SPL – A Simple Programming Language

Ron Kneusel
oneelkruns@hotmail.com

December 2007

AM-DG

Table of Contents

Introduction

This document describes the SPL language, compiler, and supporting files. It details how to install it, how to configure it and how to program using it.

What is SPL?

SPL stands for “Simple Programming Language”, and it describes both the SPL programming language and the compiler for that language. The compiler takes SPL source code and produces output for the 6502 microprocessor. The 6502 is an old 8-bit microprocessor that was common in personal computers of the late 1970s and early 1980s including the Apple II line, for which SPL is somewhat tailored.

The SPL compiler is implemented in Python and as such runs on all modern operating systems and even some old ones like VMS, if you are fortunate enough to have an old VMS machine around. The assembly output is automatically assembled using an old C-based assembler. This assembler is generic C and pre-compiled binaries are provided for Linux, Macintosh, and Windows computers. If you want the VMS version, email me, as it works just fine there, too.

The SPL compiler produces reasonably compact code well suited to a simple architecture like that of the 6502. It also comes with a library of assembly functions as well as SPL modules for higher-level routines.

The SPL language itself is Forth-like. It is stack-based but is not another Forth. Note the the source code generated is vanilla 6502, not 65C02 which included additional op-codes. The as6502 assembler is also 6502 specific.

Why?

Why not? I've always toyed with the idea of a simple cross-compiler for the Apple II, something that could generate compact, reasonably fast code. So, that alone is justification enough for me. I'm putting it out for others in case they find it helpful or at least amusing. Funny as it may sound, I still use my Apple IIs for serious work (ie, paying work). They are handy when dealing with small serial devices, either to talk to the device or to mimic the device during development with a desktop system. As an aside, most of that work has been done with QForth, a simple Apple II Forth that I have maintained, bug fixed, and enhanced over the years (I did not write it, however). I use custom assembly words on a stock Apple IIgs running in 8-bit mode as this allows for serial speeds up to 57600 baud easily enough. Most of the devices I use are in the 4800 to 38400 baud range.

SPL is based on PIC0, another, even simpler, language I wrote for the PIC 10F series microcontrollers from Microchip. That compiler, should you be interested, should be available on the same website on which you found SPL. If you can't find it, send me an email.

Installation and Configuration

The SPL compiler requires Python to be installed. If you are on Linux/Unix you probably already have it. If not, or you are on Windows, then get it here, www.python.org, and install it.

Special note for Windows users: copy python.exe to C:\WINDOWS (or add it so that it is in your PATH). This is required to use the command prompt for compiling.

Configuration

SPL needs to know the location of the lib and include directories. These are read from environment variables. If the environment variables are not found it is assumed that you are running the compiler in the spl/ directory and it will simply look for lib/ and include/ in that directory.

To define your environment variables under Linux/Mac do something like:

export SPL_INCLUDE_PATH = <your spl/include/ path>
export SPL_LIB_PATH = <your spl/lib/ path>

for bash. Your .bashrc might be a good place for these. If using csh the idea is the same, just the syntax changes:

setenv SPL_INCLUDE_PATH <your spl/include/ path>
setenv SPL_LIB_PATH <your spl/lib/ path>

Note that the trailing "/" is required.

For Windows, you need to set the environment variables in a different way. Go to My Computer, right-click and choose "Properties". Then select "Advanced" and "Environment Variables". Enter the new variables setting the path to the location you installed SPL. For example, you might set SPL_LIB_PATH to C:\spl\lib\

The above is the minimum configuration necessary to use SPL. Naturally, one is free to wrap the call to the compiler in a shell script or .bat file.

Using the Compiler

The compiler is a command line only tool, there is no GUI or IDE. You are free to chose your own editor. My favorites are vi and kate for Linux and EditPlus for Windows. This section will describe the location of source code files, how to call the compiler, and command line options.

Source Code Files

All source code files necessary for your application must be given on the command line. The expected extension for all files is .spl and it will be supplied if not given. If the mentioned source code file is not

found in the current working directory, ie, the directory in which you call the compiler, then SPL will look for a file with that name in the include directory. Therefore, any modules you wish to re-use should be placed there for easy access. Just mention them on the command line and they will be found. To SPL, all the source code files are treated as one single large file. It is not necessary to have defined a function before referring to it, therefore, no forward references or prototypes are required.

