The BASIC816 Language Manual

PJW

20 September 2019

About BASIC816

BASIC816 is an implementation of the BASIC programming language for the Western Design Center's 65816 microprocessor. It is focused on providing a simple BASIC interpreter for the C256 Foenix computer designed by Stefany Allaire. BASIC816 has a few design goals and even a couple of anti-goals:

- It should provide a retro-computing feel.
- It should be simple to use and easy to learn.
- It should provide essential access to storage and the C256's abilities.
- It should be expandable, allowing advanced users to customize it or extend it.
- It should be a clean-room implementation, unencumbered by copy-right.
- It need not be the fastest programming language available.
- It need not provide all the advancements in programming languages developed since the 1980s.

As such, BASIC816 is a fairly traditional, tokenized, line-number based implementation of BASIC, similar to those implementation of BASIC on the 8-bit computers of the 1970s and 1980s.

Screen Editing

BASIC816 for the C256 includes a simple screen editor. To enter a program, just type the lines, being sure to press the return key at the end of each line. The interpreter will not attempt to process the line until you press return, and the text it will process is whatever text is on the line of the screen where your cursor is. If you need to edit a line that's on the screen, you can use the cursor keys to move to the line and make your change. A few other special keys are supported:

Arrow Keys The arrow keys will move the cursor on the screen in a natural fashion.

Backspace The backspace key deletes the character to the left of the cursor.

Delete The delete key deletes the character under the cursor.

Insert The insert key inserts a space at the cursor (pushing the character that was under the cursor to the right).

Although not an edit key, the SysRq key is currently assigned as a "programmer's key" and will interrupt the currently running BASIC program and open the monitor. It should work for any program (even machine language), so long as the keyboard interrupt logic has not been intercepted. Likewise, the Scroll Lock key will pause any screen output from the interpreter while it is on. This includes output from a program that is producing a lot of text and even program listings.

Data Types

BASIC816 supports three data types:

Integer The integers in BASIC816 are 32-bit signed integers and may have values from $-2,147,483,648$ to $2,147,483,647$. Examples of integers are 0, 1, 42, -1, -128.

String Text data are stored in strings. A string may be entered into a BASIC816 program by enclosing it in double quote marks. Strings are represented internally as null-terminated ASCII strings, that is, there is no length data on the string, the end of which is marked by a 0 byte. Strings may be up to 65536 bytes in length.

Array Arrays allow multiple data items to be stored under a single name and accessed by an index. Two types of arrays are supported: arrays of integer and arrays of strings. Arrays may be just one-dimensional, using a single number as index, or they may be multi-dimensional, where each element has a unique combination of numbers serving as an index.

Note: when BASIC816 leaves the alpha stage of development, it will include a fourth type: 32-bit floating point numbers as well as arrays of 32-bit floating point numbers.

Literal Values

Integers and strings may both be entered directly into programs as literal values. Strings are designated with double quote marks, but there is no character escape syntax as in other languages. If you need to enter a character you cannot type, you will have to use the `CHR$` function. Integers may be entered as a sequence of digits with an optional minus sign at the start. The default base for integers is decimal, but hexadecimal numbers may be entered with the `&H` prefix.

- `"Hello, world!"`
- `12345`
- `-562`
- `&HFFD2`

Variables

A BASIC816 program can assign data to variables using the `LET` statement or an implicit `LET` statement. Variables need not be declared before use, with the exception of array variables, which must be declared through the `DIM` statement.

Variable names may be up to eight characters long. The first character must be an alphabetic character from A through Z. Subsequent characters of the name may be A–Z, 0–9, or the underscore character (_). The type of the variable is indicated by a character at the end of the variable name. Integer variables are indicated by the percent sign (%), and string variables are indicated by the dollar sign ($). These type designations must be used in all references to the variable and may be considered as part of the name. This means that a program can have six different variables with the same "name" but different types (e.g. `A`, `A%`, `A$`, `A()`, `A%()`, and `A$()`).

Expressions

An expression is a sequence of values, operators, and function calls which will have some data value as a *result*. An expression may have a numeric (integer) result or a text (string) result.

