# MODULAR ASSEMBLY LANGUAGE PROGRAMMING

APPLE TUTORIAL

**With today's assemblers,**
***it takes only a few simple rules to develop a library of independent, relocatable subroutines.***

Back in 1977 when the Apple II computer first appeared, programmers had very limited tools with which to work. The only resident assembler that was available for the Apple was the mini-assembler in the Integer BASIC ROM. The mini-assembler did little more than translate 6502 mnemonics into their corresponding hexadecimal opcodes. Consequently, most of the programs that were written with it were fairly short and simple, especially by today's standards. But then, the original Apple II's only had 16K of RAM, anyway.

A few years later, better assemblers for the Apple II started to appear. One of the first of these was EDASM, a combined text editor/assembler program produced by Apple Computer, Inc., and available as part of their Programmer's Workbench.* Now it was possible to write assembly language code that utilized labels and comments. In addition, statements could be easily inserted and deleted anywhere in the program — another feature that was missing in the old Apple II's mini-assembler. In principle, there seemed to be almost no limit to the size of the programs that could now be written, given enough memory.

## PROBLEMS WITH BIG PROGRAMS

As programs became bigger and bigger, several problems, which had been less noticeable in smaller programs, became evident. For one thing, the sheer bulk of a long program made it a nuisance to work with. Anytime a change was needed, no matter how trivial, the *entire* assembly language source file had to be loaded, resaved and reassembled.

Another problem in dealing with very long assembly language programs was trying to remember which label names had already been used and which ones hadn't. One way to resolve this problem might have been to use a set of meaningless labels (such as X1, X2, X3, etc.) and record them on a separate piece of paper. But it is better programming practice to assign labels whose names are meaningful (like START, LOOP or EXIT, for example). However, once such a label is assigned, it can never be used again anywhere else in the program.

## SUBDIVIDING BIG PROGRAMS

One approach to writing a big program was to subdivide the source file into several separate source files. These smaller sections of code were easier to handle, and each could be assembled separately. However, any time one of the sections had to be modified, all of the other sections that followed it had to be reassembled to a slightly different memory location to make room for the changes. The reassembly caused all of the entry points to change. So every piece of code that accessed those entry points had to be updated and reassembled, too.

One solution was to leave a cushion of unused memory behind each section of code so that if changes were needed, there would be room for expansion. But in a microcomputer, where memory resources are limited, such solutions were hardly elegant. (Besides,

*The examples require an assembler that can generate relocatable object modules (e.g., EDASM, Apple ProDOS Assembler, Merlin Pro and ORCA/M) and an appropriate linking loader. The resulting programs run under either DOS 3.3 or ProDOS.*

*EDASM and its linking loader are also available as part of The P.A.C.K. (Programmer's Assembly Language Construction Kit), published by Interactive Arts, 2715 Porter St., Soquel, CA 95073, (408) 475-7047.

Murphy's Law dictates that no matter how big a cushion you leave, you will always end up needing at least one byte more.

Subdividing a big program seems to suffer from two separate (but related) problems:

1. The resulting object files contain position-dependent code. This means that each section of code can execute properly only if it is loaded at the memory location that was specified when the source file was assembled.
2. Even if each section of code could be written so that it would execute properly anywhere in memory, there would still be the problem of getting the separate sections to "find" one another.

Both of these problems are resolved by writing relocatable assembly language code.

## ABSOLUTE VS. RELOCATABLE

Most good assemblers allow the programmer to select whether the resulting object code will be absolute (position-dependent) or relocatable.

Absolute code (the predominant type in most software for the Apple) has the advantage that it is easy to load into memory. Since it's simply a binary file, it can be loaded and/or executed by issuing a single command (i.e., you can simply BLOAD the file).

Relocatable code has the advantage that it can be loaded into almost any area of memory and still execute properly. In addition, relocatable code contains all of the information necessary to allow other separate, relocatable modules to find each other in memory and connect themselves together. However, this process of loading, relocating and connecting requires the aid of a separate utility program called a linking loader.