Invoking the Compiler

The compiler is called as a normal Python script though #! is set so that you can make the spl.py file itself an executable under Linux or Mac. If no arguments are given a simple help is displayed:

```
$ python spl.py

SPL Compiler (25-Nov-2007)

Use: spl file1 [file2 ...] [-o outbase] [-t type] [-sys] [-org n] [-var m] [-stack p]
         [-prodos]

Where:

    file1     =  first source code file (.spl extension assumed)
    file2...  =  additional source code files
    outbase   =  base file name for output files (NO extension)
    type      =  output file type:
                     bin = compiled binary
                     asm = assembly source only
                     a2t = text file for EXEC on Apple II
                     dsk = a disk file for Apple II emulators
                     r65 = source file for the r65 65C02 simulator
    sys       =  create a ProDOS system file (-t a2t and -t dsk only)
    org       =  set the origin address to n
    var       =  set the VARTOP value (variables grow down from here)
    stack     =  set the stack address (up to 256 bytes)
    prodos    =  set the VARTOP value to allow 3 ProDOS file buffers
```

optional arguments are in square brackets. Therefore, at a minimum, a single .spl source code file must be given and that is all. Output files generated are written to the current working directory. If no output base filename is given the value in the source code is used, the default being “out”.

Note that the compiler is “smart” and that it will only output routines that are actually used. This is particularly helpful when using a library of routines as only those pieces of code that are actually used will be compiled. SPL does not have a “runtime library” which will waste space with code that is not used.

Command Line Options

The SPL compiler supports a few command line options, all of which are optional. They are described here:

Option	Description
`-o outbase`	Defines the base filename for the output files. Exactly which files are created depends

<table>
<tr><td></td><td>upon the output type selected but all will have this base name and different extensions.</td></tr>
<tr><td>-t type</td><td>Defines the output file type. These determine which files are created. Possible options are:
<table>
<tr><th>Type</th><th>Output</th></tr>
<tr><td>bin</td><td>This is the default. A .bin file is the binary output of the assembler. Therefore, a .s file, the assembly source, and a .bin file will be created.</td></tr>
<tr><td>asm</td><td>This option produces the .s assembly source code file only.</td></tr>
<tr><td>a2t</td><td>Creates .s, .bin and .txt files. The .txt file is a text file for Apple II computers running ProDOS. If this file is EXEC-ed on an Apple II it will load the compiled program into memory and store it on disk.</td></tr>
<tr><td>dsk</td><td>Creates .s, .bin and .dsk files. A .dsk file is a disk image file for Apple II emulators. This file can be used as a normal 140k floppy disk in an Apple II emulator such as AppleWin. The disk image will have a single file on it named “A.OUT” and it will be a standard binary file unless the -sys option is set in which case it will be a ProDOS SYS file.

Note that there is a file in the library directory called “blank.dsk”. This is a blank Apple II disk image and this file is copied and modified to create the output disk image.</td></tr>
<tr><td>r65</td><td>Creates .s, .bin and .r65 files. The .r65 file is nearly identical to the .bin file but the first two bytes are the low and high byte of the starting address at which the compiled code is to be run. This is for a future 6502 simulator but the file format might prove useful elsewhere.</td></tr>
</table></td></tr>
<tr><td>-sys</td><td>Include this option to produce a ProDOS SYS file on the output disk image. Only meaningful if -t dsk is given.</td></tr>
<tr><td>-org n</td><td>Override the default code origin given near the start of the spl.py file and use the address n as the origin. The number n must be given in valid SPL format.</td></tr>
<tr><td>-var m</td><td>Override the default VARTOP location given in the spl.py file. VARTOP is the address at which variable space will start. Note that variable space starts at VARTOP and works downward towards lower memory.</td></tr>
<tr><td>-stack p</td><td>Override the stack address given in the spl.py file. Normally, it is wise to allow 256 bytes for stack space but this is not required if it is known that only a few values will ever be on the stack at one time.</td></tr>
<tr><td>-prodos</td><td>Set VARTOP to allow room for three ProDOS file buffers. These are used with the fopen command (0, 1, 2).</td></tr>
</table>

A Terse Stack Primer

If you are already familiar with Forth (or another stack-based language), feel free to skip this section.

For those new to stack-based languages read on as there is a bit of a learning curve to using the stack.

The stack is a last on-first off data structure. Think of a stack of cafeteria trays where new trays are always taken from the top. In a stack-based language tokens, which are sequences of characters separated by whitespace, are either numbers which are pushed on the stack or functions (traditionally, "words") which are executed and pull their arguments off the stack. In the case of SPL, strings enclosed in double quotes are also considered tokens. The string itself cannot be pushed on the stack, what gets pushed on the stack is the address of where that string is in memory.

