micro
cosm

MCA-430 Assembler and Utilities

for Texas Instruments MSP430

Assembler, Linker,
Library Manager,
Object to Text Converter

User's Guide

DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment on any part of MicroCOSM Software Technologies. While the information contained herein is assumed to be accurate, MicroCOSM Software Technologies assumes no responsibility for any errors or omissions. In no event shall MicroCOSM Software Technologies, its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

COPYRIGHT NOTICE

For convenience, the short name 'MicroCOSM-ST' is used hereinafter to refer to MicroCOSM Software Technologies.

Preface

MCA-430 Assembler User's Guide provides information about MicroCOSM-ST Macro Assembler for Texas Instruments MSP430 microcontroller family.

The complete tool set available for the MSP430 also includes MicroCOSM-ST **MCC-430** C Compiler and several software/hardware tools supplied by Phyton, Inc., used for development, debugging and "burning" code into target microcontroller. All tools are integrated under Phyton **Project-430 IDE** but can be invoked from the command line as well.

This document contains detailed reference information about the following components:

- **MCA-430** Assembler
- **MCLINK** Linker and Utilities

For information on other components, please refer to the related documentation.

We believe this document thoroughly describes all issues related to MCA-430. If nevertheless after reading this manual you cannot get an answer to your question, please e-mail it to MicroCOSM-ST technical support service at *support@microcosm-st.com*.

For information about Phyton, Inc. products, please read the related documentation provided by Phyton, Inc. or visit *www.phyton.com*

Document Conventions

The following style conventions are used in this manual:

Style	Used for
`courier`	Source Assembler text; display on the console.
`parameter`	The actual value that you should enter to replace the parameter name in the program or the command line.
[`optional`]	Optional parameter that can be skipped in an assembler directive or instruction.
bold	Names of attributes and command line options. Also used in text to stress importance of a statement.
CAPITAL	Assembler operators and directives, predefined assembler constants and variables, address space names (memory types).

Document Overview

This guide is divided into following chapters:

Introduction, provides the summary information about the MCA-430 Assembler**,** the MCLINK Linker and Utilities, with a short usage example.

Chapter 1. Basic Conceptions, introduces basic conceptions such as address spaces, segments, modules, and symbolic names and their attributes. The latter allow managing allocation of code and data in available physical memory, checking operand type in instructions and detecting errors on assembly and linking stages.

Chapter 2. MCA-430 Assembler, contains an overview of the assembly language and a detailed description of all MCA-430 operators, directives and macro tools. An additional section covers issues related to the MSP430 target architecture support.

Chapter 3. MCLINK Linker, explains main functions of MCLINK Linker and describes stages of the linking process, such as resolving external references, establishing set of modules to be linked, linking segments and allocating them in address spaces, and binding the object code to physical addresses.

Chapter 4. MCLIB Librarian, describes library creation and management with the MCLIB Librarian.

Chapter 5. MCDUMP Utility, describes the MCDUMP Object-to-Text Converter.

Appendices, which contain summaries of Assembler directives, Assembler operators and variables, description of Assembler, MCLINK, MCLIB, and MCDUMP command line options, and description of Assembler, MCLINK, MCLIB, and MCDUMP diagnostic messages.

References

Assembly language programming assumes knowledge of the target processor architecture and the instruction set. For detailed information on the MSP430 microcontrollers, please refer to [1a-1c].

For additional information on development of mixed C/Assembler projects, see [2].

If you have not used Phyton/MicroCOSM-ST tool sets before, please read sections of [3] about running MicroCOSM-ST tools from Phyton Integrated Development Environment to learn how to work with the projects.

1a. MSP430x4xx Family. User's Guide. *2002 Texas Instruments SLAU56B.*
1b. MSP430x3xx Family. User's Guide. *2000 Texas Instruments SLAU012.*
1c. MSP430x1xx Family. User's Guide. *2001 Texas Instruments SLAU49A.*
2. MCC-430 C Compiler for MSP430. User's Guide. *2003 MicroCOSM-ST.*
3. Phyton IDE Online Help. *2003 Phyton, Inc.*

Contents

Introduction

This guide describes the following tools: the MCA-430 Assembler, MCLINK Linker, MCLIB Librarian, and MCDUMP Object-to-Text Converter.

MCA-430 Assembler is a powerful cross-assembler which translates the source text to relocatable object modules for linking. MCA-430 main features:

- Has an extensive set of directives and operators
- Strict operand type checking
- Powerful macro capabilities and repeating blocks
- Local labels and names in functions and macros
- Supports copying the code from the Flash/ROM to the RAM for execution
- Allows assigning C types to variables and functions

MCLINK Linker is a tool which links one or more relocatable object files produced by the MicroCOSM-ST Assembler or Compiler and, if necessary, libraries to create executable code for target microcontroller. MCLINK main features:

- Provides all required functionality to produce PROMable code
- Produces detailed debugging information for all entities in the user program
- Generates informative map-file with customizable contents
- Performs C type checking for variables and functions
- Supports 2 priority levels of library modules to extend the flexibility of library usage

MCLIB Librarian is a utility for creating and managing libraries. MCLIB is used to create libraries from the relocatable object file, as well as to add, list, delete, replace, extract or move modules to/from libraries, and change the type of modules contained in libraries to 'program', 'library', or 'low-priority'.

MCDUMP Object-to-Text Converter is a utility for converting object files, libraries, and executable files into readable text form.

Filename Extensions

The following filename extensions are used:

Extension	Contents and description
.MCA	Source code file; contains ASCII text, which is the input for the MCA-430 Assembler.
.MCO	Relocatable object file, generated by the MCA-430 Assembler or MCC-430 Compiler. Contains program code and control information. Usually, a set of the object files is the input for the MCLINK Linker.
.MCL	Library file generated by the MCLIB Librarian from the relocatable object files.
.LST	Listing file generated by the MCA-430 Assembler.
.MAP	Listing file generated by the MCLINK Linker to document the linking process.
.MCE	Executable MCE-file in MicroCOSM-ST/Phyton format, generated by the MCLINK Linker. This file contains executable code and debugging information and can be used for debugging with the Phyton IDE (see IDE Online Help).
.HEX	Absolute Intel HEX file generated by the MCLINK Linker for CODE memory.

.ZAX File with debugging information, in the ZAX-format, generated by the MCLINK Linker.

Usage Example

The following example demonstrates how to use MCA-430 and MCLINK to obtain a file with code which can be loaded in the microprocessor memory and/or used for debugging. The MCA-430 Assembler command line options are described in *Appendix C. MCA-430 Command Line Interface.* The MCLINK Linker command line options are described in *Appendix D. MCLINK Command Line Interface.* For information on Phyton IDE, please refer to *IDE Online Help*.

1) Create a file named MYPROG.MCA with the following contents:

```
    .RSEG _MyData, data
    res    .dsi
    .public res

    .RSEG STACK, data
    .align 1
           .ds 40h              ;reserve 64 bytes for stack

    .RSEG _MyConst, code
    sernum .dcb '2613'

    .ASEG _Reset,code
    .org 0FFFEh
           .dcw START

    .RSEG _MyCode,code
START:
;Initialize SP
    MOV  #.sfe STACK, R5
    ADD  #1, R5
    MOV  R5, SP

    MOV  #2,res
    PUSH res
    ADD  res, 0(SP)
    POP  res                    ;res = 4
STOP:
    JMP  $
.END
```

2) Assemble your source file to a relocatable object file using the MCA-430:

```
MCA430 –l –d MYPROG.MCA
```

The following files will be created:

MYPROG.MCO – Relocatable object file (input for the Linker)
MYPROG.LST – Listing file

3) Use the MCLINK Linker to obtain the resulting executable file:

```
MCLINK -F MI -m MYPROG.MCO
```

The following files will be created:

MYPROG.MCE – Executable file in the MicroCOSM-ST/Phyton format
MYPROG.HEX – Absolute object file for CODE memory in the Intel HEX-format
MYPROG.MAP – Map file

Chapter 1. Basic Conceptions

This chapter introduces such basic conceptions as address spaces, segments, modules, symbolic names, and their attributes. They allow managing allocation of code and data in available physical memory, checking operand type in instructions and detecting errors at assembly and link stages.

1.1. Introduction

The process of development of software for embedded systems undergoes several stages: writing the source texts of a program; assembling/compiling the source texts into relocatable object code; and, linking the set of relocatable object modules to obtain executable file for debugging and/or a file with absolute object code for the target system.

Typically, programs are developed based on the modular approach, where functionally or logically complete parts of the program are arranged as separate small modules. Small subprograms and modules are easier to comprehend, design, and test than large programs. Modular approach has another benefit – the opportunity to reuse in new projects documented and bug-free modules with necessary functions (e.g., functions for interacting with peripherals).

Selected relocatable object modules can be used to build *libraries*. Furthermore, some modules in a library can be assigned the 'library' attribute. At link time, the modules with the 'library' attribute are included in the resulting executable file only if the program contains references to code or data in those modules.

Pure assembler projects contain only assembler modules. In mixed Assembler/C projects some functions are written in C and some in assembler. In this case, assembly functions can be called from C modules and vice versa.

Assembler modules contain processor op-code mnemonics followed by the instruction operands, which are commonly represented by *symbolic names*. Symbolic names in MCA-430 are assigned a set of attributes such as 'allocation', 'type', 'operand type', and 'relocatability', that enable the Assembler and Linker to check whether these names are used correctly as operands in instructions. Code or data are placed in *segments*. The Linker places segments in the *address spaces* that correspond to various types of physical memory in the embedded system (ROM, RAM, Flash, EEPROM, etc).

The MCLINK Linker links input object modules, adding the library modules, if necessary. It can generate the executable files in the following formats:

- Executable file with code and debugging information in the MicroCOSM-ST/Phyton format (*.MCE)
- Absolute object file in the Intel HEX format for CODE memory (*.HEX)

These files can be used for debugging in the Integrated Development Environments for embedded systems. MCE-files contain additional debugging information, which provides extra functionality when debugging is made in Phyton IDE.

1.2. Modules

The program being developed is often divided into functional units or *modules*, each accomplishing certain tasks. Every module can contain both executable code and necessary data. The data and labels that need to be visible in other modules are declared as public. Accordingly, modules can contain *external references* designating names declared as public in other modules.

At the first – assembling/compiling – stage, source text in every module is processed individually by the Assembler/Compiler and relocatable object modules are generated. At this stage, external references are not resolved. At the second stage, all relocatable modules are processed by the Linker; the external references are resolved, addresses are adjusted, and all relocatable modules are merged into the resulting absolute object file. Such method of processing of the source texts is called *separate compilation.*

It is possible to use the same symbolic names in different modules to identify different data or labels (so-called name hiding). For example, a variable COUNTER may be used in one module as a counter of bytes received through serial communication link, and also may be used in another module for counting the bytes transmitted. In this case, variables identified by COUNTER are different in different modules. These variables are sometimes called local variables in a module.

Due to separate compilation, the whole program does not need not to be re-compiled after changes are introduced in one or several modules. It is necessary to recompile only the modules that were changed. After recompilation, all modules again need to be processed by the Linker. This saves time when a large program is developed.

If a program is divided to functionally independent and documented modules, the following advantages are gained:

- Executable code and data is hidden inside the modules, which reduces the possibility of "induced" errors.
- If it is necessary to introduce changes into a structured program, the scope for making the corrections is limited to single modules, not the whole source text.
- The modules can be used in a new project.

In a program written in a high-level language (e.g. C) every source text file is a module.

In assembler program the source text within one file can be divided into several modules using special Assembler directives. Thus, an object file generated by Assembler may contain multiple modules.

There are three types of MCA-430 modules: *program*, *library*, and *low-priority library* modules. While the program modules are always linked to the resulting file unconditionally, the library modules are linked only when they contain public names that can resolve external references from other modules. Low-priority modules are linked last and only if they can resolve references yet unresolved by the usual library modules. For detailed description of type of modules supported by the MCA-430 Assembler, see *Chapter 4. MCLIB Librarian.*

1.3. Address Spaces

The architecture of many embedded systems provides access to separated *address spaces*, which may have specific layouts. The MSP430 design on the contrary, is notable for its all memory types mapped into the common 64K address area. All of the MSP430 microprocessors have the RAM for storing modifiable data and either the ROM (or One Time Programmable memory, OTP) or Flash (or Multiple Time Programmable, MTP) memory for storing program code and constant/fixed data. Various chip versions can have different sizes of RAM and ROM/Flash memory.

Two address spaces – CODE (for ROM or Flash memory) and DATA (for RAM) – represent the MSP430 memory layout. The MCLINK Linker places segments with code and data at specific addresses in these address spaces using the 'allocation' attribute.

1.3.1. Address Space 'Allocation' Attribute

Each address space, as well as each segment, has an attribute called *allocation.* The Linker sets up two address spaces with allocations **code** and **data,** which are listed below:

Address Space Name	Maximum Allowed Address Range	Allocation	Default Address Range
CODE	200h…0FFFFh	**code**	0FC00h..0FFFFh
DATA	000h…0FBFFh	**data**	0..280h (200h + 128 locations)

The default address ranges presume the configuration with 1K ROM and 128 byte RAM.

CODE address space is associated with the physical ROM or Flash (whichever is present). This address space is intended for storing code and constant data. Note, that the addresses 0FFE0h–0FFFFh should be reserved for interrupt vectors. See for details *Setting Interrupt Vectors* section in Chapter 2.

DATA address space is associated with the physical RAM that is used to store data modified at run time. Note that the 0-1FFh address range is used to map SFRs and peripheral module registers. This address range is reserved automatically when you include the appropriate file containing the SFRs and peripheral modules definitions. See for details *SFRs and Peripheral Module Registers* section in Chapter 2.

Changing the Size of Address Spaces
The sizes of address spaces can be redefined to correspond to the actual memory layout of the selected MSP430 device. You can specify/change the address range for each address space using the **-A** Linker option. The address ranges of the redefined address spaces must not overlap; otherwise, the Linker will produce an error message. If the sizes of address spaces are not specified with the **-A** Linker option, the Linker uses the default address ranges mentioned above. Banking, or declaration of new address spaces, is not allowed either for **code** or for **data** allocation.

In IDE the adjustment of the address space boundaries is made automatically after selection of the required MSP430 device. To see how to set custom memory sizes in the IDE, please refer to *IDE Online Help.*

If you run the MCLINK from the command line, use one of the supplied Linker response files . You can also generate the response file for MCLINK from IDE and use it as a template. Please refer to *IDE Online Help.*

The following example demonstrates how to manually redefine address spaces. Assume your system is equipped with 256 bytes of RAM and 8K of ROM (e.g., the MSP430C1331). Then, you should add the following options to the Linker response file (or the command line) to specify this memory layout:

```
-A (data)DATA(0h-2FFh)    #modifiable data will be placed
                          #in the range 200-2FFh (0-1FFh is
                          #reserved for SFR's and
                          #peripheral modules)

-A (code)CODE(DFFFh-FFFFh)#code and fixed data will be placed
                          #in the range DFFFh-FFFFh
```

1.4. Segments

Segment is a separate memory area for code or data within an address space. Each segment has a name assigned by the user. A segment can be either absolute or relocatable/overlay, and has 'allocation' attribute. In a module, code and data may be allocated in several segments. The Linker processes relocatable segments defined with the same name in different modules as parts of the same segment, i.e. combines them into one contiguous block.

1.4.1. Segment Types

There are three types of segments: *absolute*, *relocatable*, and *overlay* segments. The segment type is specified by one of three directives .ASEG, .RSEG, or .OSEG used for segment declaration.

