APPENDIX A

Differences Between MASM 6.1 and 5.1

For the many users who come to version 6.1 of the Microsoft Macro Assembler directly from the popular MASM 5.1, this appendix describes the differences between the two versions. Version 6.1 contains significant changes, including:

- An integrated development environment called Programmer's WorkBench (PWB) from which you can write, edit, debug, and execute code.
- Expanded functionality for structures, unions, and type definitions.
- New directives for generating loops and decision statements, and for declaring and calling procedures.
- Simplified methods for applying public attributes to variables and routines in multiple-module programs.
- Enhancements for writing and using macros.
- Flat-model support for Windows NT and new instructions for the 80486 processor.

The **OPTION M510** directive (or the /Zm command-line switch) assures nearly complete compatibility between MASM 6.1 and MASM 5.1. However, to take full advantage of the enhancements in MASM 6.1, you will need to rewrite some code written for MASM 5.1.

The first section of this appendix describes the new or enhanced features in MASM 6.1. The second section, "Compatibility Between MASM 5.1 and 6.1," explains how to:

- Minimize the number of required changes with the **OPTION** directive.
- Rewrite your existing assembly code, if necessary, to take advantage of the assembler's enhancements.

New Features of Version 6.1

This section gives an overview of the new features of MASM 6.1 and provides references to more detailed information elsewhere in the documentation. For full explanations and coding examples, see the documentation listed in the cross-references.

The Assembler, Environment, and Utilities

Most of the executable files provided with MASM 6.1 are new or revised. For a complete list of these files, read the PACKING.TXT file on the distribution disk. The book *Getting Started* also provides information about setting up the environment, assembler, and Help system.

The Assembler

The macro assembler, named ML.EXE, can assemble and link in one step. Its new 32-bit operation gives ML.EXE the ability to handle much larger source files than MASM 5.1. The command-line options are new. For example, the /Fl and /Sc options generate instruction timings in the listing file. Command-line options are case-sensitive and must be separated by spaces.

For backward compatibility with MASM 5.1 makefiles, MASM 6.1 includes the MASM.EXE utility. MASM.EXE translates MASM 5.1 command-line options to the new MASM 6.1 command-line options and calls ML.EXE. See the *Reference* book for details.

H2INC

H2INC converts C include files to MASM include files. It translates data structures and declarations but does not translate executable code. For more information, see Chapter 20 of *Environment and Tools.*

NMAKE

NMAKE replaces the MAKE utility. NMAKE provides new functions for evaluating target files and more flexibility with macros and command-line options. For more information, see *Environment and Tools.*

Integrated Environment

PWB is an integrated development environment for writing, developing, and debugging programs. For information on PWB and the CodeView debugging application, see *Environment and Tools*.

Online Help

MASM 6.1 incorporates the Microsoft Advisor Help system. Help provides a vast database of online help about all aspects of MASM, including the syntax and timings for processor and coprocessor instructions, directives, command-line options, and support programs such as LINK and PWB.

For information on how to set up the help system, see *Getting Started*. You can invoke the help system from within PWB or from the QuickHelp program (QH).

HELPMAKE

You can use the HELPMAKE utility to create additional help files from ASCII text files, allowing you to customize the online help system. For more information, see *Environment and Tools.*

Other Programs

MASM 6.1 contains the most recent versions of LINK, LIB, BIND, CodeView, and the mouse driver. The CREF program is not included in MASM 6.1. The Source Browser provides the information that CREF provided under MASM 5.1. For more information on the Source Browser, see Chapter 5 of *Environment and Tools* or Help.

Segment Management

This section lists the changes and additions to memory-model support and directives that relate to memory model.

Predefined Symbols

The following predefined symbols (also called predefined equates) provide information about simplified segments:

Predefined Symbol	Value
@stack	**DGROUP** for near stacks, **STACK** for far stacks
@Interface	Information about language parameters
@Model	Information about the current memory model
@Line	The source line in the current file
@Date	The current date
@FileCur	The current file
@Time	The current time
@Environ	The current environment variables

For more information about predefined symbols, see “Predefined Symbols” in Chapter 1.

Enhancements to the ASSUME Directive

MASM automatically generates **ASSUME** values for the code segment register (CS). It is no longer necessary to include lines such as

```
ASSUME CS:MyCodeSegment
```

in your programs. In addition, the **ASSUME** directive can include **ERROR**, **FLAT**, or *register***:***type*. MASM 6.1 issues a warning when you specify **ASSUME** values for CS other than the current segment or group.

For more information, see “Setting the ASSUME Directive for Segment Registers” in Chapter 2 and “Defining Register Types with ASSUME” in Chapter 3.

Relocatable Offsets

For compatibility with applications for Windows, the **LROFFSET** operator can calculate a relocatable offset, which is resolved by the loader at run time. See Help for details.

Flat Model

MASM 6.1 supports the flat-memory model of Windows NT, which allows segments as large as 4 gigabytes. All other memory models limit segment size to 64K for MS-DOS and Windows. For more information about memory models, see “Defining Basic Attributes with .MODEL” in Chapter 2.

Data Types

MASM 6.1 supports an improved data typing. This section summarizes the improved forms of data declarations in MASM 6.1.

Defining Typed Variables

You can now use the type names as directives to define variables. Initializers are unsigned by default. The following example lines are equivalent:

```
var1    DB      25
var1    BYTE    25
```

Signed Types

You can use the **SBYTE**, **SWORD**, and **SDWORD** directives to declare signed data. For more information about these directives, see “Allocating Memory for Integer Variables” in Chapter 4.

Floating-Point Types

MASM 6.1 provides the **REAL4**, **REAL8**, and **REAL10** directives for declaring floating-point variables. For information on these type directives, see “Declaring Floating-Point Variables and Constants” in Chapter 6 .

Qualified Types

Type definitions can now include distance and language type attributes. Procedures, procedure prototypes, and external declarations let you specify the type as a qualified type. A complete description of qualified types is provided in the section “Data Types” in Chapter 1.

Structures

Changes to structures since MASM 5.1 include:

- Structures can be nested.
- The names of structure fields need not be unique. As a result, you must qualify references to field names.
- Initialization of structure variables can continue over multiple lines provided the last character in the line before the comment field is a comma.
- Curly braces and angle brackets are equivalent.

For example, this code works in MASM 6.1:

```
SCORE           STRUCT
  team1         BYTE    10 DUP (?)
  score1        BYTE    ?
  team2         BYTE    10 DUP (?)
  score2        BYTE    ?
 SCORE          ENDS

 first   SCORE  {"BEARS", 20,        ; This comment is allowed.
                 "CUBS",  10 }

        mov     al, [bx].score.team1 ; Field name must be qualified
                                     ;   with structure name.
```

You can use **OPTION OLDSTRUCTS** or **OPTION M510** to enable MASM 5.1 behavior for structures. See “Compatibility between MASM 5.1 and 6.1,” later in this appendix. For more information on structures and unions, see “Structures and Unions” in Chapter 5.

Unions

MASM 6.1 allows the definition of unions with the **UNION** directive. Unions differ from structures in that all fields within a union occupy the same data space. For more information, see “Structures and Unions” in Chapter 5.

Types Defined with TYPEDEF

The **TYPEDEF** directive defines a type for use later in the program. It is most useful for defining pointer types. For more information on defining types, see "Data Types" in Chapter 1, and "Defining Pointer Types with TYPEDEF" in Chapter 3.

Names of Identifiers

MASM 6.1 accepts identifier names up to 247 characters long. All characters are significant, whereas under MASM 5.1, names are significant to 31 characters only. For more information on identifiers, see "Identifiers" in Chapter 1.

Multiple-Line Initializers

In MASM 6.1, a comma at the end of a line (except in the comment field) implies that the line continues. For example, the following code is legal in MASM 6.1:

```
longstring      BYTE    "This string ",
                        "continues over two lines."
bitmasks        BYTE    80h, 40h, 20h, 10h,
                        08h, 04h, 02h, 01h
```

For more information, see "Statements" in Chapter 1.