Operators are common mathematical symbols which will perform some sort of computation on two values. These are the usual sort of thing: addition, subtraction, multiplication, division, and so on. Operators are typically evaluated left to right, but have an operator precedence which can alter the order of execution. For instance multiplication operators are evaluated before addition. This precendence can be altered by using parentheses to enclose a sub-expression that should be evaluated as a unit.

Commands

Commands in BASIC816 are keywords which triggers some action but which may only be used at the interactive prompt. Commands may not be used within a BASIC816 program.

Operators	Purpose
^	Exponent
*, /, MOD	Multiply, Divide, modulo
+, -	Addition, Subtraction
<, >, = , <=, >=, <>	Comparison operators
NOT	Bitwise Negation
AND	Bitwise AND
OR	Bitwise OR

Table 1: Operators in order of descending precedence.

CONT

Continue execution of the current program from the point immediately after the STOP statement that was executed. It is an error to use this command if the current program was not interrupted by the STOP command or if there is no current program.

DIR

Print a listing of the files on the current file device. *NOTE:* currently, this command only works with a small RAM disk in bank 0x16. Eventually, this command will work with the IDE drive and SD card.

NEW

NEW clears all of BASIC816's memory. All program and variable data are erased, and a new program may be entered.

LIST [<start>] [- <end>]

LIST types a program listing to the console screen. It accepts two optional line-numbers to limit the listing. The first line number specifies the smallest line number to list; if it is not specified, there is no lower limit. The second line number specifies the largest line number to list; if it is not specified, there is no upper limit.

LOAD <filename>

Load the BASIC program with the given file name.

NOTE: currently, this command only works with a small RAM disk in bank 0x16. Eventually, this command will work with the IDE drive and SD card. This command will also eventually support other file types and loading destinations.

RUN

Starts execution of the program from the first line of the code.

Statements

Statements are keywords which trigger some action and may be used either at the interactive prompt or within a program. A BASIC816 program can be seen as a series of lines, each of which must have one or more statements on it. If a line has more than one statement, each statement is separated from the next by a colon (:).

CLR

Erases all variable definitions. Any variable that was defined prior to the use of CLR will be undefined. Also all data stored directly or indirectly through variables will be returned to free memory. This is similar to NEW, except that the program may be running and is left intact.

CLS

Clears the text screen and moves the cursor to the home position.

CALL <address>[, <a>[, <x>[, <y>]]]

Call a machine language subroutine at <address>. If the subroutine returns at all to the caller, it must return using the RTL instruction. Values may be provided for the A, X, and Y registers and may be 16-bit values.

DATA <value>[, ...]

Provide numeric or string data in the program which may be retrieved into a variable using the READ statement.

DIM <variable>(<size>[, ...])

Declare an array named <variable>. The array can have many dimensions, each with the size provided. An array can have up to 127 dimensions, and the size can be up to 256, but in practice the entire block of memory consumed by the array cannot exceed 65,536 bytes (including the "book keeping" memory allocated to keep track of the array (which is at most 256 bytes).

END

Stops execution of the program. The only way to restart execution after END has been executed is to use the RUN command.

FOR <variable> = <initial> TO <target> [STEP <increment>]

The FOR statement marks the beginning of a loop that will repeat a specified number of times. It starts by assigning an <initial> value to a <variable> and executing the following statement until the matching NEXT statement is encountered. It will then either add 1 to <variable> or <increment>, if provided. So long as <variable> is not <target>, the statements between the FOR and NEXT will be executed.

GET <variable>

Waits for the user to press a key, the assigns the character of that key to the string <variable> provided.

GOTO <line number>

Continue execution with the first statement on the line with the given <line number>. It is an error to GOTO a <line number> that does not exist in the program.

GOSUB <line number>

Call a subroutine starting at the line with the given <line number>. A subsequent RETURN statement will return the program to the statement following the GOSUB. It is an error to GOSUB a <line number> that does not exist in the program.

IF <expr> THEN <line>

Evaluates <expr> and examines the result. If the result is not zero, execution continues with the first statement on the line with number <line>. Otherwise, execution continues with the next statement after the IF.

Note: This statement will be receiving considerable improvements in subsequent versions of BASIC816.

INPUT [<message>;] <variable>

If a literal string message is provided, print the message followed by a question mark. If no message is provided, just print a question mark. In either case, wait for the user to type a line of input on the keyboard and put the value typed into the provided variable.

