appending a numeric value after the branch direction identifier (`F` or `B`).

Here is some sample code showing a possible application of anonymous labels:

```
; convert HEX digit in lower nibble to ASCII
cnvlo ldx #7
@     lda r1,x
      and #$0f
      ora #$30
      cmp #$3A      ; numeral?
      bcc @F        ; yes, no correction needed
      adc #6        ; convert to ASCII A-F (carry is set)
@     jsr prtchr
      dex
      bpl @B1
      rts
```

While it would be a bad programming practice to allow large distances between branch instructions or complicated interlacing of the anonymous labels they reference, short code sections such as the above relieve the programmer of having to invent symbolic names simply to appease the assembler. The basic rule is to use anonymous labels only when the branch targets are nearby and easily visible on the screen or printout.

It is worth noting that some programmers rely heavily on meaningful labels as a substitute for the liberal use of comments to explain their code. This is hardly optimal because of the terse form of symbolic labels. With anonymous labels there isn't much choice for documenting the code except to add comments. This is probably for the better, because labels such as `PLOADY`, `UGTRES0`, etc. quickly lose their significance.

To minimize the potential for abusing this feature, only branching and not jumping to anonymous labels is supported.

3.8.2 Range Errors

Range errors only occur when branch instructions attempt to access a program location which is beyond reach from the current address. Recall that a branch distance of `0` simply targets the next instruction following the branch instruction. The branch distance is a signed quantity and can extend from −128 bytes to 127 bytes from the address of the next instruction. If the distance to the target exceeds the allowable range, a *range error* will be reported by the assembler. See Chapter 6 for more information on assembler error reporting.

Note that if the branch operand is entered as an 8-bit immediate value, instead of a symbolic one, no branch error should occur.

Chapter 4

Source Files

Assembler source files are ASCII text files which comply with the syntax rules for **cbA65**. The filename must have a `.cba` extender. The two requirements for all source files are that the file must contain an `.ORG`[1] directive before generation of any code or data and must contain an `.END` directive at the end.

Although each program will be associated with a single master source file, there may be any number of other files included to complete the program. This simplifies the sharing of critical modules among different programs. See Section 4.1 for a further discussion on include files.

4.1 Include Files

The syntax requirements for include files are somewhat relaxed from those of source files. First, no restrictions on the filename or its extender are imposed. Second, include files occupy a limited space within the source file and only need to comply with the syntax requirements of the source file over that region. In other words, if the source file already has specified the value of the program counter, no `.ORG` directive is required in the include file. On the other hand, the include file may contain any valid directives or instructions.

Include files may be nested up to 9 levels. This should be sufficient for any reasonable application. **cbA65** keeps track of the nesting levels and the filenames at each level and can report any errors it detects by filename and file line number.

[1] It is possible to substitute a program counter assignment for the `.ORG` directive.

Here is an example of a simple program which uses include files nested to 2 levels.

```
;
;   nestinc.cba -- example of nested include files.
;
        .org $200
        cmp #0
        bne @f
        jmp fadd
@       cmp #1
        bne @f
        jmp fsub
@       brk
;
;   number storage
        .org $400
        .include mathcon.inc
fadd    brk
fsub    brk
        .end
```

This is the contents of `mathcon.inc`,

```
k0      .byte $00,$00,$00,$00,$00,$00,$00,$00
k1      .byte $00,$00,$10,$00,$00,$00,$00,$00
        .include pi.inc
sq2     .byte $00,$00,$14,$14,$21,$35,$62,$37
```

and this is the contents of `pi.inc`.

```
pi      .byte $00,$00,$31,$41,$59,$26,$53,$59
```

The listing file, with the default `srclines=y` printing option, looks like this.

```
 nestinc.cba   cbA65 v.1.00a, Feb 28 2008  -- Thu Feb 28 08:59:11 2008


 000001                  ;
 000002                  ;   nest0.cba -- example of nested include files.
```

```
 000003                  ;
 000004 0200                     .org $200
 000005 0200 C9 00               cmp #0
 000006 0202 D0 03               bne @f
 000007 0204 4C 20 04            jmp fadd
 000008 0207 C9 01       @       cmp #1
 000009 0209 D0 03               bne @f
 000010 020B 4C 21 04            jmp fsub
 000011 020E 00          @       brk
 000012                  ;
 000013                  ;   number storage
 000014 0400                     .org $400
 000015                          .include mathcon.inc
i100001 0400 00 00 00 00 k0      .byte $00,$00,$00,$00,$00,$00,$00,$00
        0404 00 00 00 00
i100002 0408 00 00 10 00 k1      .byte $00,$00,$10,$00,$00,$00,$00,$00
        040C 00 00 00 00
i100003                          .include pi.inc
i200001 0410 00 00 31 41 pi      .byte $00,$00,$31,$41,$59,$26,$53,$59
        0414 59 26 53 59
i100004 0418 00 00 14 14 sq2     .byte $00,$00,$14,$14,$21,$35,$62,$37
        041C 21 35 62 37
 000016 0420 00          fadd    brk
 000017 0421 00          fsub    brk
 000018                          .end

NUMBER OF SYMBOLS: 6
```

It is easy to follow the listing to determine which file is active and on which line each entry is placed.

Chapter 5

Output Files

In this chapter, a more complete discussion of the output files generated by **cbA65** is provided. Each of the file types is taken in order its major features are explained.

5.1 Log File

A log file is written every time **cbA65** is run on a source file. The log file associated with `myfile.cba` is `myfile.log`. Pass numbers along with errors and warnings are written to the log file for later reference.

In the simplest cases with a successful assembly, the log file will consist of verifications for each pass followed by a report of some program statistics. An example log file looks like:

```
 upper.cba   cbA65 v.1.00a, Feb 28 2008  -- Fri Feb 29 12:41:10 2008

Pass 1: OK
Pass 2:
    WARNING! No .END directive. Assuming End-Of-File.
Pass 3: OK
Pass 4:
    WARNING! Indirect operand crosses page in line 30 of file upper.cba
        jmp (badaddr)
       ADDRESS = $5FF
TITLE:  "Test program with various snippets", NOT TO BE PUBLISHED
```

```
CODE CHECKSUM: 0x65F4
NUMBER OF SOURCE LINES: 76
NUMBER OF LISTING LINES: 117
NUMBER OF SYMBOLS: 22
NUMBER OF UNREFERENCED LABELS: 10
NUMBER OF REFERENCES: 23
NUMBER OF ANONYMOUS LABELS: 2
NUMBER OF ASSEMBLER WARNINGS: 2
MAX. INCLUDE FILE NESTING LEVEL: 1
NUMBER OF CPU INSTRUCTIONS: 28
```

Notice that this log file contains two warnings. The first warns that no `.END` directive was found. The second is that an indirect address crossing a page boundary was detected.

5.2 Program Listing

The program listing file is generally the most important of the output files written by any assembler. It links the programmer's source code with the machine code generated by the assembler. It is invaluable for analyzing the output code, for detecting and correcting errors and for general debugging or optimizing.

5.2.1 Listing File Format

Fig. 5.1 shows the format of a page of an output listing file.

