

## OVERVIEW

All devices in the MAXQ family of microcontrollers share one of three central core designs. These designs, known as the Q10, Q20 and Q30 cores, give all MAXQ microcontrollers a shared core instruction set and architecture. This allows applications and tools created for one MAXQ microcontroller to be reused for other microcontrollers in the MAXQ family.

## MAXQ Development Tools

The following tools are provided as part of the MAX-IDE development environment. These tools may be used when writing assembly code for any MAXQ microcontroller.

- **MaxQAsm** is a general-purpose MAXQ assembler which can be used either automatically from MAX-IDE or manually from the command line. It includes support for all three MAXQ core types and can be used to generate machine code for any MAXQ microcontroller by means of configurable option switches and include files.
- **Macro** is a macro preprocessor which can be used either automatically from MAX-IDE or manually from the command line. This preprocessor works in combination with MaxQAsm to provide additional features including macros, equates, C-style redefinitions, and constants.

## References

This guide covers the use of the standard MAX-IDE assembler and preprocessor to develop applications for MAXQ microcontrollers, as well as issues which commonly arise when writing MAXQ assembly code. Additionally, the following references should be used when developing applications for any MAXQ microcontroller.

- The *MAXQ Family User's Guide* (<http://www.maxim-ic.com/MAXQUG>) contains detailed information on the common features shared by MAXQ microcontrollers, including the MAXQ instruction set itself. It also includes information on the system registers (which are used to control core functions) as well as information on peripherals commonly found on MAXQ microcontrollers, which include timers, port pins, serial USART and SPI interfaces, hardware multipliers, and others.
- The functionality specific to a particular MAXQ microcontroller is detailed by two documents which are available from the QuickView page for that device – the datasheet and the User's Guide supplement. The datasheet provides an overview of functionality and describes the device's pinout, packaging, electrical characteristics and timing requirements. The User's Guide supplement describes the device from a programming standpoint including memory architecture, additional peripherals and peripheral registers, while also listing any differences between the device's functionality and the generic functionality described in the *MAXQ Family User's Guide*.
- The latest version of this guide is available online at:  
<http://files.dalsemi.com/microcontroller/maxq/information/MaxQCoreAsm.pdf>
- The *MaxQ Development Tools Guide*, which discusses features and usage of the MAX-IDE development environment and the Microcontroller Tool Kit (MTK), is available online at:  
<http://files.dalsemi.com/microcontroller/maxq/information/MaxQDevTools.pdf>
- Finally, on-line help for MAX-IDE itself is available from the Help menu under **Help → Contents**.

## USING THE MAXQ ASSEMBLER

The MAXQ assembler, MaxQAsm, is a multi-pass assembler which can be used to generate machine code for any MAXQ microcontroller based on the Q10, Q20 or Q30 core. It may be used either automatically as part of the MAX-IDE environment or directly from the Windows command line.

### Invoking MaxQAsm from the Command Line

Running MaxQAsm from the command line with no parameters will display version information and list command line options.

```
D:\MaxQAsm>maxqasm
```

```
Dallas Semiconductor/MAXIM MaxQ Assembler, Version 2.047.0060
```

```
Last build Mar  2 2007 at 06:00:57
```

```
Error, No file specified
```

```
Usage: maxqasm [options] <filename>
```

```
options
```

```
-l          generate list file (default: no list file)
-q          quiet mode (default: verbose)
-b          allow DB statements (default: disallow in MAXQ20)
-w          wide data (MAXQ20) mode (default: MAXQ10)
-w32       wide data with 32 bit Accumulator (MAXQ30)
-acc=<n>    accumulators supported, n (default:16)
-h          standard hex mode (default: hex386 mode)
```

The <filename> is the name of the MAXQ assembly file to be processed; only one file name may be specified on the command line. The assembly file must be in plain text ASCII format and may have any extension (".asm" is typically used, or ".mpp" if the file has already been processed by Macro, but neither of these are required). The extension must be provided on the command line as part of the file name; ".asm" will not be assumed as a default.

A number of command line options may be specified to modify the default behavior of MaxQAsm; these are detailed below.

**Table 1. MaxQAsm Command Line Options**

Option	Function
-l	If this option is included, an assembler listing file will be generated along with the output .hex file. The default behavior is not to generate a list file.
-q	If this option is included, the assembler will run silently, generating no output on the command line for a successful assembly run. The default behavior is to output the version header and usage information as shown above. Any error messages will be output whether or not this option is included.
-b	By default, DB statements are not allowed to avoid word alignment issues. Including this option allows DB statements to be used.
-w	If this option is included, assembly will be targeted for a Q20 type core. Default behavior is to target assembly for a Q10 type core.
-w32	This option turns on additional features (including 32-bit wide accumulators) to target the Q30 core.
-acc=8 -acc=16 -acc=32	Any of these options may be specified to define how many accumulators the target device has. For "-acc=8", registers A[0] through A[7] will be recognized, for "-acc=16", A[0] through A[15] will be recognized, and so on. The default mode is 16 accumulator registers.
-h	If this option is included, the output file will be generated in standard Intel hex format. Default behavior is to generate hex-386 output files which include the extended linear address records as needed.

## Invoking MaxQAsm from Within MAX-IDE

When a project is compiled from MAX-IDE, all assembly files listed in that project's file window are first preprocessed individually. After preprocessing is complete, all of the resulting .mpp files are concatenated together in the order that their source files appeared in the project file window and assembled as a single file.

The following command line options are set when MaxQAsm is invoked from MAX-IDE.

- `-l` is used to generate a merged listing file for all assembled project code.
- `-q` is used to suppress unneeded header and version output.
- `-b` is used to allow DB statements to be used.
- `-w` is used if the processor is a Q20 type.
- `-w32` is used if the processor is a Q30 type.
- `-acc=8` or `-acc=32` is used if necessary, if the processor has more or less than the default of 16 accumulators.

The processor family type (Q10, Q20 or Q30) as well as the number of accumulators are defined in the device configuration file assigned to the MAX-IDE project. The device configuration files are installed by default with MAX-IDE and are located in the \Devices\MaxQ subdirectory under the MAX-IDE installation directory. Each device type has a specific configuration file (for example, the configuration file for the MAXQ2000 is MaxQ2000.cfg) which must be used for MAX-IDE to operate correctly when loading and debugging that device. The device configuration file for a MAX-IDE project may be set by selecting **Device → Options** from the menu.

Since all source files in a project are merged together before being assembled, there is no need to export identifiers from one file for use in another. All files in a project share a common namespace for identifiers. However, since files are run through Macro for preprocessing individually, each source file must include any preprocessing directives (equates, defines, macros) that are to be used in that file. Typically, this is done by keeping any preprocessing directives that are to be shared by more than one source file in a common include file, which can then be included by any source file that requires it.

## MaxQAsm Output Files

The MaxQAsm assembler generates a hex file, a list file or both each time it is run. The list file will only be generated if the `-l` command line option is used, and the hex file will only be generated if no errors occur.

### List File

The list file, which can be generated by including the `-l` command line option, is written to `<basename>.lst`, where `<basename>` is the name of the assembly source file without the extension.

As an example, consider the following MAXQ assembly source file, sample.asm.

```
segment code
org 0000h

    move    GR, #01234h
    ljump   $

segment data
org 0000h

    db      01h, 02h, 03h, 04h
    db      "01234567"
    dw      0ABCDh

end
```

If assembled with "maxqasm -l -b", the list file generated for this file (sample.lst) would be as follows.

```
+-----+
| Dallas Semiconductor/MAXIM MaxQ Assembler, Version 2.047.0060 |
+-----+
Last build Mar  2 2007 at 06:00:57

Command line: -l -b sample.asm
Assembling sample.asm for MAXQ10, using 16 accumulators, Fri Mar 02 11:25:32 2007
```

```
Line: Addr: Opcode
=====
1:
2:          segment code
3:          org 0000h
4:
5: 0000: 0B12
5: 0001: 5E34      move    GR, #01234h
6: 0002: 0B00
6: 0003: 0C02      ljump  $
7:
8:          segment data
9:          org 0000h
10:
11: 0000: 0102      db     01h, 02h, 03h, 04h
        0304
12: 0002: 3031      db     "01234567"
        3233
        3435
        3637
13: 0006: ABCD      dw     0ABCDh
14:
15:          end
Parser terminated successfully.
```

Note that the list file contains all of the original source code as well as the machine code generated for each source code line. Both of the instructions in this source file require prefixing, and so two instruction words are generated for each of them.

## Hex Files

If the assembly completes without any errors, the resulting machine code will be written to <basename>.hex, where <basename> is the original source file name minus its extension. Additionally, if any of the source file was assembled to a data segment, an additional hex file will be generated as <basename>\_d.hex, containing the data segment code. See the *Code and Data Segments* section for more details.

For the sample.asm file shown above, two hex files are output (sample.hex and sample\_d.hex) as follows.

### sample.hex

```
:020000040000FA
:08000000120B345E000B020C30
:00000001FF
```

### sample\_d.hex

```
:020000040000FA
:0E000000010203043031323334353637CDABD4
:00000001FF
```

## Hex File Formatting

The hex files generated by MaxQAsm are written in Intel hex file format. This is a standardized, ASCII-based data record format used by many compilers and assemblers.

Each record in the hex file occupies one line, beginning with a colon character “:” and ending with a carriage return or line feed. Following the colon are five fields which are encoded in hexadecimal ASCII. Each byte is written most significant character first.

**Table 2. Hex File Record Fields**

Field	# of Characters	Function
Length	2	The number of bytes (not characters) in the Data field for this record. For type 0 records – Any value. For type 1 records – 00. For type 4 records – 02.
Address	4	For type 0 records – The starting address that the Data field should be loaded at. For type 1 records – Not used (0000). For type 4 records – Not used (0000).
Record Type	2	For MAXQ hex files, three different record types are used.  00 – Data Record <ul style="list-style-type: none"> <li>This type of record represents data to be loaded into the program or data memory of the MAXQ microcontroller.</li> </ul> 01 – End of File Record <ul style="list-style-type: none"> <li>This type of record indicates the end of the hex file.</li> </ul> 04 – Extended Linear Address Record <ul style="list-style-type: none"> <li>This type of record is used to set the upper 16 bits of the current address.</li> <li>If the “-h” command line switch is used, these records will not be generated.</li> </ul>
Data	2 * Length	For type 0 records – The data to be loaded into program or data memory. For type 1 records – Empty field. For type 4 records – The 4-digit upper address value.
Checksum	2	The two’s complement checksum (1 + NOT(Length + Address + Type + [Data...]) of all bytes in the record.

