Macross 6502

an assembler for people who hate assembly language
by
Chip Morningstar

Lucasfilm Ltd. Games Division
July 7, 1986

ABSTRACT

This document describes the 6502 version of *Macross*, a super-duper cross-assembler that has actually been used!

Introduction

Macross is a generic cross assembler for a variety of different microprocessors. This document describes the 6502 version of *Macross*. *Macross* differs from many macro assemblers in that it provides a number of "higher level" constructs not traditionally found in assembly language. These include block-structured flow-of-control statements (`if`, `while`, etc.) and the ability to define record-oriented data structures (`struct`). In addition, it contains a powerful macro capability that is based on syntactic structural manipulations rather than simple text substitution. *Macross* is, in fact, a complete block-structured programming language in its own right which is interpreted at assembly time.

General Form of *Macross* Statements

Stylistically, much of *Macross* is patterned after **C**. In particular, the form of many keywords and of block structured entities is derived from **C**. Unlike **C** however, *Macross* follows the convention of more traditional assemblers that statements are delimited by line boundaries (i.e., one statement per line, with the end of a line ending a statement).

In general, spaces and tabs are treated as whitespace characters and are ignored. Therefore, such characters may be used as the assembly language programmer desires to format his or her program according to personal taste. For the convenience of the programmer, *Macross* relaxes the syntax rule that newlines always end a statement: newlines may also be treated as whitespace characters (again, for purposes of formatting) in places where it would be syntactically unambiguous to do so (i.e., where a statement obviously cannot terminate, such as after a comma). For example:

```
byte     1, 2, 3, 4,
         5, 6, 7, 8
```

is allowed and is equivalent to

```
byte     1, 2, 3, 4, 5, 6, 7, 8
```

Comments begin with a semicolon ("`;`") and continue to the end of the line, as is common in many assemblers. In addition, *Macross* supports **C** style comments bracketed by "`/*`" and "`*/`".

As with most assemblers, *Macross* statements are allowed to begin with a label (or several labels, if you like). A label is denoted by an identifier followed by a colon ("`:`"). There is no requirement that the label start in column 1, or anything like that. Labels, if present, merely must precede anything else in a statement.

An identifier is just what you'd expect from all your years of programming experience: a string of letters and digits that must begin with a letter. As is traditional in Unix* land, the underscore character ("_") is considered to be a letter (smacks of hubris that, but tradition is tradition). Departing from Unix tradition, upper- and lower-case characters are not distinct from each other for purposes of distinguishing identifiers. If you use mixed case for stylistic reasons, *Macross* will remember the case of the letters in an identifier when it was first defined, so that symbol table dumps and cross-reference listings will retain whatever case usage style you've adopted. There is, in principle, no restriction imposed upon the length of identifiers.

The Language

In what follows, things in `this typewriter like typeface` are keywords and characters that are used literally. Things in *italics* are other kinds of syntactic entities. Double brackets ("⟦" and "⟧") enclose things that are optional, while brackets followed by an asterisk ("*") enclose things that may be optionally repeated zero or more times.

1. The Instruction Statement

The most elementary *Macross* statement is the instruction statement, wherein the programmer specifies a machine instruction to be assembled. The instruction statement is

⟦ *label* ⟧* *opcode* ⟦ *operand* ⟦ `,` *operand* ⟧* ⟧

just like every assembler ever made (except **a65**, of course). *Opcode* is an identifier which is either a machine instruction mnemonic (a list of which mnemonics are accepted by *Macross* is given in **Appendix F** at the end of this document) or a macro name. For example:

```
and         foobar
someMacro   foo, bar, baz, bletch
```

The operands of an instruction may be any of the various types of operands allowed by the various addressing modes of the target processor. In the case of the 6502, these are:

1.1. Direct addressing

Direct addresses take the form

expression

and are used both for instructions that use direct addressing and ones that use relative addressing (the offset is computed automatically by *Macross*).

1.2. Indirect addressing

Indirect addresses take the form

`@` *expression*

Of course, the only 6502 instruction which accepts an indirectly addressed operand is `jmp`.

* Unix is a footnote of Bell Laboratories.

1.3. Immediate operands

Immediate operands take the form

expression

In the 6502, immediate mode operands are restricted to eight bit quantities. *Macross* will give an error message if the operand value is larger than this.

1.4. Indexed addressing

Indexed addressing operands take the forms

`x [` *expression* `]`
`y [` *expression* `]`

An alternate form of indexed addressing which is supported by *Macross* allows the symbolic selection of a field of a `struct` pointed to by an index register

`x .` *identifier* ⟦ `.` *identifier* ⟧*
`y .` *identifier* ⟦ `.` *identifier* ⟧*

This is explained in greater detail in the sections on `structs` and expressions below.

1.5. Pre-indexed indirect addressing

Pre-indexed indirect addressing is specified by operands of the form

`@ x [` *expression* `]`

As with ordinary indexed addressing, there is a form of pre-indexed indirect addressing which uses `struct` fields

`@ x .` *identifier* ⟦ `.` *identifier* ⟧*

1.6. Post-indexed indirect addressing

Post-indexed indirect addressing is specified by operands of the form

`y [ @` *expression* `]`

There is no `struct`-oriented form of post-indexed indirect addressing since there doesn't seem to be any consistent interpretation of such a thing that makes sense.

1.7. Register addressing

The only register in the 6502 which is used as an operand in its own right is the accumulator

`a`

For the sake of completeness, so that macros may have them as operands, *Macross* also allows either of the index registers to be used as operands

`x`
`y`

These are equivalent to

```
x[0]
y[0]
```

Note that `a`, `x` and `y` are reserved words in the *Macross* language and so cannot be used as labels, variable names, etc. It might seem natural to call a variable `x` but you can't. Sorry.

1.8. Text operands

For the sake of macros, text strings may also be used as operands

"*any string you like*"

The same conventions regarding escaped characters (using "\") that are followed by **C** are followed by *Macross*. These are documented in **Appendix E**. Note that on many target machines the codes that these escape sequences stand for are meaningless. They are provided primarily as a convenience for writing calls to `printf()`.

2. The Flow of Control Statements

Macross provides a number of statements which allow program flow of control to be specified in a **C**-like block structured fashion. This include a conditional execution statement (`if`) and three conditional loop statements (`while`, `do-while` and `do-until`). These statements assemble into the appropriate conditional branches and jumps to realize the desired construct.