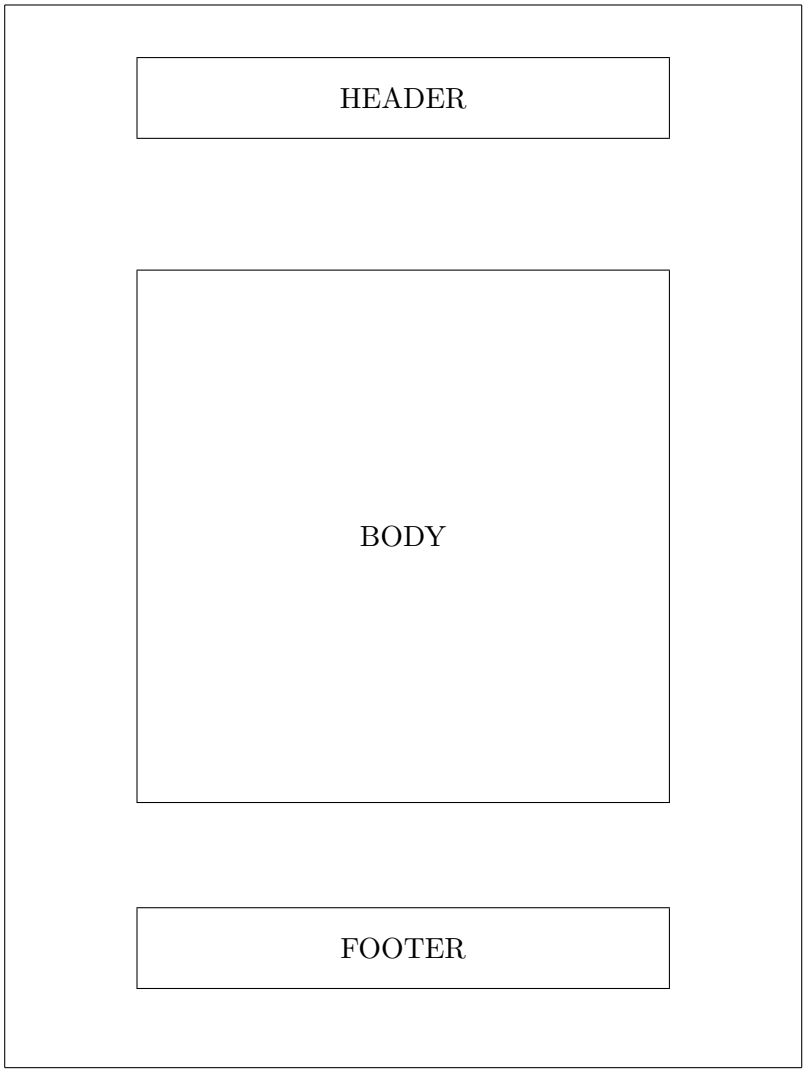


Figure 5.1: Listing File Page Format

"BASIC Programmer's Toolkit for Commodore Pet BASIC v2.0"

```
000686                  ;   string for a match with one of the Toolkit commands. The count of the
000687                  ;   found command is used to index into a jump table, (TKADDRHI & TKADDRLO)
000688                  ;
000689 B2DB A2 FF       FNDCMD      ldx #$FF
000690 B2DD E8          @           inx                ; Scan buffer.
000691 B2DE BD 00 02                lda INBUFF,x
000692 B2E1 30 CC                   bmi MODCHK
000693 B2E3 C9 20                   cmp #' '
000694 B2E5 F0 F6                   beq @B             ; Skip spaces.
000695 B2E7 B9 1A B3    CMDLP       lda CMDLIST,y      ; Get next character from
000696 B2EA F0 C3                   beq MODCHK         ;  list of Toolkit cmds.
000697 B2EC 5D 00 02                eor INBUFF,x
000698 B2EF D0 04                   bne @F
000699 B2F1 C8                      iny                ; Bump indices on every
000700 B2F2 E8                      inx                ;    matched character.
000701 B2F3 10 F2                   bpl CMDLP
000702 B2F5 C9 80       @           cmp #$80           ; Accept keyword if mismatch
000703 B2F7 F0 0A                   beq GOTTKN         ;    is caused by terminator.
000704 B2F9 C8          @           iny                ; Otherwise skip characters
000705 B2FA B9 19 B3                lda CMDLIST-1,y    ;    to next keyword.
000706 B2FD 10 FA                   bpl @B
000707 B2FF E6 05                   inc COUNT          ; Update keyword counter.
000708 B301 D0 D8                   bne FNDCMD         ; Branch always (try next cmd).
000709 B303 E6 77       GOTTKN      inc TXTPTR         ; Bump TXTPTR past this
000710 B305 CA                      dex                ;  keyword. No need to bump
000711 B306 D0 FB                   bne GOTTKN         ;  high byte (always $02).
000712 B308 A6 05                   ldx COUNT
000713 B30A E0 02                   cpx #$02
000714 B30C 30 02                   bmi @F
000715 B30E 68                      pla
000716 B30F 68                      pla
000717 B310 68          @           pla
000718 B311 BD 4E B3                lda TKADDRHI,x     ; Execute Toolkit command.
000719 B314 48                      pha
000720 B315 BD 59 B3                lda TKADDRLO,x
000721 B318 48                      pha
000722 B319 60                      rts
000723                  ;
000724                  ; This is the Toolkit keyword (command) list. The last character in each
000725                  ; string has the high bit set to facilitate keyword counting.
000726                  ;
000727 B31A 52 55 CE    CMDLIST     .strx 'RUN'
000728 B31D 41 55 54 CF             .strx 'AUTO'
000729 B321 53 54 45 D0             .strx 'STEP'
000730 B325 54 52 41 43             .strx 'TRACE'
       B329 C5
000731 B32A 4F 46 C6                .strx 'OFF'
000732 B32D 52 45 4E 55             .strx 'RENUMBER'
       B331 4D 42 45 D2
000733 B335 44 45 4C 45             .strx 'DELETE'
       B339 54 C5
000734 B33B 48 45 4C D0             .strx 'HELP'
000735 B33F 46 49 4E C4             .strx 'FIND'
```


SAMPLE PAGE OF LISTING FILE

The header consists of an information line which documents the source file name, the assembler version and the time and date. It also includes the title, as defined by the `.TITLE` directive.

The footer consists of the current page number.

5.3 Cross Reference File

A cross reference file can be specified following either the `.FILES` or `.PRINT` directive by including the `XREF` option. For example, `.print xref` will cause the assembler to generate a cross reference file `*.xrf` which shows the line number where each symbolic label was defined and all lines which reference this label.

A snippet from a cross reference file generated during assembly of the `mathpac` routines.

```
mathpac.cba   cbA65 v.1.00a, Jan  5 2008  -- Sun Jan 06 07:43:19 2008

MATHPAC: A KIMATH SUPPLEMENT

SYMBOL SYMBOL       SYMBOL       SRC LINE   CROSS
NUMBER NAME         VALUE         DEFINED   REFERENCES

00000  ABS        = 13587 ($3513)    709*   422   997
00001  ACOS       = 13374 ($343E)    620*   999
00002  ADD        = 63496 ($F808)     70*   394   418   462   475   527
                                            567   581   595   627   646
                                            799   969
00003  ADDL       = 12799 ($31FF)    364*   356
00004  ADDM       = 12804 ($3204)    367*   314   334
00005  ALOG       = 13049 ($32F9)    483*   731   1001
00006  ALOG1      = 13061 ($3305)    488*   498
00007  ALOG2      = 13072 ($3310)    493*   484   487
00008  ALOG3      = 13125 ($3345)    518*   514
00009  ARC1       = 13538 ($34E2)    689*   687
00010  ARC2       = 13541 ($34E5)    690*   678
00011  ARCSET     = 13513 ($34C9)    677*   620   628
00012  ARGYH      = 9     ($0009)     19*   775   776   845
00013  ARGYL      = 8     ($0008)     18*   93    101   145   160   169
                                            772   773   842   859   945
```

Note that the symbol address is printed in both decimal and hex for the convenience of the programmer. Line numbers are all decimal.

The above listing shows the symbols sorted in alphabetical order. However, it is also possible to specify sorting by the line number at which they were defined using the `SSORT` option following the `.PRINT` directive.

The syntax for SSORT is: SSORT = char, where char is 'A' or 'L'. 'A' stands for alphabetical symbol sort in the cross-reference file (*.xrf), and 'L' stands for sorting by line number where the symbol is defined. The default is to sort alphabetically.

5.4 Object files

In this section, a brief description of each of the object file types is given. These optional additional files are specified as options to the `.FILES` directive.

5.4.1 H65 Format

This file type is intended to provide a printable version of the binary object file produced by that assembler. It is a simple, ASCII hex translation of the object file which can be easily edited by hand.

The file format is line oriented as shown in the following layout.

```
ADDR             DATA
4 hex chars      16 pairs of hex chars
```

The address to load the following line of data or code is given as a 16-bit address expressed as a 4-character hex number. The data consists of 8-bit bytes expressed as 2-character hex numbers.

Here is an example of a typical *.h65 file.

```
B000 4C 7F B2 A9 0A 8D E2 03 A9 00 8D E3 03 85 83 85
B010 7C 85 81 A9 64 85 82 85 80 60 20 70 00 F0 32 B0
B020 17 20 73 C8 48 A5 12 A6 11 85 83 86 82 85 81 86
B030 80 68 F0 1D C9 2C F0 03 4C 03 CE 20 70 00 B0 F8
```

No data type identification or control information is included in the file. No checksum is present, so unrestricted editing is allowed.

Use `H65` as an option in the `.FILES` directive to create this file.

5.4.2 H6X Format

This format is similar to the H65 format, except that the object code bytes are grouped according to CPU opcodes. That is, if an opcode is output it is placed on its own line after the memory address. All operands are placed on the same line. Data bytes are listed individually one per line.

A portion of the H6X file corresponding to the previous file is given below.

```
B000 4C 7F B2
B003 A9 0A
B005 8D E2 03
B008 A9 00
B00A 8D E3 03
B00D 85 83
B00F 85 7C
B011 85 81
B013 A9 64
B015 85 82
B017 85 80
B019 60
```

Use `H6X` as an option in the `.FILES` directive to create this file.

5.4.3 Intel Hex Format

The Intel Hex file format was designed to facilitate the loading of binary object files into RAM or other programmable memory space for use by processors, emulators, EPROM programmers, etc. It provides an image of the binary file using only ASCII characters and simplifies the transfer of those images from one device to another.

A complete description of the Intel Hex Format is given in Appendix F.

Use `INTHEX` as an option in the `.FILES` directive to create this file.

5.4.4 MOS Technology Hex Format

MOS Technology developed a file format which is similar in purpose and structure to the Intel Hex format. It differs primarily in the calculation of the checksum for each line in the file.

Although this file format is little used, it is included for completeness. See Appendix G for the file spec.

Use `MOSHEX` as an option in the `.FILES` directive to create this file.

5.4.5 Motorola S Record Format

The Motorola S-Record format is another printable file format which translates a binary object file into an ASCII character form which is readable and easily transferred to other systems. It is primarily intended for use in emulators and by EPROM programmers.

See Appendix H for complete details.

Use `MOTSREC` as an option in the `.FILES` directive to create this file.

5.4.6 Binary ROM Image Format

This format produces a binary image of the program. It is the programmer's responsibility to set the initial `.ORG` location to the appropriate value so the code symbolic addresses are correct for the application. The file is padded up to nearest power of two, so the file length matches typical ROM sizes. The maximum file size is 32768 bytes.

For ROM applications external symbolic addresses should be specified with the `.EQU` directive, so that no memory space is allocated. For example,

```
porta   .equ    $C000
```

will identify a port address without allocating any storage for it.

To create a binary object file suitable for use in a ROM or PROM use `BIN=ROMSIZE` as an option in the `.FILES` directive, where `ROMSIZE` is a number or expression equal to the binary image size required. `ROM` is a synonym for `BIN` and may used instead.

Chapter 6

Error and Warning Messages

The following error messages are supported by `cbA65`:

- Syntax error.
- Symbol not found.
- Divide by zero.
- Bad expression.
- Unmatched parentheses.
- Duplicate label.
- Bad operation.
- Symbol table full.
- Wrong size for type.
- Branch out of range.
- Invalid quantity.
- Invalid opcode or expression.
- Unknown directive.
- String too long.

In addition to the error messages, the following warning messages are supported:

- No .END directive.
- Label truncated.
- Indirect operand crosses page.

Although the error messages are more-or-less self explanatory, the warning messages deserve a few comments.

The absence of an `.END` directive is not fatal, since a reasonable default action is to simply take the end of the source file as the end of the program. The `.END` directive is not rendered useless, however, because it provides a terminator which may be followed by proprietary programmer information which is included in the source code but will not be included in the listing.

Labels are truncated after 12 characters, but may still assemble correctly as long as the first 12 characters of every label are unique.

cbA65 warns the user if the following condition occurs when using indirect addressing:

```
000029 0200 6C FF 05            jmp (badaddr)
       .
       .
       .
000067 05FF 33 02       badaddr .word stop
```

In this situation, since the target address straddles a page boundary, the actual address used by the jump will not be taken from locations $5FF and $600, but from $5FF and $500. While this anomalous condition is almost certainly not intended by the programmer, it is a known behavior of the 65xx processor family and could conceivable be used deliberately. Therefore, **cbA65** detects the condition and issues a warning, rather than an error.