Comments in Extended Lines

MASM 5.1 allows a backslash (\ ) as the line-continuation character if it is the last nonspace character in the line. MASM 6.1 permits a comment to follow the backslash.

Determining Size and Length of Data Labels

The **LENGTHOF** operator returns the number of data items allocated for a data label. MASM 6.1 also provides the **SIZEOF** operator. When applied to a type, **SIZEOF** returns the size attribute of the type expression. When applied to a data label, **SIZEOF** returns the number of bytes used by the initializer in the label's definition. In this case, **SIZEOF** for a variable equals the number of bytes in the type multiplied by **LENGTHOF** for the variable.

MASM 6.1 recognizes the **LENGTH** and **SIZE** operators for backward compatibility. For a description of the behavior of **SIZE** under **OPTION M510**, see "Length and Size of Labels with OPTION M510," later in this appendix. For obsolete behavior with the **LENGTH** operator, see also "LENGTH Operator Applied to Record Types," page 356.

For information on **LENGTHOF** and **SIZEOF**, see the following sections in chapter 5: "Declaring and Referencing Arrays," "Declaring and Initializing

Strings," "Declaring Structure and Union Variables," and "Defining Record Variables."

HIGHWORD and LOWWORD Operators

These operators return the high and low words for a given 32-bit operand. They are similar to the **HIGH** and **LOW** operators of MASM 5.1 except that **HIGHWORD** and **LOWWORD** can take only constants as operands, not relocatables (labels).

PTR and CodeView

Under MASM 5.1, applying the **PTR** operator to a data initializer determines the size of the data displayed by CodeView. You can still use **PTR** in this manner in MASM 6.1, but it does not affect CodeView typing. Defining pointers with the **TYPEDEF** directive allows CodeView to generate correct information. See "Defining Pointer Types with TYPEDEF" in Chapter 3.

Procedures, Loops, and Jumps

With its significant improvements for procedure and jump handling, MASM 6.1 closely resembles high-level–language implementations of procedure calls. MASM 6.1 generates the code to correctly handle argument passing, check type compatibility between parameters and arguments, and process a variable number of arguments. MASM 6.1 can also automatically recast jump instructions to correct for insufficient jump distance.

Function Prototypes and Calls

The **PROTO** directive lets you prototype procedures in the same way as high-level languages. **PROTO** enables type-checking and type conversion of arguments when calling the procedure with **INVOKE**. For more information, see "Declaring Procedure Prototypes" in Chapter 7.

The **INVOKE** directive sets up code to call a procedure and correctly pass arguments according to the prototype. MASM 6.1 also provides the **VARARG** keyword to pass a variable number of arguments to a procedure with **INVOKE**. For more information about **INVOKE** and **VARARG**, see "Calling Procedures with INVOKE" and "Declaring Parameters with the PROC Directive" in Chapter 7.

The **ADDR** keyword is new since MASM 5.1. When used with **INVOKE**, it provides the address of a variable, in the same way as the address-of operator (**&**) in C. This lets you conveniently pass an argument by reference rather than value. See "Calling Procedures with INVOKE" in Chapter 7.

High-Level Flow-Control Constructions

MASM 6.1 contains several directives that generate code for loops and decisions depending on the status of a conditional statement. The conditions are tested at run time rather than at assembly time.

Directives new since MASM 5.1 include **.IF**, **.ELSE**, **.ELSEIF**, **.REPEAT**, **.UNTIL**, **.UNTILCXZ**, **.WHILE**, and **.ENDW**. MASM 6.1 also provides the associated **.BREAK** and **.CONTINUE** directives for loops and **IF** statements. For more information, see "Loops" in Chapter 7 and "Decision Directives" on page 171.

Automatic Optimization for Unconditional Jumps

MASM 6.1 automatically determines the smallest encoding for direct unconditional jumps. See "Unconditional Jumps" in Chapter 7.

Automatic Lengthening for Conditional Jumps

If a conditional jump cannot reach its target destination, MASM automatically recasts the code to use an unconditional jump to the target. See "Jump Extending," page 169.

User-Defined Stack Frame Setup and Cleanup

The prologue code generated immediately after a **PROC** statement sets up the stack for parameters and local variables. The epilogue code handles stack cleanup. MASM 6.1 allows user-defined prologues and epilogues, as described in "Generating Prologue and Epilogue Code" in Chapter 7.

Simplifying Multiple-Module Projects

MASM 6.1 simplifies the sharing of code and data among modules and makes the use of include files more efficient.

EXTERNDEF in Include Files

MASM 5.1 requires that you declare public and external all data and routines used in more than one module. With MASM 6.1, a single **EXTERNDEF** directive accomplishes the same task. **EXTERNDEF** lets you put global data declarations within an include file, making the data visible to all source files that include the file. For more information, see "Using EXTERNDEF" in Chapter 8.

Search Order for Include Files

MASM 6.1 searches for include files in the directory of the main source file rather than in the current directory. Similarly, it searches for nested include files in the directory of the include file. You can specify additional paths to search with the /I command-line option. For more information on include files, see "Organizing Modules" in Chapter 8.

Enforcing Case Sensitivity

In MASM 5.1, sensitivity to case is influenced only by command-line options such as /MX, not the language type given with the **.MODEL** directive. In MASM 6.1, the language type takes precedence over the command-line options in specifying case sensitivity.

Alternate Names for Externals

The syntax for **EXTERN** allows you to specify an alternate symbol name, which the linker can use to resolve an external reference to an unused symbol. This prevents linkage with unneeded library code, as explained in "Using EXTERN with Library Routines," Chapter 8.

Expanded State Control

Several directives in MASM 6.1 enable or disable various aspects of the assembler control. These include 80486 coprocessor instructions and use of compatibility options.

The OPTION Directive

The new **OPTION** directive allows you to selectively define the assembler's behavior, including its compatibility with MASM 5.1. See "Using the OPTION Directive" in Chapter 1 and "Compatibility between MASM 5.1 and 6.1," later in this appendix.

The .NO87 Directive

The **.NO87** directive disables all coprocessor instructions. For more information, see Help.

The .486 and .486P Directives

MASM 6.1 can assemble instructions specific to the 80486, enabled with the **.486** directive. The **.486P** directive enables 80486 instructions at the highest privilege level (recommended for systems-level programs only). For more information, see Help.

The PUSHCONTEXT and POPCONTEXT Directives

The directive **PUSHCONTEXT** saves the assembly environment, and **POPCONTEXT** restores it. The environment includes the segment register assumes, the radix, the listing and CREF flags, and the current processor and coprocessor. Note that **.NOCREF** (the MASM 6.1 equivalent to **.XCREF**) still determines whether information for a given symbol will be added to Browser information and to the symbol table in the listing file. For more information on listing files, see Appendix C or Help.

New Processor Instructions

MASM 6.1 supports these instructions for the 80486 processor:

80486 Instruction	Description
BSWAP	Byte swap
CMPXCHG	Compare and exchange
INVD	Invalidate data cache
INVLPG	Invalidate Translation Lookaside Buffer entry
WBINVD	Write back and invalidate data cache
XADD	Exchange and add

For full descriptions of these instructions, see the *Reference* or Help.

Renamed Directives

Although MASM 6.1 still supports the old names in MASM 5.1, the following directives have been renamed for language consistency:

MASM 6.1	MASM 5.1
.DOSSEG	**DOSSEG**
.LISTIF	**.LFCOND**
.LISTMACRO	**.XALL**
.LISTMACROALL	**.LALL**
.NOCREF	**.XCREF**
.NOLIST	**.XLIST**
.NOLISTIF	**.SFCOND**
.NOLISTMACRO	**.SALL**
ECHO	**%OUT**
EXTERN	**EXTRN**
FOR	**IRP**
FORC	**IRPC**
REPEAT	**REPT**
STRUCT	**STRUC**
SUBTITLE	**SUBTTL**

Specifying 16-Bit and 32-Bit Instructions

MASM 6.1 supports all instructions that work with the extended 32-bit registers of the 80386/486. For certain instructions, you can override the default operand size with the **W** (word) and the **D** (doubleword) suffixes. For details, see the *Reference* or Help.