So, the general rule is, argument values get pushed on the stack, functions pull arguments off the stack and return values to the stack. This has certain advantages, for example, functions can take an arbitrary number of arguments and return an arbitrary number of values.

Traditionally, the documentation for a function includes the a "stack effect" comment. This is a comment that shows what values are expected on the stack and what values are returned to the stack by the function. For example, the function "+" has a stack effect comment like "(a b -- a+b)" which means that "+" expects two arguments on the stack, listed from left to right being bottom to top, and returns a single value to the stack which is the sum of the arguments. Notice that functions consume their arguments removing them from the stack. For example, do implement the expression (1+3)*4/6 you would enter:

```
1 3 + 4 * 6 /
```

which would leave the answer on the stack. To print the 16-bit value on the top of the stack use "." (dot). Writing mathematical expressions in this way is known at postfix or RPN ("Reverse Polish Notation") for a Polish mathematician who first recommended this approach. Postfix expressions never need parentheses or operator precedence.

SPL includes functions for manipulating the stack. Some of them are with their stack effect comments are:

```
dup     ( a -- a a )          duplicate the top-most stack item
drop    ( a -- )              drop the top-most stack item
swap    ( a b -- b a )        swap the top two stack items
over    ( a b -- a b a )      copy the next-to-top item to the top
nip     ( a b -- b )          drop the next-to-top stack item
rot     ( a b c -- b c a )    rotate the top 3 stack items
depth   ( -- n )              put the number of items on the stack on the stack
```

All are described more fully in the reference section below. Recall, these words work on the given top-most stack items. They have no effect on any items below the ones they use.

The few paragraphs above barely scratch the surface of the sort of expressions that are possible in a stack-based language. For a much richer stack-based language take a look at Forth, especially modern Forth systems.

The SPL Language

SPL was designed to be simple and low-level but not so low-level as to be hard to use. In SPL all variables are global and there are no types. Variables are defined with the number of memory locations (bytes) to be reserved for the variable. If the variable is a character one byte is needed, if a 16-bit integer, 2 bytes are needed, and if a 32-bit integer 4 bytes are needed (32-bit arithmetic is fully supported). Larger memory structures and arrays can be built by indexing into regions of bytes reserved for a variable. Referring to a variable pushes the address of the variable on the stack just as in Forth.

Numbers

SPL supports 8-bit, 16-bit and 32-bit signed numbers. The stack is 16-bit so a 32-bit number is represented as two 16-bit values, low then high. This is how traditional Forth systems dealt with 32-bit numbers.

Numeric constants can be given in decimal (signed or unsigned), or as unsigned hexadecimal or unsigned binary. For hexadecimal use the C style "0x" prefix. For binary use "0b" (case is irrelevant). Number use up as much space as is needed to represent them unless they are followed by a "%" or "#" suffix to make them 16-bit or 32-bit respectively. For example,

```
123       #  8-bit
1234      #  16-bit
-123      #  16-bit, negative numbers always 16-bit or more
0xff      #  8-bit
0x3344    #  16-bit
0b1101%   #  16-bit
0xff#     #  32-bit
0xff%     #  16-bit
1%        #  16-bit
```

Number sizes matter most when using data blocks (described below). If an 8-bit number is put on the stack the high byte will simple be zero.

Variables and Constants

SPL supports constant as well as variables. A constant pushes its value on the stack when referenced, a variable pushes its address. Variables and constant may be declared anywhere in the source code outside of a function or data block declaration. They are defined as:

var <name> <#bytes>
const <name> <value>

A valid variable name (and constant, and function and data block name) consists of letters, numbers, and underscores. It must start with a letter, can be of any length, and *is* case sensitive.. For example, consider the following variable and constant declarations:

```
var i             2          -  reserve 2 bytes
```

```
var base_address 512           -  room for a data structure
var cost           4           -  a 32-bit integer

const COUT      0xfded         -  refer to a ROM location
const sevenBits 0b01111111     -  a binary value
const PORT0     254            -  can read/write with c@ and c!
```

Variables are at a fixed location in memory, if you need to refer to a specific memory location use a constant. SPL does not support dynamic memory allocation (yet).