A similar situation exists with the *indexed indirect* addressing mode when the zero page base address is the last byte in page zero. The assembler also issues a warning in this case. Note that the *indirect indexed* also harbors this problem, but it cannot be detected at assemble time because the target address depends on an index which is determined at run time.

Keep in mind that the zero page indexed operations *wrap around* to remain in page zero whenever the base address plus the index exceeds 255. This condition can only be detected at run time, so no warnings are issued by the assembler.

Part II

The 65xx CPU

Chapter 7

65xx Processor Family

7.1 Processor Model

USAGE	ADDRESS
IRQ VECTOR	FFFE/FFFF
RES VECTOR	FFFC/FFFD
NMI VECTOR	FFFA/FFFB
ROM I/O RAM	0200/****
STACK PAGE	0100/01FF
ZERO PAGE	0000/00FF

Table 7.1: Basic Memory Map for 6502 Systems

The memory map[1] for various members of the 65xx processor family differ in the number of available address lines, the specific clocking hardware and the presence or absence of certain

[1]The map shown in 7.1 is drawn with addresses increasing from bottom to top. For code snippets or fragments addressing will increase from top to bottom because code is normally written and listings are printed this way.

interrupt lines. Instruction sets are identical across the family, but the physical addresses for memory locations may be system dependent.

A programmer may not need to have any special knowledge of the internal architecture of a microprocessor, but must have a detailed knowledge of its registers and their operation. The following diagram shows the simplest representation of the register set of the 65xx processors.

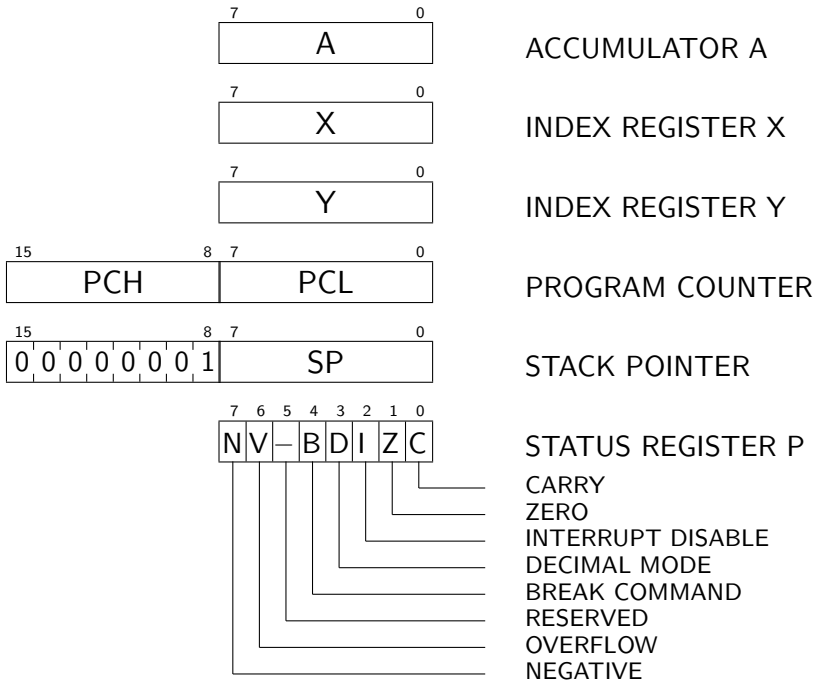


Figure 7.1: 65XX Register Programming Model

This model identifies the following registers:

ACCUMULATOR – This is the primary register used for data movement and manipulation,

INDEX REGISTER – Two index registers, with slightly different capabilities, provide offsets from the current base address,

PROGRAM COUNTER – Holds current memory address,

STACK POINTER – Current index in processor stack,

STATUS REGISTER – Flags and status bits.

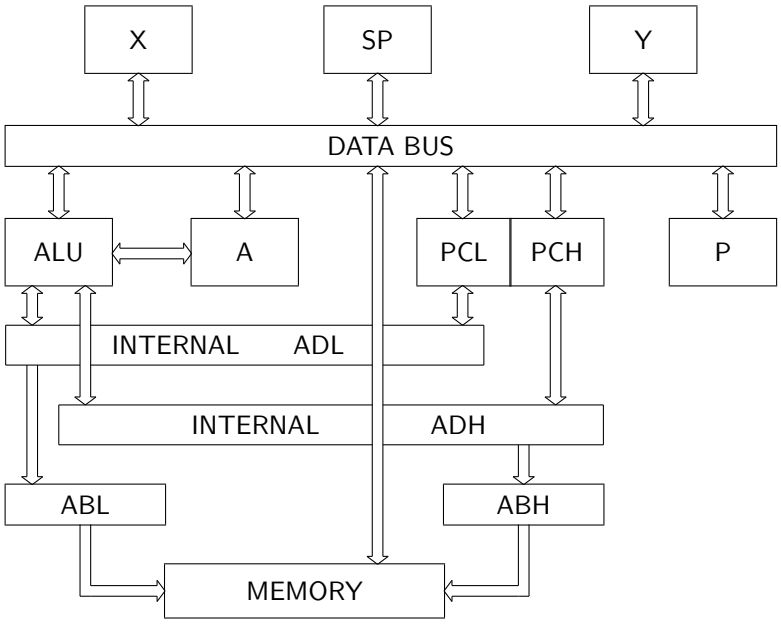


Figure 7.2: Simplified 65XX Internal Block Diagram

Note that all registers except the program counter are 8 bit registers. The program counter uses 16 (or fewer) bits.[2]

A more elaborate model of the 65xx processor follows. This model clarifies the internal arrangement of the registers in a hardware oriented diagram.

The abbreviated labels are defined as follows:

ALU – Arithmetic/logic unit,

PCL,PCH – Program counter (low 8 bits, high 8 bits),

ADL,ADH – Internal address buss (low 8 bits, high 8 bits),

ABL,ABH – External address buss (low 8 bits, high 8 bits).

[2]The stack pointer can be modelled as an extended 8 bit register with a 9th bit fixed at 1.

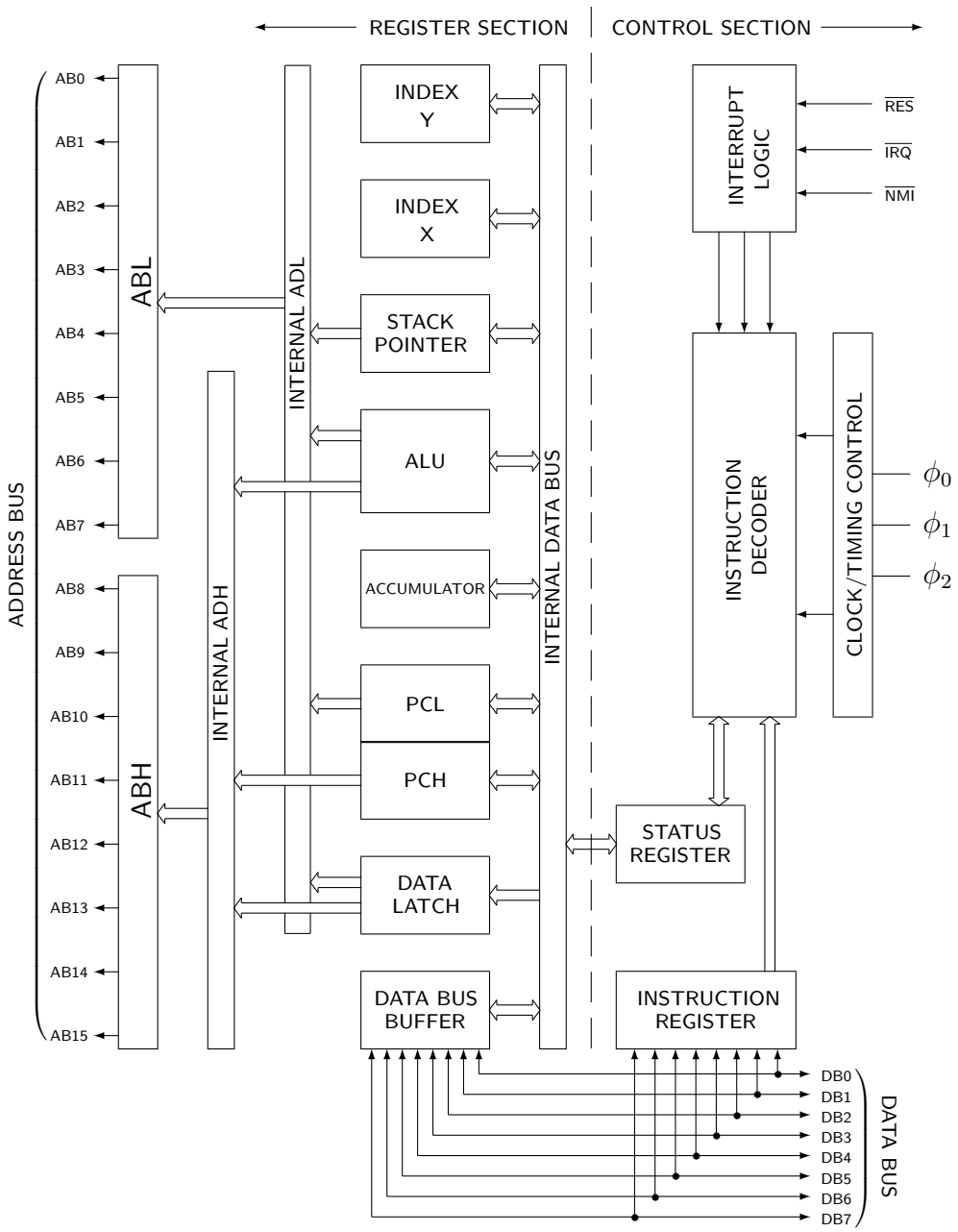


Figure 7.3: 65XX Internal Architecture

7.2 Instruction Set

There are many ways to organize the 65xx instruction set for study. In this section, the instruction mnemonics are organized by general operational function. The complete instruction set organized by opcode can be found in the chart in Appendix A.