## MODULAR PROGRAMMING

You now have an effective means of writing a big program: you merely write it as a set of small relocatable modules, and then use the linking loader to "glue" the modules together. But just how do you subdivide any given program? Do you simply write out the entire program as one long source listing, and then arbitrarily chop it up into pieces?

The concept of subdivision should not be interpreted as merely a physical grouping of code, but as a logical division of the overall task. For example, suppose you wanted to write a program that plays chess. Would you write one big program that starts out by inputting your move and ends by printing out the computer's move? You might be able to design the program to work that way. But it would make a lot more sense to design it as a collection of smaller, simpler subroutines, each of which focuses on a separate aspect of the chess game. These subroutines could then be combined together to form a chess-playing program. The subroutines (modules) that could constitute such a chess-playing program might include:

- An input/output module for specifying moves
- A module to evaluate the status of the board
- A module to determine the allowable moves for each of the chess pieces
- A module to display the board on the TV screen

and so on, as needed.

The advantages of creating a program from a set of separate modules (as opposed to creating one long program) are:

1. Each module can be designed and written separately.
2. Each module can be debugged separately.
3. Modules can be used more than once in the program, when similar tasks are performed.
4. A modular program is easy to understand and maintain.

There are no hard-and-fast rules when it comes to deciding just how to partition a programming problem into subroutines. Sometimes the problem has fairly obvious dividing lines and common sense indicates using separate subroutines. But there is always more than one way of slicing baloney, and the partitioning you decide upon today may not seem all that logical tomorrow.

The problem of partitioning is really a recursive one, since each subroutine also represents a little program in its own right. This suggests the divide and conquer approach to partitioning: if a subroutine is too long or too complicated, subpartition it further into smaller and simpler subroutines.

But be careful! Partitioning a program too much can result in a lot of nearly trivial modules. This means slower execution speed, since it takes a minimum of 12 microseconds just to enter and exit a subroutine. Carried to a ridiculous extreme, partitioning could lead to a set of modules, each containing only *one* 6502 instruction! These "modules" are already available — they're the entire instruction set of the 6502.

## SUBROUTINE CALLING PARAMETERS

A subroutine is a module of code that is separate from another body of code. Program control flows into the subroutine via a JSR instruction, and flows back out of the subroutine via an RTS instruction.

Sometimes information (data) must flow into and out of the subroutine as well. When information must be passed between a subroutine and another body of code, it is passed using calling parameters. A calling parameter can be passed to and from a subroutine by putting the parameter:

**Method 1:** into a memory location inside the subroutine itself.
**Method 2:** into a memory location outside of the subroutine.
**Method 3:** into one of the 6502's registers.

(It is also possible to use the 6502's hardware stack for passing parameters. But since the stack is located at memory locations $0100-$01FF, this method of passing parameters can be regarded as an extension of method 2 above.)

## WRITING RELOCATABLE MODULES

Writing relocatable code is generally not very different from writing absolute code. In order to generate relocatable object code, you merely need to specify some additional pseudo-ops (assembler directives) at the beginning of your source code. Pseudo-ops are slightly different for each of the assemblers currently available for the Apple II, but the fundamental concepts are the same. We can get an idea of how they are implemented by looking at the pseudo-ops used by EDASM.

## PSEUDO-OPS FOR RELOCATABLE CODE

EDASM provides three pseudo-ops for use in writing relocatable code: REL, ENTRY and EXTRN.

### REL

To produce relocatable object code, the source code must include a REL pseudo-op as the first instruction of the listing. (Normally, this REL pseudo-op alone would have sufficed to generate relocatable object code. But, because of a minor design flaw in the EDASM assembler, it is necessary to follow the REL pseudo-op with an ORG $1000 pseudo-op.)

### ENTRY and EXTRN

Whenever a name (symbol) is referenced in the operand field (the third column) of a source listing, that same name must be defined somewhere else in the program or an error message will result. Usually, the name is defined by making it a label of another instruction in the same module of code. For example, if a program contains the instruction:

```
JMP CONTINUE
```

the name CONTINUE usually appears as a label somewhere in the same source listing.

