

# Structured Programming in Assembly Language

Author: Dick Munroe

Cottage Software Works, Inc.

## Overview

The stored program computer in its modern form was developed in the late 1940s. About 15 minutes after the first program was written (using patch panels and/or toggle switches on the front panel) engineers started looking for ways to make writing programs easier. Some of the initial attempts at making programming easier were what are now considered third generation languages, e.g., Fortran II. However that still left the problem of writing low level (close to the hardware) code. Fortran and the other languages developed at the time were unsuited to the problem of dealing directly with hardware. As a result, low level languages, more or less universally called assembly languages, were developed. These languages shared a number of common characteristics. These are:

- One-for-one mapping from operands in the language to machine instruction
- Direct access to hardware resources, such as registers, I/O mechanisms, etc.

And they made programming at the lowest level of the computer substantially easier. They worked.

Over time, these assembly languages acquired additional mechanisms for making programming easier, most notably a macro processing capability through which programmers could extend the macro language. Most frequently "macros" were used to capture frequently-used code sequences, for example, saving registers at the entrance to routines, restoring them at return, etc. Ultimately the purpose of a macro was to reduce the opportunity for making mistakes. By putting common instruction sequences into macros, a whole class of errors was eliminated making the writing of assembly language programs more reliable and faster.

At the same time, substantial research was being done at the higher levels of programming (what most would consider application programming) resulting in a variety of more or less general purpose programming languages, Fortran 4, PL/1, Algol, C, Pascal, and a veritable tower of Babel of others. All of these had interesting features and approaches with respect to how programs were written. For example, Fortran included statements that were equivalent to the ubiquitous compare/branch assembly language instructions (IF (A .EQ. B) GOTO ...). This was probably one of the very first uses of a "design pattern" in the history of modern computing.

Most interesting of all was the gradual elimination of the branch instruction in many of the languages (notably Pascal which takes the elimination of GOTO to ridiculous lengths). The GOTO or branch language construct was replaced by a variety of flow of control constructs such as:

- IF THEN ELSE
- FOR loops
- DO WHILE/UNTIL loops

The "elimination" of the GOTO and the use of the alternative flow of control constructs became known as "structured programming." Structured programming is probably the single biggest contributor to the quality and quantity of code produced since the 1960s. There are several reasons for this and they will be discussed later.

However, assembly language programmers (and there are a lot of us, though fewer now with the advent of the RISC machine and the use of higher level languages for OS and driver development) largely missed the advantages of structured programming.

This article discusses the benefits of structured programming and how to do structured programming in assembly language, specifically Macro-32. The techniques discussed here have been used in a number of real time environments on a variety of platforms.

## What is structured programming?

Well there are a lot of answers to that question. The most common one is “goto-less” programming.

In fact, “structured programming” is little more than an enforced discipline that encodes information directly into the “structure” of a program that makes some of the characteristics of the program easier to understand by a programmer other than the original author. This is also useful if the original author returns to that program after a substantial hiatus.

In the case of most higher level languages, the discipline is enforced by the language. For example, Pascal literally has no GOTO in the language. C and C++ do have GOTO but also have flow-of-control constructs that mitigate their use. In both languages, it is possible to write well structured, easy to understand programs. It is also possible to write well structured, difficult to understand programs. For good examples of such things, see the winners of the obfuscated C contest held yearly. The obfuscated C programs all work, many do useful things, and all are virtually incomprehensible by design.

So, goto-less programming is not a panacea. What use is it then?

Without discipline, none.

What structured programming enables is the ability to encode additional information into the body of the program that makes it easier to understand. Specifically it let the programmer encode a visual representation of the flow of control. In turn, this lets the programmer use extra pieces of his/her brain to understand the program. The additional information is encoded by indenting the bodies of the various flow-of-control constructs so as to make them stand out visually. Thus the visual cortex is engaged to help understand the program. Humans evolved using their eyes to detect predators and a substantial portion of the brain is dedicated to visual processing. As a consequence, anything which adds visual information to a program makes understanding the program easier because more of the brain is used when working with the program.