NOTE: Currently, this statement supports only string values and variables.

NEXT

Close the matching FOR loop.

POKE <address>, <value>

Write the 8-bit <value> to the memory location at <address>. It is an error to try to write a value that requires more than 8-bits.

POKEW <address>, <value>

Write the 16-bit <value> to the memory location at <address>. The low byte of <value> will be written to <address>, and the high byte of <value> will be written to the following byte in memory. It is an error to try to write a value that requires more than 16-bits.

POKEL <address>, <value>

Write the 24-bit <value> to the memory location at <address>. The low byte of <value> will be written to <address>, and the middle byte of <value> will be written to the following byte in memory, and the high byte of <value> will be written to the next byte in memory. It is an error to try to write a value that requires more than 24-bits.

PRINT [<value> [,/;]] ...

Write the textual representation of <value> to the screen. If more than one <value> is provided, they must be separated by either a comma (,) or a semicolon (;). If a comma is used, the two items will be separated by a TAB. If a semicolon is used, the two items will be printed one after the other. A PRINT statement will print a carriage return as the last thing, unless the statement is ended with a semicolon.

REM <comment>

Inserts a comment into the BASIC816 program. All characters after the REM until end of the line will be ignored.

READ <variable>[, ...]

Read one of more values out of the DATA statements in the program into <variable>. The data read must have a compatible type to the variable. Each variable read will advance a data cursor forward one data element. If a READ is executed when the cursor has reached the end of the data elements, it is an error.

RESTORE

Resets the data cursor to the first data element of the first DATA statement.

STOP

Stops execution of the program in such a way that the user can restart it with the CONT command.

Functions

ABS(<value>)

Returns the absolute value of <value>. If the parameter is negative, it is converted to its positive equivalent (for instance, ABS(-5) will evaluate to 5).

ASC(<text>)

Returns the ASCII code for the first character of <text>. Example: ASC("A") returns 65.

CHR$(<value>)

Returns the character corresponding to the ASCII code in <value>. Example: CHR$(65) returns "A".

DEC(<hex>)

Returns an integer that is conversion of the hexadecimal number in the string <hex>. Example: DEC("A0") returns 160.

HEX$(<value>)

Returns a string that is the hexadecimal representation of the integer value passed. Example: HEX$(160) returns "A0".

LEFT$(<text>, <count>)

Returns the left-most <count> characters of the string <text>.

Example: LEFT$("Hello", 3) returns "Hel".

MID$(<text>, <first>, <count>)

Returns a substring of the string <text>. The parameter <first> specifies the number of the first character to use, where a 0 is the number of the first character in the source string. The parameter <count> indicates how many characters should be returned.

Example: MID$("Hello", 2, 3) returns "llo".

PEEK(<address>)

Returns the byte stored in memory at location <address>.

PEEKW(<address>)

Returns the 16-bit word stored in memory at location <address>. The low byte of the returned value is at <address>, and the high byte is at address + 1.

PEEKL(<address>)

Returns the 24-bit word stored in memory at location <address>. The low byte of the returned value is at <address>, the middle byte is at `address + 1`, and the high byte is at `address + 2`.

RIGHT$(<text>, <count>)

Returns the right-most <count> characters of the string <text>.
Example: `LEFT$("Hello", 3)` returns `"llo"`.

SGN(<value>)

Returns the sign of the number <value>. If the number is negative, the result is `-1`, if the number is positive, the result is `1`, and if the number is zero, the result is `0`.
Example: `SGN(-25)` returns `-1`.

SPC(<value>)

Returns a string containing <value> spaces.
Example: `SPC(5)` returns a string of five spaces.

STR$(<value>)

Returns a string containing the decimal representation of the number <value>.
Example: `STR$(25)` returns `"25"`.

TAB(<value>)

Returns a string containing <value> TAB characters.
Example: `TAB(2)` returns a string of two TABs.

VAL(<text>)

Returns the numeric value represented by the string of decimal digits in <text>.
Example: `VAL("42")` returns 42.

C256 Specific Statements

BASIC816 includes a number of statements to support features of the C256 Foenix.

CLRPIXMAP

Sets all pixels in the bitmap to 0 (transparent). The bitmap must have been setup with the `PIXMAP` statement before this statement can be used.