Macro Enhancements

There are significant enhancements to macro functions in MASM 6.1. Directives provide for a variable number of arguments, loop constructions, definitions of text equates, and macro functions.

Variable Arguments

MASM 5.1 ignores extra arguments passed to macros. In MASM 6.1, you can pass a variable number of arguments to a macro by appending the **VARARG** keyword to the last macro parameter in the macro definition. The macro can then reference additional arguments relative to the last declared parameter. This procedure is explained in "Returning Values with Macro Functions" in Chapter 9.

Required and Default Macro Arguments

With MASM 6.1, you can use **REQ** or the := operator to specify required or default arguments. See "Specifying Required and Default Parameters" in Chapter 9.

New Directives for Macro Loops

Within a macro definition, **WHILE** repeats assembly as long as a condition remains true. Other macro loop directives, **IRP**, **IRPC**, and **REPT**, have been renamed **FOR**, **FORC**, and **REPEAT**. For more information, see "Defining Repeat Blocks with Loop Directives" in Chapter 9.

Text Macros

The **EQU** directive retains its old functionality, but MASM 6.1 also incorporates a **TEXTEQU** directive for defining text macros. **TEXTEQU** allows greater flexibility than **EQU**. For example, **TEXTEQU** can assign to a label the value calculated by a macro function. For more information, see "Text Macros" in Chapter 9.

The GOTO Directive for Macros

Within a macro definition, **GOTO** transfers assembly to a line labeled with a leading colon(**:**). For more information on **GOTO,** see Help**.**

Macro Functions

At assembly time, macro functions can determine and return a text value using **EXITM**. Predefined macro string functions concatenate strings, return the size of a string, and return the position of a substring within a string. For information on writing your own macro functions, see “Returning Values with Macro Functions” in Chapter 9.

Predefined Macro Functions

MASM 6.1 provides the following predefined text macro functions:

Symbol	Value Returned
@CatStr	A concatenated string
@InStr	The position of one string within another
@SizeStr	The size of a string
@SubStr	A substring

For more information on predefined macros, see “String Directives and Predefined Functions” in Chapter 9.

MASM 6.1 Programming Practices

MASM 6.1 provides many features that make it easier for you to write assembly code. If you are familiar with MASM 5.1 programming, you may find it helpful to adopt the following list of new programming practices for programming with MASM 6.1. The list summarizes many of the changes covered in the following section, “Compatibility Between MASM 5.1 and 6.1.”

- Select identifier names that do not begin with the dot operator (**.**).
- Use the dot operator (**.**) only to reference structure fields, and the plus operator (**+**) when not referencing structures.
- Different structures can have the same field names. However, the assembler does not allow ambiguous references. You must include the structure type when referring to field names common to two or more structures.
- Separate macro arguments with commas, not spaces.
- Avoid adding extra ampersands in macros. For a list of the new rules about using ampersands in macros, see “Substitution Operator” in Chapter 9 and “OPTION OLDMACROS,” page 372.
- By default, code labels defined with a colon are local. Place two colons after code labels if you want to reference the label outside the procedure.

Compatibility Between MASM 5.1 and 6.1

MASM 6.1 provides a “compatibility mode,” making it easy for you to transfer existing MASM 5.1 code to the new version. You invoke the compatibility mode through the **OPTION M510** directive or the /Zm command-line switch. This section explains the changes you may need to make to get your MASM 5.1 code to run under MASM 6.1 in compatibility mode.

Rewriting Code for Compatibility

In some cases, MASM 6.1 with **OPTION M510** does not support MASM 5.1 behavior. In several cases, this is because bugs in MASM 5.1 were corrected. To update your code to MASM 6.1, use the instructions in this section. This usually requires only minor changes.

Many of the topics listed here will not apply to your code. This section discusses topics in order of likelihood, beginning with the most common. In addition, you may have conflicts between identifier names and new reserved words. **OPTION NOKEYWORD** resolves errors generated from the use of reserved words as identifiers. See "OPTION NOKEYWORD," page 376, for more information.

Bug Fixes Since MASM 5.1

This section lists the differences between MASM 5.1 and MASM 6.1 due to bug corrections since MASM 5.1.

Invalid Use of LOCK, REPNE, and REPNZ

Except in compatibility mode, MASM 6.1 flags illegal uses of the instruction prefixes **LOCK**, **REPNE**, and **REPNZ**. The error generated for invalid uses of the **LOCK**, **REPNE**, and **REPNZ** prefixes is error A2068:

```
instruction prefix not allowed
```

Table A.1 summarizes the correct use of the instruction prefixes. It lists each string instruction with the type of repeat prefix it uses, and indicates whether the instruction works on a source, a destination, or both.

Table A.1 Requirements for String Instructions

Instruction	Repeat Prefix	Source/Destination	Register Pair
MOVS	**REP**	Both	DS:SI, ES:DI
SCAS	**REPE/REPNE**	Destination	ES:DI
CMPS	**REPE/REPNE**	Both	DS:SI, ES:DI
LODS	--	Source	DS:SI
STOS	**REP**	Destination	ES:DI
INS	**REP**	Destination	ES:DI
OUTS	**REP**	Source	DS:SI

No Closing Quotation Marks in Macro Arguments

In MASM 5.1, you can use both single and double quotation marks (' and ") to begin strings in macro arguments. The assembler does not generate an error or warning if the string does not end with quotation marks on a macro call. Instead, MASM 5.1 considers the remainder of the line to be part of the macro argument containing the opening quote, as if there were a closing quotation mark at the end of the line.

By default, MASM 6.1 now generates error A2046:

```
missing single or double quotation mark in string
```

so all single and double quotation marks in macro arguments must be matched.

To correct such errors in MASM 6.1, either end the string with a closing quotation mark as shown in the following example, or use the macro escape character (**!**) to treat the quotation mark literally.

```
; MASM 5.1 code
MyMacro     "all this in one argument

; Default MASM 6.1 code
MyMacro     "all this in one argument"
```

Making a Scoped Label Public

MASM 5.1 considers code labels defined with a single colon inside a procedure to be local to that procedure if the module contains a **.MODEL** directive with a language type. Although the label is local, MASM 5.1 does not generate an error if it is also declared **PUBLIC**. MASM 6.1 generates error A2203:

```
cannot declare scoped code label as PUBLIC
```

If you want to make a label **PUBLIC**, it must not be local. You can use the double colon operator to define a non-scoped label, as shown in this example:

```
         PUBLIC  publicLabel
publicLabel::                  ; Non-scoped label MASM 6.1
```

Byte Form of BT, BTS, BTC, and BTR Instructions

MASM 5.1 allows a byte argument for the 80386 bit-test instructions, but encodes it as a word argument. The byte form is not supported by the processor.

MASM 6.1 does not support this behavior and generates error A2024:

```
invalid operand size for instruction
```

Rewrite your code to use a word-sized argument.

Default Values for Record Fields

In MASM 5.1, default values for record fields can range down to $–2^n$, where n is the number of bits in the field. This results in the loss of the sign bit.

MASM 6.1 allows a range of $–2^{n–1}$ to $2^{n–1}$ for default values. Illegal initializers generate error A2071:

```
initializer too large for specified size
```

Design Change Issues

MASM 6.1 includes design changes that make the language more consistent. These changes are not affected by the **OPTION** directive, discussed later in this appendix. Therefore, the changes require revisions in your code. In most cases, the necessary revisions are minor and the circumstances requiring changes are rare.

Operands of Different Size

MASM 5.1 does not require operands to agree in size, as the following code illustrates:

```
        .DATA?
        var1    DB      ?
        var2    DB      ?
        .CODE
        .
        .
        .
        mov     var1, ax        ; Copy AX to word at var1
```

The operands for the **MOV** instruction do not match in size, yet the instruction assembles correctly. It places the contents of AL into `var1` and AH into `var2`, moving a word of data in one step. If the code defined `var1` as a word value, the instruction

```
        mov     var1, al
```

would also assemble correctly, copying AL into the low byte of `var1` while leaving the high byte unaffected. Except at warning level 0, MASM 5.1 issues a warning to inform you of the size mismatch, but both scenarios are legal.