Code or data located in absolute segments are bound to absolute physical addresses already in the source text. When a name is defined in an absolute segment, the values assigned to this name will not be modified by the Linker.

The addresses of relocatable and overlay segments are adjusted by the Linker. The Assembler simply calculates the addresses of code and data based on the displacement from the origin of the fragment of segment located in a given module. At link time, the total size of the segment is calculated and the segment is allocated starting from a particular physical address. Given this starting address the absolute addresses for code and data are computed.

Binding of code and data located in the overlay segments is also performed at link time only. The Assembler determines the location of code or data in a similar way to relocatable segments. The segment declared in the input file as overlay is processed by the Linker similar to relocatable segment, except for the following: (1) the Linker sets the size of an overlay segment equal to the size of the largest fragment of this segment; (2) the Linker places all fragments of a particular overlay segment starting from the same physical address.

Note, when an absolute segment is declared, the Assembler generates an information in the relocatable object file to prevent the Linker from placing relocatable or overlay segments in that address range. Thus, the user does not need to keep relocatable or overlay segments from overlapping with the absolute ones. However, if two or more absolute segments are declared within one address space and their address ranges intersect then it is the user's responsibility to track overlapping of the fragments of the code and to prevent that.

1.4.2. Segment 'Allocation' Attribute

If a new segment is declared in a module, the 'allocation' attribute must be specified in the declaration statement. Code and initial (non-modifiable) data should be placed in code segments (with the **code** allocation attribute). Memory locations for modifiable data should be reserved in data segments. The following are the 'allocation' attributes and the allowable access methods:

Memory Type	Allocation Attribute	Allowable Access Method
code memory (ROM/Flash)	**code**	Fetching code, word/byte instructions
data memory (RAM)	**data**	Word/byte instructions, fetching code*

* In data segment declared with extended segment declaration directive format. See section *Segment Declaration and Selection* in *Assembler Directives, Chapter 2*.

Knowing the memory type of each operand the Assembler can check that the data is accessed correctly. The 'allocation' attribute is assigned to every operand in the MCA-430, which is achieved by the following mechanism: code and data are placed in segments and the 'allocation' attribute of the segment is automatically assigned to every identifier declared in that segment.

1.4.3. Allocating Segments in Address Spaces

The Linker can automatically allocate segments in the address spaces. The segments with allocation **data** fall into the DATA address space, i.e. in the RAM. The segments with allocation **code** fall into the CODE address space, i.e. in the ROM/Flash.

If a **data** segment is declared using the extended format, the Linker places all code from that segment in the CODE address space, whereas addresses of all names and labels are adjusted to the DATA address space. See for details *Segment Declaration and Selection* section in Chapter 2.

1.5. Symbolic Names

1.5.1. Local, External and Public Names

Names defined within a module can be divided into two categories: *local* names, which should only be used in that particular module, and *public* names specified using the .PUBLIC directive. References to public names defined in other modules are supported. Such references are called external references. External names used for external references should be declared with the .EXTRNx directives. Public names can be referenced from other modules, where they will be external names, and therefore public names must be unique in the project.

The same local names can be used to identify various objects in different modules, without conflicting. Furthermore, MCA-430 allows using local names/labels in functions, defined with the .FUNC directive, and local names in macros .

For details, see sections *Functions* and *Macro Tools* in *Chapter 2. MCA-430 Assembler*.

1.5.2. 'Relocatability' Attribute

The *relocatability* attribute tells the Linker whether it needs to determine the value of a name at link time. This value is normally an address in the memory. If the name type is absolute – the 'relocatability' attribute is ABS – then its value is evaluated at assembly time and the Linker does not need to evaluate it. If the name is relocatable – the 'relocatability' attribute is REL or EXT – then its value is unknown at assembly time and it must be evaluated by the Linker.

The relocatability attribute of the names defined in absolute segments and names of the numeric constants is ABS. Names defined in relocatable segments have the REL relocatability attribute. External names have the EXT relocatability attribute. Every expression, not only names, has relocatability attribute. For example, numeric constants have the ABS relocatability attribute.

See also: *Type Conversion and Checking in Expressions* section in *Chapter 2. MCA-430 Assembler*.

The relocatability attribute may accept the following values:

Value	Description
ABS	Absolute value
REL	Relocatable value
EXT	External value
SEG	Segment (for segments only)
SFB	Start address of a segment (result of the .SFB operator)
SFE	End address of a segment (result of the .SFE operator)

1.5.3. Name Types and 'Type' Attribute

Every symbolic name defined in a program has the *type* attribute. The following is a list of the available types:

Type	Numeric Value	MCA-430 Constant	Description
nothing	0	-	undefined type
void	2	.VOID	only for compatibility with C
char	4	.CHAR	8-bit, signed integer
unsigned char	5	.UCHAR	8-bit, unsigned integer
int	8	.INT	16-bit, signed integer
unsigned int	9	.UINT	16-bit, unsigned integer
long	10	.LONG	32-bit, signed integer
unsigned long	11	.ULONG	32-bit, unsigned integer
float	12	.FLOAT	32-bit, floating-point number

Assembler types are used for the following purposes:

- Information about the type of a variable is used in debugging. The Debugger, knowing the size and sign of a variable, will show the right value in the correct format upon request. For example, if a variable is defined with the .DSC directive, which makes it an 8-bit signed number, the Debugger will format and show its value exactly as an 8-bit signed number.
- Assigning of type attributes to names in assembly programs is necessary for linking assembler and C modules to one program.

Note: C allows deriving of new types from the basic types; new type may be a synonym of another type (**typedef**) or a pointer, array, structure, union, etc.

The Linker checks whether type attributes of public and external names coincide. If, for instance, the variable VAR in one module is of type .CHAR and is declared with the .PUBLIC directive and another module references this variable with the .EXTRNB directive, then in the second module this variable will be of the .UCHAR type, which will force the Linker to generate a warning message. Type checking can be disabled with the Linker **-w** option.

The .DCx, .DSx, .LABELx, .EXTRNx assembler directives automatically assign type attributes to data being declared (see section *Assembler Directives* in *Chapter 2. MCA-430 Assembler* for details).

In fact, every operand or name has the 'type' attribute. For example, a name defined with the .DB2 directive will be of type **nothing** (0).

1.5.4. 'Operand Type' Attribute

The *operand type* attribute is used by the Assembler to check whether the instruction code and the size and/or alignment of operands match.

Operand Type	Size	Alignment
BYTE	8 bit	byte
WORD	16 bit	word
DWORD	32 bit	word
UNTYPED	length is unknown	unknown

When a name has an operand type assigned to it, the Assembler will perform type checking in instructions. This can be helpful for detecting some errors. A typical example is a missing "#" before a constant:

```
mov 256, R5    ;warnings here – UNTYPED operand without allocation
mov #256,R5    ;correct, if a number was to be used
```

In the above example the 'operand type' attribute of the operand is UNTYPED, which will cause the Assembler to produce a warning message (additional warning will be connected with the absence of allocation attribute). It is not recommended to have UNTYPED names/numeric values used as memory addresses in instructions operating with data. For instance, to access SFRs and peripheral modules, use the names defined in the supplied include files.Operand type checking can be disabled with the **-a** command line option (see *Appendix C. MCA-430 Command Line Interface*).

Using the .DCx, .DSx, .LABELx, and .EXTRNx assembler directives automatically assigns types to data defined by them. In addition, you can use special directives to declare the operand type, such as .BYTE, .WORD, .DWORD (see *Assembler Directives* section in *Chapter 2. MCA-430 Assembler*).

In fact, all operands and names have the 'operand type' attribute (numeric constants, for instance, have the UNTYPED operand type). You can redefine the operand type of an expression with the .BYTE, .WORD, .DWORD (see *Assembler Operators* section in *Chapter 2. MCA-430 Assembler*) and use the expression as an operand in instruction.

Example:

```
      .RSEG _MyReg,reg
MyWord .dsw              ;MyWord obtains WORD operand type
MyByte .dsb              ;MyByte obtains BYTE operand type
MyArr  .ds 5
      .RSEG _MyCode,code
mov MyArr,MyByte         ;op-type mismatch for MyArr
mov MyArr+1,MyWord       ;op-type mismatch for both operands
.byte MyArr              ;now op-type of MyArr is BYTE
mov MyArr,MyByte         ;ok
```

```
mov MyArr+1,.byte MyWord ;ok: op-type of the 2nd operand is redefined
                         ;with .byte operator
```

1.5.5. ‘Allocation’ Attribute

The *allocation* attribute lets the Assembler know which addressing mode is allowed when accessing a memory location indicated by a symbolic name. The name defined as an address in a segment inherits the allocation attribute of the segment.

The 'allocation' attributes of external names are specified in the .EXTRNx directive. The Linker checks whether the allocation attribute of the external name matches allocation attribute of the corresponding public name.

The 'allocation' attribute is not assigned to a name which is not associated with any address (e.g., name declared as a numeric constant using the .EQU directives, etc.). For details, see *Assembler Directives* section in *Chapter 2. MCA-430 Assembler*.

Chapter 2. MCA-430 Assembler

In this chapter, you will find MCA-430 Assembler specific information, including usage of expressions, operators, assembler directives, special characters, macros and functions, as well as MSP430 specifics support.

2.1. Command Line Format

Usage:

```
MCA430.EXE [options] source_file1 [source_file2 […]]
```

`Source_file` is a file with assembler source text. `Options` are any of MCA-430 options. Options each start with the "-" (minus character), followed by a flag letter which selects the option. Options may be given in any order or omitted entirely. The flag letter may be followed by additional text relating to the option. Options are separated from each other and from the source name by spaces. Options are case sensitive.

If the source filename has the standard .MCA extension then the extension can be omitted.

The following command line options are accepted by the MCA-430:

Option	Description
-I*path*	Search for include files in the specified directories
-N	Insert file at the beginning of the source file
-u	Ignore character case
-d	Generate debugging information
-a	Disable instruction operand type checking
-r	Disable detection of jumps made to the data memory
-l	Generate listing file
-J*path*	Place object file in the specified directory
-L*path*	Place listing file in the specified directory
-x	Include cross-reference table in the listing
-c	Include false conditionals in the listing
-g	Include macro definitions in the listing
-e	Include macro expansions in the listing
-w	"Wide" listing output
-p	Split listing into 40 line pages
-P*nn*	Split listing into pages of the specified length
-E*nn*	Terminate assembling after n errors
-W*nn*	Display not more than n warnings
-b	Produce beep if error is detected
-s	Display the number of processed lines
-h or -?	Display a brief description of options
@*filename*	Append a response file to the command line

2.2. Assembler Syntax

Assembler syntax is a set of rules that a developer must follow when writing source text of a program. Improperly written source text will cause error messages produced by the Assembler.

2.2.1. Vocabulary and Grammar

As any other language, the assembly language has vocabulary and grammar. The vocabulary is a set of keywords used in expressions, such as terms and tokens, and the grammar is a set of rules defining the ways to combine terms to express ideas (notions) and actions.

Assembly language source line is the basic assembly language unit. The source line may contain assembler mnemonics, directives and/or comments.

2.2.2. Lexical Elements

Lexical elements in Assembler are the basic language elements such as names and labels, keywords, numbers, and character strings.

Identifiers are symbolic names defined by the user and are used in the source text to specify addresses, constants, macros, and so on. The identifier is a combination of alphabetic characters (either upper or lower case), digits, question marks "?", and underscore characters "_". The following restrictions apply to the identifiers:

- Identifier length can not exceed 255 characters.
- Identifiers are case sensitive (unless the **-u** command line option is specified or an appropriate option is set in the IDE). For example, `BUFFER` and `buffer` are different identifiers.
- Identifier may not contain any delimiters or spaces.
- Identifier may not begin with a digit.

The *keywords* are predefined identifiers and have a special meaning in the Assembler. Keywords are used to denote predefined names such as:

- mnemonics of MSP430 instructions (such as ADD, MOV, and so on);
- microprocessor registers (R0..R15, PC, SP, and SR);
- names of allocation attributes (code, data).

The keywords are not case sensitive.

You can use custom *names* to designate variables, labels, macros, functions, etc., in your program.

Label is a name that is used to mark a certain location in the program. Labels can be used to organize jumps in the program flow without the need to manually calculate the addresses for jump destinations.

Assembler can operate with integer *numbers*. Numbers in Assembler can be in decimal, hexadecimal, octal, and binary numerical notation.

Character string is a sequence of ASCII characters embraced in single quotes ('). To use a quotation mark as one of the characters in a character string, it should be written twice ('). Maximum allowed character string length is 255 characters.

Note, unlike in C, character strings in Assembler do not include the terminating null character. If you need a null-terminated string, you should insert the null byte at the end of the string.

2.2.3. Assembler Statements

A program in Assembler is a set of statements, each written in a separate line:

```
statement
statement
...
statement
```

The following basic rules apply to the Assembler statements:

- Assembler does not support line splitting
- Every line should end with a line feed and a carriage return symbols
- Spaces may be used anywhere in the line except inside the tokens
- Source text may contain empty lines
- Statement with a label can only be placed inside a segment

2.2.3. 1. Symbols and Delimiters

The source text of an assembler program may contain alphabetic characters, digits, symbols, and delimiters. All symbols are allowed in comments. Delimiters are used to separate terms (tokens) in the Assembler expressions. Below is the list of delimiters with brief descriptions:

Symbol	Code	Name	Description
SPACE	20H	Space	Separates fields, identifiers
HTAB	09H	Horizontal tabulation	Same as the above
'	-	Single quote	Delimits character strings
.	-	Period	Directive/operator prefix, period operator
,	-	Comma	Separates arguments
;	-	Semicolon	Starts comments
:	-	Colon	Designates labels
"	-	Quote	Floating-point number prefix
()	-	Parentheses	Delimits expressions

2.2.3. 2. Source Line Format

Assembler source text line may consist of a label, operation, operands, and comments.

Syntax:

```
[label:] [<operation> [operands]] [;comments] <CR,LF>
```

or

```
name <operation> [operands] [;comments] <CR,LF>
```

Label/Name

```
[label:]
```

or

```
name
```

This field contains either a label or a name. Label ends with a colon and can be placed in a separate line. Names can be put only before instruction mnemonics, macro calls and Assembler directives.

Operation

```
<operation>
```

This field contains an instruction mnemonic, a directive, or a macro name.

Operand

```
[operands]
```

This field contains operands of the operation. The contents and syntax of this field may vary according to the operation specified earlier in the line.

Comments

```
[;comments]
```

Serves for making comments for any line in the program text. Comments are ignored by the Assembler but are included in the listing file. Comments field should start with a semicolon (**;**). Any text following the semicolon is comments.

2.2.3. 3. Assembler Directives

Assembler directives are used to control the assembly process, declare program constants and define constants in the ROM, reserve space for variables in the RAM, define and switch segments, change the location counter, etc.

The syntax of a directive statement is the following:

```
[name] <directive> [parameters] [;comments]
```
or
```
<directive> name [;comments]
```

Directives are reserved words. All directives start with a period and are not case sensitive.

2.2.3. 4. Forward and Backward References

Referencing a name/label that is defined further in the program is called a *forward reference.* If a name is first defined and then used then it is a *backward reference.* Examples:

```
Backward Reference    Forward Reference

     MOV a,#5               AND b,#0F0h
LOOP:                       JNZ SKIP
     DEC a                  MOV b,#0
     CP a,#0          SKIP:
     JNZ LOOP               …
```

Certain restrictions apply to usage of forward references in some expressions. Any such restrictions are described when they apply to an expression.