When partitioning a program into relocatable modules, it is often necessary for instructions in each of the separate modules to be able to reference some of the labels contained in the other modules. Such labels are termed "global symbols." Labels that are referenced only from the current module, and not from any of the other listings, are termed "local symbols."

EDASM provides two pseudo-ops, ENTRY and EXTRN, for dealing with global symbols. The EXTRN pseudo-op is used to declare labels that are referenced in the current module but are defined in another module. The ENTRY pseudo-op is used to declare labels that are defined in the current listing, and can be externally referenced from other listings. Any label not specifically declared with an ENTRY pseudo-op (i.e., a global label) defaults to being local.

Relocatable modules can be thought of as "little black boxes" with sockets leading into them and cables coming out of them. Each socket is analogous to an ENTRY, and each cable and plug is analogous to an EXTRN (see **Figure 1**).

***FIGURE 1: Subroutines As Boxes With Matched Cables and Receptacles***

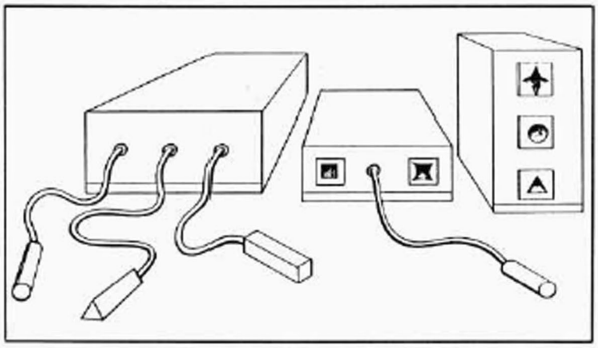
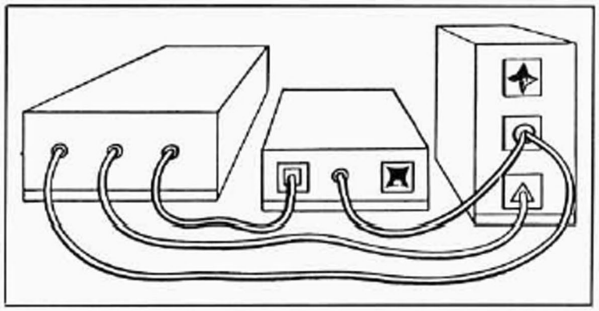


***FIGURE 2: Connecting Subroutines***



Each cable can be plugged into one, and only one, matching socket in some other black box. Plugging the cables into their corresponding sockets is analogous to linking the modules together (see **Figure 2**).

Notice that not all of the sockets need to have cables plugged into them, but every cable must be able to find its corresponding socket. In other words, any name may be declared to be an ENTRY, even if that name never gets referenced from any other subroutine. But if a name is declared to be an EXTRN, that name must be defined as an ENTRY in some other module. Notice also that every socket must be unique (two different modules cannot both have the same ENTRY name), but that any socket can accommodate more than one cable — as long as the plug on the cable matches the socket.

It is important to understand that *any* label can be a global symbol if it's simply declared to be an ENTRY. ENTRYs are not limited to merely specifying entry points to sections of code; an ENTRY can also refer to the start of a block of data.

## A RELOCATABLE MAIN PROGRAM AND SUBROUTINE

Let's look at a relocatable subroutine that has one calling parameter (see **Example 1**). This subroutine emits a brief tone from the Apple's built-in speaker. (For convenience, the listing includes the relative line numbers as they would appear in the EDASM text editor.)

Notice that the SOUND subroutine uses method 2 (a memory location outside of the subroutine) for passing its calling parameter, PERIOD; in **line 7** of **Example 1**, the parameter is arbitrarily assigned to memory location $0300. (The bigger the number in PERIOD, the lower the frequency of the sound.)