Variables put their address on the top of the stack when referenced. How to access the contents of the address depends upon the type of data stored there. Direct access for 8-, 16- and 32-bit words is provided in SPL. To get and set variables use:

```
@   ( addr -- v )  fetch the 16-bit value at addr (low then high bytes)
c@  ( addr – v )       fetch the byte at addr  (still 16-bit on the stack)
d@  ( addr – lo hi )   fetch the 32-bit value at addr (two 16-bit words)
!   ( v addr -- )      store the 16-bit stack value as a 16-bit stack
c!  ( v addr -- )      store the byte in v at addr
d!  ( lo hi addr -- ) store a 32-bit value at addr
```

Some examples,

```
var k 2            Define k as a 16-bit integer
var c 1            Define c as byte
var d 4            Define d as a 32-bit number
k @                Get the 16-bit value at k
123 k !            Store the 16-bit value, 123, in k
d @ dup + d d!     Get a 32-bit value, double it, and store it back in memory
key c c!           Store the character typed in c
```

Strings

SPL supports two kinds of string constants. The first are those defined in a function. These strings push the address of their text on the stack, most often for use with words like `disp`. Note that unlike other languages, these strings are fair game for manipulation by the running program and that it is possible to store their address in a variable. Strings are enclosed in double or single quotes and may not span more than one line of source code. The second kind of string constant is declared outside of a function and is available for use by any function. Referring to this string will also push its address on the stack.

Some examples:

```
def main
  “Hello world!”  #  A string constant, pushes its own address on the stack
  disp cr         #  display the string and move to the next line
end

str prompt1   “This is prompt one!”
str abc       “abcdefg...”
```

Control Structures

True to form, SPL only supports two control structures. There are infinite loops defined by "`{`" (loop start) and "`}`" (loop end) and an "`if`", "`else`", "`then`" structure.

Loops

SPL loops are all infinite and must be escaped with a variation of "`break`". Loops may be nested as in other programming languages. There are several options for breaking out of a loop or for causing it to move to the next iteration. These are given here:

Keyword	Effect
`break`	Unconditional break. Exit the inner-most loop immediately.
`?break`	Conditional break. Exit the inner-most loop if the top of stack item is true (not zero).
`?0break`	Conditional not-break. Exit the inner-most loop if the top of stack item is false (zero).
`cont`	Jump to the beginning of the inner-most loop.
`?cont`	Conditional continue. Jump to the beginning of the loop if the top of stack item is true.
`?0cont`	Conditional not-continue. Jump to the beginning of the current loop if the top of stack item is false.

This simple looping mechanism allows for all the looping constructs expected in a structured programming language without resorting to goto statements.

Constructing While, Repeat, and For Loops in SPL

This section illustrates how to create structured programming loops in SPL.

While

A while loop is a top-tested loop:

```
while (<condition_is_true>) do
  <loop body>
end
```

where the <loop body> is executed repeatedly as long as <condition_is_true> returns a true value. For example,

```
i = 0
while (i < 100) do
  print i
  i = i + 1
end
```

becomes,

```
0 i !                  #  set i to zero
{                      #  start a loop
  i @ 100 >= ?break    #  if i is 100 or greater, exit
  i @ . cr             #  print i
  i @ 1+ i !           #  increment i by 1
}
```

or even,

```
0 i !                  #  set i to zero
{                      #  start a loop
  i @ 100 < ?0break    #  if i is not less than 100, exit
  i @ . cr             #  print i
  i @ 1+ i !           #  increment i by 1
}
```

Repeat

A repeat loop is bottom-tested:

repeat
  <loop body>
until (<condition_is_true>)

for example,

```
i = 0
repeat
  print i
  i = i + 2
until (i > 100)
```

becomes,

```
0 i !
{
  i @ . cr
  i @ 2+ i !
  i @ 100 > ?break
}
```

or,

```
0 i !
{
  i @ dup . cr
  2+ dup i !
  100 > ?break
}
```

For

```
for i = 1 to 100 do
  print i
next i
```

becomes,

```
1 i !
{
  i @ . cr
  i @ 1+ dup i !
  100 > break
}
```

or

```
1 i !
{
  i @ dup 100 > break
  i @ . cr
  i @ 1+ i !
}
```

If..Else..Then

Conditional expressions are built from "`if`", "`else`" and "`then`". Unlike many languages, but like Forth, SPL follows the form of:

<condition> if <true-part> else <false-part> then
<condition> if <true-part> then (no else part)

In addition to "`if`" SPL also has "`0if`" which is a "not if". Why will be seen in a little bit.