2.3. Assembler Expressions

Expression is a sequence of operands and operators. Expressions that can be evaluated at assembly time are called *absolute expressions*. Expressions where the result can not be known until linking are called *relocatable expressions.*

There are *constant* and *address* expressions. Constant expressions are evaluated by the Assembler at assembly time using 32-bit arithmetic. Numbers, character strings, and named constants are the simplest constant expressions. If a constant expression is simply a number, it can not exceed 2^{32}-1, otherwise, the Assembler will generate the "out of range" error message. If a constant expression is a compound expression and the result exceeds 2^{32}-1, the higher bits are ignored. Values of the address expressions are addresses in the memory. Address expressions are evaluated either by the Assembler or by the Linker. The simplest address expressions are labels, program counter, and segment names.

Expression is a syntax structure, which can be:

- A single *basic operand* (no operators, the simplest form of expression)
- An *unary operator* (applied to an expression)
- A *binary operator* (applied to a couple of expressions)

2.3.1. Operands

The following are the basic operands (simplest expressions):

- Numbers
- Character strings
- User defined names
- $ symbol (Program Counter)
- Segment names

2.3.1. 1. Numeric Constants

Numeric constant (or number) is a sequence of digits ending with a suffix specifying base. Hexadecimal numbers should always start with a digit. Thus, if a hexadecimal number starts with a letter you should add a leading zero at the beginning. For instance, ABCDh is written as 0ABCDh.

Number	Radix	Digits	Suffix
Binary	2	01	B
Octal	8	01234567	O
Decimal	10	0123456789	D* (optional)
Hexadecimal	16	0123456789ABCDEF	H

* Suffix is optional – decimal format is default

Assembler stores all numeric constants in the memory as 32-bit integer values, so they must fit in 32 bits. Floating-point number format is also supported. For example, you can write using the floating point number prefix (see *Symbols and Delimiters* section earlier in this chapter): `"3.2645E+4.` Floating-point numbers are stored as 32-bit numbers, in accordance with the ANSI/IEEE 754-1895 standard for real single-precision normalized numbers.

Note, 32-bit floating point numbers can not be used in arithmetic expressions. Such numbers can only be used for memory initialization (providing the initial values for variables and constants in tables) in the programs containing single-precision real number calculations that comply with the ANSI/IEEE 754-1895 standard.

Numeric constant attributes:

Allocation	Operand Type	Type	Relocatability
<none>	UNTYPED	Nothing	ABS

2.3.1. 2. Character Strings

MCA-430 Assembler allows using ASCII characters in expression to generate a numerical value. If a character string is 4 characters or less, it can be used as a constant in arithmetic expressions. In this case, the value of the string will be the number composed of the ASCII codes of the characters in the string. For example, the value of string '1234' becomes 031323334h. The rightmost symbol of the string will be placed in the least significant byte (byte with the lowest address).

If a character string contains more than 4 characters then it may be used only in statements with directives (for example, .DCx or .ERROR) or in comparison operations (see *Relational Operators* section in *Assembler Operators*).

Attributes of the character strings (not longer than 4 characters):

Allocation	Operand Type	Type	Relocatability
<none>	UNTYPED	nothing	ABS

2.3.1. 3. User Defined Constants

You can use your own names to describe constants. The constant name should be defined with the directive .EQU or .DEFINE. Constants defined with the .DEFINE directive retain their values across all modules in the source file, while the constants defined with the .EQU directive retain their values only within the current module. There are also predefined Assembler constants. The full list of the can be found in the *Predefined Constants* section in this chapter. Examples of using constants:

```
Blue  .DEFINE 2                    ; Blue equals to 2
Green .DEFINE 3                    ; Green equals to 3
K .EQU 5                           ; K equals to 5
MOV #(K+2), R5                     ; 7 is loaded in R5
```

For details on directives, see section *Assembler Directives* further in this chapter.

2.3.1. 4. Labels/Names

A label is an address expression. Its value is the address of labeled location or the address of the variable in the memory.

2.3.1. 5. Program Counter

When the Assembler generates code, it uses its internal variable to trace addresses. This variable is called *program counter* (location *counter*). The current program counter value is always stored in this variable, which can be accessed by the $ symbol. The value of this internal variable is equal to the value of program counter BEFORE the current line is assembled.

Each segment has its own program counter and the Assembler uses program counter of the active (current) segment as the current counter.

Because a segment can be either relocatable or absolute, the value returned by the $ symbol has either ABS or REL relocatability attribute respectively.

Attributes of the value designated by the $ symbol:

Allocation	Operand Type	Type	Relocatability
Obtained from the current segment	UNTYPED	nothing	Obtained from the current segment

2.3.1. 6. Segment Names

The value of a *segment name* used in an expression is equal to the origin (absolute address) of the segment fragment in the current module. The use of segment names in expressions is restricted.

Segment name value attributes:

Allocation	Operand Type	Type	Relocatability
Obtained from segment	-	-	SEG

2.3.2. Type Conversion and Checking in Expressions

Every operand in an expression has a set of attributes:

- Allocation
- Operand type
- Type
- Relocatability

When an expression is processed the operand attributes are checked for compatibility and are converted according to a set of rules listed below.

Note that the 'type' attributes of operands are not checked, and the resulting 'type' attribute is always **nothing** (i.e. 0).

If the result of evaluated expression has the 'relocatability' attribute ABS then the expression is called *absolute*. Otherwise, the expression is called *relocatable*. If not explicitly specified otherwise in the operation description, the following type checking and conversion rules are applied.

2.3.2. 1. Binary Operations

Allocation attributes of both operands should the same if assigned. If either of the operands has no allocation assigned, the result obtains the allocation from the other operand. If both operands have no allocation then the result also has no allocation. In a special case of subtraction of two relocatable operands declared in one segment, the result has no allocation, since it is a numeric value.

Operand type attributes of the operands should either be the same, or one of the operands should be UNTYPED. If operand types are the same, the result will have the same type as the operands. If operand types are not the same and one of the operands is UNTYPED then the result will obtain the operand type of another operand. If operand types are not the same and neither of the operands is UNTYPED then the Assembler will generate a warning message and the result will be UNTYPED.

Relocatability attribute is checked and converted according to the following procedure. All operations are allowed with operands that have the ABS relocatability. The result of such operations will also have the relocatability ABS.

The following binary operations are allowed on operands with the REL relocatability:

- addition to an ABS relocatability operand (the result is REL),
- subtraction of an ABS relocatability operand from a REL relocatability operand (the result is REL); and
- subtraction of two REL operands declared in the same segment (the result is ABS).

No other binary operations with the REL relocatability operands are allowed.

Note, non-ABS operands can not be subtracted from the ABS operands.

The following binary operations are allowed on operands with the EXT relocatability:

- addition to an ABS relocatability operand (the result is EXT); and
- subtraction of an ABS relocatability operand from an EXT relocatability operand (the result is EXT).

Other binary operations with the EXT relocatability operands are not allowed.

The operands with the SEG relocatability (segment names) can be added to the ABS operands. The ABS operands can be subtracted from the SEG operands. In both cases, the 'relocatability' attributes of the results are SEG. Other binary operations are not allowed.

The following binary operations are allowed for operands with relocatability SFB/SFE:

- addition to an ABS relocatability operand (the result relocatability is SFB or SFE); and
- subtraction of an ABS relocatability operand from a SFB relocatability operand (result is SFB) or from a SFE relocatability operand (the result is SFE).

Other binary operations with SFB/SFE relocatability operands are not allowed.

2.3.2. 2. Unary Operations

Allocation attribute of the operand is transferred to the result of the operation.

Operand type attribute of the result is the same as the operand type of the operand, except for the special operand type conversion operations (.BYTE, etc.).

Relocatability attribute is checked and converted according to the following rules. The .SFB and .SFE operators are applicable only to the operands with the relocatability SEG (segment names). 'Relocatability' attribute of the result is .SFB or .SFE respectively. Other unary operations are prohibited for the SEG relocatability operands.

The result of a unary operator applied to an ABS operand will have the ABS 'relocatability' attribute.

In byte/word extraction operations (with the .HWRD, .LWRD, .HIGH, .LOW, .BYTE3, and .BYTE4 operators) the 'relocatability' attribute of the result is inherited from the operand and **no further arithmetic operations can be made with this result**.

2.3.3. Predefined Variables

Predefined variables are values that are used to get information about some parameters in the module.

2.3.3. 1. .UPPERCASEONLYMODE

The variable .UPPERCASEONLYMODE is equal to .TRUE (-1) if the Assembler is set to make no distinction between the lower and upper case letters in identifiers, otherwise it is equal to .FALSE (0). The MCA-430 Assembler is case sensitive by default. You can override this setting by the **–u** command line option (to see how to use this option in the IDE, please refer to *IDE Online Help*).

2.3.4. Predefined Constants

The following is the list of predefined Assembler constants:

Constant	Value	Purpose
.TRUE	-1 (0FFFFFFFFH)	Logical True
.FALSE	0	Logical False
.NOCHECK	1	Type definition
.VOID	2	C-compatible type definition
.CHAR	4	Type definition
.UCHAR	5	Type definition
.INT	8	Type definition
.UINT	9	Type definition
.LONG	10	Type definition
.ULONG	11	Type definition
.FLOAT	12	Type definition
.ELLIPSIS	17	C-compatible type definition
.VOID_PTR	100	C-compatible type definition
.CHAR_PTR	101	C-compatible type definition

.UCHAR_PTR	102	C-compatible type definition
.INT_PTR	103	C-compatible type definition
.UINT_PTR	104	C-compatible type definition
.LONG_PTR	105	C-compatible type definition
.ULONG_PTR	106	C-compatible type definition
.FLOAT_PTR	107	C-compatible type definition

2.4. Assembler Operators

MCA-430 Assembler provides a set of operators that are described below.

2.4.1. Addition and Subtraction

Operator	Description	Syntax
+	Arithmetic addition	operand1 + operand2
-	Arithmetic subtraction	operand1 - operand2

2.4.2. Multiplication and Division

Operator	Description	Syntax
*	Multiplication	operand1 * operand2
/	Unsigned division	operand1 / operand2
.MOD	Unsigned remainder selection	operand1 .MOD operand2
.IDIV	Signed division	operand1 .IDIV operand2
.IMOD	Signed remainder selection	operand1 .IMOD operand2

2.4.3. Shift Operators

Operator	Description	Syntax
.SHL	Arithmetic left shift	.SHL operand
.SHR	Arithmetic right shift	.SHR operand
.SHRL	Logical right shift	.SHRL operand

2.4.4. Bitwise Operators

Operator	Description	Syntax
-	Two's complement	- operand
~ or .INV	One's complement	.INV operand
.NOT	Logical negation*	.NOT operand
& or .AND	Bitwise logical AND	operand1 & operand 2 operand1 .AND operand2
| or .OR	Bitwise logical OR	operand1 | operand 2
^ or .XOR	Bitwise logical XOR	operand1 ^ operand 2

* The result of the logical negation operator .NOT is .TRUE (-1) if the value of its operand is equal to 0, and .FALSE (0) otherwise.

2.4.5. Relational Operators

Syntax:

```
expression <relational_operator> expression
```

Operator	Description
<, .LT	signed LESS THAN
<=, .LE	signed LESS THAN OR EQUAL
>, .GT	signed GREATER THAN
>=, .GE	signed GREATER THAN OR EQUAL
.ULT	unsigned LESS THAN
.UGT	unsigned GREATER THAN
==, .EQ	EQUAL
<>, .NE	NOT EQUAL

Relational operations return .TRUE (-1, 0FFFFFFFFh) if the result of comparison is true, and .FALSE (0) if the result of the comparison is false. Your can compare not only numbers but also strings, even if they are **longer** than 4 characters. In this case:

- Strings are equal, if they are fully identical
- A string is greater than another string if the first one is farther from the top of the list arranged in alphabetical order. For example:

```
'ABCDEFG' is greater than 'ABCD'
'ABCCCCC' is less than 'ABCD'
```

2.4.6. Byte/Word Extraction Operators

Syntax:

```
.HWRD  expression
.LWRD  expression
.HIGH  expression
.LOW   expression
.BYTE3 expression
.BYTE4 expression
```

These unary operators are byte/word extraction operators. The return values are the following:

Operator	Description
.HWRD	High word of the operand
.LWRD	Low word of the operand
.HIGH	High byte of the operand's low word
.LOW	Low byte of the operand's low word
.BYTE3	Low byte of the operand's high word
.BYTE4	High byte of the operand's high word

There are some restrictions that apply to using the result of these operations. For details, see section *Type Conversion and Checking in Expressions* in this chapter.

2.4.7. Setting Operand Type of Expression

Syntax:

```
.BYTE    expression
.WORD    expression
.DWORD   expression
.UNTYPED expression
```

These operators set the 'operand type' attribute of an expression. The operators enable type checking for the *expression* specified when this expression is used as operand in Assembler instructions. The following operand type attributes are set to the expressions:

Operator	Operand Type Attribute
.BYTE	BYTE
.WORD	WORD
.DWORD	DWORD
.UNTYPED	UNTYPED

Example:

```
.EXTRNW (data) VarW
.EXTRNB (data) VarB
.RSEG MyCode, code

;copying VarB to lower byte of VarW:
mov.b VarB, VarW                    ;warning
mov.b VarB, .BYTE VarW              ;ok
sxt VarW;
```

2.4.8. Miscellaneous Operators

2.4.8. 1. .SFB, .SFE

Syntax:

```
.SFB segment_name
.SFE segment_name
```

The .SFB and .SFE unary operators can be applied only to the segment names defined in the current module. .SFB returns the start address of the segment and .SFE returns the end address of the segment. The result of the expression is estimated at link time. There are strict limitations applied to using these expressions results. For details, see *Type Conversion and Checking in Expressions*, Chapter 2.

2.4.8. 2. .OFFSET

Syntax:

```
.OFFSET expression
```

Returns the displacement of the operand, residing in a relocatable/overlay segment, from the origin of this segment in the current module. *Expression* is a relocatable, non-external expression. Forward references are forbidden. The operation result is a numeric value of the UNTYPED operand type.

2.4.8. 3. .ALLOCATION

Syntax:

```
.ALLOCATION expression
```

Returns a numeric value corresponding to the allocation attribute of the operand. Forward references are not allowed. This operator is useful in macros for conditional code generation. The operator has very low precedence, so the parentheses are often required:

```
.IF (.ALLOCATION FOO) .EQ 2
```

This operator can return the following values:

Allocation	Returned Value
<none>	0
code	1
data	2

Example:

```
.EXTRNB(data) foo
.RSEG SegCode,code
zoo:
koo .SET 2
qq .SET (.ALLOCATION koo)            ;0
qq .SET (.ALLOCATION zoo)            ;1
qq .SET (.ALLOCATION foo)            ;2
qq .SET (.ALLOCATION noo)            ;Error, forward reference
noo .SET 1
```

2.4.8. 4. .OPTYPE

Syntax:

```
.OPTYPE expression
```

Returns a numeric value corresponding to the operand type of the *expression.* Forward references are not allowed. This operator is helpful in macros for conditional code generation. This operator priority is very low, so the parentheses are often necessary:

```
.IF (.OPTYPE foo) .EQ 2
```

The values returned by the operator are the following:

Operand Type	Returned Value
UNTYPED	0
BYTE	1
WORD	2
DWORD	3

Example:

```
.RSEG MyData,data
.EXTRNB(data) boo             ;define external unsigned char boo
.EXTRND(data) poo             ;define external unsigned long poo
zoo .DSI                      ;define integer (signed word) zoo
too:                          ;define untyped name too
aa = .OPTYPE too              ;optype returns 0
aa = .OPTYPE boo              ;optype returns 1
aa = .OPTYPE zoo              ;optype returns 2
aa = .OPTYPE poo              ;optype returns 3
aa = .OPTYPE noo              ;ERROR - forward reference
```

```
noo .SET 2                      ;define untyped name noo
.END
```

2.4.8. 5. .TYPE

Syntax:

```
.TYPE name
```

Returns the number corresponding to the type of the *name*. Forward references are not allowed. The operand of this unary operator cannot be an expression. The returned values are the following:

Value	Type	MCA-430 Constant	Description
0	nothing	-	undefined type
4	char	.CHAR	8-bit, signed integer
5	unsigned char	.UCHAR	8-bit, unsigned integer
8	int	.INT	16-bit, signed integer
9	unsigned int	.UINT	16-bit, unsigned integer
10	long	.LONG	32-bit, signed integer
11	unsigned long	.ULONG	32-bit, unsigned integer
12	float	.FLOAT	32-bit, floating-point number

2.4.8. 6. .DEFINED

Syntax:

```
.DEFINED name
```

Returns .TRUE (-1) if *name* is defined and .FALSE (0) otherwise. Forward references always return .FALSE. External reference returns .FALSE if the external name was not declared earlier in the program.