7.2.1 Data Move (Load/Store/Transfer)

These instructions are used to move 8-bit bytes of data from one place to another. Moves may be from register to memory, from memory to register, or from register to register. There are no instructions which support memory to memory transfers.

`LDA` – Load accumulator from memory.

`LDX` – Load `X` from memory.

`LDY` – Load `Y` from memory.

`PHA` – Push accumulator onto stack.

`PHP` – Push processor flags onto stack.

`PLA` – Pull accumulator from stack.

`PLP` – Pull flags from stack.

`STA` – Store a in memory.

`STX` – Store `X` in memory.

`STY` – Story `Y` in memory.

`TAX` – Transfer accumulator to `X`.

`TAY` – Transfer accumulator to `Y`.

`TSX` – Transfer stack pointer to `X`.

`TXA` – Transfer `X` to accumulator.

`TXS` – Transfer `X` to stack pointer.

`TYA` – Transfer `Y` to accumulator.

7.2.2 Data Modify

These instructions modify data.

Add/Sub

ADC – Add memory to accumulator.

SBC – Subtract memory from accumulator.

See the section on Tips and Tricks 7.4 for an efficient method for subtracting the accumulator from memory.

Logic

The following logic operations are supported.

`AND` – Bitwise `AND` of memory to accumulator.

`EOR` – Bitwise `XOR` of memory to accumulator.

`ORA` – Bitwise `OR` of memory to accumulator.

Increment/Decrement (Read/Modify/Write

These instructions operate on memory or on the `X` or `Y` registers. When operating on memory, the are classed as *read, modify, write* instructions.

`DEC` – Decrement memory.

`DEX` – Decrement `X` register.

`DEY` – Decrement `Y` register.

`INC` – Increment memory.

`INX` – Increment `X` register.

`INY` – Increment `Y` register.

The increment and decrement instructions operate on 8-bit bytes and will wrap around if the result is greater than 255 or smaller than 0. No flags are affected except the `Z` flag which will be set if the result is zero.

Shift and Rotate (Read,Modify,Write)

Single bit shifts and rotates in either direction are supported by the 65xx processors. These operations can be performed on the accumulator or on a memory location. When performed on memory, the are classed as *read, modify, write* instructions.

The mnemonics associated with the shift and rotate opcodes are:

`ASL` — Arithmetic Shift Left

`LSR` — Logical Shift Right

`ROL` — Rotate Left

`ROR` — Rotate Right

A word about the terminology is in order. In computer literature a *logical* shift is one which replaces the vacated bit with a zero. This operation is widely used as a substitute for the multiplication or division of an integer by 2. For unsigned numbers this technique yields satisfactory results.

A problem occurs, however, when a logical shift right is applied to a signed quantity. Specifically, if a negative number is shifted right and the most significant bit is replaced by a zero, the sign is changed. An *arithmetic* shift right would be expected to preserve the sign of the number. The 65xx family opcodes do not support this operation.[3] Note that the arithmetic shift left is the same as a logical shift left.

The distinction between a *shift* and a *rotate* is simply that a *shift* replaces a vacated bit position with a zero, and a *rotate* replaces it with the value of the carry flag. The following diagrams should clarify the operations.

To rotate the accumulator without circulating through the carry, see the section on Tips and Tricks 7.4.

[3] An arithmetic shift right of the accumulator can be simulated with the following code sequence: `CMP #$80, ROR`.

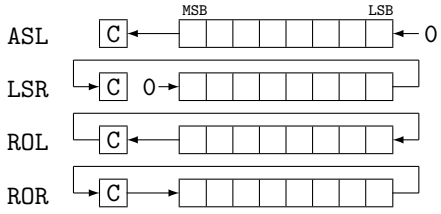


Figure 7.4: Shifting and Rotating

7.2.3 Data Test

This section details the behavior of instructions which are intended to set processor flags based on the contents of registers or memory locations.

Compare Instructions

The 65xx processors support only unsigned comparisons. That is, when two bytes are compared the states of the affected flags are those associated with comparing two 8-bit integers in the range from 0 to 255.[4]

The compare instructions are:

`CMP` – Compare memory with accumulator.

`CPX` – Compare memory with `X`.

`CPY` – Compare memory with `Y`.

The processor flags support program flow control by using one of the branch instructions following an operation which affects them. Of the eight branch instructions, six use flags affected by a compare.[5]

In order to correctly interpret the results of a comparison, the programmer must refer to the `Z` and `C` flags. On occasion the `N` flag is also of interest. Since comparisons are normally done in decision loops, it is important to understand how the resulting flag states can be used to control program flow.

[4] Although the numbers are unsigned for the purpose of manipulating the `Z` and `C` flags, the state of the `N` flag will also be affected.

[5] `BCC`,`BCS`,`BEQ`,`BNE`, `BMI` and `BPL`

Table 7.2 shows how the compare (CMP) instruction, which compares the value in the accumulator (A) register with a value in memory (M), affects the appropriate flags. The N flag

Comparison Result	Carry Flag	Zero Flag	Negative Flag
A > M	Set	Clear	?
A = M	Set	Set	Clear
A < M	Clear	Clear	?

Table 7.2: Status Flags for Simple Compares

is of limited use in making decisions using the compare instructions. However, there is a unique combination of Z and C flags which distinguish each possible result. The Z flag alone unambiguously distinguishes the cases where the value in the accumulator is equal to the value in memory or unequal to it. The C flag alone unambiguously distinguishes the cases where the value in the accumulator is less than the value in memory from those where it is not.

To interpret other relations will require testing combinations of the two flags. A complete list of possible flag states following a compare is shown in the Table 7.3.

Comparison Result	Carry Flag	Zero Flag	Negative Flag
A > M	Set	Clear	?
A ≥ M	Set	?	?
A = M	Set	Set	Clear
A ≠ M	?	Clear	?
A ≤ M	?	?	?
A < M	Clear	Clear	?

Table 7.3: Complete Table of Comparison Status Flags

In some cases, the relation tests involve two separate operations. The order of testing may be significant. See Appendix E for code examples.

Bit Instructions

There are two versions of the BIT instruction. One for page zero addressing and one for absolute addressing. The BIT instruction copies bits 6 and 7 of the target location into the processor V and N flags, respectively. The processor Z flag will hold the result of *ANDing* the target location contents with the accumulator.

This peculiar instruction is primarily used to test dynamic bits from hardware ports. It is often used in polling applications.

BIT – Set processor bits according to memory and accumulator.

7.2.4 Branch Instructions

The 65xx processors support eight instructions for controlling program flow. These instructions test the processor flags which were affected by previous operations and fall through to the next instruction or branch to a programmed target address, depending on the result. The branch instructions are:

`BCC` – Branch on carry clear (`C` = 0).

`BCS` – Branch on carry set (`C` = 1).

`BEQ` – Branch if equal (`Z` = 1).

`BNE` – Branch if not equal (`Z` = 0).

`BMI` – Branch if minus (`N` = 1).

`BPL` – Branch if plus (`N` = 0).

`BVC` – Branch if no overflow (`V` = 0).

`BVS` – Branch if overflow (`V` = 1).

7.2.5 Flag Control

Some of the processor flags can be set or cleared directly. These are:

`CLC` – Clear carry (`C` = 0).

`CLD` – Clear decimal mode (`D` = 0).

`CLI` – Clear IRQ disable (`I` = 0).

`CLV` – Clear overflow (`V` = 0).

`SEC` – Set carry (`C` = 1).

`SED` – Set decimal mode (`D` = 1).

`SEI` – Set IRQ disable (`I` = 1).

Note that there is no instruction to set the IRQ disable flag. Also be aware that the `V` flag can be set externally.

7.2.6 Unconditional Jumps

The following instructions are associated with unconditional program jumps, returns, etc.

`JMP` – Jump direct or indirect to a new location.

`JSR` – Jump to subroutine.

`RTI` – Return from interrupt.

`RTS` – Return from subroutine.

`NOP` – No operation (included here for completeness).

7.3 Addressing Modes

The 65xx processor architecture supports a number of remarkable addressing modes. Some of these involve complicated indirection which is best explained by operational models. See Appendix A for a summary of the complete instruction set.

Implied (No Address Mode)

Some instructions do not require any addressing mode specification. For example, `BRK`, `CLC`, `DEX`, `PHA` and `TAY`. These instructions are used for special operations, flag manipulation or register transfers. The involved registers, if any, are intrinsic in the opcode.

Accumulator

Accumulator addressing mode applies to instructions which have multiple addressing modes including one which refers to the accumulator. For example, `ROR` is supported for memory operands or the accumulator. This addressing mode may be specified by using the letter `A` as an operand, or simply omitting the operand. **cbA65** omits the operand so that `A` may be used as a symbol or label.

Immediate

Immediate addressing mode is specified by prepending a leading pound sign `#`to the operand. The operand is an 8-bit byte whose numeric value is to be used directly. Contrast this with zero page addressing where the operand is an address whose contents are to be used.

Depending on the assembler, an immediate operand may be a decimal or hexadecimal value, an ASCII character, or the value of a symbol.

```
lda #$ff        ; load accumulator with 255
```

Negative numbers are also supported as immediate operands. The assembler recognizes contexts which require 8-bit or 16-bit operands and will perform correctly.

Absolute

Absolute addressing is supported by all instructions which must access memory locations throughout the entire address space (65536 locations). An absolute address is usually specified by its decimal value or by a 4-character hex number. For example, `$8000 = 32768`.

Many of the 65xx opcodes specify operations on locations identified by absolute addresses.

```
eor $8000           ; xor accumulator with contents of 32768
```

Zero Page

Zero page addressing is supported by some instructions where a shortened address length provides an improvement in performance. With this addressing mode the operand (address) consists of a single byte. All instructions which support zero page addressing also support absolute addressing.

```
lda ptr1            ; move pointer 1 in z.p. to pointer 2
sta ptr2
lda ptr1+1
sta ptr2+1
```

If an instruction operand is an 8-bit value and the instruction supports zero page addressing, most assemblers will code the zero page instruction automatically. **cbA65** also defaults to zero page addressing in this case, but allows the programmer to override the default and force absolute addressing. See Section 3.5.