MASM 6.1 does not accept instructions with operands that do not agree in size. You must specifically "coerce" the size of the memory operand, like this:

```
        mov     BYTE PTR var1, al
```

Conflicting Structure Declarations

MASM 5.1 allows you to declare two or more structures with the same name. Each declaration replaces the previous declaration. However, the field names from previous declarations still remain in the assembler's list of declared values.

MASM 6.1 does not allow conflicting declarations of a structure. It generates errors A2160 through A2165 for each conflicting declaration. The errors note a specific conflict, such as conflicting number of fields, conflicting names of fields, or conflicting initializers.

Forward References to Text Macros Outside of Expressions

MASM 5.1 allows forward references to text macros in specialized cases. MASM 6.1 with **OPTION M510** also permits forward references, but only when the text macro is referenced in an expression. To revise your code, place all macro definitions at the beginning of the file.

HIGH and LOW Applied to Relocatable Operands

In some cases, MASM 5.1 accepts **HIGH** and **LOW** applied to relocatable memory expressions. For example, MASM 5.1 allows this code sequence:

```
; MASM 5.1 code
EXTRN   var1:WORD
var2    DW      0
        mov     al, LOW var1    ; These two instructions yield the
        mov     ah, HIGH var1   ; same as mov ax, OFFSET var1
```

However, the instruction

```
        mov ax, LOW var2
```

is not legal. MASM 6.1 generates error A2105:

```
HIGH and LOW require immediate operands
```

The **OFFSET** operator is required on these operands in MASM 6.1, as shown in the following. Rewrite your code if necessary.

```
; MASM 6.1 code
        mov     al, LOW OFFSET var1
        mov     ah, HIGH OFFSET var2
```

OFFSET Applied to Group Names and Indirect Memory Operands

In MASM 6.1, you cannot apply **OFFSET** to a group name, indirect argument, or procedure argument. Doing so generates error A2098:

```
invalid operand for OFFSET
```

LENGTH Operator Applied to Record Types

In MASM 5.1, the **LENGTH** operator, when applied to a record type, returns the total number of bits in a record definition.

In MASM 6.1, the statement `LENGTH recordName` returns error A2143:

```
expected data label
```

Rewrite your code if necessary. The new **SIZEOF** operator returns information about records in MASM 6.1. For more information, see “Defining Record Variables” in Chapter 5.

Signed Comparison of Hexadecimal Values Using GT, GE, LE, or LT

The rules for two’s-complement comparisons have changed. In MASM 5.1, the expression

```
0FFFFh GT -1
```

is false because the two’s-complement values are equal. However, because hexadecimal numbers are now treated as unsigned, the expression is true in MASM 6.1. To update, rewrite the affected code.

RET Used with a Constant in Procedures with Epilogues

By default in MASM 6.1, the **RET** instruction followed by a constant suppresses automatic generation of epilogue code. MASM 5.1 ignores the operand and generates the epilogue. Remove the argument if necessary. See “Generating Prologue and Epilogue Code” in Chapter 7.

Code Labels at Top of Procedures with Prologues

By default in MASM 5.1, a code label defined on the same line as the first procedure instruction refers to the first byte of the prologue.

In MASM 6.1, a code label defined at the beginning of a procedure refers to the first byte of the procedure after the prologue. If you need to label the start of the prologue code, place the label before the **PROC** statement. For more information, see “Generating Prologue and Epilogue Code” in Chapter 7.

Use of % as an Identifier Character

MASM 5.1 allows **%** as an identifier character. This behavior leads to ambiguities when **%** is used as the expansion operator in macros. Since **%** is not allowed as a character in MASM 6.1 identifiers, you must change the names of any identifiers containing the **%** character. For a list of legal identifier characters, see “Identifiers” in Chapter 1.

ASSUME CS Set to Wrong Value

With MASM 6.1 you do not need to use the **ASSUME** statement for the CS register. Instead, MASM 6.1 generates an automatic **ASSUME** statement for the code segment register to the current segment or group, as explained in “Setting the ASSUME Directive for Segment Registers” in Chapter 2.

Additionally, MASM 6.1 does not allow explicit **ASSUME** statements for CS that contradict the automatically set **ASSUME** statement.

MASM 5.1 allows CS to be assumed to the current segment, even if that segment is a member of a group. With MASM 6.1, this results in warning A4004:

```
cannot ASSUME CS
```

To avoid this warning with MASM 6.1, delete the **ASSUME** statement for CS.

Code Requiring Two-Pass Assembly

Unlike version 5.1, MASM 6.1 does most of its work on its first pass, then performs as many subsequent passes as necessary. In contrast, MASM 5.1 always assembles in two source passes. As a result, you may need to revise or delete some pass-dependent constructs under MASM 6.1.

Two-Pass Directives

To assure compatibility, MASM 6.1 supports 5.1 directives referring to two passes. These include **.ERR1**, **.ERR2**, **IF1**, **IF2**, **ELSEIF1**, and **ELSEIF2**. For second-pass constructs, you must specify **OPTION SETIF2**, as discussed in "OPTION SETIF2," page 377. Without **OPTION SETIF2**, the **IF2** and **.ERR2** directives cause error A2061:

```
[[ELSE]]IF2/.ERR2 not allowed : single-pass assembler
```

MASM 6.1 handles first-pass constructs differently. It treats the **.ERR1** directive as **.ERR**, and the **IF1** directive as **IF**.

The following examples show you how you can rewrite typical pass-sensitive code for MASM 6.1:

- Declare `var` external only if not defined in current module:

```
; MASM 5.1:
    IF2
        IFNDEF  var
            EXTRN var:far
        ENDIF
    ENDIF

; MASM 6.1:
    EXTERNDEF   var:far
```

- Include a file of definitions only once to speed assembly:

```
; MASM 5.1:
    IF1
        INCLUDE file1.inc
    ENDIF

; MASM 6.1:
    INCLUDE FILE1.INC
```

- Generate a `%OUT` or `.ERR` message only once:

```
; MASM 5.1:
    IF2
        %OUT This is my message
    ENDIF

    IF2
        .ERRNZ A NE B
    ENDIF

; MASM 6.1:
    ECHO This is my message

    .ERRNZ A NE B    <ASSERTION FAILURE: A NE B>
```

- Generate an error if a symbol is not defined but may be forward referenced:

```
; MASM 5.1:
    IF2
        .ERRNDEF    var
    ENDIF

; MASM 6.1:
        .ERRNDEF    var
```

For information on conditional directives, see “Conditional Directives,” Chapter 1.

IFDEF and IFNDEF with Forward-Referenced Identifiers

If you use a symbol name that has not yet been defined in an **IFDEF** or **IFNDEF** expression, MASM 6.1 returns FALSE for the **IFDEF** expression and TRUE for the **IFNDEF** expression. When **OPTION M510** is enabled, the assembler generates warning A6005:

```
expression condition may be pass-dependent
```

To resolve the warning, place the symbol definition before the conditional test.

Address Spans as Constants

The value of offsets calculated on the first assembly pass may not be the same as those calculated on later passes. Therefore, you should avoid comparisons with an address span, as in the following examples:

```
IF (OFFSET var1 - OFFSET var2) EQ 10
WHILE dx LT (OFFSET var1 - OFFSET var2)
REPEAT OFFSET var1 - OFFSET var2
```

However, the **DUP** operator allows such an expression as its count value. The assembler evaluates the **DUP** count on every pass, so even expressions involving forward references assemble correctly.

You can also use expressions containing span distances with the **.ERR** directives, since the assembler evaluates these directives after calculating all offsets:

```
.ERRE OFFSET var1 - OFFSET var2 - 10, <span incorrect>
```

.TYPE with Forward References

MASM 5.1 evaluates **.TYPE** on both assembly passes. This means it yields zero on the first pass and nonzero on the second pass, if applied to an expression that forward-references a symbol.

MASM 6.1 evaluates **.TYPE** only on the first assembly pass. As a result, if the operand references a symbol that has not yet been defined, **.TYPE** yields a value of zero. This means that **.TYPE**, if used in a conditional-assembly construction, may yield different results in MASM 6.1 than in MASM 5.1.