Of course the visual information must be added consistently, otherwise the programmer’s eyes get confused and the additional information can be obscured. The guidelines for adding visual information to programs are relatively straightforward. Basically, every flow-of-control structure must be introduced consistently. The code executed with the flow of control must be indented to show the scope of the flow-of-control construct. This indentation must be large enough to separate the code visually while not being so large so that the eye “skips” over the indented code as though it were divorced from the flow of control. In general, indents of less than three spaces is too little and more than 5 spaces is too much. However, the human brain is enormously flexible and as long as the rules for encoding flow-of-control information are consistently followed within a program, practically anything will work.

Encoding is therefore largely a matter of personal style. In the open source community, there are at least a dozen different popular structured programming styles. In my experience, Digital Equipment Corporation and the code generated in support of its many products and product lines was unique in that the same structured programming style was used for many of them. Today much of the programming consistency is supported by languagesensitive editors such as EMACS and LSEDIT. Virtually every integrated development environment (IDE) also enforces one or more programming styles by being sensitive to the indentation in use at the time in the code being

written. All modern programming editors can be customized to support virtually any programming style.

Again, the assembly language programmers have largely missed out on the advantages of modern editors and IDEs because assembly language itself doesn't provide any direct support for structured programming.

So what are the benefits of structured programming in assembly language and how can it be supported?

## Benefits of Structured Programming

One common error in assembly language is a property of the flexibility of assembly language. Assembly language programming is inherently untyped. The assembly language programmer may treat any given piece of data as any type, byte, word, longword, ASCII, EBCDIC, null terminated, counted string, etc. By itself, structured programming doesn't deal with this source of errors. OpenVMS (and before it, RSX-11M) used naming conventions to denote data types, byte, word, longword, text, etc. The strict use of naming conventions provides an easy visual mechanism for the programmer to make sure that the type of comparison matches the data type being compared.

Another common error is also a function of data typing. In particular, comparisons against all the common data types can usually be done in either signed or unsigned modes. Since the difference between a signed branch versus an unsigned branch is frequently a single character (for example, in Macro-32, BGTR versus BGTRU for branch on greater than versus branch on greater than unsigned), it is easy to forget that a data type is unsigned, or to simply make a typing error and leave off the "U". Again, by itself, structured programming doesn't help with these errors but naming conventions help facilitate the discovery and correction of such errors.

Another common error is not getting the sense of the comparison correct. I program in assembly language on a variety of machines. Some machines test their operands from left to right ( $A < B$ ) when using comparison instructions. Others test their operands from right to left ( $B < A$ ). Switching from machine to machine can lead to programming errors simply from forgetting details of the machine architecture. Again, structured programming, by itself, doesn't help here either.

The place where structured programming does help is understanding the flow of control through a maze of assembly language instructions. Properly designed, the tools to do structured programming in assembly language will help with the other problems as well.

Needless to say, I'm not the first to think of this. During the development of the Record Management System (RMS) on the PDP-11, Ed Marison, et al., developed a package of macros that addressed virtually all of the defects of assembly language for the PDP-11. Unfortunately, this package of macros (known as Super Mac) took forever to assemble, but the increased programmer productivity and higher quality in terms of number of bugs was felt, correctly, to more than offset the amount of time it took to assemble any given portion of RMS.

I developed and have used a similar macro package for 20 years now on a wide variety of embedded systems (PDP-11, Z8000, Motorola 68K) and any number of driver development projects (mostly on OpenVMS). This macro package focused mostly on what I feel are the largest problems associated with assembly language programming, specifically, exposing the structure (flow of control) of a program written in assembly language.