Example:

```
.IF (.NOT .DEFINED MyFunc)
  .EXTRNF MyFunc
.ENDIF
```

2.4.8. 7. .DATE

Syntax:

```
.DATE absolute_expression
```

Returns a numeric value. The meaning of the value depends on the argument.

Argument	Result	Value
1	current time (seconds)	0-59
2	current time (minutes)	0-59
3	current time (hours)	0-23
4	current date (day)	1-31
5	current date (month)	1-12
6	current date (year)	year minus 1900

Argument of the .DATE operator should be an absolute expression within the 1-6 range.

2.4.9. Operator Precedence

The following table lists the operators in the order of priority. Operators with highest priority (1) are evaluated first. If there is more than one operator with a certain priority, the leftmost operator is evaluated first followed by each subsequent operator with the same priority.

Operator	Description	Priority
-	Two's complement (arithmetic inversion)	1
~ or .INV	One's complement (bitwise inversion)	1
.NOT	Logical negation	1
.HWRD	High word extraction	1
.LWRD	Low word extraction	1
.HIGH	High byte of low word extraction	1
.LOW	Low byte extraction	1
.BYTE3	Low byte of high word extraction	1
.BYTE4	High byte of high word extraction	1
.SFB	Returns start address of a segment	1
.SFE	Returns end address of a segment	1
.DATE	Returns current date	1
.WORD	Changes operand type to WORD	1
.BYTE	Changes operand type to BYTE	1
.DWORD	Changes operand type to DWORD	1
.UNTYPED	Changes operand type to UNTYPED	1
.DEFINED	Returns 0, if the name is undefined	1
.BLANK	Returns 0, if macro parameter is specified	1
.PARM	Returns the text of actual argument	1
.TYPE	Returns type of operand	1
*	Multiplication	2
/	Unsigned division	2
.MOD	Unsigned remainder selection	2
.IDIV	Signed division	2
.IMOD	Signed remainder selection	2
.SHL	Left shift	2
.SHR	Arithmetic right shift	2
.SHRL	Logical right shift	2
+	Addition	3
-	Subtraction	3
& or .AND	Bitwise logical AND	4
| or .OR	Bitwise logical OR	5
^ or .XOR	Bitwise logical XOR	5
< or .LT	Signed LESS THAN	6
<= or .LE	Signed LESS THAN OR EQUAL	6
> or .GT	Signed GREATER THAN	7
>= or .GE	Signed GREATER THAN OR EQUAL TO	7
.ULT	Unsigned LESS THAN	7

.UGT	Unsigned GREATER THAN	7
== or .EQ	EQUAL TO	7
<> or .NE	NOT EQUAL TO	7
.ALLOCATION	Returns operand allocation attribute	8
.OPTYPE	Returns operand type attribute of the operand	8

2.5. Assembler Directives

In this section you will find the detailed description of MCA-430 directives including conditional assembling and listing control directives. See also *Appendix A. Assembler Directives Summary* for complete list of all Assembler directives.

2.5.1. Module Declaration

2.5.1. 1. .PMODULE, .LMODULE

Syntax:

```
.PMODULE module_name
.LMODULE module_name
```

Declares the beginning of a program or a library module. *Module_name* assigns the module name, under which the module will be placed in the object file. The length of a module name may not exceed 255 symbols.

Note that in order to include modules into a library all the module names must be unique. Otherwise, the MCLIB Librarian will produce the "duplicate module" error message. For details on MCLIB, refer to *Chapter 4. MCLIB Librarian*.

.LMODULE defines a library module that will be linked to the program only if there are external references to this module. .PMODULE defines a program module which, unlike a library module, is always linked to the program.

The scope and accessibility of non-public identifiers and labels defined in the module are limited to this module.

If the file contains the source text of one module only and there are no .LMODULE or .PMODULE directives then only one program module will be generated as a result of translation. This module will have the same name as the source file (path and extension are omitted).

2.5.1. 2. .LMODULE2

Syntax:

```
.LMODULE2 module_name
```

Declares the beginning of a low-priority library module. Other than that, this directive is the same as the .LMODULE directive.

Low-priority library modules are scanned by the Linker after scanning the standard library modules. Hence, a low-priority library module is linked to the program only if it contains such external references that can not be resolved by standard library modules.

2.5.1. 3. .ENDMOD

Syntax:

```
.ENDMOD
```

Indicates the end of a module. Every module in the file should end with this directive, except the last one, which ends with the directive .END.

2.5.1. 4. .END

Syntax:

```
.END
```

Indicates both the end of a module and the end of the source file. The last module in the file must end with this directive.

Note: Even if the file contains only one module and none of the .PMODULE, .LMODULE, or .LMODULE2 directives are specified, the .END directive is still required.

2.5.2. Including File

2.5.2. 1. .INCLUDE

Syntax:

```
.INCLUDE '[path\]filename'
```

Includes the contents of the *filename* file in the source text immediately after this directive.

Single quotation marks are obligatory. Include file nesting is allowed. If *path* is omitted then the Assembler first searches for the file in the current directory and then in directories specified in the Assembler **-I** command line option (refer to *IDE Online Help* to see how to specify the project directories by include files in the IDE).

If *path* is specified and starts with a drive letter or a backslash (\), it is assumed to be absolute (if only the backslash is used, the path is absolute on the current drive). Otherwise, the Assembler uses *path* as relative to the current directory or the directories specified in the **-I** command line option. Examples:

```
.INCLUDE 'config.inc'        ;CONFIG.INC will be searched in the current
                             ;directory and ones specified in
                             ;the -I option
.INCLUDE 'inc\config.inc'    ;CONFIG.INC will be searched in the 'INC'
                             ;subdirectory of the current directory
                             ;(relative path) and of the ones specified
                             ;in the -I option
.INCLUDE 'c:\inc\config.inc' ;CONFIG.INC will be searched in the 'C:\INC'
                             ;directory
.INCLUDE '\inc\config.inc'   ;CONFIG.INC will be searched in the
                             ;'drive:\INC' directory (absolute path),
                             ;where drive is the current drive letter
```

2.5.3. Segment Declaration and Selection

2.5.3. 1. .ASEG, .RSEG, .OSEG

Syntax:

```
.RSEG segment_name [,allocation]
.OSEG segment_name [,allocation]
.ASEG segment_name [,allocation]
```

Use these directives to declare a new segment or select an existing segment that has been declared earlier in the source text.
.RSEG directive declares/selects a relocatable segment.
.OSEG directive declares/selects an overlay segment.
.ASEG directive declares/selects an absolute segment.
For more details on relocatable/overlay/absolute segments, refer to *Chapter 1. Basic Conceptions* in this manual.

Segment_name indicates the segment that you want to create or switch to. *Allocation* is the allocation attribute of the segment (see *Segment 'Allocation' Attribute*, Chapter 1). If you are declaring a new segment in the current module, you should specify the segment allocation attribute. When you are switching to a previously declared segment, the allocation attribute is optional. If still specified, it must be the same as in the segment declaration. When selected, each segment should be specified using the same directive it has been declared with. Otherwise, the Assembler will generate the "segment type conflict" error.

If a segment is used in various modules, the segment type and allocation attributes must be the same in all modules; otherwise, the Linker will generate an error message.

Extended Format of Segment Declaration

In certain situations, it is useful to have code executed from RAM. For example, code that controls Flash memory write/erase can be located in the Flash memory itself and should be copied at runtime into RAM to avoid conflict when the write/erase procedure runs.

Such functionality is implemented in the MCA-430 via extended segment declaration format. When you declare a new data segment using the extended segment declaration format, you can write code and initialize data in such a segment.

Syntax:

```
.RSEG data_segment_name, data (code_segment_name)
.OSEG data_segment_name, data (code_segment_name)
```

The Linker will place code (and initialized data) defined in the segment *data_segment_name* in the current module into the **code** segment *code_segment_name* (ROM/Flash); allocate the **data** segment *data_segment_name* in the RAM and adjust all addresses of the labels and names defined in *data_segment_name* as if they are located in the segment *data_segment_name* with the allocation **data**. As a result, you can copy such code from the ROM/Flash to the RAM at run time and execute it afterwards.

The segment *code_segment_name* may be declared before; in this case it should have the allocation **code** and should be relocatable. If it was not declared before, an implicit declaration is assumed:

```
.RSEG code_segment_name, code
```

Note: if there is any code written in the *code_segment_name* segment in the current module, space in the *data_segment_name* will be reserved for it; the names defined in the *code_segment_name* segment will obtain values in the *data_segment_name*, i.e. will have the allocation **data**. If the extended declaration format is used more than once for a data segment, the code segment name should be the same in all statements; otherwise, the Assembler will produce an error message.

Note: absolute segments cannot be declared in the extended format.

Example:
1) File 'reload.inc':

```
.LSTOUT -
Reload .MACRO srcSeg,dstSeg
;the macro copies contents srcSeg into dstSeg
;R13..R15 register are used
  mov   #.sfb srcSeg,R13
  mov   #.sfe srcSeg,R14
  sub   R13,R14
  inc   R14                    ;srcSeg size in bytes
```

```
  jz    ~quit
  mov   #.sfb dstSeg,R15
~loop:
  mov.b @R13+,0(R15)
  inc   R15
  dec   R14
  jnz   ~loop
~quit:
.LSTOUT .
.ENDMAC
```

2) File 'reload.mca':

```
.PMODULE MOD1
.OSEG  MyRamCode,data(ReloadCode)
.FUNC    MyRamFunc
.PUBLIC  MyRamFunc
;...
  ret
.ENDF

.ENDMOD

.PMODULE MOD2
.INCLUDE 'reload.inc'

.OSEG  MyRamCode,data(ReloadCode)

.EXTRN(data)MyRamFunc
.FUNCTYPE 0 MyRamFunc(0)

.RSEG  MyCode,code
;...
Reload ReloadCode,MyRamCode    ;macro call
;...
CALL  #MyRamFunc
;...
.END
```

2.5.3. 2. .ENDSEG

Syntax:

```
.ENDSEG
```

.ENDSEG switches the Assembler to the previous segment, which has been in use before the last occurrence of the .RSEG, .OSEG, or .ASEG directive.

2.5.4. Memory Initialization

2.5.4. 1. .DCx

.DCB, .DCW, .DCD, .DCC, .DCI, .DCL, .DCR
Syntax:

```
[name] .DCB expr_or_string [,expr_or_string] …
[name] .DCW expression [,expression] …
[name] .DCD expression [,expression] …
[name] .DCC expr_or_string [,expr_or_string] …
[name] .DCI expression [,expression] …
[name] .DCL expression [,expression] …
[name] .DCR expression [,expression] …
```

This is a group of directives that are used to declare constant data and tables in ROM (reserve memory with initializing). Each of these directives specifies necessary data alignment, 'operand type' and 'type' attributes for a name. *Name* is an optional parameter.

Note: By default, these directives can be used only in segments that have the **code** allocation attribute. In the segment with the allocation **data** these directives can be used only if the segment is declared using the extended format. See details for the .ASEG/.RSEG/.OSEG directives earlier in this chapter.

Character strings can also be specified in the .DCB and .DCC directives. The .DCR directive assigns the **float** type attribute to a name, therefore, it is expected (but not verified by the Assembler) that a floating point number will be specified in the expression.

The range for the value of the expression depends on the size of memory area that is being reserved. For relocatable expressions, the value range is checked at link time.

Directive	Value Range
`.DCB, .DCC`	–128...255
`.DCW, .DCI`	–32768...65535
`.DCD, .DCL, .DCR`	–2147483647...4294967295

Directive	Operand Type	Type
`.DCB`	BYTE	unsigned char
`.DCW`	WORD	unsigned int
`.DCD`	DWORD	unsigned long
`.DCC`	BYTE	char
`.DCI`	WORD	int
`.DCL`	DWORD	long
`.DCR`	DWORD	float

2.5.4. 2. .DB1

Syntax:

```
[name] .DB1 expression_or_string [,expression_or_string] …
```

Reserves the required number of bytes in the object file and assigns values to them. This directive can be used, for instance, to generate byte tables in ROM. *Expression* can be absolute or relocatable and should be within the -128...+255 range. If the expression is relocatable then checking whether it is within this range occurs at link time. If a string is specified then every symbol of the string initializes one byte in the memory. Strings can be longer than 4 characters.

If the *name* is specified the following attributes are assigned to it:

Value	Allocation	Operand Type	Type	Relocatability
Current value of the Assembler Program Counter	Allocation of the current segment	UNTYPED	nothing (i.e. 0)	Relocatability of the current segment

Note: The operand type should be assigned to a name. Then, the Assembler can detect, for example, if a byte operand is used in a word instruction. Also, it is advisable to assign type to name. This can be achieved by consequently specifying the corresponding directives (e.g., .DB1/.TYPE/.BYTE), although it is simpler to use the .DCB or .DCC directives.

2.5.4. 3. .DB2, .DB4

Syntax:

```
[name]  .DB2 expression [,expression] …
[name]  .DB4 expression [,expression] …
```

Defines 2-byte or 4-byte data and assigns values to them. *Expression* in the .DB2 directive can be either absolute or relocatable but its value must reside within the -32768 to 65535 boundaries. *Expression* in the .DB4 directive can be either absolute or relocatable but its value must be within the -2147483647 to 4294967295 boundaries. A floating point number can be used as the *expression*. If the expression is relocatable then checking whether it is within this range is performed at link time.

If the *name* is specified, the following attributes are assigned to it:

Value	Allocation	Operand Type	Type	Relocatability
Current value of the Assembler Program Counter	Allocation of the current segment	UNTYPED	nothing (i.e. 0)	Relocatability of the current segment

Note: The operand type should be assigned to a name. Also, it is advisable to assign the type to a name. This can be achieved by consequently specifying the corresponding directives (e.g., .DB2/.TYPE/.WORD), although is simpler and easier to use the .DCW, .DCI, .DCD, .DCL, and .DCR directives.