The <condition> is read off the top of the stack by `if`. If the value read is not zero, it is regarded as true. This is not the same as Forth which views -1 as true. Zero, then, is false. Conditional expressions are built on the stack using relational operators and logical comparisons. So, "`if`" means, "execute the next set of functions if the top of stack item is true" and "`0if`" means, "execute the next set of functions if the top of stack item is false". Use `then` to match each `if`.

Some examples,

```
x @ 3 < if                  #  if (x < 3) then
  "too small"               #    print "too small"
else                        #  else
  "just right"              #    print "just right"
then disp                   #  endif
```

```
key 27 = if quit then      #  if (key == 27) then quit

x @ 0 >=                   #  if (x >= 0) and (x < 16) then
x @ 16 < and if            #    print "in range"
  "in range"               #  endif
then disp
```

Functions

SPL programs are built from functions. These are defined by the user as in any other programming language. A function consists of:

def <name>
 <function_body>
end

where <name> is any valid identifier name (letters or digits or underscores, starting with a letter, and is case sensitive). Note that all SPL program must have a "main" function. Since SPL is a stack-based language, there is no need to declare arguments and return values. Simply pull values off the stack and push values on the stack. Functions need not be defined in a certain order so there is no need for forward references. The <function_body> is a sequence of zero or more numbers or function calls.

For example,

```
def hello  # ( -- )
  "Hello world!" disp cr
end
```

defines a function which takes no arguments from the stack and returns no values to the stack. Whereas,

```
def square  # ( a -- a+a )
  dup +
end
```

defines a new function which takes one value from the stack and replaces it with that value*2. Lastly, consider,

```
def many  # ( n -- )
  {
    1- dup ?0break
    hello
  }
end
```

which creates a function that takes one value from the stack and calls the previously defined `hello` function that many times. Note that `dup` is necessary as `?0break` consumes its argument and we want to leave the current count (which counts down to zero) on the stack as well.

Data Blocks

Sometimes it is necessary to define a sequence of values, say as embedded machine code or as a table of constants. For this, use a data block:

data <name>
  <numbers>
end

where <name> is a name for this block, with all the requirements of a function name, and <numbers> is a sequence of zero or more numbers, of any size, which will simply be compiled one at a time into memory pointed to by the address of the data block. Referring to a data block pushes its address onto the stack. For example,

```
data cout
  0x20 0xed 0xfd 0x60
end
```

will define a data block of bytes which implement a small 6502 machine code program to print the character in the accumulator on an Apple II. The data block could be called like,

```
def main
  65 areg       #  values in "areg" are loaded into the accumulator
  cout execute  #  when the execute function is called (dittos for X and Y)
end
```

a data block can also be used as a table:

```
data cubes
  1 8 27 64 125 216
end

def main
  2 dup . 32 emit 1- cubes + c@ . cr
end
```

which prints 8 since 2**3 = 8. Notice that this table has entries that are exactly one byte in size.

As another example, consider this data block:

```
data block0
  1
  -1
  1%
  1#
  -1%
  -1#
end
```

the output assembly code for this block is:

```
A00001:
```

```
.byte 1           ;  1, a single byte
.byte 255         ;  0xFFFF = -1, 16-bits
.byte 255
.byte 1           ;  0x0001, 16-bits
.byte 0
.byte 1           ;  0x00000001, 32-bits
.byte 0
.byte 0
.byte 0
.byte 255         ;  0xFFFF = -1, 16-bits
.byte 255
.byte 255         ;  0xFFFFFFFF = -1, 32-bits
.byte 255
.byte 255
.byte 255
```

Notice that -1 without a suffix becomes a 16-bit quantity. SPL understands that bytes are unsigned quantities. (Pity that Java doesn't understand that!)

SPL Language In a Nutshell

In a nutshell, SPL is:

1. Stack-based, which handles all arithmetic (8-, 16-, and 32-bit integer, signed)
2. Loops are started with "{" and end with "}". Use a variation on "break" or "cont" to control the iteration. Loops may be nested.
3. Conditional expressions using "if" .. "else" .. "then" as in Forth
4. Structured, no goto, functions defined with "def" and "end". Arguments taken from the stack and returned on the stack.
5. All variables and string constants are global
6. Data and embedded machine code can be entered via data blocks using "data" and "end" statements.

Intrinsic Library Functions

SPL comes with a library of just over 100 library words in assembly language. These can be used in your programs to implement many of the functions that you will typically need, from arithmetic to I/O. The complete list is given below. These routines are in the library directory in assembly format.