## Introducing Simple Mac

The simple structured macro package (Simple Mac) has virtually eliminated my most common errors in assembly programming and has substantially improved my ability to revisit and understand programs that I've written years ago. Since I'm a pragmatic programmer (use what you need

when you need it), Simple Mac also makes it easy to spot where I don't use proper structured programming by making it possible to use labels only where unexpected branches occur rather than everywhere a branch destination is required.

Listing 1 shows a basic structured program using Simple Mac.

```
.LIBRARY /SMPMAC.MLB/

SM32INIT

ONE:      .BLKL
TWO:      .BLKL

START.MODULE
SMPMAR_EXAMPLES:

IF #1 SET.IN R0
THEN
    MOVAB ONE,TWO
ELSE
    MOVAB TWO,ONE
END IF

10$:

BEGIN BLOCK_TEST
    IF RESULT IS VC LEAVE BLOCK_TEST
    MOVAB ONE,TWO
END BLOCK_TEST

IFW R0 EQLU #0 GOTO 10$

IFL <ADDL ONE,TWO> IS PLUS THEN <MCOML TWO,TWO>

REPEAT
    MOVL ONE,TWO
    IF TWO GEQL ONE AND TWO NEQU #-1 NEXT
    MOVL TWO,ONE
END

DECRU ONE FROM #43 TO #-44 BY #13
    MCOML TWO,TWO
NEXT
    MCOML TWO,ONE
END

DECRU ONE FROM #43 TO #-44 BY R0
    MCOML TWO,TWO
NEXT
    MCOML TWO,ONE
END

DECR ONE FROM #43 TO #-44 BY #13
    MCOML TWO,TWO
NEXT
    MCOML TWO,ONE
END

DECR ONE FROM #43 TO #-44 BY R0
    MCOML TWO,TWO
NEXT
```

```

MCOML TWO,ONE
END

DECRU ONE FROM ONE TO #-44 BY #1
MCOML TWO,TWO
GOTOW 10$
MCOML TWO,ONE
END

DECR ONE FROM ONE TO #-44 BY #1
MCOML TWO,TWO
LEAVE
MCOML TWO,ONE
END

REPEAT
    ON.ERROR THEN <GOTOW 10$>
DECRS ONE TO #13 BY #25

REPEAT
    ON.NOERROR LEAVE MCOML TWO,TWO
DECRU ONE TO #13 BY #25

REPEAT
    ON.ERROR THEN <GOTOW 10$>
INCRS ONE TO #13 BY #25

REPEAT
    ON.ERROR THEN <GOTOW 10$>
INCRS ONE TO #13 BY R0

REPEAT
    ON.NOERROR LEAVE
MCOML TWO,TWO
INCRU ONE TO #13 BY #25

REPEAT
    ON.NOERROR LEAVE
MCOML TWO,TWO
INCRU ONE TO #13 BY R0
REPEAT
    ON.ERROR THEN <GOTOW 10$>
DECRS ONE TO #0 BY #1

REPEAT
    ON.ERROR THEN <GOTOW 10$>
DECRS ONE TO #0 BY R0

REPEAT
    ON.NOERROR LEAVE
MCOML TWO,TWO
DECRU ONE TO #0 BY #1

REPEAT

```

```

        ON.NOERROR LEAVE
        MCOML TWO,TWO
    DECRU ONE TO #0 BY R0

    REPEAT
        ON.ERROR THEN <GOTOW 10$>
    INCRS ONE TO #13 BY #1

    REPEAT
    ON.NOERROR LEAVE
        MCOML TWO,TWO
    INCRU ONE TO #13 BY #1

    SCASE R0 FROM 10 TO 30
    SET
        $CASE OUTRANGE
            MCOML ONE,ONE
        END

        $CASE 10 TO 15
            MCOML TWO,TWO
        END

        $CASE 16 TO 20,10$
        $CASE INRANGE
            MOVL R0,ONE
        END
    END

    SCASE R0 FROM 10 TO 30
    SET
    $CASE 10 TO 15
        MCOML TWO,TWO
    END

    $CASE 16 TO 20,10$

    $CASE INRANGE
        MOVL R0,ONE
    END
    END

    END MODULE
    .END

```

Listing 1  
Simple Mac example program.