***EXAMPLE 1: Cooperative Subroutine***

```
 1              REL
 2              ORG     $1000
 3  *
 4              ENTRY   SOUND
 5  *
 6  CLOCK       EQU     $06
 7  PERIOD      EQU     $0300
 8  SPKR        EQU     $C030
 9  *
10  SOUND       LDA     #$10
11              STA     CLOCK
12  LOOP        LDY     PERIOD
13  WAIT        NOP
14              NOP
15              DEY
16              BNE     WAIT
17              STA     SPKR
18              DEC     CLOCK
19              BNE     LOOP
20              RTS
```

***EXAMPLE 2: Main Program for Use With Cooperative Subroutine***

```
 1              REL
 2              ORG     $1000
 3  *
 4  *  THE "MOANING APPLE"
 5  *
 6              EXTRN   SOUND
 7  *
 8  PARAM       EQU     $0300
 9  DATA        EQU     $F800
10  *
11              LDA     #$80
12              STA     PARAM
13              LDX     #$00
14  LOOP        LDA     DATA,X
15              AND     #$20
16              BEQ     DECR
17  INCR        INC     PARAM
18              INC     PARAM
19              BEQ     INCR
20              JMP     TOSUB
21  DECR        DEC     PARAM
22              DEC     PARAM
23              BEQ     DECR
24  TOSUB       JSR     SOUND
25              INX
26              JMP     LOOP
```

Now, let's look at a main program that might call the SOUND subroutine (see **Example 2**). Since the subroutine's ENTRY name is SOUND, the main program declares the name SOUND to be an EXTRN (in **line 6**). This allows the name SOUND to be referenced (in **line 24**), even though it is not defined anywhere in the listing.

Both the main program and the subroutine use LOOP as the name of a label in their respective source listings. Since the label LOOP was not declared to be a global symbol, there is no conflict.

## INDEPENDENT SUBROUTINES

In the previous example, we wrote a main program that called a cooperative subroutine. The term "cooperative" means that the main program and its subroutine were each written with the other in mind. For example, both were written with the idea that the calling parameter would be passed via memory location $0300. Furthermore, neither the Y-Register nor the zero-page location $0006 were used by the main program, so the subroutine could use them freely.

When writing an independent subroutine, you cannot assume that there will be such cooperation between your routine and all of the other routines. There is no way to know which index registers or memory locations will be available for use when a subroutine is called. The index registers and any absolute memory locations (such as $0006 and $0300 in the previous example) may already be in use by some other routine. This problem is especially acute in the case of the zero page memory locations ($0000-$00FF), since most of them have already been claimed by the System Monitor or BASIC. Therefore, when you write an independent subroutine, you must be careful not to "step on the toes" of any other software that may also be in use. There are few hard-and-fast rules describing how to write independent subroutines, but the important point to remember is that *no* cooperation can be assumed.

***EXAMPLE 3: Independent Subroutine***

```
 1            REL
 2            ORG    $1000
 3 *
 4            ENTRY  SOUND
 5 *
 6 CLOCK      EQU    $06
 7 SPKR       EQU    $C030
 8 *
 9 PERIOD     DS     1
10 *
11 SOUND      TXA
12            PHA
13            LDA    CLOCK
14            PHA
15            LDA    #$10
16            STA    CLOCK
17 LOOP       LDX    PERIOD
18 WAIT       NOP
19            NOP
20            DEX
21            BNE    WAIT
22            STA    SPKR
23            DEC    CLOCK
24            BNE    LOOP
25            PLA
26            STA    CLOCK
27            PLA
28            TAX
29            RTS
```

To illustrate one approach to writing independent subroutines, let's rewrite the MOANING APPLE example. First, the SOUND subroutine becomes the listing shown in **Example 3**. The main program that calls this subroutine looks like **Example 4**.

There are several important differences between **Example 1** (the cooperative subroutine) and **Example 3** (the independent subroutine). First, notice that memory location $0006 and the X-Register must both be saved somewhere (such as on the stack) before they can be used (see **lines 11-14** of **Example 3**) because they may currently be used by the calling program. When the subroutine is finished with them, they are restored to their previous states. (Notice that we no longer need to use the Y-Register.)