2.1. If statement

The `if` statement has the following form

```
⟦ label ⟧*   if ( condition ) {
                  ⟦ statement ⟧*
             } ⟦ elseif ( condition ) {
                  ⟦ statement ⟧*
             } ⟧* ⟦ else {
                  ⟦ statement ⟧*
             } ⟧
```

condition is either the name of one of the target processor's hardware condition codes such as can be tested for in a conditional branch instruction (e.g., `carry`, `overflow`, etc.— the complete list is in **Appendix B**) or one either of these negated using the "logical not" operator ("`!`") or the name of one of the more complex conditions which *Macross* understands (`geq`, `lt`, etc., discussed shortly). The condition is used to determine the appropriate type of branch instruction(s) to use. For example,

```
if (plus) {
     statements-1
} elseif (carry) {
     statements-2
} else {
     statements-3
}
```

expands into this (the labels are made up for illustrative purposes only):

```
        bmi temp1
        statements-1
        jmp temp3
temp1:  bcc temp2
        statements-3
```

```
         jmp temp3
temp2:   statements-3
temp3:   whatever follows
```

The keyword `elseif` may be used as shown, or specified as two separate keywords, `else if`, depending on the programmer's whim.

Macross knows about certain conditions which are more complex than those than can be realized with single conditional branch instructions. These conditions correspond to the results of comparison operations (such as `geq` — "greater than or equal to") that may require rather complicated sequences of conditional branches to implement. These may be used in any location where an ordinary condition may be used. One simply should keep in mind that they can result in a non-trivial amount of code being generated, if one is concerned about speed of execution. The complete list of these complex conditions along with the object code that they produce is given in **Appendix B**.

2.2. **While** statement

The `while` statement has the following form

```
⟦ label ⟧*   while ( condition ) {
                 ⟦ statement ⟧*
             }
```

condition is as described above for the `if` statement. An example of the `while` statement would be

```
         while (!carry) {
              statements
         }
```

which would turn into

```
         bcs  temp1
temp2:   statements
         bcc temp2
temp1:   whatever follows
```

2.3. **Do-while** statement

The `do-while` statement is similar to the `while` statement except that the condition is tested at the bottom of the loop. It has the form

```
⟦ label ⟧*   do {
                 ⟦ statement ⟧*
             } while ( condition )
```

For example

```
         do {
              statements
         } while (equal)
```

which is equivalent to

```
temp:    statements
         beq temp
```

2.4. **Do-until** statement

The `do-until` statement is the same as the `do-while` statement except that the sense of the condition is negated. It has the form

⟦ *label* ⟧* `do {`
        ⟦ *statement* ⟧*
    `} until (` *condition* `)`

For example

```
do {
    statements
} until (equal)
```

which is equivalent to

```
temp:   statements
        bne temp
```

3. The Data Statements

The data statements allow the allocation of memory space and the storage of constant data. These statements are like the ones found in most assemblers. There are several different forms, each for a different type of data.

3.1. **Block** statement

The `block` statement allocates blocks of memory without initializing the bytes to any particular value (actually, the loader will in all likelihood initialize these to 0, but it is probably not really wise to rely on this). It has the form

⟦ *label* ⟧* `block` *expression* ⟦ `,` *expression* ⟧*

The *expression*s are the sizes of the blocks to reserve, expressed in bytes.

3.2. **Align** statement

The `align` statement aligns the current location counter to an integer multiple of some value (e.g., to align with a word boundary). It has the form

⟦ *label* ⟧* `align` *expression*

The *expression* is the multiple to which the current location counter is to be aligned. For example,

```
align       2
```

would align the current location to a word boundary, while

```
align       0x100
```

would align to a page boundary.

3.3. **Constrain** statement

The `constrain` statement provides a means of constraining a portion of code or data to be located within a region of memory bounded by addresses of integer multiples of some value (e.g., within a page). Its form is

> `constrain (` *boundary* `) {`
> ⟦ *statement* ⟧*
> `}`

Boundary may be any expression which evaluates to a number. The *statement*s are assembled normally. If assembling in absolute mode, an error message is issued if the current location counter crosses an integer multiple of *boundary*. If assembling in relocatable mode, information about the constraint will be output in the object file and the contents of the constrained block will be relocated as needed to satisfy the constraint (note that this means that it is unsafe to assume that the things in the assembly source immediately before the `constrain` statement, the contents of the constrain block itself, and the things in the assembly source immediately after the `constrain` statement will be located in contiguous locations in the eventual target machine address space). For example,

> `constrain (0x100) {`
> *statements*
> `}`

constrains the given statements to all fit within a page.

3.4. **Word** statement

The `word` statement allocates words, i.e., two byte chunks, of memory. It takes the form

⟦ *label* ⟧* `word` *expression* ⟦ `,` *expression* ⟧*

The *expression*s must evaluate to quantities that can be contained in 16 bits, of course. For example,

```
word     0x1234, foobar
```

would allocate two words, the first of which would be initialized to the hexadecimal value `0x1234` and the second to whatever the value of `foobar` is.

3.5. **Dbyte** statement

The `dbyte` statement is just like the `word` statement, except that the word is byte-swapped in memory. Its form is

⟦ *label* ⟧* `dbyte` *expression* ⟦ `,` *expression* ⟧*

3.6. **Long** statement

The `long` statement allocates longwords, i.e., four byte chunks, of memory. It takes the form

⟦ *label* ⟧* `long` *expression* ⟦ `,` *expression* ⟧*

The *expression*s must evaluate to quantities that can be contained in 32 bits, of course. For example,

```
long     0x12345678, foobar
```

would allocate two longwords, the first of which would be initialized to the hexadecimal value `0x12345678` and the second to whatever the value of `foobar` is.

3.7. **Byte** statement

The `byte` statement is similar to the `word` and `dbyte` statements, except that it allocates single byte chunks. Its form is

⟦ *label* ⟧* `byte` *expression* ⟦ `,` *expression* ⟧*

An *expression*, in this case, is either an ordinary expression (see **Expressions**, below) which must evaluate to an 8-bit quantity, indicating the value for a single byte to be reserved, or a string (see above, under the discussion of text operands), indicating that the characters in the string should be placed in memory at the current location.

3.8. **String** statement

The `string` statement is much like the `byte` statement, except that the values indicated are followed in memory by a zero byte. This enables the convenient declaration and allocation of NULL terminated character strings. This feature is of little use in the 6502 version of *Macross* but is provided for compatability with future versions targeted at more sophisticated processors. The form of the `string` statement is

⟦ *label* ⟧* `string` *expression* ⟦ `,` *expression* ⟧*

3.9. **Struct** statement

The `struct` statement enables the declaration and allocation of record-oriented data structures. There are two forms of the `struct` statement, the first of which declares a `struct` record type, and the second of which causes space to be set aside in memory for a `struct` that has already been declared. The form of the first type of `struct` statement is

⟦ *label* ⟧* `struct {`
    ⟦ *dataStatement* ⟧*
`}` *name*

*dataStatement*s are any of the data statements described in this section (section 3). *Name* becomes the name of the `struct`. Any labels inside the `struct` become *fields* of the data structure which may be referred to later in expressions using the "`.`" operator, as in **C**. A more complete description of the semantics of `struct`s is given in the section below on expressions.