Absolute Indexed

Many instructions support indexing by the `X` or `Y` registers. Indexing refers to the addition of the value in the index register to the address specified by the operand. For absolute indexed mode, the operand is an absolute (16-bit) address and the index register is part of the operand. An example of indexing using the `Y` register is

```
lda msgtbl,y
```

where `msgtbl` is an absolute address and `y` specifies that the `Y` register is to be added to the address. `Y` would most likely be an offset to a particular message in the table.

There are only two index registers, but some instructions support only one of them.

Zero Page Indexed

This addressing mode is analogous to zero page addressing in that a short (1-byte) base address is supported. The indexing works the same as for absolute indexing, except that the resulting address wraps around to remain in page zero.

Indirect Indexed

An example of the addressing mode called *indirect indexed* is shown in Figure 7.5. This example illustrates the case where `PTR` is a symbolic quantity representing the zero page address: `$C2`. At the time the instruction `LDA (PTR),Y` is executed, the `Y` register contains `$35`. After completion of the instruction, the `A` register contains `$7C`.

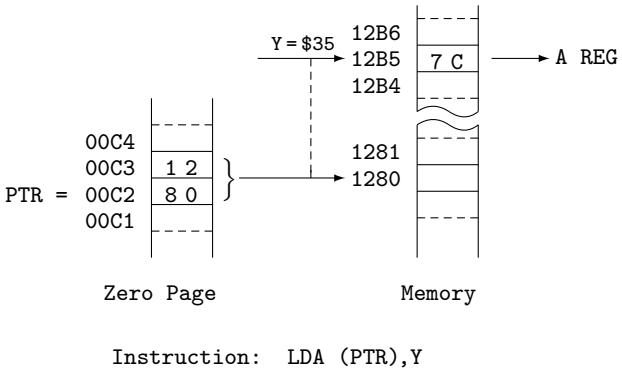


Figure 7.5: Indirect Indexed Addressing Mode

A quirk in this instruction occurs if the low byte of the zero page address (PTR) is located at `$FF`. In this case the expected address high byte will be at location `$00` instead of the

expected `$100`. The assembler can detect this condition and will report a warning.[6]

Indexed Indirect

Indexed indirect addressing is illustrated in Figure 7.6. Although this mode and the previous mode are sufficiently complex to cause some confusion, an examination of the figures with due attention to the terminology should make the distinction clear.

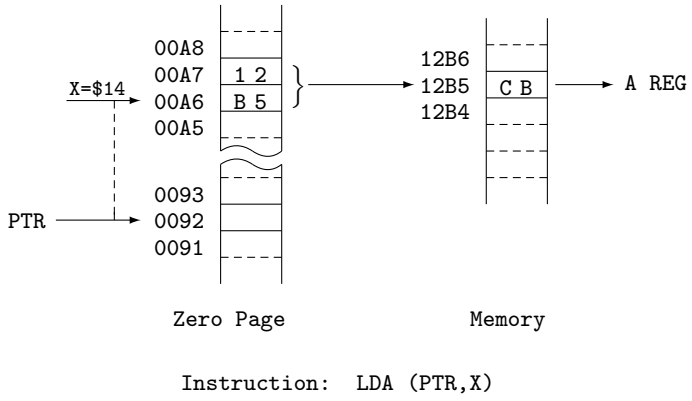


Figure 7.6: Indexed Indirect Addressing Mode

This addressing mode is also subject to page-wrapping quirks, but they cannot be detected at assemble time.

7.4 Tips and Tricks

It is sometimes necessary to perform an operation which is not directly supported by a particular processors instruction set. The current trend toward CISC (Complex Instruction Set Computers) has improved the number of built-in CPU features massively. But for the lowly 65xx processors, the programmer is burdened with implementing them in software.

For example, *multiply*, *divide*, *square root*, and a variety of floating point functions are available on modern processors. These must be done in software for the 65xx.

At a much lower level, there are several operations which are not available on the 65xx but which are easily implemented with a few instructions. These are the focus of this section.

[6] Although this condition is most likely an error, it is legal and will not abort the assembly.

7.4.1 Shift Arithmetic Right

This instruction is not available in the 65xx instructions set, but can be implemented with the following code.

```
CMP #$80            ; get sign bit into carry
ROR                 ; preserve sign during right shift
```

This sequence will effectively divide a signed quantity by 2.

7.4.2 8-Bit Rotates

The 65xx processors implement *rotate-through-carry* operations. These rotate the accumulator, but insert the carry bit in the rotation path. This is, effectively, a 9-bit rotate. See Section 7.2.2. For rotating multi-byte quantities, this is the ideal rotation strategy.

However, it is sometimes necessary to simply rotate the accumulator without intervening bits. These 8-bit rotates can be done as shown.

```
ROTATE RIGHT                ROTATE LEFT

   pha                         pha
   ror                         rol
   pla                         pla
   ror                         rol
```

7.4.3 Where Am I?

A useful trick to aid in tracing problems in complex code uses a special memory location to hold an execution address. This location can be examined after an unscheduled halt, break or crash to determine a recent execution point. To use the method, it is only necessary to sprinkle copies of code similar to the following with suitable labels throughout the executable.

```
.
.
```

```
        .
        jsr nxtloc          ; just push current address...
nxtloc  pla
        sta traceaddr       ; ...and retrieve it for reference.
        pla
        sta traceaddr+1
        .
        .
        .
```

7.4.4 Using a Return for a Jump

It is sometimes desirable to have a JMP instruction with a programmable target address. This example of self-modifying code is not possible if the only locations available are in ROM and cannot be changed at run time.

One way around this is to push the target address on the processor stack and execute an `RTS` instruction.[7] The processor adds 1 to the address popped from the stack, so it is necessary to subtract 1 from the address pushed.

```
lda #>(prtstr-1)      ; jump to 'prtstr' routine
pha
lda #<(prtstr-1)
pha
rts
```

Recall that the immediate mode symbol `#` is not really required since the < and > symbols imply immediate mode.

7.4.5 Subtracting From Memory

All math operations in the 65xx are performed on the accumulator. Subtracting the contents of a memory location from the accumulator is directly supported. The reverse operation, subtracting the accumulator from a memory location, is not directly supported. Manipulating the data so the subtraction occurs in the accumulator is awkward, to say the least. However, if the quantity in the accumulator is converted to a negative value, the contents of memory may be added to it and the result saved back with improved efficiency. Since

[7] Some programmers, with tongue in cheek, have referred to this operation as a *'come from'* instruction.

the conversion to negative requires the two's complement of the accumulator, the following code will work.

```
eor #$ff    ; form 1's complement of accumulator
sec         ; add carry for 2's complement
adc memloc  ; get result
sta memloc  ;  and save it.
```

Part III

Advanced Techniques

Chapter 8

Using Tables

It is well-known that precomputed tables can boost the performance of many algorithms. This chapter discloses a few table-driven methods for performing useful computations and conversions, with caveats.

8.1 Number Conversion

The 65xx family of processors support both binary and BCD math operations.

Here is a module which converts a BCD number < 65536 to 16-bit binary. Although the method is reasonabley efficient and requires very little code, the penalty to be paid is the requirement for a 208 byte table.

```
;
;   bcd2bin.cba -- convert BCD number (n < 65536) to binary.
;   BCD number in ($82,$83,$84) -> binary number in ($80,$81)
;                   lo,....,hi                          lo, hi
;
;   C. Bond, 2008
    .title BCD to Binary Conversion Program
    .files h65
        * = $80
bin     * = *+2
bcd     * = *+3
        .org $200
```

```
tstit   lda #$35        ; test entry point
        ldy #$55
        ldx #$06
;
;   bin2bcd
;
;   convert a BCD number in $82,$83,$84 to binary in $80,$81
;
bcd2bin sta bcd         ; a:lo byte, y:mid byte, x:hi byte
        sty bcd+1
        stx bcd+2
        ldy #$f
;
;   1) shift bcd down through bin
;   2) update bcd values
;   3) decrement counter
;   4) loop to 1) until done
;

mloop   lsr bcd+2       ; shift number field down
        ror bcd+1
        ror bcd
        ror bin+1
        ror bin
;
;   update bcd register
;
        ldx bcd
        lda cnvtbl2,x
        sta bcd
        ldx bcd+1
        lda cnvtbl2,x
        sta bcd+1
        ldx bcd+2
        lda cnvtbl2,x
        sta bcd+3
        dey
        bpl mloop
        brk

        .org $400
cnvtbl2 .byte $00,$01,$02,$03,$04,$05,$06,$07,$05,$06,$07,$08,$09,0,0,0
```

```
        .byte $10,$11,$12,$13,$14,$15,$16,$17,$15,$16,$17,$18,$19,0,0,0
        .byte $20,$21,$22,$23,$24,$25,$26,$27,$25,$26,$27,$28,$29,0,0,0
        .byte $30,$31,$32,$33,$34,$35,$36,$37,$35,$36,$37,$38,$39,0,0,0
        .byte $40,$41,$42,$43,$44,$45,$46,$47,$45,$46,$47,$48,$49,0,0,0
        .byte $50,$51,$52,$53,$54,$55,$56,$57,$55,$56,$57,$58,$59,0,0,0
        .byte $60,$61,$62,$63,$64,$65,$66,$67,$65,$66,$67,$68,$69,0,0,0
        .byte $70,$71,$72,$73,$74,$75,$76,$77,$75,$76,$77,$78,$79,0,0,0
        .byte $50,$51,$52,$53,$54,$55,$56,$57,$55,$56,$57,$58,$59,0,0,0
        .byte $60,$61,$62,$63,$64,$65,$66,$67,$65,$66,$67,$68,$69,0,0,0
        .byte $70,$71,$72,$73,$74,$75,$76,$77,$75,$76,$77,$78,$79,0,0,0
        .byte $80,$81,$82,$83,$84,$85,$86,$87,$85,$86,$87,$88,$89,0,0,0
        .byte $90,$91,$92,$93,$94,$95,$96,$97,$95,$96,$97,$98,$99,0,0,0
        .end
```