FILL <x0>, <y0>, <x1>, <y1>, <color>

Fill the rectangle with corners at (x_0, y_0) to (x_1, y_1) with the color specified by the <color> index number. The actual color displayed will depend on the RGB values provided for that index in the color lookup table currently used by the bitmap. The `PIXMAP` statement must have been used before using this statement.

GRAPHICS <mode>

Sets the graphics mode of the Vicky chip. This statement just sets the master control register of the Vicky chip to whatever value is provided in `<mode>`, which should be a number in the range 0–255.

The bits of the master control register can be seen described in Table 2.

Bit	Purpose
7	Global disable: If set, this disables all Vicky output.
6	Gamma Correction: If set, this enables gamma correction.
5	Sprites: If set, this enables the sprite engine.
4	Tiles: If set, this enables the tile engine.
3	Bitmap: If set, this enables the bitmap engine.
2	Graphics: If set, this enables the graphics blocks.
1	Text Overlay: If set, text will be displayed over graphics.
0	Text: If set, this enables the text display engine.

Table 2: Vicky Master Control Regiser

LINE <x0>, <y0>, <x1>, <y1>, <color>

Draw a line on the bitmap display from positions (x_0, y_0) to (x_1, y_1) in the color specified by the `<color>` index number. The actual color displayed will depend on the RGB values provided for that index in the color lookup table currently used by the bitmap. The `PIXMAP` statement must have been used before using this statement.

MONITOR

Enter the machine language monitor.

PIXMAP <visible>, <lut> [, <address>]

Controls the visibility of the bitmap display (the bitmap engine must be enabled with the `GRAPHICS` command). The parameter `<visible>` must be a number. If it is 0, the bitmap will be hidden. If it is any other value, it will be shown.

The number of the color lookup table to use must also be provided as `<lut>`, which must be the number of one of the graphics color lookup tables.

Optionally, an address may be provided for the first byte of the bitmap. This address must be in the video memory area starting at B00000. If no address is provided, it will default to B00000.

PLOT <column>, <row>, <color>

Set the pixel at the specified `<row>` and `<column>` to the color with the given `<color>` index number. The actual color displayed will depend on the RGB values provided for that index in the color lookup table currently used by the bitmap. The `PIXMAP` statement must have been used before using this statement.

SETBGCOLOR <red>, <green>, <blue>

Sets the background color to be used in graphics modes. The color is specified as red, green, and blue components, all of which must be in the range 0 – 255.

SETBORDER <visible> [, <red>, <green>, <blue>]

Sets the visibility and, optionally, color of the border of the C256's screen, given <red>, <green>, and <blue> intensity values, which must be between 0–255. The parameter <visibility> must be a number: if it is 0, the border will not be shown; if it is any other number, the border will be visible.

SETDATE <day>, <month>, <year>

Sets the date on the C256's real time clock, given the date as three numbers: <day>, <month>, and <year>. The day number must be from 1–31. The month number must be from 1–12. The year number must be from 0–99.

SETCOLOR <lut>, <color>, <red>, <green>, <blue>

Define the RGB intensities for a color in one of the Vicky chip's color lookup tables. The color lookup table is a number <lut> in the range 0–9. The number of the color to change is <color>, which is in the range 0–255 for all graphics lookup tables and 0–14 for the two text mode lookup tables. The RGB intensities are provided by the parameters <red>, <green>, and <blue>, which must be in the range 0–255.

The association of <lut> number to the color lookup table in Vicky is shown in Table 3.

LUT	Lookup Table	Max Color
0	Graphics LUT 0	255
1	Graphics LUT 1	255
2	Graphics LUT 2	255
3	Graphics LUT 3	255
4	Graphics LUT 4	255
5	Graphics LUT 5	255
6	Graphics LUT 6	255
7	Graphics LUT 7	255
8	Text Foreground LUT	15
9	Text Background LUT	15

Table 3: Color lookup table numbers

SPRITE <number>, <lut> [, <address>]

Set the color lookup table and the address for sprite <number>. The parameter <lut> must be the number of the graphics lookup table the sprite should use (0–7). The optional parameter <address> would be the first byte of the sprite's bitmap data and must be an address in the video RAM (starting at address B00000). If no address is provided, the sprite will default to B00000.