The first form of the `struct` statement, called a "`struct` definition", lays out the constituent parts of a data structure and gives those names to those parts. The second form of the `struct` statement, called a "`struct` instantiation",

⟦ *label* ⟧* `struct` *name*

causes storage for the `struct` named by *name* to be allocated. A `struct` definition *may not* contain another `struct` definition, but it *may* contain a `struct` instantiation. For example,

```
struct {
    pointer:block 2
    class:  block 1
} fooThing
```

would create a `struct` called `fooThing`. Then,

```
fooLabel:    struct   fooThing
```

would allocate one at the current location at the address labeled `fooLabel`. This could then be used as follows:

```
and     fooLabel.class
jmp     @fooLabel.pointer
```

which would AND the accumulator with the `class` field of the `struct` and then jump to wherever the `pointer` field pointed to. If the `x` index register already contained the address of this `struct`, then one could say

```
and     x.class
```

4. The Symbol Definition Statements

The various symbol definition statements allow the declaration of symbolic variables and values and the definition of macros and functions.

4.1. **Define** statement

The `define` statement enables the programmer to create symbolic names for values. It has two forms. The first

`define` *symbolname*

creates a new symbol, *symbolname* (an identifier), and gives it the special value `unassigned`. Any attempt to take the value of an unassigned symbol will cause an error message from the assembler. The symbol will, however, cause the `isDefined()` built-in function (see **Expressions**, below) to return `TRUE` if passed as an argument. It is also an error to `define` a symbol that has already been `define`d.

The second form of the `define` statement

`define` *symbolname* = *expression*

creates the symbol and gives it the value obtained by evaluating *expression* (see **Expressions**, below). Actually, what `define` does is create a symbolic name for *expression* and the save this expression away in a secret place. This means that symbols in *expression* may be forward references, e.g., labels that haven't been encountered yet. It is also possible to forward reference to symbols that are defined by future `define` statements, for example:

```
define foo = bar + 2
define bar = 47
```

effectively defines `foo` to be `49`. Beware, however, as there is no way for the assembler to detect mutually recursive references of this sort, so that

```
define foo = bar + 2
define bar = foo + 2
```

will be happily swallowed without complaint, until you actually try to use `foo` or `bar` in an instruction, whereupon *Macross*'s expression evaluator will go into infinite recursion until it runs out of stack space and crashes the assembler (it looks to see what `foo` is and sees that it's `bar + 2`, so it looks to see what `bar` and see that it's `foo + 2`, so it looks to see what `foo` is... To have the assembler detect and signal this error would, in the general case, add much complication and inefficiency (read: make your programs assemble a lot more slowly) for little return).

The scope of symbols defined in either of these two ways extends in time from the definition itself to the end of the assembly.

4.2. **Variable** statement

The `variable` statement enables the programmer to declare symbolic variables for future use. Similar to the `define` statement, it has two forms. The first

`variable` *symbolname*

creates a variable named *symbolname* and gives it the special value `unassigned`, just like the analogous `define` statement.

The second form of the `variable` statement

`variable` *symbolname* = *expression*

creates the variable and gives it the value obtained by evaluating *expression*. The scope of variables defined in either of these two ways extends from the `variable` statement itself to the end of the assembly (i.e., the variable is global).

The difference between the `define` statement and the `variable` statement is that the `define` statement creates what is in essence a constant whereas the `variable` statement creates a true variable. The value of a variable may change (e.g., it may be assigned to) during the course of assembly. In addition, the expression which establishes a symbol's value in a `define` statement may contain forward references (i.e., labels whose values are unknown because they haven't been encountered yet) whereas the expression assigning an initial value to a variable must be made up of terms all of whose values are known at the time the `variable` statement is encountered in the assembly.

A variable may also be declared as an array, using the form

`variable` *symbolname* `[` *length* `]`

where *length* is an expression that indicates the number of elements the array is to have. *Macross* arrays are zero-based, so the elements are indexed from 0 to *length*-1. As with ordinary variables, the elements of the array may be initialized in the `variable` statement using a statement of the form

`variable` *symbolname* `[` *length* `]` `=` *expression*
⟦ `,` *expression* ⟧*

The *expression*s are assigned sequentially into the elements of the array. If the array length is greater than the number of *expression*s given, the remaining elements are filled with zeroes. Of course, you should not specify more than *length* expressions or the assembler will complain at you.

4.3. **Macro** statement

The `macro` statement is used to define macros (surprise!). Its syntax is

`macro` *macroname* ⟦ *argumentname* ⟦ `,` *argumentname* ⟧* ⟧ `{`
    ⟦ *statement* ⟧*
`}`

where *macroname* is just that and the *argumentname*s are identifiers corresponding to the formal parameters of the macro (in the classical fashion). When the macro is called, the call arguments are bound to these symbols and then *Macross* assembles the *statement*s which form the macro body. The scope of these symbols is limited to the inside of the macro body and their values go away when the macro expansion is completed. The *statement*s may be any valid *Macross* statements except for `macro` statements and `function` statements (i.e., macro and function definitions may not be nested).

Statement labels used inside macros can be made local to the macro by preceding the label identifier with a dollar sign ("$"). For example,

```
macro   fooMac  arg {
        jmp     $foo
        wordarg
$foo:   nop
}
```

defines a macro named `fooMac` that emits a word of data that gets jumped over. The label `$foo` is local to the macro: both the reference to it in the first line of the macro and its definition on the third will only be seen inside the macro. Each time the macro is called, the `jmp` will refer to the location two instructions ahead, and any other macros that might contain `$foo` will not affect this nor will they be affected by this.

It is possible to define macros which take a variable number of arguments. This is accomplished by following the last argument in the `macro` statement by `[ ]`. This declares the argument to be an array, which gets assigned a list of all of the parameters not accounted for by the other declared arguments. This array may be interrogated with the `arrayLength()` built-in function (to find out how many extra parameters there were) and accessed just like a regular array. For example,

```
macro   enfoon  precision, args[] {
    mvariable len = arrayLength(args)
    mvariable i
    wordprecision
    mfor (i=0, i<len, ++i) {
        byteargs[i]
    }
}
```

declares the macro `enfoon` that takes one or more parameters. The first parameter is bound to `precision`, while the remainder are collected in an array that is bound to `args`. It emits the first parameter as a word value and the remaining parameters as bytes.

4.4. **Function** statement

The `function` statement is used to define functions. Its syntax is

> `function` *funcname* `(` ⟦ *argumentname* ⟦ `,` *argumentname* ⟧* ⟧ `) {`
> ⟦ *statement* ⟧*
> `}`

where *funcname* is the name of the function and the *argumentname*s are identifiers corresponding to the formal parameters of the function. When the function is called, the call arguments are evaluated and then bound to these symbols and then *Macross* assembles the *statement*s which form the function body. The scope of these symbols is limited to the inside of the function body and their values go away when the function evaluation is completed. As with macro definitions, the *statement*s may be any valid *Macross* statements except for the `macro` and `function` statements. A function may return a value using the `freturn` statement, which is described below.