The Simple Mac example program doesn't do anything except demonstrate that using Simple Mac allows the programmer to focus on the implementation instead of worrying about how to implement the flow of control through the program. In the above example, the necessary code to actually implement the flow of control would substantially outnumber the actual executable code in the program (not a normal situation, but for complex programs this can appear to be the case). Additional documentation in the form of comments, discussion about the purpose of the program,

why each block exists and what each statement is doing would also appear in a real program, adding to the ease of maintenance.

I have used Simple Mac in many environments. Of course, its principal use is in the development of Macro-11 and Macro-32 programs. I've written many device drivers for OpenVMS compatible with both the VAX and AXP versions of the system using Simple Mac. I've written system services and a variety of other applications in assembly language using Simple Mac. I've also developed embedded systems using Simple Mac on processor architectures other than the 16 and 32 bit Digital/Compaq/HP machines. In these cases, the macro processing capabilities of the assembler in the development environment was not sufficient to implement Simple Mac directly. Under these circumstances I found it necessary to write a preprocessor that converted the Simple Mac statements into assembly language which were then processed by the embedded system's development environment. By leaving the Simple Mac statements embedded in the generated assembly language source files, debugging was straightforward.

Since developing Simple Mac 20+ years ago, I've written several hundred thousand lines of assembly language on several different processor architectures. Use of Simple Mac has virtually eliminated the most common of my programming errors in assembly language and substantially improved my ability to maintain the assembly language code that I've written.

## Conclusion

Like all tools, Simple Mac must be used where and when appropriate. Most of the benefits of Simple Mac are simply copies of capabilities inherent in all modern high level languages. Given the choice between implementing in assembly language and any higher level language (save possibly Cobol) I will always choose a higher level language. But when, for whatever reasons, it's necessary to write code in assembly language, I use Simple Mac.

In summary, use of Simple Mac, along with good naming conventions and strong programming discipline can significantly improve programmer productivity and reduce maintenance costs for projects written in assembly language.

## Simple Mac syntax elements

Module	A group of assembly language source lines which begin with a START.MODULE, end with an END.MODULE, and [may] include one or more Simple Mac statements.
Module declaration	A callable unit (CALL/CALLS/CALLG).
Macro statement	Any valid assembly source statement, except one of the Simple Mac statements.
Block statement	Any of the block-structured statements: BEGIN, REPEAT, CASE, IF-THEN-ELSE, REPEAT, etc.
Block type	LOOP BLOCK CASE Segment name INNER OUTER REPEAT INCR DECR
Segment name	1-15 character symbolic name given to a

	program segment by a BEGIN statement.
label	Any valid MACRO address label.
condition	operand relation operand operand SET.IN/CLR.IN operand operand ON.IN/OFF.IN operand operand MASK.ON/MASK.OFF RESULT IS relation <macro-statement> IS relation
Conditional expression	condition condition AND condition condition OR condition
Asm constant expr	Any assembly time constant expression. It must be possible to evaluate the expression at assembly time, not link time.
Case range expression	asm-cons-expr TO asm-cons-expr <sup>1</sup> asm-cons-expr
\$Case range expression	case-range-expression INRANGE OUTRANGE <case-range-expression, ...>
Operand	Any valid assembly language operand.
Status	ERROR NOERROR