While bugs may still exist, of course, these routines have been extensively tested using the Sim65 65C02 simulator for the Macintosh (classic, not OS-X). This simulator should also be on the web site where you found SPL.

Apple II Specific Library Functions

SPL generates 6502 assembly code which is not tied to any specific machine. However, the library words to reflect a bias towards Apple II computers. The non-ProDOS related library words that are

Apple II specific are:

`accept, ch, cv, cr, cls, d.$, inverse, normal, quit, input, .$, .2$`

Also, these library utility routines are Apple II specific:

chout.s
rdkey.s
rdykey.s

To use SPL with another 6502 based computer, say a Commodore or Atari, these routines will need to be rewritten or dropped. Zero-page access will probably also need to be updated to reflect how the new system uses zero page.

In addition, there is a set of library words which relate to the Apple II ProDOS disk operating system. These, naturally, are Apple II specific as well:

fclose, fcreate, fdestroy, feof, fflush, fgetc, fopen, fputc, fread, fseek, ftell, fwrite, getcwd, read_block, rename, finfo@, finfo!, setcwd, write_block, time, date

These are described below in the library function reference.

External Library Modules

External library are library modules written in SPL and placed in the include directory. This is a good location for routines that you use frequently (relatively speaking) or for libraries specific to a particular machine.

Library Function Reference

All the library routines are described here along with how to use them.

Function	Stack	Description
`abs`	(a -- |a|)	Return the absolute value
`accept`	(addr -- len)	Get a line of text up to 255 characters and copy it to addr. Returns the length of the line. Apple II specific.
`+`	(a b -- a+b)	Add the top two stack items (16-bit)
`+!`	(addr b --)	Add b to the number stored at address (16-bit)
`and`	(a b -- n)	Logical AND of a and b
`areg`	(n --)	Store the 8-bit value on the stack in the location reserved for the A register when execute called. This allows loading the A register with a value before calling a machine language routine.

@	(addr -- n)	Fetch the 16-bit number at addr and push it on the stack (lo/hi)
d@	(addr -- n)	Fetch the 32-bit number at addr and push it on the stack as two 16-bit numbers.
c@	(addr -- n)	Fetch the 8-bit character at addr and push it on the stack as a 16-bit number (high byte is zero)
b.and	(a b -- a&b)	Bit-wise AND of the top two stack items
bclr	(n b -- n')	Clear bit b of n
b.or	(a b -- a|b)	Bit-wise OR of the top two stack items
bset	(n b -- n')	Set bit b of n
btest	(n b -- 0|1)	Test if bit b of n is set (1) or clear (0)
b.xor	(a b -- a^b)	Bit-wise exclusive-OR of the top two stack items
ch	(x --)	Set the horizontal cursor position to n. Apple II specific.
cls	(--)	Clear the text screen. Apple II specific.
cmove	(a b n --)	Move n bytes from address a to address b
~	(n -- ~n)	Flip the bits of the top stack item (1's complement)
cr	(--)	Output a carriage return (move to the next text line). Apple II.
cv	(y --)	Set the vertical cursor position. Apple II.
dabs	(n -- |n|)	Absolute value of the 32-bit number on the stack (2 16-bit numbers)
d+	(a b -- a+b)	Add the two 32-bit numbers on the stack
d/	(a b -- a/b)	Divide the two 32-bit numbers on the stack (signed)
d/mod	(a b -- a/b a%b)	Divide two 32-bit numbers and push the quotient and remainder
depth	(-- n)	Push the number of stack items on the stack
d=	(a b -- a==b)	a == b (32-bit)
d>=	(a b -- a>=b)	a >= b (32-bit)
d>	(a b – a>b)	a > b (32-bit)
disp	(addr --)	Print the null terminated string at addr on the screen. Uses chout routine.
d<=	(a b – a<=b)	a <= b (32-bit)
d<	(a b – a<b)	a < b (32-bit)
dmod	(a b – a%b)	a mod b (32-bit)
d*	(a b – a*b)	32-bit multiplication (32-bit answer)
dnegate	(a -- -a)	Change the sign of the 32-bit number on the stack
d<>	(a b – a<>b)	a <> b (32-bit)