Just as you can define macros that take variable numbers of arguments, so too can you define functions. The mechanism is the same. For example

```
function sum(args[]) {
    mvariable len = arrayLength(args)
    mvariable i
    mvariable result = 0
```

```
        mfor (i=0, i<len, ++i) {
            result += args[i]
        }
        freturn(result);
    }
```

which simply returns the sum of its arguments.

4.5. **Undefine** statement

The `undefine` statement allows symbol and macro definitions to be removed from *Macross*' symbol table. It takes the form

> `undefine` *symbolname* ⟦ `,` *symbolname* ⟧*

The named symbols go away as if they never were — they are free to be defined again and the `isDefined()` built-in function will return `FALSE` if passed one of them as an argument.

5. Macro Body Statements

Macross provides several statements which are primarily intended to manage the flow of control (or, rather, the "flow of assembly") within a macro or function definition. Some of these statements are analogs to the flow of control statements described above in section 2. However, one should keep in mind that these statements are executed interpretively at assembly time, whereas the previously described statements result in machine code in the target-processor-executable output of the assembly process.

Although these statements are intended primarily for use within macros and functions, their use is not restricted and they may be used at any point in a program. In particular, the `mif` statement (to be described shortly) is the means by which conditional assembly may be realized.

5.1. Blocks

The construct

```
{
    ⟦ statement ⟧*
}
```

is called a *block* and is in fact a valid *Macross* statement type in its own right. Blocks are used extensively in *Macross* to form the bodies of flow-of-control statements, flow-of-assembly statements and macros.

5.2. **Mdefine** statement

The `mdefine` statement

> `mdefine` *symbolname*

or

> `mdefine` *symbolname* = *expression*

operates like (and is syntactically congruent with) the `define` statement, except that the scope of the symbol definition is restricted to the body block of the macro or function in which the `mdefine` appears. The symbol definition is invisible outside that block, though it *is* visible inside any blocks that may themselves be contained within the macro or function.

5.3. **Mvariable** statement

The `mvariable` statement

`mvariable` *symbolname*

or

`mvariable` *symbolname* = *expression*

bears exactly the same relationship to the `variable` statement that the `mdefine` statement does to the `define` statement. It declares a variable whose scope is limited to the function or macro in whose definition it appears.

5.4. **Mif** statement

The `mif` statement conditionally assembles the statements contained in the block following it:

```
mif ( condition ) {
      ⟦ statement ⟧*
} ⟦ melseif ( condition ) {
      ⟦ statement ⟧*
} ⟧* ⟦ melse {
      ⟦ statement ⟧*
} ⟧
```

unlike the `if` statement, the *condition* may be any expression whatsoever. *Macross* follows the **C** convention that the value 0 represents `FALSE` and any other value represents `TRUE` (in fact, the symbols `TRUE` and `FALSE` are "predefined" by the assembler to have the values 1 and 0 respectively). The meaning of the `mif` construct is the obvious one, but keep in mind that it is interpreted at assembly time and has no direct bearing on the execution of the resulting assembled program.

5.5. **Mwhile** statement

The `mwhile` statement repetitively assembles a block of statements so long as a given condition remains true. Its form is

```
mwhile ( condition ) {
      ⟦ statement ⟧*
}
```

As with `mif`, the *condition* may be any valid *Macross* expression and the interpretation is what it seems.

5.6. **Mdo-while** statement

The `mdo-while` statement provides an alternative to the `mwhile` statement by testing at the bottom of the loop instead of at the top. Its form is

```
mdo {
      ⟦ statement ⟧*
} while ( condition )
```

5.7. **Mdo-until** statement

The `mdo-until` statement is the same as the `mdo-while` statement except that the sense of the condition is negated. It has the form

```
mdo {
      ⟦ statement ⟧*
```

```
} until ( condition )
```

5.8. Mfor statement

The `mfor` statement provides a more general looping construct analogous to the **C** `for` loop. Its form is

```
mfor ( expression-1 , expression-2 , expression-3 ) {
     ⟦ statement ⟧*
}
```

where *expression-1* is an initialization expression, *expression-2* is a test to see if looping should continue, and *expression-3* is executed at the bottom of the loop to set up for the next time around, just as in **C**. Note that, unlike **C**, the *expressions* are separated by commas, not semicolons. This is because semicolons are used to delimit line comments.

The `mfor` statement is equivalent to

```
expression-1
mwhile (expression-2 ) {
     statements
     expression-3
}
```

5.9. Mswitch statement

The `mswitch` statement provides a means of selecting one of a number of blocks of code for assembly depending upon the value of some expression. Its form is:

```
mswitch ( selectionExpression ) {
     ⟦ mcase ( expression ⟦ , expression ⟧* ) {
          ⟦ statement ⟧*
     } ⟧*
     ⟦ mdefault {
          ⟦ statement ⟧*
     } ⟧
}
```

The way this works is as follows (it's actually easier to use than to explain): *selectionExpression* is evaluated. Each of the *expressions* associated with the various `mcase` clauses (if any) is then evaluated in turn and the resulting value compared to that of *selectionExpression*. When and if one of these values "matches" the value of *selectionExpression* the block of *statements* associated with the corresponding `mcase` clause is immediately assembled and execution of the `mswitch` statement is complete. If no such value matches and there is an `mdefault` clause, the block of *statements* associated with the `mdefault` clause is assembled. If no value matches and there is no `mdefault` clause then nothing is assembled as a result of the `mswitch`. When we say the values "match", we mean that either the expressions evaluate to the same number or to strings which are identical except for the case of alphabetic characters. For example:

```
mswitch (foo) {
     mcase ("hello", "fnord") {
          statements-1
     }
     mcase ("ZAP!") {
          statements-2
     }
     mdefault {
```

```
                    statements-3
            }
    }
```

would switch on the value of the symbol `foo`. If the value of `foo` was `"hello"`, *statements-1* would be assembled. If the value of `foo` was `"zap!"`, *statements-2* would be assembled (since the string comparison is done independent of case). If the value of `foo` was `"cromfelter"` or `47`, then *statements-3* would be assembled by default.

5.10. **Freturn** statement

A *Macross* function may (and probably will) be called from an expression in the traditional manner of functions throughout the annals of computer science. In such a situation, the programmer may wish to have a function return a value. The `freturn` statement enables this. Its form is

`freturn` ⟦ *expression* ⟧

where *expression* is any permissible *Macross* expression as described below under **Expressions**. If no *expression* is given, the macro simply returns without having a value as its result. Any attempt to use the (non-existent) value returned by a call to function which doesn't return a value will result in an error message from the assembler. Function calls will automatically return without a value upon reaching the end of the block that forms the body of the function.