SPRITEAT <number>, <column>, <row>

Sets the location of a sprite. The sprite to move is <number>. Its upper-left corner will be set to <row> and <column>.

SPRITESHOW <number>, <visible> [, <layer>]

Controls the visibility of sprite <number>. If <visible> is 0, the sprite will not be visible. If it is any other number, the sprite will be visible. Optionally, the display priority <layer> may be set, which will control which sprites and tiles this sprite will display in front of or behind when they overlap.

`TEXTCOLOR <foreground>, <background>`

Sets the color for text to be printed. The foreground color index is specified by `<foreground>`, and the background color index is specified by `<background>`. Both values must be in the range 0 – 15.

`SETTIME <hour>, <minute>, <second>`

Sets the time on the C256's real time clock, given the time as three numbers: `<hour>`, `<minute>`, and `<second>`.

C256 Specific Functions

`GETDATE$(0)`

Returns the current date from the C256's real time clock as a string in `"DD:MM:YY"` format.

`GETTIME$(0)`

Returns the current time from the C256's real time clock as a string in `"HH:MM:SS"` format.

Proposed: Extentions

The following is a proposal for an extension mechanism BASIC816 may provide in the future. It is not currently implemented in the interpreter.

While BASIC816 allows a program to use `CALL` to run any machine code loaded into memory, this does not extend the language. For more complex integration of machine language and BASIC, the interpreter will allow for the program to load a binary file containing code for extending the commands, statements, and functions of the interpreter. An extension may be loaded with the statement:

```
EXTENSION <path>
```

Executing this statement will load the extension file off the storage device, given its path. The statement will also implicitly execute a `CLR` statement, removing all variable definitions and heap storage objects. The reason for this is that the extenstions will be loaded into what would be the top of the heap, and the heap will need to be moved down to accomodate the extension. Therefore, if a program needs to use an extension, the first thing it should do is to load the extension.

Note that multiple extensions can be loaded at one time. The interpreter will assign an extension token sequence to each keyword defined by the extensions. Since multiple extensions can be used, the token sequence can vary from program to program. That means that if an extension provides a statement `PLAY`, it might have the token $F1 in one program, but in another program, it might have the token $F8.

Coding an Extension

An extension will be a binary file on a storage device (SD Card, floppy disk, hard drive). Each extension will be loaded into its own memory bank of 64KB. The extension will have free use of all memory within that bank for whatever purpose it desires. The beginning of that memory bank must be laid out in a particular structure, however (see below). While the extension will always be loaded so that the first byte of the file corresponds to the first byte of the bank, the particular bank the extension will be loaded into will be determined at runtime by the interpreter. The memory size of the computer and the load order of other extensions the program may use can affect the particular bank used. All addresses used by the extension for accessing its own memory should therefore be 16-bit addresses (*e.g.* bank relative, PC relative, *etc.*).

The following pseudo-C code is a simple map of the an extension, starting with byte 0:

```
struct {
    char magic[3];              // A three-character magic code indicating this is an extension
    uint8_t version;            // A single byte version code for the extension format
    uint16_t init_ptr;          // A 16-bit offset to any initialization code (0 if none)
    struct {
        uint8_t type;           // A byte indicating the type of the token (statement, comment, function, o
        uint16_t name_ptr;      // A 16-bit offset to an ASCIIZ string for the keyword of the token
        uint16_t exec_ptr;      // A 16-bit offset to the code to call to evaluate the token
    } tokens[];
    uint8_t eot;                // Always equal to $FF
    uint8_t code[];             // The code for the extension
}
```

The first three bytes are the required magic number (TBD), identifying the file as a BASIC816 extension. The fourth byte is a binary version number. It will be $00 for the first version. If there are any changes to the extension layout, they can be flagged here.

Next comes an offset pointer (16-bits) to an initialization routine that will be called when the extension has been loaded but before the `EXTENSION` statement completes execution. This can be used for any sort of set up housekeeping the extension needs to do. If there is nothing to do here, the offset should be set to $0000.

Next comes a token record, one per keyword provided by the extension. The record contains a byte indicating what type of keyword is being defined (statement, function, command, operator, or even punctuation). The next byte indicates the evaluation precedence (relevant for operators). Then there is a 16-bit offset pointer to the name of the keyword (an upper-case ASCIIZ string). Then there is a 16-bit offset pointer to the code to execute when executing the keyword. The subroutine at that location should return with an `RTL` instruction.