The counterpart program, which converts from 16-bit binary to BCD is shown below.

```
;
;   bin2bcd.cba -- convert 16-bit binary number to BCD
;   binary number in ($80,$81) -> BCD number in ($82,$83,$84)
;                       lo, hi                      lo,....,hi
;
;   C. Bond, 2008
    .title Binary to BCD Conversion Program
    .files h65
        * = $80
bin     * = *+2
bcd     * = *+3
        .org $200
tstit   lda #$ff        ; test entry point (convert $FFFF to 65535)
        ldy #$ff
;
;   bin2bcd
;
;   convert 16-bit binary number in a,y to BCD in $82,$83,$84
;
bin2bcd sta bin         ; a:lo byte, y:hi byte
        sty bin+1
        lda #0
        sta bcd
        sta bcd+1
        sta bcd+2
```

```
        ldy #$f
;
;   1) prep bcd number for shift
;   2) shift bin through bcd
;   3) decrement loop counter (y)
;   4) loop to 1) until done
;

mloop   ldx bcd         ; prepare bcd values for shift
        lda cnvtbl,x
        sta bcd
        ldx bcd+1
        lda cnvtbl,x
        sta bcd+1
        ldx bcd+2
        lda cnvtbl,x
        sta bcd+2
;
;   shift number field up
;
        asl bin
        rol bin+1
        rol bcd
        rol bcd+1
        rol bcd+2
        dey
        bpl mloop
        brk

        .org $400
cnvtbl  .byte $00,$01,$02,$03,$04,$08,$09,$0A,$0B,$0C,0,0,0,0,0,0
        .byte $10,$11,$12,$13,$14,$18,$19,$1A,$1B,$1C,0,0,0,0,0,0
        .byte $20,$21,$22,$23,$24,$28,$29,$2A,$2B,$2C,0,0,0,0,0,0
        .byte $30,$31,$32,$33,$34,$38,$39,$3A,$3B,$3C,0,0,0,0,0,0
        .byte $40,$41,$42,$43,$44,$48,$49,$4A,$4B,$4C,0,0,0,0,0,0
        .byte $80,$81,$82,$83,$84,$88,$89,$8A,$8B,$8C,0,0,0,0,0,0
        .byte $90,$91,$92,$93,$94,$98,$99,$9A,$9B,$9C,0,0,0,0,0,0
        .byte $A0,$A1,$A2,$A3,$A4,$A8,$A9,$AA,$AB,$AC,0,0,0,0,0,0
        .byte $B0,$B1,$B2,$B3,$B4,$B8,$B9,$BA,$BB,$BC,0,0,0,0,0,0
        .byte $C0,$C1,$C2,$C3,$C4,$C8,$C9,$CA,$CB,$CC,0,0,0,0,0,0
        .end
```

The table for this conversion only requires 160 bytes — but in this case, any table at all is too much! The reason is that the 6502 is capable of directly performing the conversion efficiently, as shown below.

```
;
;   bin2bcd.cba -- convert 16-bit binary number to BCD
;
;   C. Bond, 2008
    .title Binary to BCD Conversion Program
    .files h65
        * = $80
bin     * = *+2
bcd     * = *+3
        .org $200
tstit   lda #$ff        ; test entry point
        ldy #$ff
;
;   bin2bcd
;
;   convert 16-bit binary number in a,y to bcd in $82,$83,$84
;
bin2bcd sta bin     ; entry for user provided binary number
        sty bin+1   ; a:lo, y:hi
        lda #0
        sta bcd
        sta bcd+1
        sta bcd+2
        sed
        ldx #$f
mloop   asl bin
        rol bin+1
        lda bcd
        adc bcd
        sta bcd
        lda bcd+1
        adc bcd+1
        sta bcd+1
        lda bcd+2
        adc bcd+1
        sta bcd+2
        dex
        bpl mloop
```

```
        cld
        brk
        .end
```

I don't know of any similar algorithm for accomplishing a BCD to binary conversion.

The lesson to be learned here is twofold. First, no algorithm is necessarily best for all applications and, second, there is no substitute for creativity and out-of-the-box thinking.

8.2 Nibble Shifting

When processing packed BCD numbers it is often necessary to shift a multi-byte register by one nibble (1 BCD digit). This challenge offers fertile ground for trading code size for speed.

Here is sample code for shifting an 8-byte BCD register left one digit.

```
;
;   bcd_shfta.cba -- routine to shift a packed multi-byte BCD register
;                    left by 1 digit (4-bits)
;
        .org $200
shfl    ldy #3          ; shift 8-byte 'reg' left one nibble
lloop   ldx #6
        asl reg+1,x
@       rol reg,x
        dex
        bpl @B
        dey
        bpl lloop
        brk
        .org $300
reg     .byte $00,$00,$12,$34,$56,$78,$98,$76
        .end
```

This code is compact and straightforward. However, it is *slow*. It requires about 388 processor cycles to complete the shift.

Here is a simple alternative routine which takes more code, but accomplishes the shift in about 345 cycles.

```
;
;   bcd_shftb.cba -- alternative routine to shift a packed multi-byte
;                    BCD register left by 1 digit (4-bits)
;
temp    .equ $80
        .org $200
shfl    lda #0
        sta temp
        ldx #7
@       lda reg,x
        pha
        asl
        asl
        asl
        asl
        ora temp
        sta reg,x
        pla
        lsr
        lsr
        lsr
        lsr
        sta temp
        dex
        bpl @B
        brk
        .org $300
reg     .byte $00,$00,$12,$34,$56,$78,$98,$76
        .end
```

The following version accomplishes the same thing as the previous two routines but, at the cost of two tables, reduces the computation overhead to 157 CPU cycles — over *twice* as fast.

```
;
;   bcd_shftc.cba -- table-driven routine to shift a multi-byte BCD register
;                    left by one digit.
;
        .org $200
shfl    ldx #7      ; shift 'reg' left one digit.
        lda #0
@       ldy reg,x
```

```
        ora tlh,y
        sta reg,x
        lda thl,y
        dex
        bpl @B
        brk

        .org $300
reg     .byte $00,$00,$12,$34,$56,$78,$98,$76
        .org $400
;
;   This table, and the one following, support high speed
;   shifting of multibyte packed BCD registers.
;
tlh .byte 0,$10,$20,$30,$40,$50,$60,$70,$80,$90,0,0,0,0,0,0
    .byte 0,$10,$20,$30,$40,$50,$60,$70,$80,$90,0,0,0,0,0,0
    .byte 0,$10,$20,$30,$40,$50,$60,$70,$80,$90,0,0,0,0,0,0
    .byte 0,$10,$20,$30,$40,$50,$60,$70,$80,$90,0,0,0,0,0,0
    .byte 0,$10,$20,$30,$40,$50,$60,$70,$80,$90,0,0,0,0,0,0
    .byte 0,$10,$20,$30,$40,$50,$60,$70,$80,$90,0,0,0,0,0,0
    .byte 0,$10,$20,$30,$40,$50,$60,$70,$80,$90,0,0,0,0,0,0
    .byte 0,$10,$20,$30,$40,$50,$60,$70,$80,$90,0,0,0,0,0,0
    .byte 0,$10,$20,$30,$40,$50,$60,$70,$80,$90,0,0,0,0,0,0
    .byte 0,$10,$20,$30,$40,$50,$60,$70,$80,$90,0,0,0,0,0,0
;
;   'thl' maps the upper digit of its BCD index into the
;   low digit.
;
;   Example:
;
;           ldy #$35
;           lda thl,y
;
;   Now the 'a' register contains #$03; i.e. the '3' in '35'
;   is shifted to the low digit position and the high digit
;   position is cleared.
;
thl .byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
    .byte 1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
    .byte 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2
    .byte 3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3
    .byte 4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4
```

```
.byte 5,5,5,5,5,5,5,5,5,5,5,5,5,5,5,5
.byte 6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6
.byte 7,7,7,7,7,7,7,7,7,7,7,7,7,7,7,7
.byte 8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8
.byte 9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9
    .end
```

Chapter 9

Hashing 65xx Mnemonics

One of the early tasks in writing programming tools for 65xx processors is that of efficiently identifying opcode mnemonics. It is desirable to be able to scan a token consisting of ASCII characters and determine quickly whether this token is a valid 65xx opcode. The direct and naïve approach is to attempt to match the characters in the token to an entry in a list of opcodes. In the worst case, when the token is not in the list, this can waste many machine cycles.

A more sophisticated approach uses a *hash* table. The optimal strategy is easily stated:

1. Manipulate the token to produce a numeric value,
2. Retrieve a pointer from the table using the numeric value as an index,
3. Compare the entry pointed to in the opcode list with the given token.

The result of the compare will be *match* or *no match*.

Optimal hashing is rarely possible. In some cases the resulting hash tables are too large to be practical. Sometimes the hashing algorithm, which generates an index from the token, produces duplicate numbers for different tokens. All these problems can be managed with simple support strategies, but in our case an optimal solution is possible.[1]

The following test code exercises a 'perfect' hash algorithm for screening 65xx opcodes. There are 56 unique opcodes (not counting address mode modifiers), and each one is present in a prepared list.

[1] See the computer science literature for a wealth of information on hashing strategies.

The candidate string submitted to the routine for testing will produce a return value of 1 if it is a valid opcode and 0 otherwise. The ideal hashing is accomplished by generating a value in the range from 0 to 255 for each 3-character string submitted. If the strings are valid 65xx opcodes, there will be a unique value for each one. This value is used as an index into a table which contains the index of each opcode in the list.

For example, the string `ADC` will yield the hash code value, `$F4`. At position `$F4` (244) in the table is the value 1. The opcode list, mne[ ], contains `ADC` at position 1. Hence, a simple compare of the original string with the string at mne[1] confirms the opcode.

Note that the final step is a string compare, but that only one string compare is required to confirm that the input string is a valid opcode. Other strategies require searching the list, which is fairly short, but long enough to benefit from hashing. Imperfect hashing may require several string compares before a decision is reached.

A further optimization is possible. The 65xx mnemonics are all 3 character strings. If these strings are null terminated, as would be the case in a 32-bit `C` language environment, we can process 4 characters at a time in a 32-bit integer. Then no string comparisons are required at all, just a single integer compare.