6. Miscellaneous Statements

Various useful statements don't fall into any of the above categories.

6.1. **Include** statement

The `include` statement allows the text in other files to be included in the source program being assembled, in the time-honored fashion. Its form is

`include` *filename*

where *filename* is a string value giving the name of the file to be included. `Include`d files may themselves contain `include` statements to any number of levels of recursion (within reason).

6.2. **Extern** statement

The `extern` statement allows you to declare symbols to be visible to the linker outside of the file in which they are found. Its use:

`extern` *symbol* ⟦ `,` *symbol* ⟧*

6.3. **Start** statement

The `start` statement declares the starting address of a program.

`start` *expression*

where the *expression* indicates the start address. There should be no more than one `start` statement in a program. If no start address is specified, the object file will be produced without a start address entry.

6.4. **Assert** statement

The `assert` statement provides a means of testing assembly time conditions and generating programmer specified error messages if those conditions are not satisfied. Its syntax is:

`assert (`*condition*`)` ⟦ *textString* ⟧

where *condition* is an expression such as those used in the `mif` statement. This condition is evaluated and if `FALSE` then the message *textString*, if given, is written to the standard output. The message is written in the form of a conventional *Macross* error message, giving the file and line number on which the failed assertion occured. If *textString* is omitted then a simple error message to the effect that the assert failed will be output. For example,

```
assert (foo == 1) "Hey!  You blew it."
```

would check to see that the value of the symbol `foo` is `1`, and if it isn't would issue an error message containing the string `"Hey!  You blew it."`.

6.5. **Org** statement

The `org` statement adjusts the current location counter and tells the assembler to start locating instructions and data in absolute memory locations.

`org` *expression*

The *expression* indicates the new current location. If this is an absolute address, *Macross* starts assembling at the specified absolute memory location. If, on the other hand, it is relative to a relocatable address or to the current location counter, the current location counter is simply adjusting accordingly.

6.6. **Rel** statement

The `rel` statement restores *Macross* to assembling code in a relocatable fashion, if it was not already doing so. The relocatable location resumes from wherever it was left the last time an absolute `org` stopped relocatable code assembly. *((explain this better))*

6.7. **Target** statement

The `target` statement

`target` *expression*

tells the assembler to assemble as if it had been `org`ed to a particular address without actually performing the `org`. The *expression* indicates the location to start assembling from. This must be an absolute address (and the `target` statement may only be used when assembling in absolute mode).

The result of the `target` statement is that assembly proceeds from the current location but labels will be defined as if it was proceeding from the location specified by *expression* and references to the current location counter will be offset by the difference between the actual current location counter value and the value of *expression*. This effect will persist until the next `org` or `target` statement. For example

```
        org     0x1000
        target  0x0800
foo:    rts
bar:    word    0x1234
        word    here
        org     someplaceElse
```

would first set the current location counter to `0x1000`. At location `0x1000` it would assemble an `rts` instruction while giving the label `foo` the value `0x800`. Then, at location `0x1001` it would deposit the word value `0x1234` while giving the label `bar` the value `0x801`. At location `0x1003` it would deposit the word value `0x803`. Finally, the second `org` would set the current location counter to `someplaceElse` and the effects of the `target` statement would cease.

Expressions

The expression syntax of *Macross* was chosen to be as close to that of **C** as possible. *Macross* recognizes the same set of operators as **C** with few exceptions, and the operators have the same precedence with respect to each other that everyone is used to.

Another important feature of expressions in *Macross* is that expressions by themselves on a line are valid statements. Assignment expressions, uses of the post- and pre-increment and decrement operators ("++" and "--"), and calls to functions used like procedures are the most useful applications of this. For example

```
foo = 5
bar++
printf("Hello world\n")
```

are all valid statements.

1. Primitive expressions

The most primitive expressions are identifiers, numbers, characters, character strings, array references, and function calls.

Identifiers have already been described, in the section **General Form of *Macross* Statements** above. The only thing to add here is that the special identifier `here` denotes the value of the current location counter.

Numbers may be decimal, octal, hexadecimal, binary or quarters. The form of the first three of these is as in **C**: decimal numbers are denoted by a sequence of decimal digits that does not begin with a "`0`"; octal numbers by a sequence of octal digits that *does* begin with a "`0`"; and hexadecimal numbers by a sequence of hexadecimal digits (the decimal digits plus the letters "`a`" through "`f`", in either upper or lower case), preceded by "`0x`" or "`0X`". Binary numbers and quarters are represented analogously. Binary numbers are represented by a sequence of "`0`"s and "`1`"s preceded by "`0b`" or "`0B`". Quarters are base-four numbers (for two-bit entities like Atari pixels) and are represented by the sequences of the digits `0` through `3` preceded by "`0q`" or "`0Q`" (the credit for this idea goes to Charlie Kellner).

Character constants are denoted the same as in **C**: by a single character enclosed in apostrophes ("`'`"). The same conventions about characters escaped with a backslash ("`\`") also apply. Strings may be of varying lengths and are enclosed in quotation marks ("`"`"), as discussed above in the explanation of instruction operands.

An array reference in *Macross* have the same syntactic form that they have in **C**:

array `[` *index* `]`

where *array* is an array (which is usually an identifier but can be any expression that results in an array value or a character string — note that for convenience a string can be treated as simply an array of characters) and *index* is an integer with a value between 0 and the length of the array minus one, inclusive (all *Macross* arrays are zero-based).

A function call, in *Macross*, is syntactically the same as in **C**: the name of the function being called followed by a comma-separated argument list in parenthesis. The argument list, as in **C**, may be empty. The arguments themselves may be arbitrary instruction operands (described above in section 1). A number of built-in functions are provided by *Macross* to perform a variety of useful operations. These are discussed in **Appendix C**.

2. Operators

Expressions may be constructed from the primitive elements described above and from other expressions using a variety of unary and binary operators. The operator set is patterned after **C**'s. The only **C** operators not supported are:

[1] "`?:`" (arithmetic if) — not supported for reasons of syntactic confusion on both the part of the parser attempting to parse it and the programmer attempting to use it.

[2] "`,`" (comma) — used in *Macross* as an important separator and thus not available.

[3] unary "`*`" and "`&`", `sizeof`, casts and "`->`" — not relevant here.

All of the assignment operators ("+=", "−=", etc.) *are* supported by *Macross*.

Macross reinterprets the **.** operator in that "*expression* **.** *structfieldname*" is interpreted as adding the offset value implied by *structfieldname* (i.e., the distance in bytes into a `struct` to reach the named field) to the address that is the value of *expression*.

Macross adds to the operator set the following:

[1] "`?`" — as a unary operator, takes the high order byte of the word value that is its argument.