After all the token records have been provided, the next byte should be $FF to indicate there are no more records to process. All other bytes in the file after that point are left for the extension's use. They may be arranged however the extension author desires.

C256 Character Set

The character set for BASIC816 on the C256 is assummed to be ASCII, with each character taking a full 8-bit byte. There are a number of special characters the system supports, shown in Table 4.

Code	Purpose
0x08	Delete character to the left and move cursor left
0x09	Move cursor to next tab
0x0A	Line Feed
0x0D	Carriage Return
0x11	Move cursor up a line
0x1D	Move cursor right one column
0x91	Move cursor down a line
0x9D	Move the cursor left one column

Table 4: Special Character Codes

Temporary RAM Disk

Until a full file I/O system is ready, there is temporary support for a read-only RAM disk. Only two commands are implemented to support it: `DIR` to print a directory listing, and `LOAD` to load a BASIC program.

The RAM disk is implemented as a section of memory from 0x360000 to 0x37FFFF, so it is only 128KB. Each file is preceded by a 24 byte header which describes the file and links to the next file:

```
NAME        .fill 8  ; The 8-character name of the file
EXTENSION   .fill 3  ; The 3 character extenion of the file
ATTRIBUTES  .fill 1  ; Reserved/unused
SIZE        .fill 4  ; The size of the file in bytes
HANDLE      .fill 4  ; Address of the file contents in the RAM disk
NEXT        .fill 4  ; Address of the next file header in the RAM disk
```

Included with the source code for the BASIC interpreter is a Python script, `mkvfs.py` which will read the contents of a directory and pack them in the correct format into a binary file that can be uploaded by the FoenixIDE into the RAM disk memory location. The script is very simple, however, and does not check if the resulting file is too big, so use it at your own risk! If it is too big, it will over-write the kernel.

DIR

Prints the names of all the files in the RAM disk. If no RAM disk image has been loaded, this command will behave unpredictably.

LOAD <filename>

Reads the BASIC program with the given `<filename>` as if you had typed it manually. Prints a series of dots as a simple progress indicator with one dot per line read. The files must be in simple ASCII text format.

The Monitor

BASIC816includes a simple, built-in machine language monitor. The monitor can be summoned using the `MONITOR` command, but it will also be activated if a machine language program uses the `BRK` instruction.

The monitor works like a lot of other monitors. Commands are entered as single lines of text, with a single character command code as the first character and arguments following on the rest of the line.

Monitor Commands

- `A <address> <assembly>`—Assemble a line of 65816 assembly code given a starting address.
- `C <address1> <address2> [<length>]`—Compare two blocks of memory, and lists addresses that differ. The third argument `<length>` is optional, and is the number of bytes in the blocks to compare. If no `<length>` is provided, only one byte will be compared at the two addresses.
- `D <start> [<end>]`—Attempt to disassemble the memory starting at the address `<start>` and continuing until the address `<end>` (if no `<end>` address is provided, it will attempt to disassemble 256 bytes).
- `F <start> <end> <byte>`—Fill a block of memory with a value. The block of memory is specified by the `<start>` and `<end>` addresses, and the value to store is specified by the `<byte>` argument.
- `G [address]`—Start executing the code at the provided `<address>`. The `G` command starts executing the code with a `JSL` instruction, so the code can return to the monitor with either the `RTL` or the `BRK` instruction.
- `J [address]`—Start executing the code at the provided `<address>`. The `J` command starts executing the code with a `JML` instruction, so the code can return to the monitor only with the `BRK` instruction.
- `H <start> <end> <byte> [byte]...`—Hunt through a block of memory for a sequence of `<byte>`s. The block of memory is specified byte the addresses `<start>` and `<end>`. Up to eight `<byte>`s may be provided, and a matching address will be printed only if that exact sequence of bytes starts at that address.

- `M <start> [end]`—Display the values stored in the memory block specified the addresses `<start>` and `<end>`. If no `<end>` is provided, a block of 256 bytes will be displayed. The memory dump contains both the hexadecimal values and an ASCII interpretation of the data.
- `R`—Display the values of the CPU registers at the point the monitor was invoked.
- `; <PC> <A> <X> <Y> <SP> <DBR> <DP> <NVMXDIZC>`—Update the values of of the CPU registers. These new values will take effect when the monitor starts executing other code or returns to the code that invoked it originally. All register values are expressed in hexadecimal except the status flags, which are binary values and must be either `0` or `1`.