The sample.hex file shown above has one example of each record type.

```

:020000040000FA
02          -- Length   : 2 bytes (4 characters)
 0000      -- Address  : N/A
    04      -- Type    : Extended Address
      0000  -- Data    : Sets high 16 bits of address to 0000h
        FA  -- Checksum : 1+NOT(02+00+00+04+00+00)

:08000000120B345E000B020C30
08          -- Length   : 8 bytes (16 characters)
 0000      -- Address  : 0000h
    00      -- Type    : Data
      120B345E000B020C -- Data    : Bytes to be loaded starting at 0000h
        30  -- Checksum : 1+NOT(08+00+00+00+12+0B+34+5E+00+0B+02+0C)

:00000001FF
00          -- Length   : 0 bytes
 0000      -- Address  : N/A
    01      -- Type    : End of File
      FF  -- Checksum : 1+NOT(00+00+00+01)

```

Note that since MAXQ word-wide memories, when accessed in byte mode, are little-endian, the hex file generated writes the least significant byte of each instruction word (or DW) first. So, the first instruction in the file (0B12h) is inserted into the hex file as the bytes 12h, 0Bh.

For the Q10 and Q20 cores, the address space is limited to 16 bits total, so the only extended address record which will be generated in default mode (without the `-h` switch) will be the initial record as shown above to set the extended address to 0000h.

## General Syntax

Statements processed by MaxQAsm must follow this format, which consists of up to five fields. A statement may consist of one of the following:

- Label, opcode (with parameters, if any) and comment.
- Label, opcode (with parameters, if any).
- Opcode (with parameters, if any) and comment.
- Opcode (with parameters, if any).
- Label and comment.
- Label only.
- Comment only.
- Blank line.
- END directive (must be the final line in the file).

Except for the END directive, statement lines are laid out as follows.

```
[label:] [opcode [parameter [, parameter]]] [;comment]
```

The **label** is optional. If included, it is set equal to the current assembly word address. Labels and other identifiers may be from 1 to 127 characters in length and may consist of the following characters:

- Alphabetic characters “A” to “Z” or “a” to “z”. Identifiers in MaxQAsm are case-insensitive, that is, “LABEL1” and “label1” are considered equivalent.
- Numeric characters “0” to “9”. However, a numeric character may not be the first character in an identifier (such as “0LABEL”).
- The characters “\_” (underbar), “?” (question mark), and “\$” (dollar sign).

The **opcode**, if included, must be one of the standard MAXQ assembly opcodes as defined in the *MAXQ Family User's Guide*. Opcodes in MaxQAsm are also, case-insensitive, so “addc”, “AddC” and “ADDC” are all equivalent. Each opcode has from zero to two **parameters**.

If the semicolon character “;” is included in the statement line, all text following this character is treated as a **comment** and will be ignored by both Macro and MaxQAsm. Comments are limited to a single line only.

The final line in each assembly source file must consist of the END directive, as follows.

```
end
```

## Global and Local Labels

In MAXQ assembly, **labels** mark a point in the assembly code which can be used as a destination for a JUMP or CALL or as a reference point when reading data from code space. There are two types of labels which can be used – global and local.

**Global** labels are defined as described above under “General Syntax”, and consist of an identifier followed by a colon (“:”). This type of label has global scope, which means that it can be referenced from any point at any file in the assembly project.

```

    call    Sub1

Sub1:
    push   Acc
    add    #1
    ret

```

Global labels are useful when defining entry points for subroutines and reference points for global strings and table data, since they allow the labels to be referenced from anywhere in the project. However, in some cases it is more useful for a label to have a more limited scope. For example, subroutines often contain labels in order to implement loops or other control structures within the subroutine. In general, these labels are for the use of the subroutine only and are not intended to be used by code outside the subroutine, but with global labels, there is no way to prevent such external use. Additionally, care must be taken when defining labels within subroutines to keep the names unique and avoid collisions with labels defined inside other subroutines, which often leads to code such as the following.

```

transmitBytes:
    move   LC[1], #10           ; Number of bytes to transmit
    move   Acc, #07h
transmitBytes_loop:           ; Make sure this name is unique
    call   sendNextByte
    djnz  LC[1], transmitBytes_loop

```

**Local** labels solve both of these problems by providing a way to define labels that are only visible within a subroutine or other limited scope section. These labels are defined in the same way as global labels, except they always start with a period (“.”). Local labels operate exactly the same as global labels, except for their scope. Instead of being visible throughout the entire project, the scope of local labels is restricted to the space between one global label and the next. This allows labels to be defined which are only accessible within a subroutine or other code block and have no potential for colliding with labels defined elsewhere.

**Note:** In order to use local labels, you must be running maxqasm version 2.042 or later, since this feature was not defined in previous versions.

```

    call    sub1
    call    sub2

sub1:
    move   LC[1], #10
.loopA:
    add    #1
    djnz  LC[1], .loopA    ; Jumps to .loopA within sub1
    ret

sub2:
    move   LC[1], #10
.loopA:
    add    #1
    djnz  LC[1], .loopA    ; Jumps to .loopA within sub2
    ret

next:
    nop

```

In the example code shown here, the `.loopA` label is only accessible to code between global labels `sub1` and `sub2` (in which case `.loopA` refers to `sub1.loopA`) or to code between global labels `sub2` and `next` (in which case `.loopA` refers to `sub2.loopA`). Attempting to reference `.loopA` from elsewhere in the project will result in an assembler error.

Local labels are an optional feature, but they help in keeping code cleaner and enforcing defined entry points for subroutines. Another potential use lies in declaring variables in data space whose labels are only visible to certain subroutines or blocks of code.

```

segment code
sub1:
    move   LC[1], #10
    move   DP[0], #.data1    ; Points to sub1.data1
.loopA:
    add    #1
    djnz  LC[1], .loopA
    ret

segment data
.data1:
    dw    0

segment code
sub2:
    move   LC[1], #10
    move   DP[0], #.data1    ; Points to sub2.data1
.loopA:
    add    #1
    djnz  LC[1], .loopA
    ret

segment data
.data1:
    dw    0

```

## Constants

Constant numeric values are used as immediate values in statements, macro calls and equate definitions. They may be written in one of four formats.

- Decimal (default) – consists of the characters “0” to “9” and ends with no character or, optionally, with “d” or “D”. Examples: 10, 07d, 99D.
- Binary – consists of the characters “0” to “1” and must end with either “b” or “B”. Examples: 01b, 111B.
- Hexadecimal – must begin with a numeric character “0” to “9”. The remaining characters may consist of “0” to “9”, “A” to “F” or “a” to “f” (case insensitive) and the constant must end with “h” or “H”. Examples: 10h, 0FFH.
- Single character – if a single character is written in single quotes, it will be converted to a byte ASCII value. Example: 'A'.

Labels are also considered constants, and may be used anywhere a constant would normally be used, except for DB and DW statements. In addition, the special label “\$” represents the current assembly location, so that

```
L1:
    move    GR, #L1
```

and

```
    move    GR, $
```

are equivalent.

## MaxQAsm Constant Operations

MaxQAsm supports a number of constant operations which may be processed following the macro stage of assembly. This is important because labels and the symbol '\$' have no value until the actual assembly is performed by MaxQAsm. So, for example, the line

```
    move    DP[0], #(LABEL * 2)
```

must be processed by MaxQAsm since LABEL is undefined at the macro preprocessing stage. Constant operations supported by MaxQAsm are listed in Table 3.

**Table 3. MaxQAsm Constant Operators**

Operator	Format	Description
+	$x + y$	Returns the integer sum of $x$ and $y$ .
-	$x - y$	Returns the integer difference of $x$ and $y$ .
-	$-x$	Unary minus; returns the two's complement negation of $x$ .
*	$x * y$	Returns the integer product of $x$ and $y$ .
/	$x / y$	Returns the value of $x$ divided by $y$ , with any remainder discarded (truncated to an integer).
%	$x \% y$	Returns the modulus of $x$ divided by $y$ .
	$x   y$	Returns the bitwise OR value of $x$ and $y$ .
&	$x \& y$	Returns the bitwise AND value of $x$ and $y$ .
<<	$x \ll y$	Returns the value of $x$ shifted left by $y$ bits, with the low $y$ bits filled with 0. The return value is undefined for values of $y$ greater than 32 or less than 0.
>>	$x \gg y$	Returns the value of $x$ shifted right by $y$ bits, with the high $y$ bits filled with 0. The return value is undefined for values of $y$ greater than 32 or less than 0.