The critical elements of this method implemented in `C` are:

1. A table or list of valid mnemonics for all opcodes. This is a 57 element list which contains a dummy string at position 0.

```
char* mne[57] = {
                "XXX",
                "ADC",
                "AND",
                 ...
                 ...
                "TXS",
                "TYA"};
```

2. A hash index table with 256 entries. All non-opcode hash values are 0. The hash values which result from valid opcode hashing contain the index of the opcode in the mnemonic table above.

3. The hash algorithm. This algorithm was produced by a computer search of several hashing strategies. It only requires 2 shifts and 2 XOR operations per invocation to generate the hash value. A unique value is produced for each opcode.

```
/* This table is a list of all standard 65xx opcode mnemonics. A
 * dummy entry at location 0 allows the list to index from 1.
 */
char *mne[] = {
    "XXX", "ADC", "AND", "ASL", "BCC", "BCS", "BEQ", "BNE",
    "BMI", "BPL", "BVC", "BVS", "BIT", "BRK", "CLC", "CLD",
    "CLI", "CLV", "CMP", "CPX", "CPY", "DEC", "DEX", "DEY",
    "EOR", "INC", "INX", "INY", "JMP", "JSR", "LDA", "LDX",
    "LDY", "LSR", "NOP", "ORA", "PHA", "PHP", "PLP", "PLA",
    "ROL", "ROR", "RTI", "RTS", "SBC", "SEC", "SED", "SEI",
    "STA", "STX", "STY", "TAX", "TAY", "TSX", "TXA", "TXS",
    "TYA"
    };


/* This table contains index values into the mnemonic table. The
 *  256 entries correspond to possible values generated by the
 * hash algorithm: hash_op(). For each valid opcode, the hash
 * algorithm produces a number which identifies a location in
 * in this array. The location contains an index into the
 * opcode mnemonic table mne[].
 */
int hshtbl[256] = {
 0,  0,  0,  0, 23,  0,  0, 22,  0,  0,  0,  0, 32,  0,  0, 31,
 0,  0, 21,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
45, 46,  0,  0,  0,  0, 47,  0,  0,  0,  0,  0,  0,  0,  0, 37,
 0, 26, 27,  0,  0,  0,  0,  0, 14, 15,  0,  0,  0,  0, 16,  0,
 0,  0,  0, 28, 25,  0, 48,  0,  0,  0,  0, 17,  0, 18,  0, 38,
 0,  0,  0,  0,  0,  0, 10,  0,  0,  0,  0, 24, 56,  0,  0,  0,
 0, 41,  0,  0, 54,  3, 11, 40,  0,  0,  0, 34,  0, 19, 20,  0,
 0, 29,  0,  0,  0,  0, 43,  0,  0,  0,  0,  0,  0,  0, 13,  0,
42,  0,  0,  0,  0,  0,  0,  0,  0,  0, 55,  0,  0, 33, 35,  9,
 0,  0,  0,  0,  7,  2,  0,  0,  8,  0,  0,  0, 39,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0, 53,  0,  0,  0,  0,  0, 49, 50,  0,
 0,  0,  0,  0, 52,  0,  0, 51,  0,  0,  0,  0, 36,  0,  0,  0,
 0,  0,  0,  0,  0,  0,  0,  0, 44,  0,  0,  0,  0,  0,  0, 12,
 6,  0,  0,  0, 30,  0,  0,  0,  0,  0,  0,  0,  0,  0,  5,  0,
 0,  0,  0,  0,  1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  4,  0};
```

Here is the hash algorithm. It returns an index into the above table.

```
/* The following hash algorithm is a 'perfect' hasher for
 * the official 65xx opcode mnemonics. It produces a
 * unique number in the range 1 - 255 for each of the valid
 * mnemonics. The returned number is either 0 or an index
 * into hshtbl[]. Since it is possible for other character
 * combinations to produce the same index, a test for string
 * match must follow the generation of the hash index.
 */
int hash_op(char *p)
{
    int hsh;

    hsh = (((*p++)-67) << 1);
    hsh ^= (((*p++)-67) << 3);
    hsh ^= ((*p)-67);
    hsh &= 0xff;
    return hshtbl[hsh];
}
```

Here is a simple calling routine. Invoke the call with a null-terminated three character ASCII string as an argument. Returns 1 (*true*) if the argument is a valid mnemonic and 0 (*false*) otherwise.

```
/* Test for string match to 3-letter opcode.
 * Accepts NULL terminated, 3-character string.
 */
int isopcode(char *p)
{
    int idx,s;

    s = *(int *)p;
    s &= ~0x00202020;   /* assure string is upper case */
    idx = hash_op((char*)&s);
    if (!idx) return 0;         /* failed hash test */
/* the following replaces a string compare (4 characters, including NULL) */
    if (*(int *)mne[idx] ==  s) {
        return 1;               /* pass! valid official mnemonic */
    }
    return 0;                   /* failed mnemonic compare test */
}
```

Appendices

Appendix A

65xx Opcode Chart

LSD (HEX) ⟶

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	BRK	ORA (n,X)				ORA n	ASL n		PHP	ORA #n	ASL			ORA nn	ASL nn	
1	BPL n	ORA (n),Y				ORA n,X	ASL n,X		CLC	ORA nn,Y				ORA nn,X	ASL nn,X	
2	JSR nn	AND (n,X)			BIT n	AND n	ROL n		PLP	AND #n	ROL		BIT nn	AND nn	ROL nn	
3	BMI n	AND (n),Y				AND n,X	ROL n,X		SEC	AND nn,Y				AND nn,X	ROL nn,X	
4	RTI	EOR (n,X)				EOR n	LSR n		PHA	EOR #n	LSR		JMP nn	EOR nn	LSR nn	
5	BVC n	EOR (n),Y				EOR n,X	LSR n,X		CLI	EOR nn,Y				EOR nn,X	LSR nn,X	
6	RTS	ADC (n,X)				ADC n	ROR n		PLA	ADC #n	ROR		JMP (nn)	ADC nn	ROR nn	
7	BVS n	ADC (n),Y				ADC n,X	ROR n,X		SEI	ADC nn,Y				ADC nn,X	ROR nn,X	
8		STA (n,X)			STY n	STA n	STX n		DEY		TXA		STY nn	STA nn	STX nn	
9	BCC n	STA (n),Y			STY n,X	STA n,X	STX n,Y		TYA	STA nn,Y	TXS			STA nn,X		
A	LDY #n	LDA (n,X)	LDX #n		LDY n	LDA n	LDX n		TAY	LDA #n	TAX		LDY nn	LDA nn	LDX nn	
B	BCS n	LDA (n),Y			LDY n,X	LDA n,X	LDX n,Y		CLV	LDA nn,Y	TSX		LDY nn,X	LDA nn,X	LDX nn,Y	
C	CPY #n	CMP (n,X)			CPY n	CMP n	DEC n		INY	CMP #n	DEX		CPY nn	CMP nn	DEC nn	
D	BNE n	CMP (n),Y				CMP n,X	DEC n,X		CLD	CMP nn,Y				CMP nn,X	DEC nn,X	
E	CPX #n	SBC (n),X			CPX n	SBC n	INC n		INX	SBC #n	NOP		CPX nn	SBC nn	INC nn	
F	BEQ n	SBC (n),Y				SBC n,X	INC n,X		SED	SBC nn,Y				SBC nn,X	INC nn,X	

Appendix B

65xx Processor Pinout Diagrams

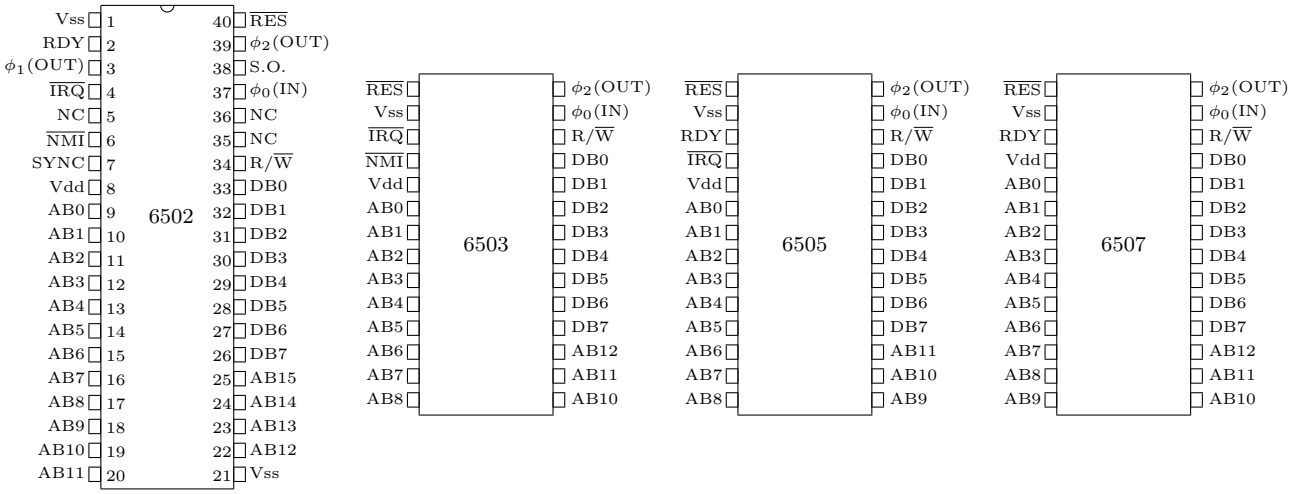


Figure B.1: Selected 650X Pinout Diagrams

Appendix C

ASCII Character Set

LSD (HEX) $\longrightarrow$

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

HEX	ASCII	DESCRIPTION	HEX	ASCII	DESCRIPTION
00	NUL	Null character	10	DLE	Device link escape
01	SOH	Start of header	11	DC1	Device control 1
02	STX	Start of text	12	DC2	Device control 2
03	ETX	End of text	13	DC3	Device control 3
04	EOT	End of transmission	14	DC4	Device control 4
05	ENQ	Enquiry	15	NAK	Negative acknowledge
06	ACK	Acknowledge	16	SYN	Synchronous idle
07	BEL	Sound bell	17	ETB	End of transmission block
08	BS	Backspace	18	CAN	Cancel
09	HT	Horizontal tab	19	EOM	End of medium
0A	LF	Line feed	1A	SUB	Substitute
0B	VT	Vertical tab (or home)	1B	ESC	Escape
0C	FF	Form feed	1C	FS	File separator
0D	CR	Carriage return	1D	GS	Group separator
0E	SO	Shift out	1E	RS	Record separator
0F	SI	Shift in	1F	US	Unit separator

Appendix D

Hex/Decimal Conversion Chart

LSD (HEX) $\longrightarrow$

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Appendix E

Branch after Compare

The following code snippets illustrate the use of processor status flags after a compare to control program flow. These examples are not optimized and are only intended to clarify the decision concepts.