2.5.5. Memory Reservation without Initialization

2.5.5. 1. .DSx

.DSB, .DSW, .DSD, .DSC, .DSI, .DSL, .DSR
Syntax:

```
name .DSB [expression]  ;unsigned value, 8 bit
name .DSW [expression]  ;unsigned value, 16 bit
name .DSD [expression]  ;unsigned value, 32 bit
name .DSC [expression]  ;signed value, 8 bit
name .DSI [expression]  ;signed value, 16 bit
name .DSL [expression]  ;signed value, 32 bit
name .DSR [expression]  ;floating-point value, 32 bit
```

This is a group of directives used to define variables and arrays of specified type (reserve memory without initialization). The *name* is the name assigned to the variable. If the *expression* is not specified or equals to 1 then one variable is defined.

If the *expression* is greater than 1, an **array** of *expression* variables of the specified size/type is defined. The e*xpression* must be of absolute type and contain no forward references.

The name inherits the allocation and relocatability attributes from the current segment. The following attributes are assigned to the name:

Directive	Operand Type	Type	Alignment
.DSB	BYTE	unsigned char	byte
.DSW	WORD	unsigned int	word
.DSD	DWORD	unsigned long	word
.DSC	BYTE	char	byte
.DSI	WORD	int	word

.DSL	DWORD	long	word
.DSR	DWORD	float	word

Example:

```
.RSEG MyData,data            ;open data segment
MyFlags .DSW                 ;define WORD variable
MyFlag1 .EQU .byte MyVar     ;define storage for low byte
MyFlag2 .EQU .byte MyVar+1   ;define storage for high byte
…
```

2.5.5. 2. .DS

Syntax:

```
[name] .DS expression
```

Reserves the number of bytes specified by the *expression* without initialization. *Expression* must be of absolute type and contain no forward references.

This directive is used only for reserving memory (e.g., for a variable in RAM). It does not generate any code simply incrementing the program counter value. If the *name* is specified, the following attributes are assigned to it:

Value	Allocation	Operand Type	Type	Relocatability
Current value of the Assembler Program Counter	Allocation of the current segment	UNTYPED	nothing (i.e. 0)	Relocatability of the current segment

Note: If you need to reserve space for a variable and assign the 'operand type' attribute (as well as the 'type' attribute) to this variable, it is better to use one of the .DSx group directives.

2.5.6. Symbol Definition

2.5.6. 1. .DEFINE, .EQU

Syntax:

```
name .DEFINE expression
name .EQU expression
```

Use these directives to define a name and assign the expression value to it. The .DEFINE directive is different from the .EQU directive in terms of scope of the name defined. The value of the name declared with the .DEFINE directive is maintained across all subsequent modules in the source program file. The scope of the name defined with the .EQU directive is limited to the current module only. Expression used in .DEFINE directive must be absolute.

All attributes of the names declared with these directives are inherited from the expressions. See also *“Type Conversion and Checking in Expressions”*.

Note that the 'type' attribute of *expression* is always **nothing**; it does not depend on the 'type' attributes of operands. If the *name* is declared as public, then the correct 'type' attribute should be assigned to it. Otherwise the linker can generate ‘type conflict’ warning messages.

In pure assembler programs, you can disable type checking with –t command line linker option and avoid the ‘type conflict’ warnings.

In mixed C/assembly programs, it is not recommended to disable type checking. If a public name in an assembly module is defined with .EQU directive as an alias of another name, the type of the alias name should be properly defined with .TYPE directive (operator .TYPE can be used to get corresponding type attribute value), for example:

```
Name1 .DSB                ; Name1 has type 'uchar'
Name2 .EQU Name1          ; Name2 has type 'nothing' here
.TYPE Name2(.TYPE Name1) ; Name2 has type of Name1, i.e. 'uchar'
```

2.5.6. 2. .SET, =

Syntax:

```
name .SET expression
name = expression
```

Use these directives to define a name and assign the *expression* value to it. Unlike .EQU and .DEFINE directives, values assigned to names with .SET and "=" may be redefined later in the current module using the same directives. Note, unlike the .DEFINE and .EQU directives, names defined with .SET and "=" can not be made public and serve as externals in other modules.

2.5.6. 3. .LABELx

.LABELB, .LABELW, .LABELD, .LABELC, .LABELI, .LABELL, .LABELR
Syntax:

```
name .LABELB
name .LABELW
name .LABELD
name .LABELC
name .LABELI
name .LABELL
name .LABELR
```

This group of directives is used to define typed labels. 'Operand type' and 'type' attributes are assigned to the label name. It is preferable to use such labels to mark the start or end address of data tables. The Assembler program counter does not change.

The following attributes are assigned to a typed label:

Value	Allocation	Relocatability
Current value of the Assembler program counter	Allocation of the current segment	Relocatability of the current segment

Directive	Operand Type	Type	Alignment
.LABELB	BYTE	unsigned char	byte
.LABELW	WORD	unsigned int	word
.LABELD	DWORD	unsigned long	word
.LABELC	BYTE	char	byte
.LABELI	WORD	int	word
.LABELL	DWORD	long	word
.LABELR	DWORD	float	word

Example 1:

```
;define a table in ROM:
.RSEG    TABLES,code         ;open segment to be placed in the ROM
  MyTable  .LABELW           ;set appropriate label
           .DCW 22,44,77,88  ;table
           .DCW 66,33,55,99  ;contents
```

Example 2:

```
;define overlapping variables:
.RSEG     MyData,data        ;open data segment
  MyVar    .LABELW           ;set WORD label
  MyVarLo  .DSB              ;define storage for low byte
  MyVarHi  .DSB              ;define storage for high byte
```

Note, that the following statement:

```
name: ...
```

where *name* is the label name, also contains a label but this label has the 'operand type' attribute UNTYPED. It is recommended to use such labels only as jump addresses, not for accessing data.

2.5.7. Program Linkage

2.5.7. 1. .PUBLIC

Syntax:

```
.PUBLIC name [,name] …
```

This directive sets the *name*(*s*) as public, i.e. accessible using external references from other modules. Such names must be defined in the current module. Macro names, module names, keywords and symbols declared as external can not be declared as public.

2.5.7. 2. .EXTRNx

.EXTRNB, .EXTRNW, .EXTRND, .EXTRNC, .EXTRNI, .EXTRNR, .EXTRNL, .EXTRNN
Syntax:

```
.EXTRNB (allocation) name [,name] …
.EXTRNW (allocation) name [,name] …
.EXTRND (allocation) name [,name] …
.EXTRNC (allocation) name [,name] …
.EXTRNI (allocation) name [,name] …
.EXTRNL (allocation) name [,name] …
.EXTRNR (allocation) name [,name] …
.EXTRNN name [,name] …
```

The directives in this group declare external names with corresponding attributes ('allocation', 'operand type', and 'type'). The .EXTRNN directive is used to declare external names without the 'allocation' attribute (such as numeric constants). Other directives can be used to declare names of external variables, array names, table names, etc.

Note: it is recommended to use the .EXTRNF directive for declaration of external names that are function addresses.

The following attributes are assigned to the declared external names:

Value	Allocation	Relocatability
External, estimated by the Linker	Allocation specified in the directive	EXT

Directive	Operand Type	Type
.EXTRNB	BYTE	unsigned char
.EXTRNW	WORD	unsigned int
.EXTRND	DWORD	unsigned long
.EXTRNC	BYTE	char
.EXTRNI	WORD	int
.EXTRNL	DWORD	long
.EXTRNR	DWORD	float
.EXTRNN	UNTYPED	nothing

2.5.7. 3. .EXTRN

Syntax:

```
.EXTRN  (allocation) name1 [,name2]…
```

This directive declares *name* as external. This name must be declared as public in some other module, otherwise an error message will be produced at link time. This directive is normally used to declare external labels.

Note, to make a correct declaration of an external function and to avoid "type mismatch" Linker warning, it is recommended to use the special directive .EXTRNF rather than the .EXTRN directive. If an external function is defined in a **data** segment declared using extended format, you can use the .FUNCTYPE directive to assign proper function type. See *Segment Declaration and Selection*, Chapter 2.

'Allocation' attribute is obligatory. Therefore, you can not use the .EXTRN directive to declare an external name, which is just a number (use the .EXRTNN to declare an external numerical constant).

This directive does not assign any attributes to external name except allocation (in fact, the 'operand type' attribute is set to UNTYPED and the 'type' attribute is set to **nothing** or 0) and it is not recommended to use it to declare data.

The Linker checks whether the attributes of external names match the attributes of the corresponding public names, so when an external name is declared in such a way, it might be necessary to specify the appropriate attributes using the following directives: .TYPE and .BYTE, .WORD, .DWORD .

Assuming the variable RAT is declared in a certain program module:

```
…
.RSEG   DC,data
…
 RAT     .DSB 1     ;data, BYTE, unsigned char
.PUBLIC  RAT        ;declare a name as public
…
```

To gain access to this variable from another module, this name should be declared as external and the appropriate types should be specified:

```
…
.EXTRN (data)RAT    ;RAT was declared in segment with allocation data
.BYTE  RAT          ;BYTE operand type is assigned
.TYPE  RAT(.uchar)  ;unsigned char type is assigned
…
```

Note that the .EXTRNx directives declare external names with the appropriate operand type and type, so the last fragment of the program can be replaced by:

```
…
.EXTRNB (data)RAT
…
```

2.5.7. 4. .EXTRNF

Syntax:

```
.EXTRNF function_name [expr1] [(expr2 [, expr3] ...)]
```

The .EXTRNF directive is used for declaring external Assembler subroutines/functions (defined with .FUNC/.ENDF in other module), and external C functions. This directive defines a *function_name* with the following attributes:

Value	Allocation	Operand Type	Type	Relocatability
Undefined until linking	**code**	UNTYPED	Corresponds to type of function, which receives parameters of types *expr2*, *expr3*, etc. and returns a value of type *expr1*	EXT

When Linker attempts to resolve the external references, it checks all attributes of the external name and matching public name in another module.

The expressions *expr1*, *expr2*, *expr3* (and other as required) must be absolute and may not contain any forward references. It is recommended to use predefined constants such as .CHAR, .UINT, etc. Predefined .NOCHECK constant in the list of function arguments means the function has unknown number of arguments of unknown types. This constant can be put only at the last position of the parameter list. The Linker handles .NOCHECK as being compatible with any type. Since the type numbers are defined only for the basic types, .NOCHECK can be used to specify arrays, structures, typedefs, unions, and pointer-to-function types, as well as any qualified type, i.e. const or volatile. See also sections *Name Types and Type Attribute* for details.

If *expr1* is not specified then it is substituted with .NOCHECK. The Linker does not check function arguments if the argument list (expr2, expr3, etc.) is not specified. The .EXTRNF directive without arguments can be used in pure assembler programs. In mixed C/Assembler programs, it is recommended to declare arguments in the .EXTRNF directive.

The .EXTRNF directive assigns *function_name* the attributes that are necessary for a function in C language. To make a mixed program work correctly, the user must organize interchanging of the parameters and return values between C and assembler functions according to the C function calling convention. For details, see *MCC-430 C Compiler User's Guide*.

Example – Various ways to declare an external function:

```
;external function receiving char
;and returning int:
.EXTRNF f1 .int (.char)

;external function receiving int
;and returning nothing:
.EXTRNF f2 .void (.int)

;external function without arguments
;and returning int:
.EXTRNF f3 .int (.void)

;external function with two int arguments
;and one argument - strucutre returning float:
.EXTRNF f4 .float (.int, .int, .nocheck)

;external function returning int
;(the linker will not check correspondence for argument list):
.EXTRNF f5 .int

;external function receiving float
;(the linker will not check correspondence for return value type):
.EXTRNF f6 (.float)

;external function (the linker will not check for compliance to the list
;of arguments and to the type of the value returned):
.EXTRNF f7
```

2.5.8. Assignment of Attributes to Names

2.5.8. 1. .BYTE, .WORD, .DWORD

Syntax:

```
.BYTE  name [,name] …
.WORD  name [,name] …
.DWORD name [,name] …
```

These directives specify the operand type attribute of a name. 'Operand type' attribute of a name can be assigned only once, but if you use the name in expression, you can redefine the operand type of the expression by one of the .BYTE, .WORD, .DWORD operators.

Note, the .DCx, .DSx, .LABELx, .EXTRNx directives set the 'operand type' attribute of the name automatically. The .BYTE, .WORD, and .DWORD directives do not change the 'allocation' attribute of the name.

These directives can not be used with the names that have the **code** allocation.

Example:

```
.ASEG Reg, data
.ORG 200h
A .DSB
.RSEG RD,data
Name1:  .DS 2          ;defines NAME1 with operand type UNTYPED
Name2:  .DS 2          ;defines NAME2 with operand type UNTYPED
.BYTE Name1            ;assigns operand type == BYTE to NAME1
.RSEG MyCode,code
mov.b Name1,A          ;OK
mov.b Name1+1,A        ;OK
```

```
mov.b Name2,A          ;warning
mov.b Name2+1,A        ;warning
.END
```

2.5.8. 2. .TYPE

Syntax:

```
.TYPE name(expression) [,name(expression)] …
```

The .TYPE directive assigns the 'type' attribute to *name*. The 'type' attribute is equal to *expression*, which should be absolute and contain no forward references. It is recommended to use the predefined constants (see section *Predefined Constants* earlier in this chapter).

The type assigned by the .TYPE directive is used for generating the source-level debugging information and type checking at link time. The Linker checks whether the type attributes of external names match the type attributes of public names.

Normally, this directive is not required as the names defined with the basic set of the Assembler directives already have the 'type' attributes properly assigned.

2.5.8. 3. .FUNCTYPE

Syntax:

```
.FUNCTYPE expr1 func_name (expr2 [, expr3] ...)
```

The directive .FUNCTYPE assigns 'type' attribute to the name *func_name*, which corresponds to the function type. The function receives arguments with the types *expr2*, *expr3* etc. and returns a result with the type *expr1*. This directive is used for function definition in mixed assembler/C programs and allows strict type checking performed by the Linker.

The expressions *expr1*, *expr2*, *expr3* ... must be absolute and contain no forward references. It is recommended to use predefined constants such as .CHAR, .UINT etc. Predefined constant .NOCHECK in the list of function arguments means unknown number of arguments of unknown type. Respectively, this constant can be put only in the last position of the parameter list. The Linker considers .NOCHECK to be compatible with any type. As type numbers are defined only for basic types, .NOCHECK is used to specify arrays, structures, typedefs, unions, and pointer-to-function types, as well as any qualified type, such as **`const`** or **`volatile`**. For additional information on type numbers, see *Name Types and 'Type' Attribute*, Chapter 1.

Normally, this directive is not needed as the .FUNC and .EXTRNF directives with parameters allow assigning the 'type' attribute properly.

2.5.9. Function Declaration

2.5.9. 1. .FUNC, .ENDF

Syntax:

```
.FUNC function_name [expr1] [(expr2 [, expr3] …)]
…
<function body>
…
.ENDF
```

These directives are used for writing functions in assembly language. The .FUNC and .ENDF directives denote the beginning and the end of function definition respectively. It is not allowed to define a function within another function, which means that the .FUNC directive can not be used again before the .ENDF directive ends the current function definition.

The name *function_name* obtains the following attributes:

Value	Allocation	Operand Type	Type	Relocatability
Current Program Counter value	**code**	UNTYPED	corresponds to type of function, which receives parameters with types *expr2*, *expr3*, etc. and returns a result with type *expr1*	Relocatability of the current segment

The .FUNC directive must be placed in **code** segment or **data** segment declared using the extended format (see description of directives under *Segment Declaration and Selection*). The .ENDF directive must be placed in the same segment as the preceding .FUNC directive. The current address of the segment relative to the beginning of the segment fragment must be greater than the address of the function entry point.
The expressions *expr1*, *expr2*, *expr3* (and other as required) must be absolute and may not contain any forward references. It is recommended to use predefined constants such as .CHAR, .UINT etc. Predefined constant .NOCHECK in the list of function arguments means that the function has unknown number of arguments of unknown types. This constant can be put only in the last position of the parameter list. The Linker considers .NOCHECK being compatible with any type. Since the type numbers are defined only for the basic types, the .NOCHECK can be used to specify arrays, structures, typedefs, unions, and pointer-to-function types, as well as any qualified types such as const or volatile. See section *Name Types and Type Attribute* in Chapter 1 for details.