dnumber	(addr – n)	Convert a null-terminated ASCII string at addr into a 32-bit signed number on the stack
d.$	(a --)	Print 32-bit number on the stack in hexadecimal. Apple II.
dprint	(a --)	Print the signed 32-bit number on the stack. Only depends on chout routine.
2drop	(a b --)	Drop the 32-bit number on the stack
drop	(a --)	Drop the 16-bit number on the stack
dsqrt	(a – sqrt(a))	Return the integer square root of the 32-bit number on the stack
d-	(a b – a-b)	a-b (32-bit)
2dup	(a b – a b a b)	Duplicate the 32-bit number on the stack
du.	(a --)	32-bit unsigned print. Only depends on chout
dup	(a – a a)	Duplicate the 16-bit number on the stack
emit	(c --)	Output the character on the stack. Only depends on chout
=	(a b – a==b)	a == b (16-bit)
execute	(addr --)	Call the machine language routine at addr. Before calling, A, X, and Y registers loaded with the values stored in areg, xreg, and yreg.
fill	(adr len val --)	Fill the len bytes of memory starting at addr with val
>=	(a b – a>=b)	a >= b (16-bit)
>	(a b – a>b)	a > b (16-bit)
input	(-- len)	Get a line of text from the keyboard to the buffer starting at 0x200. Up to 255 characters null-terminated. Apple II.
inverse	(--)	Set the printing mode to inverse. Apple II.
keyp	(-- 0|1)	Return true (1) if a key pressed, false (0) otherwise. Apple II.
key	(-- c)	Wait for a key from the keyboard. Apple II.
/	(a b -- a/b)	a / b (16-bit)
<=	(a b – a<=b)	A <= b (16-bit)
<	(a b – a<b)	A < b (16-bit)
1-	(a – a-1)	Subtract one from the top stack item (16-bit)
2-	(a – a-2)	Subtract two from the top stack item (16-bit)
mod	(a b – a%b)	A mod b (16-bit)
*	(a b – a*b)	A * b (16-bit)
negate	(a -- -a)	Change the sign of the top stack item (16-bit)
<>	(a b – a<>b)	a <> b (16-bit)
nip	(a b – b)	Drop the next to top stack item

normal	(--)	Set the printing mode to normal. Apple II.
not	(a -- !a)	Logical NOT of the top stack item
number	(addr – n)	Convert the null-terminated ASCII string at addr into a singed 16-bit number on the stack
or	(a b – a or b)	Logical OR of the top two stack items
over	(a b – a b a)	Duplicate the next to top stack item
pad	(-- addr)	Push the output text buffer address on the stack
1+	(a – a+1)	Add one to the top stack item (16-bit)
2+	(a – a+2)	Add two to the top stack item (16-bit)
.2$	(a --)	Print top of stack as two hexadecimal digits. Apple II.
.$	(a --)	Print top of stack as four hexadecimal digits. Apple II.
.	(a --)	Print the top of stack item (16-bit, signed)
quit	(--)	Call the ProDOS BYE MLI function. For exiting a SYS program. Apple II.
rot	(a b c – b c a)	Rotate the third stack item to the top of the stack
space	(--)	Output a space (ASCII 32). Uses chout.
strcmp	(a b -- -1|0|+1)	Compare two null-terminated strings returning -1 if a < b, 0 if a == b, and +1 if a > b.
strcpy	(src dest --)	Copy the null-terminated string, src, to dest
strlen	(addr -- len)	Return the length of the null-terminated string at addr
strmatch	(str pat -- 0|1)	Does the string null-terminated string, str, match the null-terminated pattern, pat? Pattern may contain * and ? Modification of a routine by Paul Guertin on www.6502.org
strpos	(adr c -- -1|idx)	Return the index of the first occurrence of c in null-terminated string adr or -1 if c not found
-	(a b – a-b)	A – b (16-bit)
2swap	(a b c d – c d a b)	Swap the two 32-bit numbers on the stack
swap	(a b – b a)	Swap the two 16-bit numbers on the stack
!	(n addr --)	Store n (16-bit) at addr (lo/hi)
d!	(d addr --)	Store n (32-bit) at addr
c!	(c addr --)	Store n (8-bit) at addr
u/	(a b – c)	Unsigned 16-bit division
u>	(a b – a>b)	Unsigned a > b comparison
u<	(a b – a<b)	Unsigned a < b comparison
umod	(a b – a%b)	Unsigned a mod b

u*	(a b – a*b)	Unsigned 16-bit multiplication
u.	(a --)	Unsigned print of the stop stack item
<<	(a n -- a<<n)	Shift a n bits left
>>	(a n -- a>>n)	Shift a n bits right
xor	(a b – a xor b)	Logical exclusive -OR
xreg	(c --)	Set the X register value to load when execute called
yreg	(c --)	Set the Y register value to load when execute called

Using ProDOS from SPL

SPL comes with ProDOS library functions. These allow SPL programs to read, write, and manipulate disk files as well as read the system clock, if present. Most ProDOS MLI commands are available from the existing set of functions.