Example operations using MaxQAsm's constant operators with immediate constant values, label address values, and the current address (\$) operator are shown below.

```

+-----+
| Dallas Semiconductor/MAXIM MaxQ Assembler, Version 2.047.0060 |
+-----+
Last build Mar  2 2007 at 06:00:57

```

Command line: -l -q -b -w asmconst.asm

Assembling asmconst.asm for MAXQ20, using 16 accumulators, Fri Mar 02 15:14:05 2007

```

Line: Addr: Opcode
=====
1:
2:          org 0
3:
4:          main:
5: 0000: 0A02      move    Acc, #1 + 1      ; 1 + 1      = 2
6: 0001: 0A04      move    Acc, #9 - 5     ; 9 - 5     = 4
7: 0002: 0A06      move    Acc, #2 * 3     ; 2 * 3     = 6
8: 0003: 0A40      move    Acc, #80h / 2   ; 80h / 2   = 40h
9: 0004: 0A03      move    Acc, #21 / 7    ; 21 / 7    = 3
10: 0005: 0A01      move    Acc, #21 % 4    ; 21 % 4    = 1
11: 0006: 0BFF
11: 0007: 0AFF      move    Acc, #-1        ; -1         = FFFFh
12: 0008: 0BFF
12: 0009: 0AAA      move    Acc, #~55h     ; ~55h      = FFAAh
13: 000A: 0B55
13: 000B: 0A77      move    Acc, #5555h | 22h ; 5555h | 22h = 5577h
14: 000C: 0A07      move    Acc, #7777h & 0Fh ; 7777h & 0Fh = 7770h
15: 000D: 0B05
15: 000E: 0A00      move    Acc, #0005h << 8 ; 0005h << 8 = 0500h
16: 000F: 0A01      move    Acc, #8000h >> 15 ; 8000h >> 15 = 0001h
17:
18: 0010: 3F44      move    DP[0], #label1
19: 0011: 3F45      move    DP[0], #label1 + 1
20: 0012: 3F88      move    DP[0], #label1 * 2
21: 0013: 0B80
21: 0014: 3F88      move    DP[0], #(label1 * 2) + 8000h
22: 0015: 0BA0
22: 0016: 3F02      move    DP[0], #(label2 * 2) + 8000h
23:
24:          org 0044h
25:
26:          label1:
27: 0044: DA3A      nop
28:
29:          org 0080h
30:
31: 0080: 3F80      move    DP[0], $
32: 0081: 3F82      move    DP[0], $ + 1
33: 0082: 3F72      move    DP[0], $ - 10h
34: 0083: 0B80
34: 0084: 3F83      move    DP[0], $ | 8000h
35: 0085: 3F05      move    DP[0], $ & 0007h
36:
37:          org 1001h
38:
39:          label2:
40: 1001: DA3A      nop

```

## Predefined Registers

Except for opcode/register combinations which are explicitly prohibited in the instruction set (such as “AND Acc” or “MOVE SP, @SP--”), any MAXQ register may be used as the *src* or *dst* field in any instruction. A number of system registers and pseudoregisters are predefined by MaxQAsm for use with any MAXQ microcontroller project.

For more information on the functions of the system registers listed here, refer to the *MAXQ Family User's Guide* and to the User's Guide Supplement for the specific MAXQ microcontroller you are using.

**Table 4A. Predefined Registers in MaxQAsm (Q10 and Q20 Cores, default or -w)**

Name	Conditions	Function
AP		Accumulator Pointer Register.
APC		Accumulator Pointer Control Register.
PSF		Processor Status Flags Register.
IC		Interrupt Control Register.
IMR		Interrupt Mask Register.
SC		System Control Register.
IIR		Interrupt Identification Register.
CKCN		System Clock Control Register.
WDCN		Watchdog Control Register.
A[0]		Accumulator 0 – 8 bits wide for Q10, 16 bits wide for Q20.
A[1]		Accumulator 1 – 8 bits wide for Q10, 16 bits wide for Q20.
A[2]		Accumulator 2 – 8 bits wide for Q10, 16 bits wide for Q20.
A[3]		Accumulator 3 – 8 bits wide for Q10, 16 bits wide for Q20.
A[4]		Accumulator 4 – 8 bits wide for Q10, 16 bits wide for Q20.
A[5]		Accumulator 5 – 8 bits wide for Q10, 16 bits wide for Q20.
A[6]		Accumulator 6 – 8 bits wide for Q10, 16 bits wide for Q20.
A[7]		Accumulator 7 – 8 bits wide for Q10, 16 bits wide for Q20.
A[8]	-acc=16	Accumulator 8 – 8 bits wide for Q10, 16 bits wide for Q20.
A[9]	-acc=16	Accumulator 9 – 8 bits wide for Q10, 16 bits wide for Q20.
A[10]	-acc=16	Accumulator 10 – 8 bits wide for Q10, 16 bits wide for Q20.
A[11]	-acc=16	Accumulator 11 – 8 bits wide for Q10, 16 bits wide for Q20.
A[12]	-acc=16	Accumulator 12 – 8 bits wide for Q10, 16 bits wide for Q20.
A[13]	-acc=16	Accumulator 13 – 8 bits wide for Q10, 16 bits wide for Q20.
A[14]	-acc=16	Accumulator 14 – 8 bits wide for Q10, 16 bits wide for Q20.
A[15]	-acc=16	Accumulator 15 – 8 bits wide for Q10, 16 bits wide for Q20.
Acc		Active accumulator.
A[AP]		Active accumulator, same as Acc except that auto inc/dec is disabled.
PFX[n]		n = 0 to 7. Prefix Register.
IP		Instruction Pointer.
SP		Stack pointer.
@++SP		<b>Destination only.</b> Preincrements SP and writes the source value to the stack.
@SP--		<b>Source only.</b> Returns the value at the top of the stack and then decrements SP.
IV		Interrupt Vector Register.
LC[0]		Loop Counter 0.
LC[1]		Loop Counter 1.
Of fs		Offset portion of the Frame Pointer.
DPC		Data Pointer Control Register.

Name	Conditions	Function
GR		16-bit General Purpose Register.
GRL		Low byte of GR.
BP		Base portion of the Frame Pointer.
GRS		Byte-swapped version of GR.
GRH		High byte of GR.
GRXL		Sign-extended version of the low byte of GR.
BP [Offs]		Frame pointer value – BP + Offs.
@BP [Offs]		Returns the value stored in memory location BP + Offs.
@BP [Offs--]		<b>Source only.</b> Returns the value stored in memory location BP + Offs, then decrements Offs.
@BP [++Offs]		<b>Destination only.</b> Increments Offs, then writes the source value to memory location BP + Offs.
DP [0]		Data Pointer 0.
@DP [0]		Reads from or writes to memory location at DP[0].
@++DP [0]		<b>Destination only.</b> Increments DP[0], then writes the source value to the memory location at DP[0].
@DP [0]++		<b>Source only.</b> Returns the value stored in memory location DP[0], then increments DP[0].
@--DP [0]		<b>Destination only.</b> Decemets DP[0], then writes the source value to the memory location at DP[0].
@DP [0]--		<b>Source only.</b> Returns the value stored in memory location DP[0], then decrements DP[0].
DP [1]		Data Pointer 1.
@DP [1]		Reads from or writes to memory location at DP[1].
@++DP [1]		<b>Destination only.</b> Increments DP[1], then writes the source value to the memory location at DP[1].
@DP [1]++		<b>Source only.</b> Returns the value stored in memory location DP[1], then increments DP[1].
@--DP [1]		<b>Destination only.</b> Decemets DP[1], then writes the source value to the memory location at DP[1].
@DP [1]--		<b>Source only.</b> Returns the value stored in memory location DP[1], then decrements DP[1].

**Table 4B. Predefined Registers in MaxQAsm (Q30 Core Only, -w32)**

Name	Conditions	Function
AP		Accumulator Pointer Register.
APC		Accumulator Pointer Control Register.
NUL		Bit bucket register; reads from this register return zero and writes to this register are a no-op.
PSF		Processor Status Flags Register.
IC		Interrupt Control Register.
SC		System Control Register.
IPR0		Interrupt Priority 0 Register.
IPR1		Interrupt Priority 1 Register.
CKCN		System Clock Control Register.
WDCN		Watchdog Control Register.
A[0]		Accumulator 0 – 32 bits wide.
A[1]		Accumulator 1 – 32 bits wide.
A[2]		Accumulator 2 – 32 bits wide.
A[3]		Accumulator 3 – 32 bits wide.
A[4]		Accumulator 4 – 32 bits wide.
A[5]		Accumulator 5 – 32 bits wide.
A[6]		Accumulator 6 – 32 bits wide.
A[7]		Accumulator 7 – 32 bits wide.
A[8]	-acc=16	Accumulator 8 – 32 bits wide.
A[9]	-acc=16	Accumulator 9 – 32 bits wide.
A[10]	-acc=16	Accumulator 10 – 32 bits wide.
A[11]	-acc=16	Accumulator 11 – 32 bits wide.
A[12]	-acc=16	Accumulator 12 – 32 bits wide.
A[13]	-acc=16	Accumulator 13 – 32 bits wide.
A[14]	-acc=16	Accumulator 14 – 32 bits wide.
A[15]	-acc=16	Accumulator 15 – 32 bits wide.
Acc		Active accumulator.
A[AP]		Active accumulator, same as Acc except that auto inc/dec is disabled.
PFX[n]		n = 0 to 7. Prefix Register.
IP		Instruction Pointer.
SP		Stack pointer.
@--SP		<b>Destination only.</b> Predecrements SP and writes the source value to the stack.
@SP++		<b>Source only.</b> Returns the value at the top of the stack and then increments SP.
LC[0]		Loop Counter 0.
LC[1]		Loop Counter 1.
Offs		Offset portion of the Frame Pointer.
BP		Base portion of the Frame Pointer.
BP[Offs].B BP[Offs].W BP[Offs].L		<b>Source only.</b> Pointer value – BP[Offs]. Value is (BP + Offs) for default and byte mode (.B), BP + (Offs x 2) for word mode (.W), and BP + (Offs x 4) for long-word mode (.L).
@BP[Offs].B @BP[Offs].W @BP[Offs].L		Returns the value stored in memory location (BP + Offs) for byte mode, (BP + (Offs * 2)) for word mode, and (BP + (Offs * 4)) for long mode..
@BP[Offs--].B @BP[Offs--].W @BP[Offs--].L		Writes or returns the value stored in memory location BP[Offs], then decrements Offs.