[2] "`/`" — as a unary operator, takes the low order byte of the word value that is its argument.

[3] "`^^`" — a binary operator, denotes logical exclusive-OR. This is simply an orthogonal extension for the sake of completeness.

Of course, parenthesis can be used at any point to override the normal precedence of the various operators. A full list of all the operators that *Macross* understands is given in **Appendix D**.

3. Expression evaluation

In order to make the most effective use of expressions in the *Macross* environment, it is helpful (and at times necessary) to understand how and when *Macross* evaluates them.

When *Macross* evaluates an expression, it may have one of three sorts of results. These are *success*, *undefined*, and *failure*. A *success* result means that *Macross* encountered no problems evaluating the expression, and whatever value it evaluated to is just used as needed. A *failure* result indicates that there was a problem of some sort. Usually this is a result of some user error. In any case, an appropriate diagnostic message will be issued by the assembler and the statement in which the expression was found will not be assembled.

An *undefined* result is where the complications, if any, arise. An expression will evaluate to an *undefined* result if one or more of the terms of the expression are undefined symbols. Usually these are labels which simply haven't been encountered yet (i.e., they are forward references). In certain contexts, such as the operand of a machine instruction, this is a legitimate thing to do, and in certain others, such as the condition of a `mif` statement, this is not allowed at all. In the latter case, an *undefined* result is just like a *failure* result. In the former case, the assembler is forced to get fancy in order to make it all work right.

What *Macross* does is squirrel away a copy of the expression along with a pointer as to where in the object code the value of the expression is supposed to go. At the end of assembly, the undefined label will presumably now be defined, and *Macross* evaluates the saved expression and pokes the result into the appropriate location. (If, at this point, the undefined label is still undefined, an error message to that effect is issued). Clearly, if an expression has side effects (such as changing the value of some global variable), this can result in some confusing behavior. The *Macross* assembler is smart enough to not let you do anything that has overt side effects in an expression that is being saved away for future evaluation. The things which are disallowed in such a case are assignments and uses of the post- and pre-increment and decrement operators ("++" and "−−"). Functions, however, may have side effects and *Macross* does not try to prevent you from using function calls in expressions that get saved for later evaluation. It can, and will, detect some, but not all, side effects during the later evaluation and give a suitable error message. This is because it is perfectly legitimate to use a function call to a function that doesn't have side effects in an expression containing forward references.

If you are now totally confused, the only thing you need remember is: **Don't ever use a call to a function that has side effects in an expression containing a forward reference.**

Appendix A — Macross 6502 Grammar

```
program:
            ⟦ statement Newline ⟧* Endfile

statement:
⟦ label ⟧*  opcode ⟦ operand ⟦ , operand ⟧* ⟧
⟦ label ⟧*  if ( condition ) block
                ⟦ elseif ( condition ) block ⟧*
                ⟦ else block ⟧
⟦ label ⟧*  while ( condition ) block
⟦ label ⟧*  do block while ( condition )
⟦ label ⟧*  do block until ( condition )
            dataStatement
            define identifier ⟦ = expression ⟧
            variable identifier ⟦ = expression ⟧
            macro identifier ⟦ identifier ⟦ , identifier ⟧* ⟧ block
            function identifier ( ⟦ identifier ⟦ , identifier ⟧* ⟧ ) block
            undefine identifier ⟦ , identifier ⟧*
⟦ label ⟧*  block
            mdefine identifier ⟦ = expression ⟧
            mif ( expression ) block
                ⟦ melseif ( expression ) block ⟧*
                ⟦ melse block ⟧
            mwhile ( expression ) block
            mdo block while ( expression )
            mdo block until ( expression )
            freturn ⟦ expression ⟧
            mfor ( expression , expression , expression ) block
            mswitch ( selectionExpression ) {
                ⟦ mcase ( expression ⟦ , expression ⟧* ) block ⟧*
                ⟦ mdefault block ⟧
            }
            constrain ( expression ) block
            assert ( expression ) ⟦ expression ⟧
            include textString
            extern identifier ⟦ , identifier ⟧*
            start expression
            org expression
            target expression
            expression

dataStatement:
⟦ label ⟧*  block expression ⟦ , expression ⟧*
⟦ label ⟧*  align expression
⟦ label ⟧*  word expression ⟦ , expression ⟧*
⟦ label ⟧*  long expression ⟦ , expression ⟧*
⟦ label ⟧*  dbyte expression ⟦ , expression ⟧*
⟦ label ⟧*  byte expression ⟦ , expression ⟧*
⟦ label ⟧*  string expression ⟦ , expression ⟧*
⟦ label ⟧*  struct { ⟦ dataStatement ⟧* } identifier
⟦ label ⟧*  struct identifier

label:      identifier :
```

operand:

expression
@ *expression*
# *expression*
a
x
y
x [*expression*]
x . *identifier* ⟦ . *identifier* ⟧*
y [*expression*]
y . *identifier* ⟦ . *identifier* ⟧*
@ x [*expression*]
@ x . *identifier* ⟦ . *identifier* ⟧*
y [@ *expression*]
textString

block: { ⟦ *statement* Newline ⟧* }

textString:

" *any string you like* "

condition:

conditionCode
! *conditionCode*

expression:

identifier
identifier (⟦ *operand* ⟦ , *operand* ⟧* ⟧)
number
here
textString
(*expression*)
− *expression*
! *expression*
~ *expression*
? *expression*
/ *expression*
expression * *expression*
expression / *expression*
expression % *expression*
expression − *expression*
expression + *expression*
expression << *expression*
expression >> *expression*
expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*
expression == *expression*
expression != *expression*
expression & *expression*
expression | *expression*
expression ^ *expression*
expression && *expression*

expression `||` *expression*
expression `^^` *expression*
expression `.` *identifier*
identifier = *expression*
identifier += *expression*
identifier −= *expression*
identifier *= *expression*
identifier /= *expression*
identifier %= *expression*
identifier &= *expression*
identifier |= *expression*
identifier ^= *expression*
identifier <<= *expression*
identifier >>= *expression*
identifier ++
identifier −−
++ *identifier*
−− *identifier*

identifier:
⟦`a-zA-Z_`⟧⟦`a-zA-Z_0-9`⟧*

number:
decimalNumber
octalNumber
binaryNumber
hexadecimalNumber
quarter

decimalNumber:
⟦`1-9`⟧⟦`0-9`⟧*

octalNumber:
`0`⟦`0-7`⟧*

binaryNumber:
`0b`⟦`01`⟧⟦`01`⟧*

hexadecimalNumber:
`0x`⟦`0-9a-f`⟧⟦`0-9a-f`⟧*

quarter:
`0q`⟦`0-3`⟧⟦`0-3`⟧*

Appendix B — Condition Codes

(6502 version)