---

<sup>1</sup> the first asm constant expression must be less than the second



Relation	EQ/EQL	Signed Equal to
	EQU/EQLU	
	NE/NEQ	Unsigned Equal to
	NEU/NEQU	
	GT/GTR	
	HI/GTU/GTR	
	U GE/GEQ	Not Equal To
	HIS/GEU/GEQU	Not Equal To
	LT/LSS	Greater than
	LO/LTU/LSSU	
	LE/LEQLOS/L	Greater than Unsigned
	EU/LEQU	
	MINUS ZERO	Greater than or Equal to
	PLUS	
	CC	Greater than or Equal to Unsigned
	CS	
	VC	
	VS	
	SET.IN /ON.IN	Less than
	CLR.IN/OFF.IN	
	MASK.ON	Less than Unsigned
	MASK.OFF	Less than or equal
		Less than or Equal to Unsigned
		Sign bit set
		Zero bit set
		Sign bit clear
		Carry Clear
		Carry Set
		Overflow Clear
		Overflow Set
		Bit set in
		Bit off in
		Bit(s) on in the masked operand
		Bit(s) off in the masked operand

Type	B (byte) W (word) L (longword) F (float, currently not implemented) Q (quadword, currently not implemented) O (octaword, currently not implemented)
Sign	S (signed) U (unsigned)

A conditional expression is true if:

- it is a single condition and that condition is true.
- it is an OR expression and either of the conditions is true.
- it is an AND expression and both conditions are true.

It is false otherwise.

The IF, UNTIL, and WHILE statements operate on the specified data types when evaluating a conditional expression. If a type is unspecified, the default type is word.

The SCASE statement operates on the specified data type when evaluating a range, otherwise word entities are used. Float values are not valid for case ranges.

The IS operation in a condition tests the settings of the current condition codes. If the first operand is the reserved word RESULT, then the current setting of those codes is tested, otherwise the first operand is assumed to be a macro statement. This macro statement is executed and the resulting condition codes are tested.

SET.IN/CLR.IN and ON.IN/OFF.IN in a conditional expression refer to a bit in the second operand, as selected by the first operand.

MASK.ON/MASK.OFF in a condition expression refer to a collection of bits in the second operand, as masked by the first operand.

A Simple Mac source file contains one or more modules. SIMPLE-MAC statements may only appear within a Simple Mac module (START.MODULE/END.MODULE).

A program block consists of one or more assembly language statements delimited by a starting statement and an END statement.

- Conditional blocks begin with IF or SCASE statements.
- Loops begin with REPEAT, INCR, or DECR statements.
- Program segments are started by BEGIN or \$CASE statements.

Multiline conditional blocks and program segments must be terminated by an END statement. Loops can be terminated by an END, UNTIL, or WHILE statement. Note that THEN and ELSE statements do not terminate a block.

Single line IF-THEN, IF-LEAVE, IF-NEXT, and IF-GOTO statements do not constitute a conditional block and do not require an END statement.

Single line \$CASE-range-expression, label statements do not constitute a program segment.

## Simple Mac statements

### BEGIN segment-name

Assigns the specific symbolic name to this program segment. The symbolic name can then be used in LEAVE statements to exit the code contained within the block. BEGIN blocks may be nested.

```

BEGIN CONTROL
  CALL INIT
  CALL CSIOV1
  ON.ERROR LEAVE CONTROL
  CALL PROCES
  CALL CLEAN
END

```

## SCASE[type] operand FROM case-range-expression

To avoid confusing the Macro-32 compiler, the CASE instruction has been renamed SCASE.

This stands for Simple Mac CASE. This is the only distinction in syntax between SMPMAC.MAC (Macro-11) and SMPMAC.MAR (Macro-32). Many processors do not directly implement a case instruction. The SCASE macro provides the hooks by which one may be implemented.