Name	Conditions	Function
@BP[Offs++] .B @BP[Offs++] .W @BP[Offs++] .L		Writes or returns the value stored in memory location BP[Offs / Offs * 2 / Offs * 4], then increments Offs.
DP[0]		Data Pointer 0.
@DP[0] .B @DP[0] .W @DP[0] .L		Reads from or writes to the memory location (byte / word / long) at DP[0].
@DP[0] .B++ @DP[0] .W++ @DP[0] .L++		Reads from or writes to the memory location at DP[0], then increments DP[0] by 1 (.B), 2 (.W) or 4 (.L).
@DP[0] .B-- @DP[0] .W-- @DP[0] .L--		Reads from or writes to memory location at DP[0], then decrements DP[0] by 1 (.B), 2 (.W) or 4 (.L).
DP[1]		Data Pointer 1.
@DP[1] .B @DP[1] .W @DP[1] .L		Reads from or writes to memory location (byte / word / long) at DP[1].
@DP[1] .B++ @DP[1] .W++ @DP[1] .L++		Reads from or writes to memory location at DP[1], then increments DP[1] by 1 (.B), 2 (.W) or 4 (.L).
@DP[1] .B-- @DP[1] .W-- @DP[1] .L--		Reads from or writes to memory location at DP[1], then decrements DP[1] by 1 (.B), 2 (.W) or 4 (.L).

Note that for registers with indexes such as the prefixes, accumulators, loop counters or data pointers, the index may be specified in any constant numeric format. So the eleventh accumulator may be referred to as A[10], A[0Ah], or even A[01010b].

## Predefined Register Bits

Although registers in module 8 (AP, APC, PSF, IC, IMR, SC, IIR, IPR0, IPR1, CKCN and WDCN) are bit addressable, names of the bits in these registers are not predefined in MaxQAsm. Except where these bits are explicitly defined as part of an opcode (such as “move C, #0”), they must be referred to in terms of their parent register.

So while the Carry bit may be cleared by the instruction

```
move C, #0
```

the Equal bit must be cleared by the instruction

```
move PSF.0, #0
```

## Data Byte (DB) Statement

The DB statement is used to insert bytes of data into the output hex file at the current assembly location. It can be used in either the code segment or the data segment, and the bytes are inserted in the order they are given.

The syntax for the DB statement takes the following form. The “DB” keyword is case-insensitive.

```
db  constant [, constant [, constant ...]]
```

The following three types of values may be used as DB constants, and more than one type may be mixed in a DB statement.

- Byte constant values (written in decimal, hex or binary) from 0 to 255. Negative values may not be used.
- Character constant values (written as '<char>')
- String constant values, written in double quotes. These must be at least one character long and may be multi-line.

```
+-----+
| Dallas Semiconductor/MAXIM MaxQ Assembler, Version 2.047.0060 |
+-----+
Last build Mar  2 2007 at 06:00:57
```

```
Command line: -l -q -b -w db.asm
Assembling db.asm for MAXQ20, using 16 accumulators, Fri Mar 02 15:31:34 2007
```

```
Line: Addr: Opcode
=====
1:
2:          org 0
3:
4: 0000: 00      db  0
5: 0001: 6565    db  101, 101, 255
      FF
6: 0003: 01FF    db  01b, 11111111b
7: 0004: 0055    db  00h, 55h, 0FFh
      FF
8:
9: 0006: 41      db  'A'
10: 0007: 4142   db  'A', 'B'
11:
12: 0008: 5374    db  "String", ".", 00h
      7269
      6E67
      2E00
13:
14: 000C: 415A    db  "A", "Z", ";#$"
      273B
      2324
```

Note that in order to use the DB statement, the “-b” command line switch must be included when invoking MaxQasm or an error will result. This command line switch is included automatically when compiling a project from MAX-IDE.

Since the DB statement may be used to generate either odd or even numbers of bytes in the hex file, it has the potential of introducing word-alignment issues. To avoid this, the assembler automatically ensures that the hex data output is padded to the next word boundary (if necessary) following each DB statement.

This means that the code

```
db 11h
db 22h
db 33h
```

will produce a total of six bytes: 11h, FFh (padding byte), 22h, FFh (padding byte), 33h, FFh (padding byte), while the code

```
db 11h, 22h, 33h
```

will only produce four bytes: 11h, 22h, 33h, FFh (padding byte), since the padding is performed following each line of the source code. The padding bytes always have the value FFh. For the code example, listed above, the hex output (with padding bytes underlined) would be:

```
:020000040000FA
:1000000000FF6565FFFF01FF0055FFFF41FF414213
:0E001000537472696E672E00415A273B2324F9
:00000001FF
```

Note that the padding bytes are not shown in the list file. The padding byte is always the most significant byte of the program memory word.

## Data Word (DW) Statement

The DW statement is similar to the DB statement, but is used to insert 16-bit words of data into the output hex file at the current assembly location. It can be used in either the code segment or the data segment.

The syntax for the DW statement takes the following form. The “DW” keyword is case-insensitive.

```
dw  constant [, constant [, constant ...]]
```

The following two types of values may be used as DW constants, and more than one type may be mixed in a DW statement.

- Word constant values (written in decimal, hex or binary) from 0 to 65535. Negative values may not be used. If a value smaller than 256 is given, the high byte will be padded to 00h.
- Character constant values (written as '<char>'). When these are used in a DW statement, the high byte is padded to 00h.

String constant values may not be used in DW statements. Since DW statements always generate even numbers of bytes, they may be mixed freely with instruction statements.

```
+-----+
| Dallas Semiconductor/MAXIM MaxQ Assembler, Version 2.047.0060 |
+-----+
Last build Mar  2 2007 at 06:00:57
```

```
Command line: -l -q -b -w dw.asm
Assembling dw.asm for MAXQ20, using 16 accumulators, Fri Mar 02 15:57:08 2007
```

```
Line: Addr: Opcode
=====
1:
2:          org 0
3:
4: 0000: 0000    dw    0, 10d
          000A
5: 0002: 0001    dw    01b, 1111111111111111b
          FFFF
6:
7: 0004: 0A00    move  Acc, #0
8:
9: 0005: FFFF    dw    0FFFFh
10: 0006: FFFF    dw    65535
11: 0007: 0041    dw    'A'
12:
13: 0008: DA3A    nop
14:
15:          end
```

Parser terminated successfully.

## Data Long/Double (DL/DD) Statement

The DL/DD statement is similar to the DW statement, but is used to insert 32-bit longwords of data into the output hex file at the current assembly location. It can be used in either the code segment or the data segment.

The syntax for the DL/DD statement takes the following form. The “DL/DD” keyword is case-insensitive.

```
dl  constant [, constant [, constant ...]]
dd  constant [, constant [, constant ...]]
```

The following two types of values may be used as DL/DD constants, and more than one type may be mixed in a DW statement.

- Word constant values (written in decimal, hex or binary) from 0 to 4294967295. Negative values may not be used. If necessary, the high byte(s) will be padded to zero.
- Character constant values (written as '<char>'). When these are used in a DL/DD statement, the 3 high bytes are padded to 00h.

String constant values may not be used in DL/DD statements. Since DL/DD statements always generate even numbers of bytes, they may be mixed freely with instruction statements.

```
+-----+
| Dallas Semiconductor/MAXIM MaxQ Assembler, Version 2.047.0060 |
+-----+
Last build Mar  2 2007 at 06:00:57
```

```
Command line: -l -q -b -w dl.asm
Assembling dl.asm for MAXQ20, using 16 accumulators, Fri Mar 02 16:03:44 2007
```

```
Line: Addr: Opcode
=====
1:
2:          org 0
3:
4: 0000: 0000    dl    0, 10d
          0000
          0000
          000A
5: 0004: 0000    dl    01b, 1111111111111111b
          0001
          0000
          FFFF
6:
7: 0008: 0A00    move  Acc, #0
8:
9: 0009: FFFF    dl    0FFFFFFFFh
          FFFF
10: 000B: FFFF    dl    4294967295
          FFFF
11: 000D: 0000    dl    'A'
          0041
12:
13: 000F: DA3A    nop
14:
15:          end
```

## The Assembler Message (\$MESSAGE) Statement

The \$MESSAGE statement is used to output a message along with the standard output (and to the list file) at assembly time. It does not affect the hex file output. This is a documentation statement, designed to indicate build versions, debug builds, and other information of that nature.

```
$message("This is a debug build!")
```

## The ORG Statement

The ORG statement is used to set the current assembly location to a word address value. The syntax is as follows.

```
org <address>
```

The address value may be given in decimal, hex or binary, and may be anywhere from zero to 0FFFFh. If extended address record generation (-h switch omitted) is enabled, the address may be anywhere from zero to 0FFFFFFh (16M x 16). (This type of addressing is used with the MAXQ30 core.)

Note that until a statement is encountered which actually generates a byte or word in the hex file (DB, DW or instruction), the ORG statement has no effect.

```
Dallas Semiconductor/MAXIM MaxQ Assembler, Version 1.018.0119
```

```
Last build May 26 2005 at 08:08:09
```

```
Assembling org.asm for MAXQ10,using 16 accumulators, Tue Aug 09 10:33:00 2005
```

```

Line: Addr: Opcode
-----
1:
2:          org  00000h
3:          org  0FFFFh
4:          org  00000h
5:
6: 0000: 0000      dw  0, 0, 0, 0, 0
          0000
          0000
          0000
          0000
7:
8:          org  256d
9:
10: 0100: 0000      dw  0, 0
          0000
11:
12:          org  10000b
13:
14: 0010: 0000      dw  0, 0
          0000
15:
16:          end

```

Although MaxQAsm will allow ORG statements to be used to set the address to any value in either the code or data segments, the actual physical ranges of memory that can be written to through the bootloader will vary from part to part. Refer to the User's Guide Supplement for the MAXQ microcontroller you are using for information on the boundaries of available code and data memory on that device.

## Code and Data Segments

Since MAXQ microcontrollers have separate code and data memory spaces, the exact meaning of a given numeric memory address will depend on whether it is a code address or data address. Since the ORG statement sets the numeric address only, MaxQAsm provides a pair of directives to differentiate between statements that should be assembled into code space and statements that should be assembled into data space.

```
segment code
segment data
```

The SEGMENT CODE statement indicates that all DB, DW, and instruction statements from this point on should be assembled into code space. Similarly, the SEGMENT DATA statement indicates that all statements from this point on should be assembled into data space. Each SEGMENT statement affects all DB, DW, and instruction statements from the point the SEGMENT statement appears until the next SEGMENT statement is reached.