Obsolete Features No Longer Supported

The following two features are no longer supported by MASM 6.1. Because both are obscure features provided by early versions of the assembler, they probably do not affect your MASM 5.1 code.

The ESC Instruction

MASM 6.1 no longer supports the **ESC** instruction, which was used to send hand-coded commands to the coprocessor. Because MASM 6.1 recognizes and assembles the full set of coprocessor mnemonics, the **ESC** instruction is not necessary. Using the **ESC** instruction generates error A2205:

```
ESC instruction is obsolete: ignored
```

To update MASM 5.1 code, use the coprocessor instructions instead of **ESC**.

The MSFLOAT Binary Format

MASM 6.1 does not support the **.MSFLOAT** directive, which provided the Microsoft Binary Format (MSB) for floating-point numbers in variable initializers. Using the **.MSFLOAT** directive generates error A2204:

```
.MSFLOAT directive is obsolete: ignored
```

Use IEEE format or, if MSB format is necessary, initialize variables with hexadecimal values. See “Storing Numbers in Floating-Point Format” in Chapter 6.

Using the OPTION Directive

The **OPTION** directive lets you control compatibility with MASM 5.1 code. This section explains the differences in MASM 5.1 and MASM 6.1 behavior that the **OPTION** directive can influence.

The **OPTION M510** directive (or /Zm command-line option) initiates all aspects of 5.1 compatibility mode. You can select from among specific characteristics of MASM 5.1 behavior with the **OPTION** arguments discussed in following sections. Each section also explains how to revise your code if you want to remove **OPTION** directives from your MASM 5.1 code.

Note If your code includes both **.MODEL** and **OPTION M510**, the **OPTION M510** statement must appear first. Wherever this appendix suggests using **OPTION M510** in your code, you can set the /Zm command-line option instead.

OPTION M510

This section discusses the **M510** argument to the **OPTION** directive, which selects the MASM 5.1 compatibility mode. In this mode, MASM 6.1 implements MASM 5.1 behavior relating to macros, offsets, scope of code labels, structures, identifier names, identifier case, and other behaviors.

The **OPTION M510** directive automatically sets the following:

```
OPTION OLDSTRUCTS       ; MASM 5.1 structures
OPTION OLDMACROS        ; MASM 5.1 macros
OPTION DOTNAME          ; Identifiers may begin with a dot (.)
OPTION SETIF2:TRUE      ; Two-pass code activates on every pass
```

If you do not have a **.386**, **386P .486**, or **486P** directive in your module, then **OPTION M510** adds:

```
OPTION EXPR16           ; 16-bit expression precision
                        ;   See "OPTION EXPR16," following
```

If you do not have a **.MODEL** directive in your module, **OPTION M510** adds:

```
OPTION OFFSET:SEGMENT   ; OFFSET operator defaults to
                        ;   segment-relative
                        ;   See "OPTION OFFSET," following
```

If you do not have a **.MODEL** directive with a language specifier in your module, **OPTION M510** also adds:

```
OPTION NOSCOPED         ; Code labels are not local inside
                        ;   procedures
                        ;   See "OPTION NOSCOPED," following
OPTION PROC:PRIVATE     ; Labels defined with PROC are not
                        ;   public by default
                        ;   See "OPTION PROC," following
```

If you want to remove **OPTION M510** from your code (or /Zm from the command line), add the **OPTION** directive arguments to your module according to the conditions stated earlier.

There may be compatibility issues affecting your code that are supported under **OPTION M510**, but are not covered by the other **OPTION** directive arguments. Once you have modified your source code so it no longer requires behavior supported by **OPTION M510**, you can replace **OPTION M510** with other **OPTION** directive arguments. These compatibility issues are discussed in following sections.

Once you have replaced **OPTION M510** with other forms of the **OPTION** directive and your code works correctly, try removing the **OPTION** directives, one at a time. Make appropriate source modifications as necessary, until your code uses only MASM 6.1 defaults.

Reserved Keywords Dependent on CPU Mode with OPTION M510

With **OPTION M510**, keywords and instructions not available in the current CPU mode (such as **ENTER** under **.8086**) are not treated as keywords. This also means the **USE32**, **FLAT**, **FAR32**, and **NEAR32** segment types and the 80386/486 registers are not keywords with a processor selection less than **.386**.

If you remove **OPTION M510**, any reserved word used as an identifier generates a syntax error. You can either rename the identifiers or use **OPTION NOKEYWORD**. For more information on **OPTION NOKEYWORD**, see “OPTION NOKEYWORD,” later in this appendix.

Invalid Use of Instruction Prefixes with OPTION M510

Code without **OPTION M510** generates errors for all invalid uses of the instruction prefixes. **OPTION M510** suppresses some of these errors to match MASM 5.1 behavior. MASM 5.1 does not check for illegal usage of the instruction prefixes **LOCK**, **REP**, **REPE**, **REPZ**, **REPNE**, and **REPNZ**.

Illegal usage of these prefixes results in error A2068:

```
instruction prefix not allowed
```

For more information on these instruction prefixes, see "Overview of String Instructions" in Chapter 5. See also "Bug Fixes from MASM 5.1," earlier in this appendix.

Size of Constant Operands with OPTION M510

In MASM 5.1, a large constant value that can fit only in the processor's default word (4 bytes for **.386** and **.486**, 2 bytes otherwise) is assigned a size attribute of the default word size. The value of the constant affects the number of bytes changed by the instruction. For example,

```
; Legal only with OPTION M510
        mov     [bx], 0100h
```

is legal in **OPTION M510** mode. Since `0100h` cannot fit in a byte, the assembler interprets the value as a word.

Without **OPTION M510**, the assembler never assigns a size automatically. You must state it explicitly with the **PTR** operator, as shown in the following example:

```
; Without OPTION M510
        mov     [bx], WORD PTR 0100h
```

Code Labels when Defining Data with OPTION M510

MASM 5.1 allows a code label definition in a data definition statement if that statement does not also define a data label. MASM 6.1 also allows such definitions if **OPTION M510** is enabled; otherwise it is illegal.

```
; Legal only with OPTION M510
MyCodeLabel:    DW      0
```

SEG Operator with OPTION M510

In MASM 5.1, the **SEG** operator returns a label's segment address unless the frame is explicitly specified, in which case it returns the segment address of the frame. A statement such as `SEG DGROUP:var` always returns `DGROUP`, whereas `SEG var` always returns the segment address of `var`. **OPTION M510** forces this same behavior in MASM 6.1.

If you do not use **OPTION M510**, the behavior of the **SEG** operator is determined by the **OPTION OFFSET** directive, as described in "OPTION OFFSET," later in this appendix.

In MASM 6.1, the value returned by the **SEG** operator applied to a nonexternal variable depends on compatibility mode:

- Without **OPTION M510**, **SEG** returns the address of the frame (the segment, group, or the value assumed to the segment register) if one has been explicitly set.
- With **OPTION M510**, **SEG** returns the group if one has been specified. In the absence of a defined group, **SEG** returns the segment where the variable is defined.

Expression Evaluation with OPTION M510

By default, MASM 6.1 changes the way expressions are evaluated. In MASM 5.1,

```
var-2[bx]
```

is parsed as

```
(var-2)[bx]
```

Without **OPTION M510**, you must rewrite the statement, since the assembler parses it as

```
var-(2[bx])
```

which generates an error.

Length and Size of Labels with OPTION M510

With **OPTION M510**, you can apply the **LENGTH** and **SIZE** operators to any label. For a code label, **SIZE** returns a value of 0FFFFh for **NEAR** and 0FFFEh for **FAR**. **LENGTH** always returns a value of 1. For strings, **SIZE** and **LENGTH** both return 1.

Without **OPTION M510**, **SIZE** returns values of 0FF01h, 0FF02h, 0FF04h, 0FF05h, and 0FF06h for **SHORT**, **NEAR16**, **NEAR32**, **FAR16**, and **FAR32** labels, respectively. **LENGTH** returns 1 except when used with **DUP**, in which case it returns the outermost count. For arrays initialized with **DUP**, **SIZE** returns the length multiplied by the size of the type.