Provides an EXTREMELY fast dispatch mechanism to a variety of possible alternative processing paths. This function is expensive in memory since the speed is achieved using a dispatch table. If the value of the case operand is outside the specified range and no OUTRANGE action is specified, the case falls through to the end. If no action is specified for some set of values of the operand, the case falls through to the end (remove the \$CASE INRANGE in the example and values 0 to 3 would fall through the CASE).

```

SCASE I FROM 0 TO 7
SET
  $CASE OUTRANGE,CASE.ERROR           Go here if out of range.

  $CASE 4 TO 7                         If 4 <= i <= 7 do this block
    CALL ON.HIBIT
  END

  $CASE INRANGE
    CALL OFF.HIBIT                     Otherwise do this block.
  END
END

```

## \$CASE \$case-range-expression[,label]

Identifies which value or range of values will be processed by the following block. If a label is specified, the \$CASE causes a jump to that label to occur. A \$CASE without a label argument defines the beginning of a block. The block may be exited via a LEAVE CASE. See the SCASE example above for usage.

## DECR[sign] loop-index FROM start TO end BY decrement

Initialize the loop index with the start value, and repeat the loop until the value of the loop index is less than the stated end value. The test for termination is either SIGNED or UNSIGNED depending on the value of the sign argument (S or U). The default is SIGNED comparison. The loop index must be of type long. This loop is a 0 trip loop, that is, if the initial value of the loop index is less than the end value, the loop is not executed. The termination test is performed at the top of the loop. If the lexical value of the loop-index and the start value are the same, no loop initialization is generated.

If the loop index is identical to the initial value, the loop assumes that the last thing done before entering the loop was to set the loop index. No special code is generated to test things. Since the 0 trip versions of these loops assume the loading of the loop index, you MUST either load the loop

index just before the DECR/INCR instruction or issue a processor appropriate comparison instruction (TSTL or CMPL for Macro-32) to get the condition codes set up properly.

## ELIF

```
ELIF[type] conditional-expression
[THEN]
    conditional-block
[ELSE
    conditional-block]
END
```

ELIF is syntactic sugar equivalent to:

```
ELSE
    IF[type] conditional-expression
        ...
    END
END
```

except that a single END statement to a series of IF-THEN-ELIF-THEN-ELIF statements will terminate all statements including the introducing IF. The purpose of the ELIF statement is to avoid generating excessive indentation with nested ifs. Excessive indentation can make a program very difficult to read and obscure the flow of control.

All forms of IF are also valid with ELIF, e.g., ELIF-LEAVE, ELIF-GOTO, etc. When used in this fashion, the ELIF terminates the current conditional block, and NO END is necessary since these forms of IF statement have no ELSE clauses.

```
IFW A EQ #14
THEN
    ...
ELIF A EQ #19
THEN
    ...
ELIF A EQ #23
THEN
    ...
ELSE
    ...
END
Conditional block is EXPLICITLY
terminated.

IFW A EQ #14
THEN
    ...
ELIF A EQ #23 GOTO DONE
Conditional block is IMPLICITLY
terminated.
```

## IF

```
IF[type] conditional-expression
[THEN]
    conditional-block
```

[ELSE  
conditional-block]

END

Executes the subsequent conditional block if the specified conditional expression is true, otherwise, it executes the optional ELSE conditional block. Type can be any of the primitive types supported by the processor architecture. For Macro-32 these are Byte, Word, Longword, Quadword, and Octaword, e.g. IFB, IFL, IFW, ...

```
IFB (R1)+ GT #0
THEN
    MOVL R0,-(R3)
    INCL COUNT
    CALL NEWLIN
ELSE
    CALL ERROR
END
```

### **IF[type] conditional-expression THEN <macro-statement>**

Executes the specified macro statement if the conditional expression is true. There are no restrictions on the macro statement. It can be anything from a single instruction, to a subroutine, to a macro-invocation in its own right. The macro-statement can, in fact, be another Simple Mac statement.

```
IFW R0 LT #MAX THEN <CALL SWQR5>
IFB R5 LSSU #TABEND THEN <ADD #3,R5>
```