The default mode is to assemble to code space, so if all statements in a source file are to be assembled into code space only, the SEGMENT CODE statement is optional. Note that the SEGMENT statements are assembler statements and not Macro directives, so when a MAX-IDE project is compiled, a SEGMENT statement in a source file will remain in effect for the rest of that source file and through all source files following it in the project window, or until another SEGMENT statement is reached.

Code segments may contain instruction statements and data statements (DB and DW). Data segments may contain data statements (DB and DW); including an instruction in a data segment will cause an assembler error. Note that data segments still operate with word addressing, so

```
segment data
org 0h

    db 01h
    dw 0000h
```

will cause a word-alignment error just as it would in a code segment.

Note that not all MAXQ bootloaders may support loading to data memory. In some instances, loading to data memory through the bootloader may only be supported for that portion of the data memory which is nonvolatile (implemented in EEPROM, flash memory or something similar). Even if loading to volatile data memory (SRAM) is supported, since the memory is volatile, it will only be loaded with the proper contents when the MAX-IDE project is initially executed. Once the MAXQ microcontroller is powered off and on or reset, the contents of the volatile data memory may no longer be the contents that were loaded from the hex file. Refer to the User's Guide Supplement for the specific MAXQ microcontroller you are using for more details.

Multiple SEGMENT statements may be used within a single source file, and it is possible to switch back and forth between code and data segments any number of times. The current assembly address is tracked separately for each segment, as shown in the list file below.

```
Dallas Semiconductor/MAXIM MaxQ Assembler, Version 1.018.0119
```

```
Last build May 26 2005 at 08:08:09
```

```
Assembling segment.asm for MAXQ10,using 16 accumulators, Tue Aug 09 11:48:44
```

```

Line: Addr: Opcode
=====
1:
2:          segment code
3:
4:          org 0010h
5:
6: 0010: 0A00      move Acc, #0
7: 0011: 0102      db   01h, 02h
8: 0012: 5555      dw   05555h
9: 0013: 5374      db   "String"
          7269
          6E67

10:
11:          segment data
12:
13:          org 0100h
14:
15:          ; move Acc, #0      ; Illegal
16: 0100: 0102      db   01h, 02h
17: 0101: 5555      dw   05555h
18: 0102: 5374      db   "String"
          7269
          6E67

19:
20:          segment code
21:
22: 0016: 0A0A      move Acc, #10
23:
24:          end

```

As discussed previously, running MaxQAsm on a source file with SEGMENT DATA statements included will generate a separate hex file, <basename>\_d.hex. For the example above, the following two files are generated.

```
segment.hex
```

```
:020000040000FA
:0E002000000A01025555537472696E670A0A90
:00000001FF
```

```
segment_d.hex
```

```
:020000040000FA
:0A01000001025555537472696E67D1
:00000001FF
```

These hex files are both loaded automatically by MAX-IDE when the project is executed.

## Automatic Prefixing

All MAXQ instructions are the same size – a single 16 bit word. However, certain operations require more information than will fit in that 16 bit space. For these operations, a *prefixing* instruction must be used to preload the additional information required before the instruction is executed. This additional information is stored in the Prefix (PFX) register.

Prefixing is required under the following circumstances.

- When using a 16-bit immediate value as a source, and the high byte of that immediate value is nonzero.
- When using a 24-bit/32-bit immediate value as a source, and the high byte(s) of that immediate value are nonzero (MaxQ30 core only).
- When accessing any register with an index greater than 15 as a source.
- When accessing any register with an index greater than 7 as a destination.

For these types of operations, prefix instructions will be generated automatically by MaxQAsm. The following list file shows examples of operations which do and do not require prefixing.

```

Line: Addr: Opcode
=====
1:
2: 0000: 0955      move  A[0], #055h          ; No prefix - 8 bit immediate
3: 0001: 0934      move  A[0], #00034h       ; No prefix - high byte is zero
4: 0002: 0B12
4: 0003: 0934      move  A[0], #01234h       ; Prefix for nonzero high byte
5: 0004: 2B00
5: 0005: 0955      move  A[8], #055h         ; Prefix for 8+ index destination
6: 0006: 89F0      move  A[0], M0[0Fh]       ; No prefix
7: 0007: 1B00
7: 0008: 8900      move  A[0], M0[10h]       ; Prefix for 16+ index source
8: 0009: 3B00
8: 000A: 8900      move  A[8], M0[10h]       ; Prefix for source+destination
9:
10:                end

```

## Manual Prefixing

It is possible to perform the required prefixing operations explicitly in assembly code by including the required prefixing instructions. This allows precise control of the code generated and opens up a few additional possibilities for operations.

Information loaded into the Prefix register PFX remains there for a single cycle only (except for the MaxQ30 core), so the prefixing instruction(s) must always come immediately before the instruction that requires the prefix information. The MAXQ core ensures that the instruction sequence will never be interrupted by generating an *interrupt exception window* (see the MAXQ Family User's Guide for more details) when the PFX register is loaded.

To provide the high nonzero byte for a 16-bit immediate source, all that is required is to load the PFX register with the high byte value.

```

2: 0000: 0B12
2: 0001: 0934      move  A[0], #01234h

```

is equivalent to...

```

4: 0002: 0B12      move  PFX[0], #012h
5: 0003: 0934      move  A[0], #034h

```

The additional information needed when selecting extended source and destination register indexes is provided by writing to a particular PFX index, from 0 to 7.

**Table 5. Prefix Register Indexing**

PFX Destination	Accesses Destination Index	Accesses Source Indexes
PFX[0]	0 – 7	0 – 15
PFX[1]	0 – 7	16 – 31
PFX[2]	8 – 15	0 – 15
PFX[3]	8 – 15	16 – 31
PFX[4]	16 – 23	0 – 15
PFX[5]	16 – 23	16 – 31
PFX[6]	24 – 31	0 – 15
PFX[7]	24 – 31	16 – 31

When writing PFX in this manner to set the extended bits for source and destination, the following instruction should contain only the low-order bits of the source and destination (0-7 for destination and 0-15 for source) to avoid generating an additional, unneeded prefix instruction. This is identical to the code automatically generated by the assembler.

If both the source and destination are registers, the actual value written to PFX is unused, but it must still be written to select the high-order destination and source bits.

```

7: 0004: 8000    move  M0[0], M0[0]          ; No prefix needed

9: 0005: 1B00    move  PFX[1], #00h         ; Manual version
10: 0006: 8000    move  M0[0], M0[0]
11: 0007: 1B00
11: 0008: 8000    move  M0[0], M0[16]       ; Automatic version

13: 0009: 2B00    move  PFX[2], #00h         ; Manual version
14: 000A: 9000    move  M0[1], M0[0]
15: 000B: 2B00
15: 000C: 9000    move  M0[9], M0[0]        ; Automatic version

17: 000D: 3B00    move  PFX[3], #00h         ; Manual version
18: 000E: 9020    move  M0[1], M0[2]
19: 000F: 3B00
19: 0010: 9020    move  M0[9], M0[18]       ; Automatic version

21: 0011: 4B00    move  PFX[4], #00h         ; Manual version
22: 0012: 8000    move  M0[0], M0[0]
23: 0013: 4B00
23: 0014: 8000    move  M0[16], M0[0]       ; Automatic version

25: 0015: 5B00    move  PFX[5], #00h         ; Manual version
26: 0016: B040    move  M0[3], M0[4]
27: 0017: 5B00
27: 0018: B040    move  M0[19], M0[20]      ; Automatic version

```

```

29: 0019: 6B00      move  PFX[6], #00h      ; Manual version
30: 001A: F070      move  M0[7], M0[7]
31: 001B: 6B00
31: 001C: F070      move  M0[31], M0[7]    ; Automatic version

33: 001D: 7B00      move  PFX[7], #00h      ; Manual version
34: 001E: 8000      move  M0[0], M0[0]
35: 001F: 7B00
35: 0020: 8000      move  M0[24], M0[16]   ; Automatic version

37: 0021: 0B12      move  PFX[0], #012h     ; Manual version
38: 0022: 0034      move  M0[0], #034h
39: 0023: 0B12
39: 0024: 0034      move  M0[0], #01234h   ; Automatic version

41: 0025: 2B12      move  PFX[2], #012h     ; Manual version
42: 0026: 0034      move  M0[0], #034h
43: 0027: 2B12
43: 0028: 0034      move  M0[8], #01234h   ; Automatic version

45: 0029: 4B12      move  PFX[4], #012h     ; Manual version
46: 002A: 0034      move  M0[0], #034h
47: 002B: 4B12
47: 002C: 0034      move  M0[16], #01234h  ; Automatic version

49: 002D: 6B12      move  PFX[6], #012h     ; Manual version
50: 002E: 0034      move  M0[0], #034h
51: 002F: 6B12
51: 0030: 0034      move  M0[24], #01234h  ; Automatic version
52:
53:                end

```

Manual prefixing opens up a few new possibilities. For example, any two 8-bit registers can be merged into any 16-bit register, without requiring GR to be overwritten.

```

53: 0031: ABE8      move  PFX[2], CKCN
54: 0032: 89F8      move  A[0], WDCN      ; Sets A[8] = CKCN:WDCN

```

### Prefixing with the MaxQ30 Core

When using the MaxQ30 core (-w32 switch), prefixing operates identically to the MaxQ10 and MaxQ20 cores except that the prefix register PFX may be loaded back-to-back to load larger values. The last load of PFX before the instruction is used to set the high-order destination and source bits; previous loads must be to PFX[0] or PFX[1]. So, for example, loading a 32-bit immediate value into one of the accumulator registers assembles to:

```

4: 000001: 0B12                ; move PFX, #12h
4: 000002: 0B34                ; move PFX, #34h
4: 000003: 0B56                ; move PFX, #56h
4: 000004: 1978      move  A[1], #012345678h ; move A[1], #78h

```

which takes four cycles to execute.