The *Macross* `if`, `while`, `do-while` and `do-until` statements make use of symbols denoting the hardware condition codes of the target processor which may be used as the conditions upon which conditional branches are base. In the 6502 version of *Macross*, these are the recognized condition code symbols:

Conditions which generate simple branches

`carry`	tests carry bit
`equal` `zero`	tests zero bit
`neq`	(equivalent to, e.g., `!equal`)
`minus` `negative`	tests negative bit
`plus` `positive`	(equivalent to, e.g., `!minus`)
`overflow`	tests overflow bit

Conditions which generate complex branches

1. lt — less than (valid after `cmp` or `sbc`)

For example,

```
        if (lt) {
            ...stuff...
        }
```

generates

```
        bcs temp
        ...stuff...
temp:
```

2. leq — less than or equal to (valid after `cmp` or `sbc`)

For example,

```
        if (leq) {
            ...stuff...
        }
```

generates

```
        beq temp1
        bcs temp2
temp1:
        ...stuff...
```

```
temp2:
```

3. geq — greater than or equal to (valid after `cmp` or `sbc`)

For example,

```
        if (geq) {
            ...stuff...
        }
```

generates

```
        bcc temp
        ...stuff...
temp:
```

4. gt — greater than (valid after `cmp` or `sbc`)

For example,

```
        if (gt) {
            ...stuff...
        }
```

generates

```
        bcc temp
        beq temp
        ...stuff...
temp:
```

5. slt — signed less than (valid after `sbc` only)

For example,

```
        if (slt) {
            ...stuff...
        }
```

generates

```
        bvs temp1
        bpl temp3
        bmi temp2
temp1:  bmi temp3
temp2:
        ...stuff...
temp3:
```

6. sleq — signed less than or equal to (valid after `sbc` only)

For example,

```
        if (sleq) {
            ...stuff...
        }
```

generates

```
        beq temp2
        bvs temp1
        bpl temp3
        bmi temp2
temp1:  bmi temp3
temp2:
        ...stuff...
temp3:
```

7. sgt — signed greater than (valid after `sbc` only)

For example,

```
        if (sgt) {
            ...stuff...
        }
```

generates

```
        beq temp3
        bvs temp1
        bmi temp3
        bpl temp2
temp1:  bpl temp3
temp2:
        ...stuff...
temp3:
```

8. sgeq — signed greater than or equal to (valid after `sbc` only)

For example,

```
        if (sgeq) {
            ...stuff...
        }
```

generates

```
        bvs temp1
        bmi temp3
        bpl temp2
temp1:  bpl temp3
temp2:
        ...stuff...
temp3:
```

Appendix C — Built-In Functions

Certain predefined built-in functions are supported by *Macross* for reasons of convenience or syntactic or semantic irregularity. They are:

`addressMode(`*operand*`)`

Returns a number whose value indicates which addressing mode *operand* represents *((define these values))*.

`apply(`*macname* ⟦ `,` *arg* ⟧* `)`

Assembles the macro whose name is specified by the `string` *macname* with the macro arguments (if any) given by the *arg*s.

`arrayLength(`*array*`)`

Returns the number of elements in the array *array*.

`atascii(`*string*`)`

Returns a string which is *string* with each character mapped through an ASCII to ATASCII (Atari's ASCII deviant character code) conversion table.

`atasciiColor(`*string*`,` *color*`)`

Returns a string which is *string* with each character mapped through the ASCII to ATASCII conversion table, and then the two-bit value specified by *color* OR'ed into the high order two bits of each character.

`isAbsoluteValue(`*operand*`)`

Returns `TRUE` if and only if *operand* is an absolute (i.e., non-relocatable) value, otherwise `FALSE`.

`isARegister(`*operand*`)`

Returns `TRUE` if and only if *operand* is `a` (i.e., the accumulator), otherwise `FALSE`.

`isBlock(`*operand*`)`

Returns `TRUE` if and only if *operand* is a block, otherwise `FALSE`.

`isBuiltInFunction(`*symbol*`)`

Returns `TRUE` if and only if *symbol* is a built-in function, otherwise `FALSE`.

`isConditionCode(`*operand*`)`

Returns `TRUE` if and only if *operand* is a condition code, otherwise `FALSE`.

`isDefined(`*symbol*`)`

Returns `TRUE` if and only if *symbol* has been defined, otherwise `FALSE`.

`isDirectMode(`*operand*`)`

Returns `TRUE` if and only if the address mode of *operand* is *direct*, otherwise `FALSE`.

`isExternal(`*symbol*`)`

Returns `TRUE` if and only if *symbol* is external (i.e., visible outside the file in which it is defined), otherwise `FALSE`.

`isField(`*symbol*`)`

Returns `TRUE` if and only if *symbol* is a field of a struct, otherwise `FALSE`.

`isFunction(`*symbol*`)`

Returns `TRUE` if and only if *symbol* is a user defined function, otherwise `FALSE`.

`isImmediateMode(`*operand*`)`

Returns `TRUE` if and only if the address mode of *operand* is *immediate*, otherwise `FALSE`.

`isIndexedMode(`*operand*`)`

Returns `TRUE` if and only if the address mode of *operand* is an indexed mode, otherwise `FALSE`.

`isIndirectMode(`*operand*`)`

Returns `TRUE` if and only if the address mode of *operand* is *indirect*, otherwise `FALSE`.

`isPostIndexedMode(`*operand*`)`

Returns `TRUE` if and only if the address mode of *operand* is *post-indexed*, otherwise `FALSE`.

`isPreIndexedMode(`*operand*`)`

Returns `TRUE` if and only if the address mode of *operand* is *pre-indexed*, otherwise `FALSE`.

`isRelocatableValue(`*operand*`)`

Returns `TRUE` if and only if *operand* is a relocatable value, otherwise `FALSE`.

`isString(`*operand*`)`

Returns `TRUE` if and only if *operand* is a string, otherwise `FALSE`.

`isStruct(`*symbol*`)`

Returns `TRUE` if and only if *symbol* is the name of a struct, otherwise `FALSE`.

`isSymbol(`*operand*`)`

Returns `TRUE` if and only if *operand* is a symbol (as opposed to an expression or a number, for example), otherwise `FALSE`.

`isXIndexedMode(`*operand*`)`

Returns `TRUE` if and only if the address mode of *operand* is *x-indexed*, otherwise `FALSE`.

`isXRegister(`*operand*`)`

Returns `TRUE` if and only if *operand* is `x`, otherwise `FALSE`.

`isYIndexedMode(`*operand*`)`

Returns `TRUE` if and only if the address mode of *operand* is *y-indexed*, otherwise `FALSE`.

`isYRegister(`*operand*`)`

Returns `TRUE` if and only if *operand* is `y`, otherwise `FALSE`.