If *expr1* is not specified then it is substituted with .NOCHECK. The Linker does not check function arguments if the argument list (*expr2*, *expr3*, etc.) is not specified. The .FUNC directive without arguments can be used in pure assembler programs. In mixed C/Assembler programs, it is recommended to declare arguments in the .FUNC directive.

See also: section *Functions* later in this chapter.

2.5.10. Address Control

2.5.10. 1. .ORG

Syntax:

```
.ORG expression
```

Sets the value of the current segment program counter equal to the value of the *expression.* The expression should not contain any forward references. Relocatability of the *expression* should be the same as the segment type, i.e. the *expression* should be absolute in the absolute segment and relocatable in the relocatable or overlay segments.

Example:

```
.RSEG MySeg
.ORG $+5
```

Byte/word extraction operators, such as .HWRD, .LWRD, etc., can not be used in the *expression.*

2.5.10. 2. .ALIGN

Syntax:

```
.ALIGN expression
```

Expression must be absolute and should not contain any forward references. This directive performs alignment of the current program counter on the boundary of 2 raised to the *expression* power:

```
0 - no alignment
1 - alignment on even addresses (multiple of 2)
2 - alignment on address multiple of 4
    so on…
```

In the absolute segments, the alignment specified by the .ALIGN directive is performed by the Assembler. In the relocatable segments the alignment is performed by the Linker by automatically adjusting the start address of each segment so that the alignment is maintained for all the .ALIGN directives in that segment. After the segments are linked, the Linker additionally checks that the alignment specified by the all the .ALIGN directives is maintained. If misalignment is detected the Linker generates the appropriate error message. Thus, the .ALIGN directive provides 100% guarantee of obtaining the required alignment.

2.5.11. Conditional Assembly

Conditional assembly directives provide a way to conditionally include and exclude blocks of the source program text from the assembly process. Selection is made upon the results of calculations performed at assembly time. Using the conditional assembly directives is convenient in macros, as this adds extra flexibility to macros.

2.5.11. 1. .IF, .ELSE, .ENDIF

Syntax:

```
.IF expression
    .
    .
 [.ELSE
    .
    . ]
.ENDIF
```

If the calculated value of *expression* is not equal to .FALSE (0) then the fragment between .IF and .ELSE (or between .IF and .ENDIF, if .ELSE is omitted) is assembled. If the *expression* value is equal to .FALSE (0), the .ELSE block (if present) is assembled.

Expression must be absolute and contain no forward references. Nesting of conditional blocks is supported.

2.5.12. Listing Control

2.5.12. 1. .LSTOUT

Syntax:

```
.LSTOUT option
```

This directive enables/disables generation of listing. *Option* is one of the three options:

- `-` listing generation is unconditionally disabled;
- `+` listing generation is unconditionally enabled;
- `.` listing generation is enabled, if the **-l** command line option is specified.

In order to exclude some fragment of the source text from the listing, use ".LSTOUT –" to disable listing generation, then use ".LSTOUT ." to restore the mode specified in the command line. The output listing file is given the same name as the source file, with the .LST extension.

2.5.12. 2. .LSTCND

Syntax:

```
.LSTCND option
```

This command controls whether the false conditionals are included in the listing file. If listing is disabled then this directive does not have any effect. *Option* is one of the three options:

- listing generation is unconditionally disabled;
+ listing generation is unconditionally enabled;
. listing generation is enabled, if the **-c** command line option is specified (default).

2.5.12. 3. .LSTMAC

Syntax:

```
.LSTMAC option
```

This command controls whether the text of macro definitions is included in the listing. This directive does not have any effect if the listing is disabled. *Option* is one of the three options:

- listing generation is unconditionally disabled;
+ listing generation is unconditionally enabled;
. listing generation is enabled, if the **-g** command line option is specified (default).

2.5.12. 4. .LSTEXP

Syntax:

```
.LSTEXP option
```

This command controls whether macro expansions are included in the listing. False conditional fragments in macros are included in the listing file only if both the macro expansion and the false conditional block listings are enabled either with the **-c** command line option or with the .LSTCND directive. The .LSTEXP directive does not have any effect if the listing is disabled. *Option* is one of the three options:

- listing generation is unconditionally disabled;
+ listing generation is unconditionally enabled;
. listing generation is enabled, if the **-e** command line option is specified (default).

2.5.12. 5. .LSTXRF

Syntax:

```
.LSTXRF option
```

This command controls whether the cross-reference tables are included in the listing. It is recommended to use this command, for example, when the source file contains several modules. In this case, you can enable cross-reference table listing generation in some modules and disable it in the other modules. Option is one of the three options:

- listing generation is unconditionally disabled;
+ listing generation is unconditionally enabled;
. listing generation is enabled, if the **–x** command line option is specified.

2.5.12. 6. .LSTWID

Syntax:

```
.LSTWID option
```

This directive enables/disables the "wide" mode for listing generation. In the "wide" mode an additional column with either the status or the included file nesting information is added to the listing. *Option* is one of the three options:

- `-` listing generation is unconditionally disabled;
- `+` listing generation is unconditionally enabled;
- `.` listing generation is enabled, if the **–w** command line option is specified.

2.5.12. 7. .TITL

Syntax:

```
.TITL 'string'
```

Specifies the title string that will be printed out on the top of every page of the listing file.

Example:

```
.TITL 'CONTROL SYSTEM ONE'
```

2.5.12. 8. .STITL

Syntax:

```
.STITL 'string'
```

Specifies the subtitle string, which will be printed out on every listing page as a subtitle.

Example:

```
.STITL 'Keyboard service routines'
```

2.5.12. 9. .PAGE

Syntax:

```
.PAGE
```

This directive starts a new page in the listing.

2.5.13. Miscellaneous Directives

2.5.13. 1. .WARNING, .ERROR, .MESSAGE

Syntax:

```
.WARNING 'string[,string...]'
.ERROR   'string[,string...]'
.MESSAGE 'string[,string...]'
```

When Assembler encounters one of these directives, it generates a message on the console containing the *string*. The message includes the appropriate caption, file name and line number. When several character strings are specified, the Assembler concatenates them into one string in the order they are listed.

User message numbers are preset: message #0 for the .MESSAGE directive, message #1 for the .ERROR directive, and message #2 for the .WARNING directive.

Example:

```
.WARNING 'Possible problem',' with stack overflow.'
```

```
.ERROR 'Invalid macro call!', ' Check the number of parameters.'
.MESSAGE 'Complete.'
```

Display on the console:

```
Warning[2] filename.ext(1,0): Possible problem with stack overflow.
Error[1] filename.ext(2,0): Invalid macro call! Check the number of
parameters.
Message[0] filename.ext(3,0): Complete.
```

2.5.13. 2. .LNKCMD

Syntax:

```
.LNKCMD 'command_line_parameters_for_Linker'
```

Passes parameters to the Linker. This directive is used to specify some of the options for the Linker directly in the source Assembler file. When the module with the .LNKCMD directive is linked to a program, the entire *command_line_parameters_for_Linker* character sequence between the quotation marks is passed to the Linker as if it were the command line parameters. Only the following options can be used:

-A: Define the address space
-C: Enable code generating in the address space
-K: Reserve ranges in address spaces with the specified allocation
-N: Reserve ranges in address space
-S: Segment allocation
-H: Specify filename extension for HEX-file
-Z: Increase segment size

See for details *Appendix D. MCLINK Command Line Interface*.

2.5.13. 3. .OBJREC

Syntax:

```
.OBJREC expression [,expression] …
```

Directs byte data into the output object file. This directive writes the byte or a sequence of bytes, specified in the *expression*(*s*), directly to the object file. The *expression* must be absolute and reside in the range from -128 to +255.

Normally, this directive is not used. However, it can be useful for implementing some of the sophisticated operations ("tricks"). Usage of this directive expects that you are familiar enough with relocatable object file format to prevent damaging the output file.

2.6. Functions

MCA-430 supports writing assembler functions using the special directives .FUNC/.ENDF. This provides the extended development capabilities, such as:

- In Assembler modules you can use *local labels* and *variables* in functions with the scope limited to the function body.
- When creating mixed C/Assembler projects some functions can be written in Assembler and called from C modules and vice versa.
- An advantage of using the .FUNC/.ENDF directives is the opportunity to apply *performance analyzer* in debugging (for details, see *IDE Online Help*).

2.6.1. Pure Assembler Programs

The .FUNC/.ENDF pair is used to define functions in Assembler programs. Inside a function you can define and use *local names* and *labels* – identifiers starting with underscore symbols. These names or labels are visible only in the function embraced in the .FUNC and .ENDF directives. This is implemented to avoid long unique identifiers for function internal names. Local names are not included in the debugging information and can not be declared as public.

Note, when a name starting with an underline symbol has been defined outside a function prior to the .FUNC directive, this name is not local in this function but is visible inside it. An external name starting with the underline symbol can also be declared in a function. Such name is not local; its scope is not limited to area defined by .FUNC and .ENDF, and it is included in the debugging information.

Example – Local labels in a pure assembler program:

```
…
.RSEG MyData,data           ;switch to the data segment
_xVar .DSI                  ;declare variable in the MyData segment
                            ;It is NOT local in the function,
                            ;because it is declared outside.
.RSEG MyCode,code           ;switch to the code segment
.FUNC my_func               ;start function
    mov _xVar,_var          ; _xVar - normal variable
    mov #0,_var+1           ; _var - local variable
    rlc _var
    jnc _1                  ;_1 - local label
    mov #0FFh,_var+1        ;
_1: ret                     ;definition of local label _1

.OSEG MyTemp,data           ;switch to the data segment
_var    .DSI                ;declare local (in function) variable
.RSEG MyCode                ;return to the code segment
.ENDF                       ;close function
…
```

2.6.2. Mixed C/Assembler Programs

When developing a mixed C/Assembler project sometimes a better approach is to write the main program in C, utilizing all features and capabilities provided by this high level language, and write some hardware specific or time-critical functions in assembler. The function can be written in assembler, translated into a separate object module, and then called from C program, keeping with the calling convention (for details, see *MCC-430 C Compiler User's Guide*). Moreover, some assembler code can be put directly in a C program using special C *pragma* directives. Conversely, a function can be written in C and then called from an assembler module. In this case, the function should be described in Assembler module as external with the .EXTRNF directive.

It is strongly recommended to use the .FUNC/.ENDF directives to define Assembler functions and the .EXTRNF directive to declare external C functions due to the following reasons. If an assembler function is called from a C function, its prototype (specifying the function argument types and return value type) should be previously declared in the C module. If the function entry point in the assembler module is simply a label, the type of the function is **nothing** and the Linker will detect type mismatch at link time and will generate the "type mismatch" warning message. If a C function is called from an assembler module, it should be declared as an external name in the module. If the .EXTRN directive is used, the type of the function is **nothing** and the Linker also will detect type mismatch at link time and will generate the "type mismatch" warning message. Use the .EXRN and .FUNCTYPE directives to declare external functions, which are defined in data segments declared using the extended format.

The strict type checking can be disabled using the Linker **-t** command line option, which will suppress linker "type mismatch" warnings, but in this case, the linker, for example, will not detect a situation when

the type and/or number of arguments of the called function do not match with the type/number of parameters in function definition. Therefore, a better approach is to use the .FUNC/.ENDF and .EXTRNF directives to properly specify the function type. Use the predefined constants to specify types of function arguments and return value.

For details, see function declaration directives under *Assembler Directives* earlier in this chapter.

Example 1 – Declaration of an external C function in the assembler program:

```
;MY_C_FUNC is an external C function declared as:
;void MY_C_FUNC(long a,...)
;
.EXTRNF MY_C_FUNC .void (.long,.ellipsis) ;declaration of the function
…                                          ;in Assembler program
…
;Next, construct the stack frame and pass the arguments
;according to C conventions.
…
call MY_C_FUNC                            ;function call
;
;Finally, remove the stack frame and extract the return value of the
;function. In this example the void function does not have a return
;value.
…
```

Example 2 – Declaration of assembler function as C function for using in C modules:

```
.FUNC MY_ASM_FUNC .int (.uint,.char,.int_ptr)
…                           ;function body
…                           ;must comply with
…                           ;the C function calling conventions
.ENDF                       ;end of function
;
.PUBLIC MY_ASM_FUNC         ;make the function accessible in
                            ;other modules written in C
```

2.7. Macro Tools

Macros are special tool for development and use of the source text 'patterns' that allow parameter substitution. When writing programs you will notice quite often that very similar sequences of instructions are repeated several times, with minor differences. For example, when writing a program you have to make a subroutine that copies 5 bytes from one memory location to another. Later you realize that you are writing the similar code, but for copying 4 bytes. If both subroutines operate in a similar way, you will see that they are nearly identical with the exception of three parameters: number of bytes and two memory addresses. It is more convenient to create that code by means of macros than to write many times similar subroutines.

Advantages provided by macros:

- As you have to type shorter text, the probability of making errors is reduced.
- Scope of names and labels that are used only within a macro can be limited so that they are "visible" to the Assembler within a macro only. Therefore, these names do not need to be unique. Macro written in such a way can be used many times, and "internal" (local) names of the macro will not be duplicated.
- If a logical error is detected within a macro, you will have to correct it only once in contrast to the situation, when you have to scan the whole source program text in order to find similar code fragments and correct the error in all of them.
- If similar program fragments are developed with a macro, it is much easier to modify those fragments as you have just to modify the macro definition only.

- The efficiency of programmers' work increases as the time is not spent for duplicating source text. Besides, the most frequently used macros can be made available to other programmers.

Macros versus subroutines (functions):

- A subroutine (function) is a part of the program executable code; it can be called from other parts of the code at the stage of program execution. A macro is defined once in the source program text and is substituted into the program text at the compilation stage as a macro expansion. There can be more than one macro substitution.
- When a function is called, the same code is executed every time. When a macro is called, a unique sequence of program lines is formed in accordance with the pattern specified by the macro definition.
- Every function call increases the time of program execution. If you use a macro, program execution time is not increased by calls, but the total program size is increased.

The sections below contain information on macro and repeating block definition directives, macro and repeating operators and special characters, as well as using the local names in macros.

2.7.1. Defining a Macro

Before a macro can be used it should be described earlier in the program. Defining a macro the user gives it a name, which is then used to call the macro. Macro definition block starts with the .MACRO directive and ends with the .ENDMAC directive. Macro definition can contain repeated blocks and calls to other macros, but can not contain other macro definitions. Nested macro definitions are not supported.

Macro definition has the following syntax:

```
macro_name .MACRO [formal_parameters_separated_by_commas]
<macro body>
.ENDMAC
```

The *macro_name* is a name which will be used further in the program to expand the macro. You can specify formal parameters for the macro if you want it to be expanded with different actual arguments. You can give any names to the formal parameters as these names are local for the macro. Even if the name is already defined in the program, parameter name will be used in the macro.

The body is any number of Assembler statements. You can use the formal parameters in these statements.