## USING THE MACRO PREPROCESSOR

The MAXQ preprocessor, Macro, is used to perform a number of operations on assembly files before they are processed by MaxQAsm. It may be used either automatically as part of the MAX-IDE environment or directly from the Windows command line.

### Invoking Macro from the Command Line

Running Macro from the command line with no parameters will display version information and list command line options.

```
D:\MaxQAsm>macro
```

```
Dallas Semiconductor Macro Preprocessor, Alpha Version 2.001.0001
```

```
Last build Aug 2 2005 at 12:06:54
```

```
Usage: macro [options] <input file>
```

```
options:
```

```
-q Quiet mode
-e- Strip "END" statement from output file
-I<Include Path> Optional include file directory
-s Suppress lines removed by macro
-D<Identifier>=<String> Add <Identifier> to symbol table with string
as predefined value
-L Generate debug information
```

The <input file> is the name of the MAXQ assembly file to be processed; only one file name may be specified on the command line. The assembly file must be in plain text ASCII format and may have any extension (“.asm” is typical but is not required). The extension must be provided on the command line as part of the file name; “.asm” will not be assumed as a default.

**Table 6. Macro Command Line Options**

Option	Function
-q	If this option is included, the preprocessor will run silently, generating no output on the command line for a successful run. The default behavior is to output the version header and usage information as shown above. Any error messages will be output whether or not this option is included.
-e-	If this option is included, the END statement will be removed from the output file.
-I<path> or -I <path>	This option may be included multiple times in a single command line. If included, this option specifies (in <path>) one or more directories to search for include files. Multiple directories may be specified by separating them with semicolons, and directory paths may be relative or absolute.
-s	Normally, lines which are only of interest to Macro and not the assembler (such as equate definitions, #define and #include lines, and macro defines) are commented out in the output file. Including this option causes such lines to be removed from the output file completely.
-D<id>=<val> or -D <id>=<val>	This option defines a preprocessor value before the input file is read. Including this option has the same effect as placing  #define <id> <val>  at the first line of the assembly source file.
-L	Including this option causes additional debugging information (in comments) to be placed into the assembly source file, including begin-module and end-module markers and line numbering information. This information is not used by the assembler, but it is scanned by MAX-IDE for debugging purposes.

## Invoking Macro from Within MAX-IDE

When a project is compiled in the MAX-IDE environment, each source file is processed by Macro individually with the command line switches “-e- -L -q”. After preprocessing is complete, the output files are concatenated in the order they appear in the MAX-IDE project window and an END statement is appended before the merged file is processed by MaxQAsm.

## Macro Output File

Macro produces a single output file with the same base name as the source file and an extension of “.mpp”. For example, running the following source file through Macro using switches “-e- -q”

```
segment code
org 0000h

    move    GR, #01234h
    ljump   $

segment data
org 0000h

    db      01h, 02h, 03h, 04h
    db      "01234567"
    dw      0ABCDh

end
```

results in the following output file.

```
segment code
org 0

    move    GR, #01234h
    ljump   $

segment data
org 0

    db      1, 2, 3, 4
    db      "01234567"
    dw      0abcdh

    ;;-- Line suppressed by Macro end
```

## Include Files

The INCLUDE directive allows common defines, equates and macros to be stored in a file which can be used by multiple source files in a project or even in multiple projects. The syntax for this directive is as follows.

```
$include (<filename>)  
or  
#include "<filename>"
```

The INCLUDE directive causes the contents of the file <filename> to be read and processed by Macro immediately before processing of the current source file continues. Once the include file has been processed, the original file resumes being processed by Macro at the line following the INCLUDE directive.

Include files may be nested by using the INCLUDE directive to include other files, so that

### **source.asm**

```
#include "def1.inc"
```

### **def1.inc**

```
#define SYMBOL1 01h  
#include "def2.inc"
```

### **def2.inc**

```
#define SYMBOL2 02h  
#include "def3.inc"
```

### **def3.inc**

```
#define SYMBOL3 03h
```

has the same effect as

### **source.asm**

```
#define SYMBOL1 01h  
#define SYMBOL2 02h  
#define SYMBOL3 03h
```

Recursive and circular include file nesting will, of course, result in errors.

**error.asm**

```
#include "error.asm"      ; Recursive reference
```

**error1.asm**

```
#include "error2.asm"
```

**error2.asm**

```
#include "error1.asm"      ; Circular reference back to error1.asm
```

**Include File Paths**

Include file names may have any type of extension, and they may be specified in the INCLUDE directive with either no path:

```
#include "defs.inc"
```

or with a relative path:

```
#include "inc\defs.inc"
```

or with an absolute path on the current drive:

```
#include "\\projects\inc\defs.inc"
```

or with an absolute path on a specified drive:

```
#include "C:\projects\inc\defs.inc"
```

If one of the two absolute path types is used, then Macro searches for the include file only in the specified directory. If the relative path version is used, or if no path is given, Macro searches the current directory (plus any relative path) followed by the list of directories, if any, given on the command line using the `-I` option (plus any relative path).

For example, given `"-I ..\lib;incl;c:\maxq"` on the Macro command line, the following include files will be searched for as follows, in the order given.

```
#include "defs.inc"      ; .\defs.inc
                        ; ..\lib\defs.inc
                        ; .\incl\defs.inc
                        ; c:\maxq\defs.inc

#include "inc\defs.inc"  ; .\inc\defs.inc
                        ; ..\lib\inc\defs.inc
                        ; .\incl\inc\defs.inc
                        ; c:\maxq\inc\defs.inc
```

## Include File Contents

Include files may contain all MaxQAsm and Macro statements and directives except for instructions. Include files may end with an END statement, but it is not required that they do so and it will be removed by Macro in any case.

## Macro Constant Operations

Macro can perform a number of operations on constant values used as immediate operands (beginning with “#”), as well as constant values used in equate definitions. For any of these operators, results extending beyond 32 bits will be truncated to the low 32 bits of the result. Negative results will be converted to a 32-bit two’s complement value.

**Table 7. Macro Constant Operators**

Operator	Format	Description
+	$x + y$	Returns the integer sum of $x$ and $y$ .
-	$x - y$	Returns the integer difference of $x$ and $y$ .
-	$-x$	Unary minus; returns the two’s complement negation of $x$ .
*	$x * y$	Returns the integer product of $x$ and $y$ .
/	$x / y$	Returns the value of $x$ divided by $y$ , with any remainder discarded (truncated to an integer).
%	$x \% y$	Returns the modulus of $x$ divided by $y$ .
MIN	MIN ( $x$ , $y$ )	If $x < y$ , returns $x$ ; otherwise returns $y$ .
MAX	MAX ( $x$ , $y$ )	If $x > y$ , returns $x$ ; otherwise returns $y$ .
	$x   y$	Returns the bitwise OR value of $x$ and $y$ .
&	$x \& y$	Returns the bitwise AND value of $x$ and $y$ .
^	$x \wedge y$	Returns the bitwise XOR value of $x$ and $y$ .
NOT !	NOT ( $x$ ) ! $x$	Returns the bitwise negation of $x$ .
HIGH	HIGH ( $x$ )	Returns the high byte (bits 8 to 15) of the word value $x$ .
LOW	LOW ( $x$ )	Returns the low byte (bits 0 to 7) of the word value $x$ .
CONST16	CONST16 ( $x$ )	Returns the low word (bits 0 to 15) of the value $x$ .
<<	$x \ll y$	Returns the value of $x$ shifted left by $y$ bits, with the low $y$ bits filled with 0. The return value is undefined for values of $y$ greater than 32 or less than 0.
>>	$x \gg y$	Returns the value of $x$ shifted right by $y$ bits, with the high $y$ bits filled with 0. The return value is undefined for values of $y$ greater than 32 or less than 0.
AND &&	$x \text{ AND } y$ $x \&\& y$	Logical AND. If both $x$ and $y$ are nonzero, returns 1; otherwise, returns 0.
OR 	$x \text{ OR } y$ $x    y$	Logical OR. If either $x$ or $y$ is nonzero, returns 1; otherwise, returns 0.
>	$x > y$	Greater than. If $x > y$ , returns 1; otherwise, returns 0.
>=	$x \geq y$	Greater than or equal to. If $x \geq y$ , returns 1; otherwise, returns 0.
<	$x < y$	Less than. If $x < y$ , returns 1; otherwise, returns 0.
<=	$x \leq y$	Less than or equal to. If $x \leq y$ , returns 1; otherwise, returns 0.
==	$x == y$	Equal to. If $x == y$ , returns 1; otherwise, returns 0.
!=	$x != y$	Not equal to. If $x != y$ , returns 1; otherwise, returns 0.

**Macro Constant Operation Examples**

```

move Acc, #1 + 1          ; 2 - Addition
move Acc, #4 - 1          ; 3 - Subtraction
move Acc, #2 * 2          ; 4 - Multiplication
move Acc, #26 / 5         ; 5 - Integer division (truncated)
move Acc, #26 % 5         ; 1 - Integer modulus
move Acc, #MIN(6,7)       ; 6 - Minimum of two values
move Acc, #MAX(6,7)       ; 7 - Maximum of two values

move Acc, #550h | 055h    ; 555h      - Bitwise OR
move Acc, #550h & 055h    ; 050h      - Bitwise AND
move Acc, #550h ^ 055h    ; 505h      - Bitwise XOR
move Acc, #HIGH(1234h)    ; 12h       - High byte of word
move Acc, #LOW(1234h)     ; 34h       - Low byte of word
move Acc, #NOT(0000h)     ; FFFFFFFFh - Bitwise negation
move Acc, #001h << 7      ; 80h      - Shift left
move Acc, #080h >> 7      ; 01h      - Shift right

move Acc, #1 AND 0        ; 0         - Logical AND
move Acc, #1 && 1          ; 1         - Logical AND
move Acc, #0 OR 0         ; 0         - Logical OR
move Acc, #1 || 10        ; 1         - Logical OR
move Acc, #5 > 3          ; 1         - Greater than
move Acc, #4 < 3          ; 0         - Less than
move Acc, #2 >= 2         ; 1         - Greater than or equal to
move Acc, #3 <= 1         ; 0         - Less than or equal to
move Acc, #1 == 1         ; 1         - Equal to
move Acc, #2 != 3         ; 1         - Not equal to
move Acc, #-1             ; FFFFFFFF - Two's complement negation

move Acc, #(2+(2*3)) << 2 ; 20h

```

```

L1:
move Acc, #L1             ; This is valid
move Acc, #HIGH(L1)       ; This will not process correctly

```

Since labels are not processed by Macro (only by MaxQAsm), they can be used as constants but cannot be used as part of a constant operation.

```

L1:
move Acc, #L1             ; This is valid
move Acc, #HIGH(L1)       ; This will not process correctly since L1 has no value
                           ; at the time of macro processing

```

## Equates

Equates, which are defined using the EQU directive, are used for numeric replacement in any place that a constant value would normally be used. They may use all the formats and constant operators defined above for constants. Labels cannot be used in equate definitions.