### **IF[type] conditional-expression LEAVE block-type**

Transfers control to the end statement of the specified block type. LEAVE searches for the innermost block that satisfies block-type. For example, you can exit a repeat loop from within a case block or a BEGIN block from within an IF block. If a segment name is used in a LEAVE, the named segment is exited. If the block type OUTER is used, the outermost block is exited, independent of block type. If the block type INNER is used, the innermost block is exited, independent of block type. The default value for block-type is INNER.

```
REPEAT
    IF A EQ #END
    THEN
        ...
        IF (R0) EQL #0 OR RO GTRU LEAVE LOOP
        ...
    ELSE
        ...
    END
    ...
END
```

### **IF[type] conditional-expression GOTO label**

If the conditional expression is true, control is transferred to the specified label. This operation isn't frequently needed, but is here for those times when a good old GOTO is just what's needed.

```
IF #ERROR SET.IN CONTROL THEN GOTO ABORT
```

## **END [COMMENT]**

Terminates the current loop, conditional, or program segment. The optional comment can be used to match end statement with block start statement, improving readability.

```
BEGIN
...
END BEGIN
```

## **END.MODULE**

Terminates the current module. A single file can contain more than one module.

## **GOTO[type] label**

Transfers execution to the specified label. A GOTOB uses a branch instruction, a GOTOW uses a jump. Unlike the other instructions, the default type for GOTO is B rather than W. The implementation of this statement is rather heavily slanted towards the VAX processor architecture and may need to be modified on other architectures. In that event, I suggest that the GOTOB statement be used to indicate transfer of control to somewhere "close" visually and GOTOW to someplace "far away."

## **INCR[sign] loop-index FROM start TO end BY increment**

### **END**

Initialize the loop index with the start value, and repeat the loop until the value of the loop index is less than the stated end value. The test for termination is either SIGNED or UNSIGNED depending upon the value of the sign argument. The default is SIGNED comparison. The loop index must be of type word. This loop is a 0-trip loop. The termination test is performed at the top of the loop. If the lexical value of the loop-index and the start value are the same, no loop initialization is generated. This statement is biased towards the VAX implementation for loops.

See also DECR.

## **LEAVE block-type**

Transfers control to the end of the specified block type.

See IF conditional-expression LEAVE block-type for examples.

## **NEXT**

Transfer control to the next iteration of the loop. Control is transferred to the loop termination tests.

```
ON.ERROR THEN <macro-statement>
ON.ERROR GOTO label
ON.ERROR LEAVE block-type
```

Perform the specific action if the low bit of R0 is clear. This is an implementation of the OpenVMS-specific error-condition code convention in which error codes are returned in R0 and the low order bit is cleared. Other processors and systems have other conventions. For example, PDP-11 operating systems generally returned success and failure by clearing and setting the carry-condition flag. The Macro-11 implementation of Simple Mac reflects this difference.

**ON.NOERROR THEN <macro-statement>**

**ON.NOERROR GOTO label**

**ON.NOERROR LEAVE block-type**

The opposite of the ON.ERROR statement.

```
REPEAT
END
```

```
REPEAT UNTIL[type] condition
END
```

```
REPEAT
UNTIL[type] condition
```

```
REPEAT WHILE[type] condition
END
```

```
REPEAT
WHILE[type] condition
```

```
REPEAT
DECR[sign] loop-index TO value BY decrement
```

```
REPEAT
INCR[sign] loop-index TO value BY increment
```

Perform the loop UNTIL the condition is true, WHILE the condition is true, or until the loop-index is less than or equal to the specified value (for decrement repeat loops) or is greater than or equal to the specified value (for increment repeat loops). The termination conditions are tested at the place where they appear in the loop. If at the top, they are tested before the loop begins and the loop is a 0-trip loop; if at the bottom, they are tested when the loop terminates and the loop is a 1-trip loop. The loop index must be initialized before entering the loop and must be of type word. This will vary from processor architecture to architecture.

## **START.MODULE**

Defines the beginning of a module. Initializes the state of Simple Mac.

## **For more information**

[Dick Munroe](#) – my resume and contact information, I'm looking for work, contract or permanent.