You can use .BLANK and .PARM operators and .NPARMS predefined variable with macros. The .BLANK operator can be used inside a macro to determine if a certain actual parameter is empty (omitted). Use the .PARM operator to get access to a macro argument by the number of its position in the parameter list. The .NPARMS variable contains the number of actual parameters passed to the macro. The .NPARMS variable must also be used only within a macro definition.

2.7.1. 1. .MACRO Directive

Syntax:

```
name .MACRO [formal_parameters]
```

This directive declares the beginning of macro definition (macro body). *Name* is the macro name to be used in macro calls. Formal parameters are symbolic names defined by the user. They should be separated by commas and there can be not more than 33 parameters. When a macro is expanded, the Assembler substitutes its formal parameters with the actual parameters. The scope for the formal parameters is limited to macro body. Macro body can contain any machine instruction mnemonics and necessary directives, for instance, conditional assembling directives. Calls to macros from within other macros are supported.

2.7.1. 2. .ENDMAC Directive

Syntax:

```
.ENDMAC
```

This directive indicates the end of macro definition or a repeating block.

2.7.1. 3. .EXITM Directive

Syntax:

```
.EXITM
```

When the .EXITM directive is encountered, the macro expansion is immediately terminated. This can be convenient when a conditional block is used in the macro body. The .EXITM directive can be located only inside a macro body or a repeating block.

2.7.1. 4. Operator .BLANK

Syntax:

```
.BLANK formal_parameter_name
```

This operator can be used inside macros only. It returns .TRUE (-1) if the actual argument passed to macro in macro call to replace the parameter with *formal_parameter_name* is empty (omitted in the macro call). Otherwise, the returned value is .FALSE (0).

Example:

```
foo .MACRO par1,par2
.IF .BLANK par1 == .FALSE
        .DCB par1
        .ELSE
        .DCB 0
        .ENDIF
.IF .BLANK par2 == .FALSE
        .DCB par2
        .ELSE
        .DCB 0
        .ENDIF
.ENDMAC
.RSEG MyCode,code
foo 1,2                         ;will be expanded to:
                                ;.dcb 1
                                ;.dcb 2
foo ,2                          ;will be expanded to:
                                ;.dcb 0
                                ;.dcb 2
foo                             ;will be expanded to:
                                ;.dcb 0
                                ;.dcb 0
.END
```

2.7.1. 5. Operator .PARM

Syntax:

```
.PARM expression
```

Returns the text of the actual argument that is specified in a macro call at position indicated by *expression.* If the parameter is not specified, .PARM operator returns nothing. This operator can be used

only in macros. The argument of the operator must be an absolute expression and must contain no forward references. For instance, `.PARM 1` returns the first parameter, `.PARM 2` returns the second parameter, etc.; `.PARM 0` returns the name of the macro.

Example:

```
...
Clear .MACRO
~parm_no = .NPARMS               ;define temporary local name
        .REPT ~parm_no           ;initiate repeating block
        mov #0,.PARM ~parm_no    ;use .PARM
        ~parm_no = ~parm_no - 1  ;decrement temporary variable
        .ENDMAC
.ENDMAC
...
.RSEG MyCode,code
Clear R5,R6,R7                   ;example using the macro
...
.END
```

Generated code:

```
mov R7,#0                        ;note the order: R7,R6,R5
mov R6,#0
mov R5,#0
```

2.7.1. 6. Variable .NPARMS

You can reference this variable only in macro definition blocks. The .NPARMS value is equal to the number of actual arguments specified in the macro call. Omitted parameters are also counted. Actually, .NPARMS equals to the number of commas (,) in the macro call plus 1.

Example:

```
foo .MACRO p1,p2
.IF .NPARMS > 0
.DCB p1
.ENDIF
.IF .NPARMS > 1
.DCB p2
.ENDIF
.ENDMAC
.RSEG MyCode,code
foo 1,2              ;expands to:
                     ;.dcb 1
                     ;.dcb 2
foo 1                ;expands to:
                     ;.dcb 1
foo ,1               ;expands to:
                     ;.dcb       - error
                     ;.dcb 1
.END
```

2.7.2. Calling a Macro

Once a macro has been defined, it can be called from anywhere in the program any number of times. A macro call consists of the macro name and actual parameters. When a macro is invoked, the call is replaced with the macro body, and all the formal parameters are replaced with the actual arguments. This process is called *macro expansion*.

Macro call syntax:

```
[label:] macro_name [actual_arguments]
```

Macro_nameI is the name of macro being called. *Actual_arguments* is a list of actual arguments separated by commas. You may specify up to 33 actual arguments. *Label* is an optional label defined by the user.

Macro can be called only within the module where it is defined.
The list of actual arguments should not necessarily be the same as the list of formal parameters specified in the macro definition. However, the number of actual arguments should not be greater than the number of formal parameters (this restriction is not applied when .NPARMS or .PARM operators are used in the macro body). In the macro body the Assembler substitutes actual parameters for all tokens corresponding to the formal parameters. If the actual argument for any of the formal parameters is not specified, the formal parameter is replaced with the NULL character (removed).

Example:

```
FillB .MACRO addr,val,count     ;macro definition
mov   #addr, R5
mov   #count,R6
~loop:
mov.b #val,0(R5)
inc   R5
dec   R6
jnz   ~loop
.ENDMAC

.RSEG MyData,data               ;array in data
MyArrayLen .EQU 63              ;array length is 63
MyArray .DSB MyArrayLen         ;array definition

.RSEG MyCode,code
FillB MyArray,55,MyArrayLen     ;macro call
.END
```

Listing:

```
 1                    FillB .MACRO addr,val,count   ;macro definition
10
11                    .RSEG MyData,data             ;array in data
12 0000003F           MyArrayLen .EQU 63            ;array length is 63
13 00000000           MyArray .DSB MyArrayLen       ;array definition
14
15                    .RSEG MyCode,code
16 00000000           FillB MyArray,55,MyArrayLen   ;macro call
 + 00000000 35400000  mov   #MyArray, R5
 + 00000004 36403F00  mov   #MyArrayLen,R6
 + 00000008           ~0001~loop:
 + 00000008 F5403700  mov.b #55,0(R5)
   0000000C 0000
 + 0000000E 1553      inc   R5
 + 00000010 1683      dec   R6
 + 00000012 FA23      jnz   ~0001~loop
 +
17                    .END
```

2.7.3. Local Names in Macros

The special symbol "~" is used in macro definitions to specify a local name that will be expanded into a unique name each time the macro is called. For example, the name `~MyName` will be replaced with the

`~number~MyName,` where the *number* is a 4-digit sequential number of the current macro call in the source file.

Note: such names can not be declared as public or external, neither can they represent segment names.

Note: the "~" symbol is also used as the bitwise inversion (one's complement) in the MCA-430. Therefore, if you need to use the bitwise inversion operation within macros, you should use the .INV operator.

Example:

```
.EXTRNB (data)foo
.ASEG Regs,data
.ORG 020h
P1IN .DSB
wait_P0IN .MACRO val
~label:
 cmp.b  val,P1IN
 jnz    ~label
.ENDMAC
.RSEG MyCode,code
wait_P1IN #33
wait_P1IN foo
.END
```

Listing:

```
 1                    .EXTRNB (data)foo
 2                    .ASEG Regs,data
 3 00000010           .ORG 020h
 4 00000010           P0IN .DSB
 5                    wait_P1IN .MACRO val
10                    .RSEG MyCode,code
11 00000000           wait_P1IN #33
 + 00000000           ~0001~label:
 + 00000000 F0902100   cmp.b  #33,P1IN
   00000004 0C00
 + 00000006 FC23       jnz    ~0001~label
 +
12 00000008           wait_P1IN foo
 + 00000008           ~0002~label:
 + 00000008 D090F6FF   cmp.b  foo,P1IN
   0000000C 0400
 + 0000000E FC23       jnz    ~0002~label
 +
13                    .END
```

2.7.4. Repeating Blocks

Repeating blocks are a tool that allows organizing repetition of part of the code without the need to write the same code several times. The repeating block is always expanded in the same place where it is defined. Special repeating block declaration Assembler directives (.REPT, .IRP, and .IRPC) provide a way to use a set of arguments that will consequently replace the formal parameter in the repetitions.

Repeating block has the following syntax:

```
header
body
.ENDMAC
```

The *body* contains any number of statements, including other repeating blocks or macro definitions. The .ENDMAC is the directive that indicates the end of the *body* and the repeating block. The *header* is one of three directives: .REPT, .IRP, or .IRPC.

Repeating blocks are a special case of macros. In fact, they are processed by the MCA-430 in a very similar way to macros, with slight differences.

2.7.4. 1. .REPT Directive

Syntax:

```
.REPT expression
```

This directive instructs the Assembler to repeat a repeating block body the number of times specified by *expression.* The expression must be absolute and contain no forward references. The value of the expression must be within the 0-65535 range.

Example:

```
.RSEG CSEG,code
PARM = 0
.REPT 5
.dcb PARM
PARM = PARM + 1
.ENDMAC
.END
```

Listing:

```
1                   .lstexp +
2                   .RSEG CSEG,code
3 00000000          PARM = 0
4                   .REPT 5
+ 00000000 00       .dcb PARM
+ 00000001          PARM = PARM + 1
+ 00000001 01       .dcb PARM
+ 00000002          PARM = PARM + 1
+ 00000002 02       .dcb PARM
+ 00000003          PARM = PARM + 1
+ 00000003 03       .dcb PARM
+ 00000004          PARM = PARM + 1
+ 00000004 04       .dcb PARM
+ 00000005          PARM = PARM + 1
8                   .END
```

2.7.4. 2. .IRP Directive

Syntax:

```
.IRP formal_parameter,(actual_arguments)
```

Consequently repeats the repeating block body with each of *actual_arguments.* Each time the block is assembled, the next actual argument in the list becomes the substitute for *formal_parameter.* The number of repetitions equals to the number of actual arguments. Actual arguments must be separated by commas; number of arguments may not exceed 32.

Example:

```
.RSEG CSEG,code
.IRP PARM,(25,36,47,58,69)
.dcb PARM
```

```
.ENDMAC
.END
```

Listing:

```
1                   .lstexp +
2                   .RSEG CSEG,code
3                   .IRP PARM,(25,36,47,58,69)
+ 00000000 19       .dcb 25
+ 00000001 24       .dcb 36
+ 00000002 2F       .dcb 47
+ 00000003 3A       .dcb 58
+ 00000004 45       .dcb 69
5                   .ENDMAC
6                   .END
```

2.7.4. 3. .IRPC Directive

Syntax:

```
.IRPC formal_parameter, 'string'
```

Consequently repeats the repeating block body with each of the characters in the *string*. Number of repetitions is determined by the number of characters in the string. Each time, the specified formal parameter is replaced with the next character enclosed in single quotes, i.e. with a string containing one character. The length of the string is limited to 400 characters.

Example:

```
.RSEG CSEG,code
.IRPC PARM,'HELLO'
.dcb PARM
.ENDMAC
.END
```

Listing:

```
1                   .lstexp +
2                   .RSEG CSEG,code
3                   .IRPC PARM,'HELLO'
+ 00000000 48       .dcb 'H'
+ 00000001 45       .dcb 'E'
+ 00000002 4C       .dcb 'L'
+ 00000003 4C       .dcb 'L'
+ 00000004 4F       .dcb 'O'
5                   .ENDMAC
6                   .END
```

You can also use .PARM operator to get the text of the actual arguments inside repeating blocks. Some special operators and other special features can also be used in macros. See details for the .BLANK and .PARM operators (see *Macro Operators*, Chapter 2) and the .NPARMS directive, earlier in this chapter.

2.7.5. Special Characters in Macros and Repeating Blocks

Special characters are symbols that you can use to implement certain programming techniques that are useful for making the code shorter, more self-descriptive, and less erroneous. For instance, you may want to call a macro from another macro or use a few variables in a repeating block that vary only in the last digit. The following special characters are used in macro and repeating block processing: <>, ^, {}, and %.

2.7.5. 1. Passing Actual Parameters (<>)

Name, expression, or character string are normally passed as actual arguments. If it is necessary to pass a parameter that contains separators (for example, commas or spaces) and which is not a character string (is not enclosed in single quotes), this parameter must be enclosed in the angular brackets: "<" and ">".

Every time a parameter in the angular brackets is passed to a macro, the brackets are removed by the macro preprocessor. Therefore, a parameter that needs to be passed from within one macro to another should be enclosed in double angular brackets: <<1,2>>.

Example:

```
CAT .MACRO par1
.DCB par1
.ENDMAC
RAT .MACRO par1,par2
CAT par1
.DCW par2
.ENDMAC
.RSEG CSEG,code
RAT     0,1
RAT     <<3+1,4+2>>,<7+8,9+10>
.END
```

Listing:

```
 1                    CAT .MACRO par1
 4                    RAT .MACRO par1,par2
 8                    .RSEG CSEG,code
 9 00000000           RAT     0,1
 + 00000000           CAT 0
 + 00000000 00        .DCB 0
 +
 + 00000002 0100      .DCW 1
 +
10 00000004           RAT     <<3+1,4+2>>,<7+8,9+10>
 + 00000004           CAT <3+1,4+2>
 + 00000004 0406      .DCB 3+1,4+2
 +
 + 00000006 0F001300  .DCW 7+8,9+10
 +
11                    .END
```

2.7.5. 2. Actual Parameters and Text Concatenation in Macros (^)

The "^" character in macros and repeating blocks is used for concatenation. If there is the "^" character on the left (right) of the formal parameter in a macro definition, it is removed when the macro is expanded, and the corresponding actual argument is concatenated with the text preceding (following) the "^" character. Spaces and tabs adjacent to "^" do not affect concatenation in any way, i.e. when macro is expanded, the spaces and tabs are removed with "^". In other words, the "^" character applied to a formal parameter in a macro definition works as a special concatenation operator.

Note that "^" is also used as a bitwise logical XOR operation, so if a bitwise logical XOR is to be applied to a formal parameter in macro, the .XOR operator must be used.

Note, concatenation inside character strings enclosed in quotes does not work. Thereby, if a macro definition contains the following text:

```
'^parameter^'
```

where *parameter* is a formal parameter name then this string will not be modified at macro expansion.

Example:

```
.lstexp +
Const_66 .equ 1
CONCAT .MACRO p1,p2
label_ ^ p1:
  mov #Const_ ^ p2,R5
.ENDMAC
.RSEG MyCode,code
CONCAT 55,66
.END
```

Listing:

```
1                    .lstexp +
2 00000001           Const_66 .equ 1
3                    CONCAT .MACRO p1,p2
7                    .RSEG MyCode,code
8 00000000           CONCAT 55,66
+ 00000000           label_55:
+ 00000000 1543        mov #Const_66,R5
+
9                    .END
```

Actual arguments can not be concatenated with the text starting with a **number**:

```
X .MACRO Y
mov #0,Y ^ 2_lo    ;concatenation will not occur here
mov #0,Y ^ _2_lo   ;but will occur here
.ENDMCA
```

2.7.5. 3. Parameter-to-String Conversion ({})

If a formal parameter in a macro or a repeating block definition is enclosed in braces ({}), the braces will be replaced with single quotes when the macro or repeated block is expanded. This operation is useful when an actual argument needs to be converted to a string after it has been passed to macro.

Spaces and tabs placed between the braces and formal parameters are ignored and removed with the "{" and "}" characters.