The **LENGTHOF** and **SIZEOF** operators in MASM 6.1 handle arrays much more consistently. These operators return the number of data items and the number of bytes in an initializer. For a description of **SIZEOF** and **LENGTHOF**, see the following sections in Chapter 5: “Declaring and Referencing Arrays,” “Declaring and Initializing Strings,” “Defining Structure and Union Variables,” and “Defining Record Variables.”

Comparing Types Using EQ and NE with OPTION M510

With **OPTION M510**, the assembler converts types to a constant value before comparisons with **EQ** and **NE**. Code types are converted to values of 0FFFFh (near) or 0FFFEh (far). If **OPTION M510** is not enabled, the assembler converts types to constants only when comparing them with constants. Thus, MASM 6.1 recognizes only equivalent qualified types as equal expressions.

For existing MASM 5.1 code, these distinctions affect only the use of the **TYPE** operator in conjunction with **EQ** and **NE**. The following example illustrates how the assembler compares types with and without compatibility mode:

```
MYSTRUCT        STRUC
  f1            DB      0
  f2            DB      0
MYSTRUCT        ENDS

; With OPTION M510

val     =       (TYPE MYSTRUCT) EQ WORD    ; True: 2 EQ 2
val     =       2 EQ WORD                  ; True: 2 EQ 2
val     =       WORD EQ WORD               ; True: 2 EQ 2
val     =       SWORD EQ WORD              ; True: 2 EQ 2

; Without OPTION M510

val     =       (TYPE MYSTRUCT) EQ WORD    ; False: MyStruct NE WORD
val     =       2 EQ WORD                  ; True:  2 EQ 2
val     =       WORD EQ WORD               ; True:  WORD EQ WORD
val     =       SWORD EQ WORD              ; False: SWORD NE WORD
```

Use of Constant and PTR as a Type with OPTION M510

You can use a constant as the left operand to **PTR** in compatibility mode. Otherwise, you must use a type expression. With **OPTION M510**, a constant must have a value of 1 (**BYTE**), 2 (**WORD**), 4 (**DWORD**), 6 (**FWORD**), 8 (**QWORD**) or 10 (**TBYTE**). The assembler treats the constant as the parenthesized type. Note that the **TYPE** operator yields a type expression, but the **SIZE** operator yields a constant.

```
; With OPTION M510

MyData  DW      0

        mov     WORD PTR [bx], 10           ; Legal
        mov     (TYPE MyData) PTR [bx], 10  ; Legal
        mov     (SIZE MyData) PTR [bx], 10  ; Legal
        mov     2 ptr [bx], 10              ; Legal

; Without OPTION M510

MyData  WORD    0

        mov     WORD PTR [bx], 10           ; Legal
        mov     (TYPE MyData) PTR [bx], 10  ; Legal
;       mov     (SIZE MyData) PTR [bx], 10  ; Illegal
;       mov     2 PTR [bx], 10              ; Illegal
```

Structure Type Cast on Expressions with OPTION M510

In compatibility mode, use the **PTR** operator to type-cast a constant to a structure type. This is most often done in data initializers to affect the CodeView information of the data label. Without **OPTION M510**, the assembler generates an error.

```
MYSTRC  STRUC
  f1    DB      0
MYSTRC  ENDS

MyPtr   DW      MYSTRC PTR 0    ; Illegal without OPTION M510
```

In MASM 6.1, the initializer type does not influence CodeView's type information.

Hidden Coercion of OFFSET Expression Size with OPTION M510

When programming for the 80386 or 80486, the size of an **OFFSET** expression can be 2 bytes for a symbol in a **USE16** segment, or 4 bytes for a symbol in a **USE32** or **FLAT** segment. With **OPTION M510**, you can use a 32-bit **OFFSET** expression in a 16-bit context. Without **OPTION M510**, you must use the **LOWWORD** operator to convert the offset size.

```
.386

; With OPTION M510

seg32   SEGMENT USE32
MyLabel WORD    0
seg32   ENDS

seg16   SEGMENT USE16 'code'                 ; With OPTION M510:
        mov     ax,  OFFSET MyLabel          ; Legal
        mov     ax,  LOWWORD OFFSET MyLabel  ; Legal
        mov     eax, OFFSET MyLabel          ; Legal
seg16   ENDS


; Without OPTION M510

seg32   SEGMENT USE32
MyLabel WORD    0
seg32   ENDS

seg16   SEGMENT USE16 'code'                ; Without OPTION M510:
;       mov     ax,  OFFSET MyLabel         ; Illegal
        mov     ax,  LOWWORD offset MyLabel ; Legal
        mov     eax, OFFSET MyLabel         ; Legal
seg16   ENDS
```

Specifying Radixes with OPTION M510

If the current radix in your code is greater than 10 decimal, MASM 6.1 allows the radix specifiers **B** (binary) and **D** (decimal) only in compatibility mode. You must change **B** to **Y** for binary, and **D** to **T** for decimal, since both **B** and **D** are legitimate hexadecimal values, making numbers such as 12D ambiguous. If you want to keep **B** and **D** as radix specifiers when the current radix is greater than 10, you must specify **OPTION M510**. For more information about radixes, see "Integer Constants and Constant Expressions" in Chapter 1.

Naming Conventions with OPTION M510

By default, MASM 5.1 does not write the names of public variables in uppercase to the object file, even when a language type of **PASCAL, FORTRAN,** or **BASIC** is specified.

Unless you use **OPTION M510,** these language types in MASM 6.1 write identifier names in uppercase, even with the /Cp or /Cx command-line options. When you link with /NOI, case must match in the object files to resolve externals.

Length Significance of Symbol Names with OPTION M510

With MASM 5.1, only the first 31 characters of a symbol name are considered significant, and only the first 31 characters of a public or external symbol name are placed in the object file.

Without **OPTION M510**, the entire name is considered significant. The maximum number of characters placed in the object file is controlled with the /H*number* command-line option, with a default of 247 (the maximum length of an identifier in MASM 6.1).

String Defaults in Structure Variables with OPTION M510

In compatibility mode, a constant initializer can override a structure field initialized with a string value. Without **OPTION M510**, only another string or a list can override a string initializer. To update your code, surround the constant override value with angle brackets or curly braces to indicate a list with one element.

```
MTSTRUCT        STRUCT
MyString        BYTE        "This is a string"
MTSTRUCT        ENDS

; With OPTION M510

MyInst          MTSTRUCT    <0>

; Without OPTION M510, either of these statements is correct

MyInst          MTSTRUCT    <<0>>
MyInst          MTSTRUCT    {<0>}
```

Effects of the ? Initializer in Data Definitions with OPTION M510

As described in “Declaring and Initializing Strings” in Chapter 5, the assembler treats the **?** initializer as either zero or as an unspecified value. In compatibility mode, however, the assembler always treats the **?** initializer as zero unless it is used with the **DUP** operator. In this case, the assembler allocates space, but does not initialize it with any value.

Current Address Operator with OPTION M510

In compatibility mode, the current address operator (**$**) applied to a structure returns the offset of the first byte of the structure. When **OPTION M510** is not enabled, **$** returns the offset of the current field in the structure.

Segment Association for FAR Externals with OPTION M510

In MASM 5.1, you must place an **EXTRN** directive for a variable in the same segment that holds the variable. For far data, this often entails opening and closing a segment just to place the **EXTRN** statement.

MASM 6.1 offers much greater flexibility in where **EXTERN** and **EXTERNDEF** statements can appear, as described in "Positioning External Declarations" in Chapter 8. However, in compatibility mode, MASM 6.1 emulates the behavior of MASM 5.1.

Defining Aliases Using EQU with OPTION M510

In MASM 5.1, you can equate one symbol with another. These equates are called "aliases."

Unless you specify **OPTION M510**, MASM 6.1 does not allow aliases defined with **EQU**. An immediate expression or text must appear as the right operand of an **EQU** directive. Change aliases to use the **TEXTEQU** directive, described in "Text Macros" in Chapter 9. This change may cause an expression to evaluate differently.