  NOTE: The output of the `R` command is in the format of the `;` command. This allows registers to be edited simply using the built in screen editing functions of BASIC816 and the monitor.
- `T <start> <end> <destination>`—"Transfer" or copy a block of data from one area of memory to another. The source of the data to copy is the block of memory specified by `<start>` and `<end>`. The destination block starts at `<destination>`.
- `W <byte>`—Sets a control value to specify what value the assembler and disassembler should assume is in the CPU status register at the start of the code. This allows you to specify what the assembler and disassembler should assume for the `M` and `X` flags in the processor status register, as these flags affect the sizes of the operands for immediate mode instructions.
- `X`—Attempt to return to the program that invoked the monitor. This will usually return to BASIC816, but it could also return to another program, if that program used a `BRK` instruction. This command will not work if the stack has been modified too much, or if the calling program is not written assuming a BRK can return (and leaves a byte of "data" immediately after the `BRK` itself).
- `> <start> <byte> [byte]...`—Write `<byte>`s of data to a block of memory. The block starts at the address `<start>`. Up to eight `<byte>`s of data can be provided to the command.

  NOTE: the `M` command returns memory in this format (the ASCII preview is ignored by `>`). This allows memory to be edited easily using the built-in screen editing features of BASIC816 and the monitor.
- `?`—Prints a brief help message to remind you how to use the commands.

Assembly and Disassembly Details

The assembler and disassembler use the standard mnemonics for the 65816 CPU, and will assemble or disassemble one instruction per line. All addresses are listed as full 24-bit addresses, even if they are 16 or 8 bit offsets. There is no support in the monitor for labels, so all addresses must be hard coded.

In many respects, the monitor works like most other simple, 6502-based monitors you may have encountered, but there are two key differences that come from the architecture of the 65816.

The M and X Flags

The `M` and `X` flags can change the length of immediate mode instructions. If thge `M` flag is set, an immediate `LDA` instruction will take a single byte operand, but if `M` is clear, the instruction requires a two byte operand. Similarly, the `X` flag will control the length of `LDX` and `LDY` instructions.

This being the case, both the assembler and the disassembler need to be told what the flags will be for a given section of code in order to either generate the correct byte sequence or to correctly interpret a sequence of bytes into an instruction.

To support this, the monitor includes a `W` command, which lets you tell the monitor what values to assume for the flags. Furthermore, both the assembler and disassembler will watch for `REP` and `SEP` instructions, and track changes to the flags to try to be intelligent about the the lengths of those instructions.

Addressing Modes

Addressing modes on the 65816 can be ambiguous. An instruction `LDA $26` could resolve to any of three different addressing modes! The table below lists the address mode syntax and the associated addressing modes and operand sizes (`d` stands for a hexadecimal digit).

Syntax	Address Mode	Operand Size (bits)
`A`	Accumulator addressing mode	N/A
`dd:dddd,X`	Absolute Indexed X Long	24
`dd:dddd`	Absolute Long	24
`dddd,X`	Absolute Indexed X	16
`dddd,Y`	Absolute Indexed Y	16
`dddd`	Absolute	16
`dd,X`	Direct Page Indexed X	8
`dd,Y`	Direct Page Indexed Y	8
`dd,S`	Stack Relative	8
`dd`	Direct Page	8
`#dddd`	Immediate	8 or 16[1]
`#dd,#dd`	MVN/MVP special	16
`#dd`	Immediate	8
`(dd,S),Y`	Stack Relative Indexed Y	8
`(dddd,X)`	Absolute Indexed X Indirect	16
`(dddd)`	Absolute Indexed	16
`(dddd),Y`	Absolute Indexed X	16
`(dd,X)`	Direct Page Indexed X Indirect	8
`(dd),Y`	Direct Page Indirect Indexed Y	8
`(dd)`	Direct Page Indirect	8
`[dddd]`	Absolute Indirect Long	16
`[dd],Y`	Direct Page Indirect Indexed Y Long	8
`[dd]`	Direct Page Indirect Long	8