To use ProDOS in your programs it is advisable to use the `-prodos` keyword when compiling. This will automatically set the `VARTOP` value so that there will be room for BASIC.SYSTEM and three file buffers. If you are careful, and know you only need say one open file at a time, you can get away with one file buffer (each is 1k = 0x400 bytes) and simply subtract that much from the default `VARTOP` value. A file buffer number, 0..2, must be given when calling `fopen`.

Also, ProDOS, being old and designed in the Pascal era, uses counted strings for pathnames. SPL is designed to work with null-terminated strings. Therefore, use `count` to destructively turn a null-terminated string (<256 chars) into a counted string. Use `uncount` to reverse this and get a null-terminated string back.

The names for the library functions have been chosen to mimic those found in the standard C library whenever possible. However, the behavior isn't always identical. For example, `fseek` here always takes an offset from the beginning of the file. Use `feof` to find the end of the file (hence, its size in bytes). A little practice should clear things up when something doesn't act as you'd expect. As a rule, these functions return a zero for no error and a one if there is an error. For example, you could use a construct like:

```
ref c@ fclose if
  "fclose #" disp .2$ cr
  return
then
```

to close a file and check for an error. If there is an error (a 1 was returned) print “fclose #” followed by the ProDOS error code as a two digit hex number.

Though hard to find these days, there are good books about ProDOS out there. For example, “Beneath Apple ProDOS” or “Apple ProDOS: Advanced Features for Programmers”. The latter was used as a guide when writing these functions. One or the other of these might be floating around the net as a

PDF file by now.

ProDOS Function Reference

Function	Stack	Description
count	(s -- s)	Destructively change a null-terminated string (<256 chars long) into a counted string. Returns the input address.
uncount	(s -- s)	Destructively change a counted string into a null-terminated string. Returns the input address.
fclose	(ref -- ec 1|0)	Close an open file
fcreate	(fname typ aux -- ec 1 | 0)	Create a new file. Cannot already exist.
fdestroy	(name -- ec 1|0)	Delete a file
feof	(ref -- ec 1 | d 0)	Return the offset (32-bit) to the file's EOF position. This is the length of the file.
fflush	(ref -- ec 1 | 0)	Write the file's buffer to disk immediately without closing the file.
fgetc	(ref -- ec 1| c 0)	Read the next character from the file put it on the stack
fopen	(path buf -- ec 1 | ref 0)	Open a file for reading and writing. The file must already exist (see fcreate).
fputc	(c ref -- ec 1 | 0)	Write the character on the stack to the file.
fread	(buf n ref -- ec 1| bytes 0)	Read a set of bytes from the file to a buffer in memory.
fseek	(d_off ref -- ec 1 | 0)	Move the file pointer (32-bit) to the specified byte number. Always relative to the start of the file.
ftell	(ref -- ec 1 | d 0)	Return the current position in the file (32-bit)
fwrite	(buf n ref -- ec 1| bytes 0)	Write a set of bytes from memory to the file.
getcwd	(addr -- ec 1 | 0)	Return the current working directory. This is a counted string. Use uncount to make it a null-terminated string.
read_block	(buf blk drv -- ec 1 | 0)	Read a disk block into memory. This is the lowest level of access allowed in ProDOS.
rename	(src dest -- ec 1 | 0)	Rename a file. Both the source and destination names must be counted strings.
finfo@	(name -- ec 1 | adr 0)	Returns the address of a structure which contains the file information returned by the GET_FILE_INFO MLI command.

finfo!	(buf name -- ec 1 | 0)	Set file info for the given file.
setcwd	(addr -- ec 1 | 0)	Change the current working directory. The new pathname must be a counted string.
write_block	(buf blk drv -- ec 1 | 0)	Write a 512 byte block of memory to disk. This is the lowest level of access allowed by ProDOS.
date	(-- dd mm yy)	Return the current date if a clock card is installed. Returns all zeros otherwise.
time	(-- hh mm)	Return the current time if a clock card is installed. Returns all zeros otherwise.