Another difference between **Example 1** and **Example 3** is the method by which the calling parameter is passed. Instead of assigning the calling parameter to an absolute memory location outside of the subroutine (method 2), we make it relocatable with the subroutine by assigning it to the memory location immediately preceding the subroutine's ENTRY point (method 1). By incorporating the parameter into the subroutine, we guarantee that no memory conflicts will occur with any other routines.

Notice that a subroutine can use a local symbol (PERIOD) to internally reference its own calling parameter (as in **line 17** of **Example 3**). The calling program, on the other hand, must externally reference it by using a negative offset from the subroutine's ENTRY name (e.g., SOUND-1 in **lines 11, 16, 17, 20 and 21** of **Example 4**).

***EXAMPLE 4: Main Program for Use With Independent Subroutine***

```
 1            REL
 2            ORG    $1000
 3 *
 4 * THE "MOANING APPLE"
 5 *
 6            EXTRN  SOUND
 7 *
 8 DATA       EQU    $F800
 9 *
10            LDA    #$80
11            STA    SOUND-1
12            LDX    #$00
13 LOOP       LDA    DATA,X
14            AND    #$20
15            BEQ    DECR
16 INCR       INC    SOUND-1
17            INC    SOUND-1
18            BEQ    INCR
19            JMP    TOSUB
20 DECR       DEC    SOUND-1
21            DEC    SOUND-1
22            BEQ    DECR
23 TOSUB      JSR    SOUND
24            INX
25            JMP    LOOP
```

When there is more than one calling parameter, each additional parameter is referenced again with its appropriate negative offset from the ENTRY name. For example, if the SOUND subroutine had several calling parameters instead of only one, then the calling program could reference the second parameter as SOUND-2, the third as SOUND-3, and so on. (An alternative to this negative offset method of accessing calling parameters would be to assign each calling parameter its own ENTRY name. However, such an approach quickly uses up all of the "good" names, and makes the linking loader's symbol tables needlessly long. And the longer the symbol tables are, the slower the linking loader runs.)

## DEFAULT VALUES FOR CALLING PARAMETERS

Another advantage to using method 1 for passing calling parameters is that it allows default values to be easily assigned. Often, a subroutine will have a calling parameter that is almost always assigned one particular value. It makes sense that this most frequently-used value should be supplied as a default instead of requiring the user to specify it each time. By assigning a default value, the user can, in effect, ignore the parameter most of the time. However, since it is still a calling parameter, it can be accessed for those rare applications that require a different value.

## CALLING INDEPENDENT SUBROUTINES FROM BASIC

Independent subroutines can pass calling parameters either by using internal locations (method 1) or by using 6502 registers (method 3). On the other hand, to call an assembly language subroutine from BASIC (where the parameters are passed using PEEKs and POKEs), only method 1 or method 2 (using a memory location outside the routine) can be used. Therefore, in order to write subroutines that are both independent and callable from BASIC, method 1 is the recommended way of passing all calling parameters.

## THE POWER OF INDEPENDENT SUBROUTINES

By writing relocatable, independent modules, it is possible to create libraries of your most frequently-used subroutines. Once they are debugged and working, *you can incorporate these modules into future programs without having to ever modify (or even look at) their source codes again!*

The implications are far-reaching. Perhaps the most exciting one is that it is now possible for a programmer to share his or her subroutines with other programmers.

Consider the following scenario: You have just received the latest issues of your two favorite computer magazines. In one is an article describing how to plot ASCII characters on the double Hi-Res screen (included is a listing of the subroutine that does the character plotting). In the second magazine, you find an article (and subroutine listing) describing how to draw lines on the double Hi-Res screen. You decide that you would like to write a program that draws and labels bar graphs.

But hold everything! — there's a fly in the ointment. Both authors have placed their subroutines at $9000. "No problem," you say, "I'll just re-ORG one of them to $8000." So you reassemble the code and finish writing your own main program. When you try to run the whole thing, though, it still doesn't work right. After hours of debugging, you finally discover that the two published subroutines use the same memory location on page zero.

Had the two authors in the above scenario published their code as cooperative subroutines, the problems would never have occurred.