The following examples illustrate the differences between MASM 5.1 code, MASM 6.1 code with **OPTION M510**, and MASM 6.1 code without **OPTION M510**:

```
; MASM 5.1 code
var1    EQU     3
var2    EQU     var1    ; var2 taken as an alias
                        ; var2 references var1 anywhere var2 is
                        ;   used as a symbol

; MASM 6.1 with OPTION M510
var1    EQU     3
var2    EQU     var1    ; var2 taken as a var2 EQU <var1>
                        ; var2 substituted for var1 whenever
                        ;   text macros substituted

; MASM 6.1 without OPTION M510
var1    EQU     3
var2    EQU     var1    ; Treated as var2 EQU 3
```

Difference in Text Macro Expansions with OPTION M510

MASM 6.1 recursively expands text macros used as values, whereas MASM 5.1 simply replaces the text macro with its value. The following example illustrates the difference:

```
; With OPTION M510

tm1     EQU     <contains tm2>
tm2     EQU     <value>

tm3     CATSTR  tm1             ; == <contains tm2>

; Without OPTION M510

tm3     CATSTR  tm1             ; == <contains value>
```

Conditional Directives and Missing Operands with OPTION M510

MASM 5.1 considers a missing argument to be a zero. MASM 6.1 requires an argument unless **OPTION M510** is enabled.

OPTION OLDSTRUCTS

This section describes changes in MASM 6.1 that apply to structures. With **OPTION OLDSTRUCTS** or **OPTION M510**:

- You can use plus operator (**+**) in structure field references.
- Labels and structure field names cannot have the same name with **OPTION OLDSTRUCTS**.

Plus Operator Not Allowed with Structures

By default, each reference to structure member names must use the dot operator (**.**) to separate the structure variable name from the field name. You cannot use the dot operator as the plus operator (**+**) or vice versa.

To convert your code so that it does not need **OPTION OLDSTRUCTS**:

- Qualify all structure field references.
- Change all uses of the dot operator (**.**) that occur outside of structure references to use the plus operator (**+**).

If you remove **OPTION OLDSTRUCTS** from your code, the assembler generates errors for all lines requiring change. Using the dot operator in any context other than for a structure field results in error A2166:

```
structure field expected
```

Unqualified structure references result in error A2006:

```
undefined symbol : identifier
```

The following example illustrates how to change MASM 5.1 code from the old structure references to the new type in MASM 6.1:

```
; OPTION OLDSTRUCTS (simulates MASM 5.1)
structname      STRUC
a               BYTE  ?
b               WORD  ?
structname      ENDS

structinstance  structname <>

        mov     ax, [bx].b              ; This code assembles
        mov     al, structinstance.a    ;   correctly only with
        mov     ax, [bx].4              ;   OPTION OLDSTRUCTS
                                        ;   or OPTION M510


; OPTION NOOLDSTRUCTS (the MASM 6.1 default)
structname      STRUCT
a               BYTE  ?
b               WORD  ?
structname      ENDS

structinstance  structname <>

        mov     ax, [bx].structname.b   ; Add qualifying type
        mov     al, structinstance.a    ; No change needed
        mov     ax, [bx]+4              ; Change dot to plus

; Alternative methods in MASM 6.1
; Either this:
        ASSUME  bx:PTR structname
        mov     ax, [bx]
; or this:
        mov     ax, (structname PTR[bx]).b
```

Duplicate Structure Field Names

With the default, **OPTION NOOLDSTRUCTS**, label and structure field names may have the same name. With **OPTION OLDSTRUCTS** (the MASM 5.1 default), labels and structure fields cannot have the same name. For more information, see “Structures and Unions” in Chapter 5.

OPTION OLDMACROS

This section describes how MASM 5.1 and 6.1 differ in their handling of macros. Without **OPTION OLDMACROS** or **OPTION M510**, MASM 6.1 changes the behavior of macros in several ways. If you want the MASM 5.1 macro behavior, add **OPTION OLDMACROS** or **OPTION M510** to your MASM 5.1 code.

Separating Macro Arguments with Commas

MASM 5.1 allows white spaces or commas to separate arguments to macros. MASM 6.1 with **OPTION NOOLDMACROS** (the default) requires commas between arguments. For example, in the macro call

```
MyMacro  var1 var2 var3, var4
```

OPTION OLDMACROS causes the assembler to treat all four items as separate arguments. With **OPTION NOOLDMACROS**, the assembler treats

```
var1 var2 var3
```

as one argument, since the items are not separated with commas. To convert your macro code, replace spaces between macro arguments with a single comma.

New Behavior with Ampersands in Macros

The default **OPTION NOOLDMACROS** causes the assembler to interpret ampersands (**&**) within a macro differently than does MASM 5.1. MASM 5.1 requires one ampersand for each level of macro nesting. **OPTION OLDMACROS** emulates this behavior.

Without **OPTION OLDMACROS**, MASM 6.1 removes ampersands only once no matter how deeply nested the macro. To update your MASM 5.1 macros, follow this simple rule: replace every sequence of ampersands with a single ampersand. The only exception is when macro parameters immediately precede and follow the ampersand, and both require substitution. In this case, use two ampersands. For a description of the new rules, see “Substitution Operator” in Chapter 9.

This example shows how to update a MASM 5.1 macro:

```
; OPTION OLDMACROS (emulates MASM 5.1 behavior)

createNames     macro   arg
    irp         tail, <Next, Last>
        irp     num,  <1, 2>
        ; Define more names of the form: abcNext1?
arg&&tail&&&num&&&?   label   BYTE
        ENDM
    ENDM
ENDM

; OPTION NOOLDMACROS (the MASM 6.1 default)

createNames     macro   arg
    for         tail, <Next, Last>  ; FOR is the MASM 6.1
        for     num,  <1, 2>        ;   synonym for irp
        ; Define more names of the form: abcNext1?
arg&&tail&&num&?   label   BYTE
        ENDM
    ENDM
ENDM
```

OPTION DOTNAME

MASM 5.1 allows names of identifiers to begin with a period. The MASM 6.1 default is **OPTION NODOTNAME**. Adding **OPTION DOTNAME** to your code enables the MASM 5.1 behavior.

If you don't want to use this directive in your source code, rename the identifiers whose names begin with a period.

OPTION EXPR16

MASM 5.1 treats expressions as 16-bit words if you do not specify **.386** or **.386P** directives. MASM 6.1 by default treats expressions as 32-bit words, regardless of the CPU type. You can force MASM 6.1 to use the smaller expression size with the **OPTION EXPR16** statement.

Unless your MASM 5.1 code specifies **.386** or **.386P**, **OPTION M510** also sets 16-bit expression size. You can selectively disable this by following **OPTION M510** with the **OPTION EXPR32** directive, which sets the size back to 32 bits. You cannot have both **OPTION EXPR32** and **OPTION EXPR16** in your program.

It may not be easy to determine the effect of changing from 16-bit internal expression size to 32-bit size. In most cases, the 32-bit word size does not affect the MASM 5.1 code. However, problems may arise because of differences in intermediate values during evaluation of expressions. You can compare the files for differences by generating listing files with the /Fl and /Sa command-line options with and without **OPTION EXPR16.**

OPTION OFFSET

The information in this section is relevant only if your MASM 5.1 code does not use the **.MODEL** directive. With no **.MODEL**, MASM 5.1 computes offsets from the start of the segment, whereas MASM 6.1 computes offsets from the start of the group. (With **.MODEL**, MASM 5.1 also computes offsets from the start of the group.)

To force MASM 6.1 to emulate 5.1 behavior, specify either **OFFSET:SEGMENT** or **OPTION M510**. Both directives cause the assembler to compute offsets relative to the segment if you do not include **.MODEL**.