`listingOff()`

If assembly listing has been enabled using the `-l` command line flag, turn listing off temporarily. Otherwise, no effect.

`listingOn()`

If assembly listing was turned off using the `listingOff()` function, turn it back on again. If listings have been globally disabled by not specifying the `-l` command line flag, this function has no effect. The `listingOff()` and `listingOn()` functions are intended to be used to together to control assembly listings of large programs. They can be used to suppress listing of large and uninteresting sections such as header files full of definitions of global values. These functions may nest: in effect `listingOff()` increments a counter and `listingOn()` decrements it. Only when the counter is zero (i.e., the number of `listingOn()`s matches the number of `listingOff()`s) does listing actually occur.

`makeArray(`*length* ⟦ `,` *element* ⟧* `)`

Creates an array of length *length* and returns it. Optionally fills the array with the values specified by the *element* expressions. If the number of *element*s given is greater than *length*, an error results.

`nthChar(`*string* ⟦ `,` *position* ⟧ `)`

Returns the *position*th character of the string *string* (position zero being the first character in the string). If *position* is omitted it defaults to zero. If *position* is greater than the length of *string*, an error results.

`printf(`*format* ⟦ `,` *arg* ⟧* `)`

A formatted print routine just like the Unix system subroutine of the same name.

`strcat(`*string1*`,` *string2*`)`

Returns a string which is the concatenation of the two operands, which must themselves be strings.

`strcmp(`*string1*`,` *string2*`)`

Returns a number which is less than, equal to, or greater than 0 depending upon whether *string1* is lexically less than, equal to, or greater than *string2*. The ASCII character set is used. The two operands, of course, must be strings.

`strcmplc(`*string1*`,` *string2*`)`

Essentially the same as `strcmp()` except that alphabetic characters are converted to lower case before being compared. The result is a case-independent string comparison. This is useful for comparing two identifier name strings to see if they represent the same symbols.

`strlen(`*string*`)`

Returns a number which is the length, in characters, of *string*.

`substr(`*string*`,` *startPos* ⟦ `,` *length* ⟧ `)`

Returns a substring of the string *string* starting from the character at start position *startPos* (counting the first character from 0) and continuing for *length* characters. If *startPos* is negative, the start position is counted from right to left (with the rightmost character position being indicated by -1) instead of the more usual left to right. If *length* is negative, *startPos* in essence denotes the end of the desired substring and *length* characters up to that position are returned. If *length* is omitted, the substring from *startPos* to the end of the string is returned, if *startPos* is positive, or to the beginning of the string, if *startPos* is negative. If any of the indices cause the substring bounds to go off the end of *string* an error results. For example,

```
substr("hello there", 6, 3)       yields "the"
substr("hello there", -8, 2)      yields "lo"
substr("hello there", 6, -3)      yields "o t"
substr("hello there", -8, -4)     yields "hell"
substr("hello there", 6)          yields "there"
substr("hello there", -7)         yields "hello"
```

`symbolDefine(`*string* ⟦ `,` *value* ⟧ `)`

Defines the symbol named by *string* (with optional value *value*) as if it had been defined with a `define` statement. For example:

```
symbolDefine(strcat("foon", "farm"), 47)
```

is equivalent to

```
define foonfarm = 47
```

`symbolLookup(`*string*`)`

A call to this function with a string operand is equivalent to a reference to the symbol that the string represents. For example,

```
and     symbolLookup("foo")
```

is equivalent to

```
and     foo
```

`symbolName` (*symbol*)

Returns a string which is the name of the symbol *symbol*. For example, `symbolName(foo)` would return `"foo"`. This can be used in conjunction with the `symbolLookup` function so that the following:

```
and     symbolLookup(strcat(symbolName(foo), "bar"))
```

is equivalent to

```
and     foobar
```

`symbolUsage` (*symbol*)

Returns a number whose value indicates what sort of symbol *symbol* is (i.e., label, function, struct field, etc.). *((define these values))*

`valueType` (*thing*)

Returns a number whose value indicates the type of *thing* (i.e., symbol, condition code, number, block, etc.). *((define these values))*

Appendix D — Operator Set

This appendix describes the (**C** derived) operators supported by *Macross*.

− *expression*	integer negation
! *expression*	logical negation (0 goes to 1, all other values go to 0)
~ *expression*	bitwise negation (ones complement)
? *expression*	high byte
/ *expression*	low byte
expression * *expression*	integer multiplication
expression / *expression*	integer division
expression % *expression*	integer modulus (remainder)
expression − *expression*	integer subtraction
expression + *expression*	integer addition
expression << *expression*	left shift
expression >> *expression*	right shift
expression < *expression*	less than
expression > *expression*	greater than
expression <= *expression*	less than or equal to
expression >= *expression*	greater than or equal to
expression == *expression*	equal to
expression != *expression*	not equal to
expression & *expression*	bitwise AND
expression | *expression*	bitwise OR
expression ^ *expression*	bitwise XOR
expression && *expression*	logical AND
expression || *expression*	logical OR
expression ^^ *expression*	logical XOR
expression **.** *identifier*	struct field selection
identifier = *expression*	assignment
identifier += *expression*	assignment with addition
identifier −= *expression*	assignment with subtraction
identifier *= *expression*	assignment with multiplication
identifier /= *expression*	assignment with division
identifier %= *expression*	assignment with modulus
identifier &= *expression*	assignment with AND
identifier |= *expression*	assignment with OR
identifier ^= *expression*	assignment with XOR
identifier <<= *expression*	assignment with left shift
identifier >>= *expression*	assignment with right shift
identifier ++	post-increment
identifier −−	post-decrement
++ *identifier*	pre-increment
−− *identifier*	pre-decrement

Appendix E — Character Escape Codes

Like **C**, *Macross* enables you to use the ‘‘\’’ character as an escape to embed quotation marks, formatting characters (such as newline) and other non-printing characters in character strings and character constants. The recognized codes are:

`\n`	newline
`\t`	horizontal tab
`\b`	backspace
`\r`	carriage return
`\f`	form feed
`\e`	escape
`\\`	backslash
`\'`	apostrophe
`\"`	quote
`\^`*c*	CONTROL-*c* (where *c* is any character).
`\`*ddd*	arbitrary byte (where *ddd* is one, two or three octal digits).

Appendix F — Recognized Opcode Mnemonics

(6502 version)

These are the 6502 opcode mnemonics recognized by *Macross*:

```
adc
and
asl
bcc
bcs
beq
bit
bmi
bne
bpl
brk
bvc
bvs
clc
cld
cli
clv
cmp
cpx
cpy
dec
dex
dey
eor
inc
inx
iny
jmp
jsr
lda
ldx
ldy
lsr
nop
ora
pha
php
pla
plp
rol
ror
rti
rts
sbc
sec
sei
sta
stx
sty
tax
tay
tsx
txa
txs
tya
```