E.1 Accumulator Greater than Memory

```
    cmp N       ; compare A with N
    bcc @F      ; FAIL if A is less than N
    beq @F      ; FAIL if A is equal to N
    jmp PASS
@   jmp FAIL
```

E.2 Accumulator Greater than or Equal to Memory

```
    cmp N       ; compare A with N
    bcc @F      ; FAIL if A is less than N
    jmp PASS
@   jmp FAIL
```

E.3 Accumulator Equal to Memory

```
    cmp N       ; compare A with N
    bne @F      ; FAIL if A is unequal to N
    jmp PASS
@   jmp FAIL
```

E.4 Accumulator Unequal to Memory

```
    cmp N       ; compare A with N
    beq @F      ; FAIL if A is equal to N
    jmp PASS
@   jmp FAIL
```

E.5 Accumulator Less Than or Equal to Memory

```
    cmp N       ; compare A with N
    beq @F      ; PASS if A is equal to N
    bcc @F      ; PASS if A is less than N
    jmp FAIL
@   jmp PASS
```

E.6 Accumulator Less Than Memory

```
    cmp N       ; compare A with N
    bcc @F      ; PASS if A is less than N
    jmp FAIL
@   jmp PASS
```

Appendix F

Intel Hex Format

Intel HEX-record Format[1]

INTRODUCTION

Intel's Hex-record format allows program or data files to be encoded in a printable (ASCII) format. This allows viewing of the object file with standard tools and easy file transfer from one computer to another, or between a host and target. An individual Hex-record is a single line in a file composed of many Hex-records.

HEX-RECORD CONTENT

Hex-Records are character strings made of several fields which specify the record type, record length, memory address, data, and checksum. Each byte of binary data is encoded as a 2-character hexadecimal number: the first ASCII character representing the high-order 4 bits, and the second the low-order 4 bits of the byte.

The 6 fields which comprise a Hex-record are defined as follows:

[1]The information in this section was obtained from Internet sources. It is believed to be reliable, but no warranty on accuracy or usage is provided.

Field		Format	
Number	Type	Chars	Description
1	Start code	1	An ASCII colon, ":".
2	Byte count	2	Number *(n)* of character pairs in data field.
3	Address	4	The 2-byte memory address to load the data field.
4	Type	2	00, 01, or 02.
5	Data	0-2n	From 0 to n bytes of code or data, ($n \leq 32$).
6	Checksum	2	Checksum over fields 2 through 5.

The checksum is the least significant byte of the two's complement sum of the values represented by all the pairs of characters in the included fields.

Each record may be terminated with a CR/LF/NULL. Accuracy of transmission is ensured by the byte count and checksum fields.

HEX-RECORD TYPES

There are three possible types of Hex-records.

00 — A record containing data and the 2-byte address at which the data is to reside.

01 — A termination record for a file of hex-records. Only one termination record is allowed per file and it must be the last line of the file. There is no data field.

02 — A segment base address record. This type of record is ignored by Lucid programmers.

HEX-RECORD EXAMPLE

Following is a typical Hex-record module consisting of four data records and a termination record.

```
:10010000214601360121470136007EFE09D2190140
:100110002146017EB7C20001FF5F16002148011988
:10012000194E79234623965778239EDA3F01B2CAA7
:100130003F0156702B5E712B722B732146013421C7
:00000001FF
```

The fields in the first data record is explained as follows:

1. **:** – Start code.
2. **10** – Hex 10 (decimal 16), indicating 16 data character pairs, 16 bytes of binary data in this record.
3. **0100** – Four-character, 2-byte, address field. Hex address 0100, indicates location where the following data is to be loaded.
4. **00** – Record type indicating a data record.
5. The next 16 character pairs are the ASCII bytes of the actual program data.
6. **40** – Checksum of the first Hex-record.

The termination record is explained as follows:

1. **:** – Start code.
2. **00** – Byte count is zero, no data in termination record.
3. **0000** – Four-character 2-byte address field, zeros (HI/LO).
4. **01** – Record type 01 is termination.
5. No data field present.
6. **FF** – Checksum of termination record.

Appendix G

MOS Technology Hex Format

The MOS Technology Hex File Format[1]

INTRODUCTION

MOS Technology's Hex file format allows program or data files to be encoded in a printable (ASCII) format which is very similar to the Intel Hex format. This allows viewing of the object file with standard tools and easy file transfer from one computer to another, or between a host and target. An individual Hex-record is a single line in a file composed of many Hex-records.

HEX RECORD CONTENT

MOS Hex Records are character strings made of several fields which specify the record length, memory address, data, and checksum. Each byte of binary data is encoded as a 2-character hexadecimal number: the first ASCII character representing the high-order 4 bits, and the second the low-order 4 bits of the byte.

The 5 fields which comprise a MOS Hex record are defined as follows:

[1]The information in this section was obtained from Internet sources. It is believed to be reliable, but no warranty on accuracy or usage is provided.

Field		Format	
Number	Type	Chars	Description
1	Start code	1	An ASCII semicolon, ";".
2	Byte count	2	Number *(n)* of character pairs in data field.
3	Address	4	The 2-byte memory address to load the data field.
4	Data	0-2n	From 0 to n bytes of code or data, ($n \leq 32$).
5	Checksum	4	Checksum over fields 2 through 4.

The checksum is the least significant word (2-bytes) of the sum of the values represented by all the pairs of characters in the included fields.

Each record may be terminated with a CR/LF/NULL. Accuracy of transmission is ensured by the byte count and checksum fields.

MOS HEX RECORD EXAMPLE

Following is a typical MOS Hex record module consisting of four data records and a termination record.

```
;10B000576F77212044696420796F75207265610624
;10B0106C6C7920676F207468726F756768206106B9
;10B0206C6C20746861742074726F75626C652006C6
;0DB030746F20726561642074686973733F05A3
;00
```

The fields in the first data record is explained as follows:

1. **;** – Start code.
2. **10** – Hex 10 (decimal 16), indicating 16 data character pairs, 16 bytes of binary data in this record.
3. **B000** – Four-character, 2-byte, address field. Hex address B000, indicates location where the following data is to be loaded.
4. The next 16 character pairs are the ASCII bytes of the actual program data.
5. **0624** – Checksum of the first MOS Hex record.

The termination record is explained as follows:

1. **;** – Start code.
2. **00** – Byte count is zero, no data or checksum in termination record.

Appendix H

Motorola S-Record Format

Motorola S-record Format[1]

INTRODUCTION

Motorola's S-record format for output modules was devised for the purpose of encoding programs or data files in a printable (ASCII) format. This allows viewing of the object file with standard tools and easy file transfer from one computer to another, or between a host and target. An individual S-record is a single line in a file composed of many S-records.

S-RECORD CONTENT

S-Records are character strings made of several fields which specify the record type, record length, memory address, data, and checksum. Each byte of binary data is encoded as a 2-character hexadecimal number: the first ASCII character representing the high-order 4 bits, and the second the low-order 4 bits of the byte.

The 5 fields which comprise an S-record are defined as follows:

[1]The information in this section was obtained from Internet sources. It is believed to be reliable, but no warranty on accuracy or usage is provided.

Field		Format	
Number	Type	Chars	Description
1	Record type	2	S-Record type — S1 or S9
2	Record length	2	The count (n) of the character pairs, excluding type and length.
3	Address	4	The 2-byte memory address to load the data field.
4	Data	0-2n	From 0 to n bytes of code or data, ($n \leq 32$).
5	Checksum	2	Checksum over fields 2 through 4.

The checksum consists of the least significant byte of the one's complement of the sum of the values represented by the pairs of characters making up the record length, address, and data fields.

Each record may be terminated with a CR/LF/NULL. Additionally, an S-record may have an initial field to accommodate other data such as line numbers. Accuracy of transmission is ensured by the record length (byte count) and checksum fields.

S-RECORD TYPES

Eight types of S-records have been defined to accommodate various encoding, transportation, and decoding needs. Lucid programmers use the 8-bit data types, the S1 and S9:

S1 – A record containing data and the 2-byte address at which the data is to reside.

S9 – A termination record for a file of S1-records. Only one S9-record is allowed per file and it must be the last line of the file. The address field for directly executable code may optionally contain the 2-byte address of the instruction to which control is to be passed. For ROM data the S9 address field is usually 0000. There is no data field.

S-RECORD EXAMPLE

The following is a typical S-record module:

```
S1130000285F245F2212226A000424290008237C2A
S113001000020008000826290018538123410018 13
S113002041E900084E42234300182342000824A952
S107003000144ED492
S9030000FC
```

The module consists of four S1 data records and an S9 termination record. The first S1 data record is explained as follows:

1. **S1** – S-record type S1, indicating a data record to be loaded/verified at a 2-byte address.

2. **13** – Hex 13 (decimal 19), indicating 19 character pairs, representing 19 bytes of binary data, follow. (2 bytes address, 16 bytes data, 1 byte checksum)
3. **0000** – Four-character 2-byte address field: hex address 0000, indicates location where the following data is to be loaded.
4. The next 16 character pairs are the ASCII bytes of the actual program data.
5. **2A** – Checksum of the first S1-record.

The second and third S1 data records also contain $13 character pairs each. The fourth S1 data record contains 7 character pairs.

The S9 termination record is explained as follows:

1. **S9** – S-record type S9, indicating a termination record.
2. **03** – Hex 03, indicating three character pairs (3 bytes) to follow.
3. **0000** – Four-character 2-byte address field, zeros.
4. **FC** – Checksum of S9-record.

Index

T

U

W

X

Z