## SUBROUTINE LIBRARIES

Most computer users will tell you that they have their own personal software libraries. By this, they usually mean that they have a pile of floppy disks lying around that contain different programs for their computer.

While such program libraries do not require any formal organization, subroutine libraries do. A subroutine library is an organized structure whose primary purpose is to provide a simple and effective way of storing and loading commonly-used modules. Most linking loaders have built-in provisions for handling subroutine libraries, but each handles them in a slightly different way.

### EDASM Subroutine Libraries

A subroutine library is similar to a public library in that both represent a storehouse of information. A public library contains collections of books and magazines; a subroutine library contains collections of subroutines and data tables.

A public library has a card catalog (directory) showing where each book and magazine can be found. Similarly, a subroutine library (as implemented by the EDASM linking loader) contains a directory (a text file) that lists the names of all the ENTRYs (subroutines and data tables) contained in the library. For each ENTRY, the directory gives the name of the relocatable file in which the ENTRY can be found.

Because the EDASM linking loader is interactive, there is no limit to the number of libraries you can have. In fact, it is possible for the same subroutine to be stored in several different subroutine libraries, just as the same book can be found in many public libraries.

Since a subroutine library's directory is simply a text file that indicates to the linking loader where it can find the object files that it needs, you may be tempted to think that a directory is nothing more than the equivalent of an EXEC file for loading. This is definitely not the case. An EXEC file would load in everything, even ENTRYs that are never used.

Invoking a library, on the other hand, loads in only those modules that are actually needed. The linking loader scans the specified directory file and, as each ENTRY name is encountered, the linking loader compares it to EXTRNs referenced by the program. If it finds a match (and if no other module defining that ENTRY is already in memory), the file containing the ENTRY is loaded.

## CONCLUSION

A computer program is a closed system of code. . . a little universe of its own in which its creator has provided everything needed for standalone operation (such as user-friendly I/O, and error handling). But programs are extremely limited in their flexibility — all you can do with a program is run it! If a purchased program lacks some desired feature, or if it doesn't work quite the way you'd like it to, you simply have to live with the problem.

A subroutine can also be a closed system of code (i.e., a black box that performs a well-defined task), but the scope of its universe is usually much smaller than that of a program. A subroutine generally cannot stand alone as a main program because it lacks support; it is essentially an "incomplete program." But it is this incompleteness that gives subroutines their tremendous flexibility.

*. . . programming in assembly language could become almost as easy and fun as programming in BASIC.*

They are analogous to electronic components (amplifiers, logic gates, etc.) which, by themselves, are relatively powerless. But by combining them with other components, there's almost no limit to the powerful and unique new products that do-it-yourselfers can create.

While there are many fully-assembled programs available as commercial products, there are virtually no subroutine modules that software do-it-yourselfers can buy. (Stores like Radio Shack supply hardware components but there are, as yet, no Subroutine Shacks for supplying software components.)

Prefabricated software modules would be particularly useful to beginning programmers, who may understand BASIC but who don't yet feel quite at home in the strange new world of assembly language programming.

One of the advantages of programming in Applesoft BASIC is that prefabricated modules (HGR, HPLOT, RND, SQR, etc.) are built into the language. Even a beginning BASIC programmer can quickly and easily access these modules to achieve interesting results.

Programming in assembly language, on the other hand, tends to be a laborious task. Each line of code does little in comparison to what can be done in a single line of a BASIC program. Worse yet, assembly language has none of the built-in modules of BASIC. To create an assembly language program that produces even a slightly interesting effect requires not only an inordinate amount of work, but also a fairly thorough understanding of the computer's hardware architecture.

One of the primary goals of this article was to suggest a standard programming methodology by which even beginning programmers could easily develop software — without first having to understand all of the hardware idiosyncrasies of their computer and its peripherals. By providing prefabricated independent modules that handle such fundamental operations as Hi-Res plotting or line drawing, programming in assembly language could become almost as easy and fun as programming in BASIC.

*Acknowledgment:* I would like to thank Lucia Grossberger for her helpful contributions and support.