To selectively enable MASM 6.1 behavior, place the directive **OPTION OFFSET:GROUP** after **OPTION M510**. In this case, you should ensure each **OFFSET** statement has a segment override where appropriate. The following example shows how **OPTION OFFSET:SEGMENT** affects code written for MASM 5.1:

```
OPTION  OFFSET:SEGMENT
MyGroup GROUP   MySeg

MySeg   SEGMENT 'data'
MyLabel LABEL   BYTE
        DW      OFFSET MyLabel          ; Relative to MySeg
        DW      OFFSET MyGroup:MyLabel  ; Relative to MyGroup
        DW      OFFSET MySeg:MyLabel    ; Relative to MySeg
MySeg   ENDS
```

In the preceding example, the first **OFFSET** statement computes the offset of `MyLabel` relative to `MySeg`. Without **OFFSET:SEGMENT**, MASM 6.1 returns the offset relative to `MyGroup`. To maintain the correct behavior with **OFFSET:GROUP**, specify a segment override, as shown in the following. The other two **OFFSET** statements already include overrides, and so do not require modification.

```
OPTION  OFFSET:GROUP
MyGroup GROUP   MySeg

MySeg   SEGMENT 'data'
MyLabel LABEL   BYTE
        DW      OFFSET MySeg:MyLabel    ; Relative to MySeg
        DW      OFFSET MyGroup:MyLabel  ; Relative to MyGroup
        DW      OFFSET MySeg:MyLabel    ; Relative to MySeg
MySeg   ENDS
```

When not in compatibility mode, the **OPTION OFFSET** directive determines whether the **SEG** operator returns a value relative to the group or segment. With **OPTION M510**, **SEG** is always segment-relative by default, regardless of the current value of **OPTION OFFSET**.

OPTION NOSCOPED

The information in this section applies only if the **.MODEL** directive in your MASM 5.1 code does not specify a language type. Without a language type, MASM 5.1 assumes code labels in procedures have no “scope”—that is, the labels are not local to the procedure. When not in compatibility mode, MASM 6.1 always gives scope to code labels, even without a language type.

To force MASM 5.1 behavior, specify either **OPTION M510** or **OPTION NOSCOPED** in your code. To selectively enable MASM 6.1 behavior, place the directive **OPTION SCOPED** after **OPTION M510**.

To determine which labels require change, assemble the module without the **OPTION NOSCOPED** directive. For each reference to a label that is not local, the assembler generates error A2006:

```
undefined symbol : identifier
```

OPTION PROC

The information in this section applies only if the **.MODEL** directive in your MASM 5.1 code does not specify a language type. Without a language type, MASM 5.1 makes procedures private to the module. By default, MASM 6.1 makes procedures public. You can explicitly change the default visibility to private with either **OPTION M510**, **OPTION PROC:PRIVATE**, or **OPTION PROC:EXPORT**.

To selectively enable MASM 6.1 behavior, place the directive **OPTION PROC:PUBLIC** after **OPTION M510**. You can override the default by adding the **PUBLIC** or **PRIVATE** keyword to selected procedures. The following example shows how to change MASM 5.1 code to keep a procedure private:

```
; MASM 5.1 (OPTION PROC:PRIVATE)
MyProc PROC NEAR

; MASM 6.1 (OPTION PROC:PUBLIC)
MyProc PROC NEAR PRIVATE
```

This is necessary only to avoid naming conflicts between public names in multiple modules or libraries. The symbol table in a listing file shows the visibility (public, private, or export) of each procedure.

OPTION NOKEYWORD

MASM 6.1 has several new keywords that MASM 5.1 does not recognize as reserved. To resolve any conflicts, you can:

- Rename any offending symbols in your code.
- Selectively disable keywords with the **OPTION NOKEYWORD** directive.

The second option lets you retain the offending symbol names in your code by forcing MASM 6.1 to not recognize them as keywords. For example,

```
OPTION NOKEYWORD:<INVOKE STRUCT>
```

removes the keywords **INVOKE** and **STRUCT** from the assembler's list of reserved words. However, you cannot then use the keywords in their intended function, since the assembler no longer recognizes them.

The following list shows MASM 6.1 reserved words new since MASM 5.1:

.BREAK	**.UNTILCXZ**	**FRSTORD**
.CONTINUE	**.WHILE**	**FRSTORW**
.DOSSEG	**ADDR**	**FSAVED**
.ELSE	**ALIAS**	**FSAVEW**
.ELSEIF	**BSWAP**	**FSTENVD**
.ENDIF	**CARRY?**	**FSTENVW**
.ENDW	**CMPXCHG**	**GOTO**
.EXIT	**ECHO**	**HIGHWORD**
.IF	**EXTERN**	**INVD**
.LISTALL	**EXTERNDEF**	**INVLPG**
.LISTIF	**FAR16**	**INVOKE**
.LISTMACRO	**FAR32**	**IRETDF**
.LISTMACROALL	**FLAT**	**IRETF**
.NO87	**FLDENVD**	**LENGTHOF**
.NOCREF	**FLDENVW**	**LOOPD**
.NOLIST	**FNSAVED**	**LOOPED**
.NOLISTIF	**FNSAVEW**	**LOOPEW**
.NOLISTMACRO	**FNSTENVD**	**LOOPNED**
.REPEAT	**FNSTENVW**	**LOOPNEW**
.STARTUP	**FOR**	**LOOPNZD**
.UNTIL	**FORC**	**LOOPNZW**

LOOPW	**PUSHCONTEXT**	**SWORD**
LOOPZW	**PUSHD**	**SYSCALL**
LOWWORD	**PUSHW**	**TEXTEQU**
LROFFSET	**REAL10**	**TR3**
NEAR16	**REAL4**	**TR4**
NEAR32	**REAL8**	**TR5**
OPATTR	**REPEAT**	**TYPEDEF**
OPTION	**SBYTE**	**UNION**
OVERFLOW?	**SDWORD**	**VARARG**
PARITY?	**SIGN?**	**WBINVD**
POPAW	**SIZEOF**	**WHILE**
POPCONTEXT	**STDCALL**	**XADD**
PROTO	**STRUCT**	**ZERO?**
PUSHAW	**SUBTITLE**	

OPTION SETIF2

By default, MASM 6.1 does not recognize pass-dependent constructs. Both the **OPTION M510** and **OPTION SETIF2** statements force MASM 6.1 to handle MASM 5.1 constructs that activate on the second assembly pass, such as **.ERR2**, **IF2**, and **ELSEIF2**.

Invoke the option like this:

OPTION SETIF2: {TRUE | FALSE}

When set to **TRUE**, **OPTION SETIF2** forces all second-pass constructs to activate on every assembly pass. When set to **FALSE**, second-pass constructs do not activate on any pass. **OPTION M510** implies **OPTION SETIF2:TRUE**.

Changes to Instruction Encodings

MASM 6.1 contains changes to the encodings for several instructions. In some cases, the changes help optimize code size.

Coprocessor Instructions

For the 8087 coprocessor, MASM 5.1 adds an extra **NOP** before the no-wait versions of coprocessor instructions. MASM 6.1 does not. In the rare case that the missing **NOP** affects timing, insert **NOP**.

For the 80287 coprocessor or better, MASM 5.1 inserts **FWAIT** before certain instructions. MASM 6.1 does not prefix any 80287, 80387, or 80486 coprocessor instruction with **FWAIT**, except for wait forms of instructions that have a no-wait form.

RET Instruction

MASM 5.1 generates a 3-byte encoding for **RET**, **RETN**, or **RETF** instructions with an operand value of zero, unless the operand is an external absolute. In this case, MASM 5.1 ignores the parameter and generates a 1-byte encoding.

MASM 6.1 does the opposite. It ignores a zero operand for the return instructions and generates a 1-byte encoding, unless the operand is an external absolute. In this case, MASM 6.1 generates a 3-byte encoding.

Thus, you can suppress epilogue code in a procedure but still specify the default size for **RET** by coding the return as

```
        ret     0
```

Arithmetic Instructions

Versions 5.1 and 6.1 differ in the way they encode the arithmetic instructions **ADC, ADD**, **AND**, **CMP**, **OR**, **SUB**, **SBB**, and **XOR**, under the following conditions:

- The first operand is either AX or EAX.
- The second operand is a constant value between 0 and 127.

For the AX register, there is no size or speed difference between the two encodings. For the EAX register, the encoding in MASM 6.1 is 2 bytes smaller. The **OPTION NOSIGNEXTEND** directive forces the MASM 5.1 behavior for **AND**, **OR**,
and **XOR**.