The syntax for defining an equate is as follows.

```
<identifier> equ <constant>
```

Identifiers in Macro follow the same rules as identifiers in MaxQAsm. They may be from 1 to 127 characters in length and may consist of the following characters:

- Alphabetic characters “A” to “Z” or “a” to “z”. Identifiers in Macro are case-insensitive; that is, “DEF1” and “def1” are considered equivalent.
- Numeric characters “0” to “9”. However, a numeric character may not be the first character in an identifier (such as “0DEF”).
- The characters “\_” (underbar), “?” (question mark), and “\$” (dollar sign).

Additionally, the <identifier> value for an equate directive must be one that has not previously been used as an identifier in a #define or macro definition.

The <constant> value for an equate directive may use any allowed constant format (binary, hex, decimal) and may contain constant operators and previously established equates and defines. **Note: The value of an equate is limited to 32 bits (0000000h – 0FFFFFFFh).**

As an example, the source file shown here:

```
X equ 1
Y equ 1000b
Z equ 10h

S1 equ X + X
S2 equ S1 << Y
S3 equ Z * S2

    move Acc, #S1
    move Acc, #S2
    move Acc, #S3
    move A[X], #0
end
```

results in the following Macro output.

```
;;-- Line suppressed by Macro X equ 1
;;-- Line suppressed by Macro Y equ 8
;;-- Line suppressed by Macro Z equ 010h

;;-- Line suppressed by Macro S1 equ 1 + 1
;;-- Line suppressed by Macro S2 equ 2 << 8
;;-- Line suppressed by Macro S3 equ 010h * 0200h

    move Acc, #2
    move Acc, #0200h
    move Acc, #02000h
    move A[1], #0
end
```

## Defines

Defines allow replacement of an identifier with any desired string of text, using the following syntax.

```
#define <identifier> <replacement>
```

The <identifier> value for a #define directive must be a valid Macro identifier that has not been previously used as an equate or macro identifier. Unlike an equate, which can only be used where a constant value would normally be used, a define will be replaced anywhere in an instruction statement, #define or equate definition, or macro body. There are a few restrictions on where #define values will be replaced by Macro.

- In order for the identifier to be replaced as its defined value, it must be found as a standalone symbol in the line, and not as part of a larger symbol. For example, if AA is defined as a replacement value, AAA will not be affected.
- Replacement is not performed inside double-quoted strings.
- Replacement is not performed inside comments.

The replacement operation is performed repeatedly on each line until no more replacement symbols are found. This means that it is possible to create one #define in terms of another #define value.

```
#define A B
#define B C
```

will result in all occurrences of the symbol “A” being replaced with “C”. Since there is no limit on the number of replacement scans, both

```
#define A A+1
```

and

```
#define A B
#define B A
```

will cause Macro failures.

### Defining Registers

One typical use of the #define directive is to define mnemonics for the peripheral registers that vary from one MAXQ microcontroller to another. These are the registers found in modules M0 to M5, depending on the particular part. See the User’s Guide Supplement for the particular MAXQ microcontroller you are using for more details.

With the addresses of the new registers determined, the #define directive can be used as follows. (Examples are from the MAXQ2000).

```
#define PO0          M0 [0]
#define PO1          M0 [1]
#define PO2          M0 [2]
#define PO3          M0 [3]
#define EIF0         M0 [6]
#define EIE0         M0 [7]
```

and so on. The new register definitions may be placed in an include file for use by any assembly source file, allowing the new registers to be used in the same way as the predefined system registers.

```
move Acc, #01h
move PO0, Acc
```

## Undefining Symbols

The `#undefine` directive can be used to undefine a replacement previously defined by a `#define` directive. The syntax for this directive is as follows.

```
#undefine <identifier>
```

The `<identifier>` should be an identifier value used in a previous `#define` directive. If the identifier is an equate or macro identifier, or if the identifier is undefined, `#undefine` will have no effect; this is not treated as an error.

Once an identifier has been cleared with `#undefine`, it may be redefined with a different value or as a different type of preprocessor symbol (macro or equate).

```
#define ONE 01h           ; Define identifier
ZZZ    equ 02h          ; Equate identifier

    move Acc, #ONE      ; move Acc, #01h

#undefine ONE            ; Remove ONE

    move Acc, #ONE      ; move Acc, #ONE

#undefine TWO            ; No effect
#undefine ZZZ            ; No effect

    move Acc, #ZZZ      ; move Acc, #02h

ONE    equ 01h          ; ONE can be reused as an equate

    move Acc, #ONE      ; move Acc, #01h

end
```

## Conditional Assembly

Macro provides a set of conditional assembly directives which allow code to be included or excluded from the assembly process based on constant value expressions. The general form of a conditional assembly block is as follows.

```
if constant || ifdef identifier || ifndef identifier
    ...code statements...
[ elseif constant
    ...code statements... ]
[ else
    ...code statements... ]
endif
```

### #if Directive

This directive is one of the three that can be used to begin a conditional assembly block. There are two equivalent syntaxes for this directive, either:

```
if constant
or
#if constant
```

In either case, the value of <constant> must be a constant expression formed of constant values, equates, numeric defines and constant operators. The behavior of this directive depends on the value of the constant expression, as follows.

- If the constant expression evaluates to zero:
  - All code following the #if directive until the next #elseif, #else or #endif directive will be **excluded**.
  - If the next directive encountered is #endif, the conditional assembly block is complete.
  - If the next directive encountered is #elseif, that directive will be evaluated as described below.
  - If the next directive encountered is #else, all code following that directive will be **included** until the #endif directive is reached, at which point the conditional assembly block is complete.
- If the constant expression evaluates to nonzero:
  - All code following the #if directive until the next #elseif, #else or #endif directive will be **included**.
  - If the next directive encountered is #endif, the conditional assembly block is complete.
  - If the next directive encountered is #else or #elseif, this directive and all following code statements and directives will be excluded and ignored until the next #endif directive is reached, at which point the conditional assembly block is complete.

**#ifdef Directive**

This directive is one of the three that can be used to begin a conditional assembly block. There are two equivalent syntaxes for this directive, either:

```
ifdef identifier
```

or

```
#ifdef identifier
```

In either case, the behavior of this directive depends on whether or not <identifier> is currently defined (has been defined with #define and has not been undefined with #undefine), as follows.

- If the identifier is currently defined:
  - All code following the #ifdef directive until the next #elseif, #else or #endif directive will be **excluded**.
  - If the next directive encountered is #endif, the conditional assembly block is complete.
  - If the next directive encountered is #elseif, that directive will be evaluated as described below.
  - If the next directive encountered is #else, all code following that directive will be **included** until the #endif directive is reached, at which point the conditional assembly block is complete.
- If the identifier is currently undefined:
  - All code following the #ifdef directive until the next #elseif, #else or #endif directive will be **included**.
  - If the next directive encountered is #endif, the conditional assembly block is complete.
  - If the next directive encountered is #else or #elseif, this directive and all following code statements and directives will be excluded and ignored until the next #endif directive is reached, at which point the conditional assembly block is complete.

**#ifndef Directive**

This directive is one of the three that can be used to begin a conditional assembly block. There are two equivalent syntaxes for this directive, either:

```
ifndef identifier
```

or

```
#ifndef identifier
```

In either case, the behavior of this directive depends on whether or not <identifier> is currently defined (has been defined with #define and has not been undefined with #undefine), as follows.

- If the identifier is currently undefined:
  - All code following the #ifndef directive until the next #elseif, #else or #endif directive will be **excluded**.
  - If the next directive encountered is #endif, the conditional assembly block is complete.
  - If the next directive encountered is #elseif, that directive will be evaluated as described below.
  - If the next directive encountered is #else, all code following that directive will be **included** until the #endif directive is reached, at which point the conditional assembly block is complete.
- If the identifier is currently defined:
  - All code following the #ifndef directive until the next #elseif, #else or #endif directive will be **included**.
  - If the next directive encountered is #endif, the conditional assembly block is complete.
  - If the next directive encountered is #else or #elseif, this directive and all following code statements and directives will be excluded and ignored until the next #endif directive is reached, at which point the conditional assembly block is complete.

*Note: Only identifiers defined with the #define directive are “defined” for the purpose of these directives. Identifiers defined as equates, macros or labels will be treated as “undefined” when evaluated by #ifdef and #ifndef.*

**#else Directive**

This directive is optional in a conditional assembly block. If used, it must be the last conditional assembly directive in the block before the #endif directive. The syntax for this directive is as follows.

```
else
```

```
or
```

```
#else
```

The behavior of this directive depends on the evaluation status of the previous #if, #ifdef, #ifndef, and #elseif directives in the conditional assembly block.

- If none of the previous sections in the conditional assembly block have had their code included, all code following this directive is **included** until the #endif directive is reached.
- If any of the previous sections in the conditional assembly block have had their code included, all code following this directive is **excluded** until the #endif directive is reached.

**#elseif Directive**

This directive is optional in a conditional assembly block. If used, it must come before any #else directive in the block. The syntax for this directive is as follows.

```
elseif constant
```

```
or
```

```
#elseif constant
```

In either case, the value of <constant> must be a constant expression formed of constant values, equates, numeric defines and constant operators. This directive is only evaluated when no code has been previously included in this conditional assembly block. If evaluated, the behavior of this directive depends on the value of the constant expression, as follows.

- If the constant expression evaluates to zero:
  - All code following the #elseif directive until the next #elseif, #else or #endif directive will be **excluded**.
  - If the next directive encountered is #endif, the conditional assembly block is complete.
  - If the next directive encountered is #elseif, that directive will be evaluated in the same manner.
  - If the next directive encountered is #else, all code following that directive will be **included** until the #endif directive is reached, at which point the conditional assembly block is complete.
- If the constant expression evaluates to nonzero:
  - All code following the #if directive until the next #elseif, #else or #endif directive will be **included**.
  - If the next directive encountered is #endif, the conditional assembly block is complete.
  - If the next directive encountered is #else or #elseif, this directive and all following code statements and directives will be excluded and ignored until the next #endif directive is reached, at which point the conditional assembly block is complete.

**#endif Directive**

This directive ends a conditional assembly block. The syntax for this directive is as follows.

```
endif constant
```

```
or
```

```
#endif constant
```

**Conditional Block Nesting**

Conditional assembly blocks may be nested. In this case, each additional `#if/#endif` block encountered is evaluated if the section it is located in is currently being included.

**Example Source File**

```

ONE      equ 01h
#define  TWO 02h

#if (ONE == 1)
    move Acc, #01          ; Included
#endif

#if (TWO == 1)
    move Acc, #02          ; Excluded
#elif (TWO == 2)
    move Acc, #03          ; Included
#else
    move Acc, #04          ; Excluded
#endif

#ifdef  THREE
    move Acc, #05          ; Excluded
#else
    move Acc, #06          ; Included
#endif

#ifndef FOUR
    move Acc, #07          ; Included
#elif (ONE == 1)
    move Acc, #08          ; Not evaluated
#else
    move Acc, #08          ; Excluded
#endif

#if ((1+1) == 2)
    #ifdef TWO              ; Evaluated
        move Acc, #09      ; Included
    #endif
#else
    #if (1)                 ; Not evaluated
        move Acc, #10     ; Excluded
    #else
        move Acc, #11     ; Excluded
    #endif
#endif

end

```

**Example Preprocessor Output File (with `-s` switch included)**

```

    move Acc, #1; Included
    move Acc, #3; Included
    move Acc, #6; Included
    move Acc, #7; Included
    move Acc, #9; Included
end

```

## Defining Macros

Macros are used to repeat frequently-occurring sections of code without the overhead of calling a subroutine. Each time a macro is called in a source file, all of the macro code is inserted at that point. Macros are defined using the `MACRO` directive, with syntax as shown below.

```
<identifier> MACRO [PARAM] [<parameter 1> [[ ,] <parameter 2> ...]]
[LOCAL <local identifier 1> [[ ,] <local identifier 2> ...]
  <macro code>
ENDM
```

A macro definition consists of the following components.

- The macro `<identifier>` is the identifier that will be used when calling the macro. It must follow the standard identifier rules and may not be an identifier which is also used as a label, equate, or define.
- The parameter list, which is optional, defines one or more parameters which are passed when the macro is called. The values for these parameters are inserted into the body of the macro.
- The local identifier list, which is optional, defines one or more local identifiers which can be used as labels within the body of the macro. Each time the macro is called, these identifiers take unique values in order to avoid conflicts with identifiers in previous expansions of the macro.
- The macro code itself appears after the parameter list (if any) and the local identifier list (if any). Macro code may contain equates, defines, and nested macro calls, which are all processed normally. However, macro code may not contain macro definitions. Comments within the macro body are retained each time the macro is expanded.
- Macro definitions must end with the `ENDM` directive.

### A Simple Macro Example

The simplest form of macro is one which contains no parameters or local identifiers. The following source file, when processed by Macro, expands into the preprocessed file shown below.

*source file*

```
delay macro                ; Start of macro definition
  nop                      ; Start of macro code
  nop
  nop
  nop                      ; End of macro code
endm                       ; End of macro definition

  delay                    ; Call delay macro

end
```

*Macro output file*

```
; BEGIN Macro delay
  nop                      ; Start of macro code
  nop
  nop                      ; End of macro code
; End Macro delay

end
```

### Macros with Input Parameters

The macro example below contains two parameters which must be passed in each time the macro is called. The first line of the macro definition defines the parameter list.

```
swap macro param Reg1, Reg2
```

This macro has two parameters, `Reg1` and `Reg2`. When the macro is called, each of these parameters must be provided with a value, such as

```
swap A[0], A[1]
```

When the macro code is expanded, each occurrence of `Reg1` will be replaced by `A[0]`, and each occurrence of `Reg2` will be replaced by `A[1]`, in the same way that the `#define` directive operates. However, macro parameters are not `#defines`, and so they cannot be referenced by `#ifdef` and `#ifndef` in the body of the macro.

Note that `defines` and `equates` used in the macro calling line are processed before the macro code is expanded, so that for

```
#define REG1 A[0]
VAL1 equ 011h
```

```
swap REG1, VAL1
```

the macro will be expanded with `Reg1` replaced by `A[0]` (not `REG1`) and `Reg2` replaced by `011h` (not `VAL1`). The following additional rules apply when replacing parameters in the expanded macro body.

- The correct number of parameters must be used when calling the macro. If too many parameters are given, a Macro error will result. If too few parameters are given, the missing parameters will not be replaced with values in the body of the macro, which will most likely result in an assembler error.
- Parameters are not replaced inside comments or double-quoted strings.
- Parameters will only be replaced in the macro body where they appear as separate words, that is, for the macro given above, “`Reg1`” will be replaced in the macro body, but “`Reg1Reg2`” will not. This is identical to the operation of `#define`.

*source file*

```
swap macro param Reg1, Reg2 ; Could also be "swap macro Reg1 Reg2"
  move GR, Reg1
  move Reg1, Reg2
  move Reg2, GR
endm

; swap A[0], A[1], A[2] ; Too many parameters (error)
  swap A[0], A[1]
  swap ; Too few parameters

end
```

*Macro output file*

```

; swap A[0], A[1], A[2]      ; Too many parameters (error)

; BEGIN Macro swap
  move  GR, A[0]
  move  A[0], A[1]
  move  A[1], GR
; End Macro swap

; BEGIN Macro swap
  move  GR, Reg1
  move  Reg1, Reg2
  move  Reg2, GR
; End Macro swap

end

```

**Macros with Local Identifiers**

Local identifiers must be used in macros which contain labels, since if a macro is simply defined as

```

loopN macro Count
  move LC[0], Count
L1:
  djnz LC[0], L1
endm

```

there will be an assembly error if the macro is called more than once, since the L1 label will be redefined at a new location. The solution to this is to use a local identifier, as follows.

```

loopN macro Count
  local L1
  move LC[0], Count
L1:
  djnz LC[0], L1
endm

```

In this way, each time a macro is expanded which contains the local identifier L1, that identifier will be replaced in the body of the macro with a unique identifier formed by merging the name of the source file, the name of the identifier, and the number of times that local identifier has been used. This allows multiple macros to use the same local identifier name without conflicts, as shown below.

*source file*

```

loopN macro Count
local L1
    move LC[0], Count
L1:
    djnz LC[0], L1
endm

loopN2 macro Count
local L1
    move LC[0], Count
L1:
    djnz LC[0], L1
endm

    loopN    #10
    loopN2  #20
    loopN    #30
    loopN2  #40

end

```

*Macro output file*

```

; BEGIN Macro loopN
    move LC[0], #0ah
macro3_asm_1L1:
    djnz LC[0], macro3_asm_1L1
; End Macro loopN

; BEGIN Macro loopN2
    move LC[0], #014h
macro3_asm_2L1:
    djnz LC[0], macro3_asm_2L1
; End Macro loopN2

; BEGIN Macro loopN
    move LC[0], #01eh
macro3_asm_3L1:
    djnz LC[0], macro3_asm_3L1
; End Macro loopN

; BEGIN Macro loopN2
    move LC[0], #028h
macro3_asm_4L1:
    djnz LC[0], macro3_asm_4L1
; End Macro loopN2

end

```

**Nested Macros**

Macro calls may be made inside a macro body, although macros may not be called recursively (a macro may not include a call to itself in its body definition). Parameters may be passed to these nested macro calls, as shown below.

*source file*

```
m1 macro p1 p2 p3
    move Acc, p1
    m2    p2, p3
endm

m2 macro p1 p2
    move Acc, p1
    m3    p2
endm

m3 macro p1
    move Acc, p1
endm

    m1 #01h, #02h, #03h

end
```

*Macro output file*

```
; BEGIN Macro m1
    move Acc, #1
; BEGIN Macro m2
    move Acc, #2
; BEGIN Macro m3
    move Acc, #3
; End Macro m3
; End Macro m2
; End Macro m1

end
```

## REVISION HISTORY

<b>Version</b>	<b>Date</b>	<b>Changes</b>
1.0	02/27/2006	<ul style="list-style-type: none"> <li>• Original release covering Q10, Q20 and Q30 cores.</li> </ul>
1.1	05/15/2006	<ul style="list-style-type: none"> <li>• Added section covering MaxQAsm constant operations.</li> <li>• Corrections to the Q30 predefined registers table.</li> </ul>
1.2	03/07/2007	<ul style="list-style-type: none"> <li>• Added section describing the new global/local label mechanism (added in maxqasm v2.042)</li> <li>• Updated and expanded MaxQAsm constant operations section.</li> <li>• Removed the <i>Labels and Word Alignment</i> section since DB statements always word-align now.</li> <li>• Updated the <i>Data Byte (DB) Statement</i> section.</li> <li>• Added the <i>Data Long/Double (DL/DD) Statement</i> section.</li> <li>• Added the <i>Assembler Message (\$MESSAGE) Directive</i> section.</li> <li>• Updated the <i>Macro Constant Operations</i> section to cover the new CONST16 operator.</li> <li>• Added a note that equates are limited to 32 bits in width.</li> </ul>