Example:

```
BOO .MACRO p1
label_ ^ p1 .DCB 'Actual parameter: ',{p1},0
.ENDMAC
.RSEG MyCode,code
BOO lambada
.END
```

Listing:

```
1                    BOO .MACRO p1
4                    .RSEG MyCode,code
5 00000000           BOO lambada
+ 00000000 41637475  label_lambada .DCB 'Actual parameter: ','lambada',0
  00000004 616C2070
  00000008 6172616D
  0000000C 65746572
  00000010 3A206C61
  00000014 6D626164
```

```
  00000018 6100
+
6                       .END
```

2.7.5. 4. Expression-to-Value Conversion in Macro Calls (%)

When a percentage sign % is put before an actual argument in a macro call, the argument value is computed by the Assembler. Instead of replacing the formal parameter with the actual argument itself, the Assembler calculates the actual argument and then puts in the calculated value.

Decimal representation is used in substitution with leading zeroes removed. The actual argument that the % sign is used with, must be an absolute expression with no forward references.

Example:

```
MOO .MACRO p1
label_ ^ p1: mov #p1,R5
.ENDMAC
.RSEG MyCode,code
MOO %66+22
.END
```

Listing:

```
1                       MOO .MACRO p1
4                       .RSEG MyCode,code
5 00000000              MOO %66+22
+ 00000000 35405800     label_88: mov #88,R5
+
6                       .END
```

2.7.6. Nested Macro Calls and Definitions

In a macro definition, you can specify a call to another macro that has been defined. The result will appear when the parent macro is called. The maximum nesting level is limited only by the available memory space.

Repeating blocks and macro definitions can contain other repeating blocks. The maximum nesting level of the repeating blocks is limited only the available working memory.

Note: the MCA-430 does not support nesting macro definitions in other macro definitions or repeating blocks.

2.8. TI MSP430 Architecture Support

2.8.1. Operand Attributes Checking in Instructions

The Assembler performs checking of the operand attributes to detect improper use of operand(s) with the single-operand and double-operand instructions, as well as conditional jumps. For example, if a byte operation instruction is used with an operand with the WORD 'operand type' attribute, the Assembler will produce a warning message.

2.8.1. 1. Double-Operand Instructions

If symbolic or absolute addressing modes are used, the Assembler checks that the 'operand type' attribute of source/destination operand is BYTE in byte instructions and WORD in word instructions. If a type mismatch is detected then the Assembler produces a warning message. If symbolic or absolute addressing mode is used with the destination operand, the Assembler checks that the 'allocation' attribute of the operand is **data**. If the 'allocation' attribute is **code** then the Assembler produces a warning about an attempt to modify constant data. These warnings can be suppressed by the **-a** Assembler option.

The Assembler/Linker controls that the value of operand in immediate addressing mode is within 0-0FFh in byte instructions and within 0-0FFFFh in word instructions. If the Assembler (or the Linker, if value is absolute) estimates that the value of an operand is out of allowable range, it produces the error message. Such Assembler diagnostic message can not be disabled with the **-a** Assembler option.

2.8.1. 2. Single-Operand Instructions

If symbolic or absolute addressing modes are used, the Assembler checks that the 'operand type' attribute of the operand is BYTE in byte instructions (RRA.B, RRC.B) and WORD in word instructions (RRA, RRC, SWPB, SXT). Also, when symbolic or absolute addressing modes are used with these instructions, the Assembler checks that the 'allocation' attribute of the operand is **data**. Warning messages are produced if a type mismatch or an allocation mismatch is detected. Such warnings can be suppressed by the **-a** Assembler option. The Assembler checks that the immediate operand addressing mode is never used with the RRA, RRC, SWPB, and SXT instructions, as this can lead to unpredictable program operation.

The PUSH instruction operand attributes are checked similar to double operand instruction source operand attributes.

Allowable attributes of the CALL instruction operand:

Addressing mode	Operand	Operand Type	Allocation	Value Range
Direct	`Rm`	–	–	–
Index	`X(Rm)`	Any	Any	0..0FFFFh
Symbolic	`ADDR`	WORD	**code**, **data**	200h..0FFFFh
Absolute	`&ADDR`	WORD	**code**, **data**	200h..0FFFFh
Indirect, Indirect auto increment	`@Rm` `@Rm+`	–	–	–
Immediate	`#N`	UNTYPED	**code**, **data**	200h..0FFFFh

The Assembler checks allocation attribute of operands in immediate addressing mode in the CALL instruction. If such operand has the allocation attribute **data** and the operand is not a label in a segment defined in the current module using the extended format of segment declaration directive, then a warning message is produced. Setting the **-r** Assembler option suppresses such warnings.

2.8.1. 3. Conditional Jumps

The Assembler will try to detect a jump made to the data segment. When a name or label is used as the jump destination in JEQ/JZ, JNE/JNZ, JC, JNC, JN, JGE, JL, and JMP instructions, the Assembler checks that the allocation attribute of the name is **code**. In other cases, the Assembler produces a warning message.

The Assembler checks that:

- the destination address is aligned on even boundary;
- the value of operand is within $(-1024+PC_{old}+2)..(+1022+PC_{old}+2)$;
- the jump destination is below the 64K boundary and above the 0x200h address, which is the upper boundary of address range reserved for the SFRs and peripheral modules.

If an external name is used as operand, the above checks are performed by the Linker.

2.8.1. 4. Emulated Instructions

Allowable attributes of the BR instruction operand are the same as of the CALL instruction operand. Other emulated instructions operands undergo the same verification as the destination operands in double operand instructions.

2.8.2. Alignment

All instructions are aligned on even addresses, i.e. word boundary. When instruction is generated and the program counter value is odd, the Assembler increments the program counter.

Bytes can be located at even or odd addresses. Words can be located only at even addresses. The Assembler performs alignment of words on even addresses. If an external byte operand is located at an odd address and is used in the current module as operand in a word instruction, the Linker detects misalignment and produces the "alignment error" message.

2.8.3. Implementation of Immediate Addressing

When immediate addressing mode is used and the operand is one of the -1, 0, 1, 2, 4, and 8 values, the Assembler replaces such operand with an operand in direct addressing mode using one of the two constant generator registers R2 and R3.

Example 1 – Immediate value is in the set of generated constants:

```
MOV #4,R12
replaced with:
MOV @R2,R12
```

Example 2 – Immediate value is not in the set of generated constants:

```
MOV #44,R12
replaced with:
MOV @PC+,R12
.DCW 44h
```

Note, if the PUSH instruction is used with the "4" and "8" values, the constant generator registers are not used due to the known "CPU4" error in the MSP430.

2.9. Programming with MCA-430

2.9.1. SFRs and Peripheral Module Registers

The Special Function Registers (SFRs) and peripheral devices are mapped into lower 512 memory addresses of the DATA address space. Thereby, the 0-1FFh address range is always reserved to prevent the Linker from allocating relocatable segments in that range. The supplied include files containing special function registers and bits definitions are located in the `Inc\` directory. Note that when working with the IDE there is no need to manually specify this file in the program, as the necessary include file is always included in the program.

2.9.2. Stack Initialization

The stack pointer (SP) must be initialized before any stack operations are performed. For this purpose, you need to declare a stack segment in the DATA address space, set the size of the stack, and set the SP register equal to the end address of the stack segment (stack is filled from the upper addresses down to the lower addresses). The size of the stack segment can be set using the .DS directive in the program, reserving necessary address range. Example:

```
.rseg STACK, data          ;declare the stack segment
   .ds 100h                ;reserve 256 bytes in the STACK segment

.rseg CODESEG, code
start:                     ;beginning of the program
...
   mov #.sfe STACK, R5     ;load the end address of STACK to R5
   add #1, R5              ;round up to even address
   and 0FFFEh, R5
   mov R5, SP              ;load R5 to Stack Pointer
```

You can also use the Linker **-Z** command line option to set the size of the stack segment or use the appropriate setting in the IDE (see *IDE Online Help* for details). Example:

```
-Z STACK(100h)            #set the STACK segment size to 256 bytes
```

2.9.3. Setting Interrupt Vectors

The upper 16 words of the CODE address space starting from 0FFE0h up to 0FFFEh are always reserved for storing hardware interrupt vectors (i.e., interrupt service routine entry addresses). The top interrupt vector located at the 0FFFEh address has the highest priority and always points to the start address of the program. The interrupt vectors should be initialized with proper values of the interrupt service routine (ISR) entry addresses. Example: assume the INT_5_ISR routine is to be assigned to service the interrupt with priority #10 (Watchdog timer). The following is the sample code for initialization of the interrupt vector:

```
.aseg INTVEC, code        ;Start/select interrupt vector segment.
.org 0FFF4h               ;Set program counter equal to
                          ;the address of the interrupt vector.
.db2 INT_5_ISR            ;Initialize the interrupt vector with
                          ;the ISR entry address.
```

In the above example the program must have the code for servicing the interrupt with priority #10 starting from the address 0FF00h and ending with the RETI instruction.

See *MCC-430 C Compiler User's Guide* for information about adding interrupt handling routines to the mixed C/Assembler program.

2.9.4. Assembly Program Example

The following example demonstrates the basic steps in writing an assembler program:

```
;Initialization of the start procedure:
.pmodule ?RESET
.aseg RESET, code
.extrnf MAIN                  ;MAIN is the start function of the program
.org 0FFFEh
.dcw MAIN                     ;Set the reset vector to MAIN
.endmod

;'Program' module declaration:
.pmodule PROG
.include 'c1331.inc'
.rseg STACK,data              ;declare a segment for stack
.rseg CODESEG,code            ;declare a segment for code

;Declare segment for variables in the RAM:
.rseg DATASEG,data
MyByteVar     .dsb            ;byte (unsigned char)
MyWordVar     .dsw            ;word (unsigned int)
MyDwordVar    .dsd            ;dword (unsigned long)
.public       MyWordVar       ;make accessible for other modules

;Define an array of words:
MyArrayLength .equ 50
MyArray       .dsi MyArrayLength

;Declare a segment for tables of constants:
.rseg CONST,code
TableByte     .dcb 55,66
              .dcb 77,88
TableFloat    .dcr "3.3, "3.4, "3.5
              .dcr "3.6, "3.7, "3.8
```

```
MyString      .dcb 'Hello, world!'

.rseg CODESEG                 ;switch to code segment
.func MAIN                    ;start main function

;The function MAIN should be declared public, then its address will be
;written as the reset vector (jump is made to MAIN upon reset)

.public MAIN

;Stack initialization:
mov   #.sfe STACK, R5         ;load end address ot the STACK to R5
add   #1, R5                  ;round up to even boundary
and   #0FFFEh, R5
mov   R5, SP                  ;load R5 to Stack Pointer

mov   #55h, R5                ;just some sample code
mov   #33h, R6                ;
add   R5,R6                   ;
clr   MyWordVar               ;
mov.b #20, MyByteVar
mov.b MyByteVar, .BYTE MyWordVar
swpb  MyWordVar
bic   MyWordVar, R6
;…
.extrnf MY_PROCEDURE          ;declare an external function
call #MY_PROCEDURE            ;call the function
jmp  $
.endf                         ;end of the MAIN function
.endmod                       ;end of module

.PMODULE MOD1                 ;open another module
.extrnw (data)MyWordVar       ;external WORD variable in the RAM
.public      MY_PROCEDURE     ;declare this function as public
.rseg MYCODE,code             ;open code segment
.func MY_PROCEDURE            ;start function
     inc MyWordVar
;    …
     ret
.endf                         ;end of function
.end                          ;end of last module and end of file

.LMODULE LIB1                 ;LIB1 will be linked only if
.rseg LIBCODE, code           ;the function MyLibFunc
.func MyLibFunc               ;is used in the program
.public MyLibFunc
;…
ret
.endf
.end
```

Chapter 3. Linker

The MCLINK Linker links modules contained in object files and libraries, into a single executable file. Linking process consists of the following stages:

- Establishing a set of modules for linking and resolving external references
- Determining the size and composition of the address spaces
- Determining the size and composition of segments
- Allocation of segments in appropriate address spaces
- Calculation of absolute values of the names defined in relocatable segments
- Evaluation of relocatable external references values
- Generating the output files and the report (MAP-file)

3.1. Command Line Format

Usage:

```
MCLINK [options] [prefix] obj_or_lib_file [prefix] obj_or_lib_file …
```

`Obj_or_lib_file` is an object file produced by the Assembler or Compiler or a library made with the MCLIB Librarian. Options each start with the "-" (minus character), followed by a flag letter which selects the option. Options may be given in any order or omitted entirely. The flag letter may be followed (with an intervening blank) by additional text relating to the option. Options are separated from each other and from the source name by blanks. Options are case sensitive. The Linker options can also be set from the IDE (see *IDE Online Help*).

If the object files have standard filename extensions (.MCO or .MCL) then they can be omitted.

The following command line options are accepted by the MCLINK:

Option	Description
-A	Define the address space
-C	Enable code generating in the address space
-K	Reserve ranges in the address spaces with the specified allocation
-N	Reserve ranges in the address space
-S	Segment allocation
-E	Specify output file name and target directory
-O*path*	Specify search paths for object files
-F	Specify output file format
-H	Specify filename extension for HEX-file
-Z	Increase segment size
-m	Generate a MAP-file
-M	Skip the specified section in MAP-file
-t	Disable type checking in the external names
-w	Linker warnings control
-h or -?	Display brief description of options
-p, -l, -o	Prefixes: change the module types
@*filename*	Append a response file to the command line

3.2. Modules for Linking

At the first stage, the Linker determines the set of modules to be linked. For this purpose, the Linker scans one after one object and library files that are specified in the command line. Normally, this requires more than one pass.

Note that all program modules are included in the list of modules for linking on the first pass.

On the second pass, the Linker checks the list of modules and adds any library modules to the list only if any public names defined in these modules can resolve external references yet unresolved.

At the end of each pass, the Linker checks for the following conditions satisfied:

- If all the external references are resolved, the Linker stops scanning input files and proceeds to the next linking phase. If there are still some unresolved references, the Linker initiates the next pass.
- If at a certain pass the Linker has not added any more modules to the list and yet there are some unresolved references, scanning of the input files is stopped and an error message is produced.

This multi-pass procedure enables the user to list the library files in the command line in any desirable order.

Note: all program modules are linked always prior to library modules. All library modules, in turn, are always linked prior to low-priority library modules.

It takes less time for the Linker to determine if a module needs to be linked when the module is located in a library rather than when it is located in an object file. The reason for this is a special header in the library, which contains the list of external and public names used in all modules included in the library.

See also: *Chapter 4. MCLIB Librarian*.

3.3. Resolving External References and Type Checking

The Linker resolves references to external names in a module by searching for matching public names in other modules. If a matching public name is found, all attributes ('allocation', 'operand type', 'type') of the public name and the external name are compared. If any one of the attribute value is varying, the warning of a possible error is produce. Type checking is disabled if the **-t** command line option is specified.

Note, an error message is displayed if the same public names are declared in two different modules.

Relocatable external references like any other relocatable references are resolved only after all segments have been allocated.

3.4. Setting up Address Spaces

The Linker always sets up the following *standard address spaces*:

Address Space	Allowed Address Range
CODE	0200h – 0FFFFh
DATA	0000h – 0FBFFh

The size of the standard address space can be changed using the Linker **-A** command line option to preserve some area in the memory for special purposes.

See also: *Address Spaces*, Chapter 1.

3.5. Linking Relocatable and Overlay Segments

After the set of modules for linking has been established, the Linker examines these modules and defines the names of all segments in the program. Segments that are used in several modules should have identical definitions in each module (and therefore must have identical type – absolute, relocatable or overlay – and identical allocation attribute) otherwise the error message is generated.

Relocatable segments. The Linker arranges the fragments in a single relocatable segment (a fragment is a part of a segment defined in one module). When a fragment is added, its alignment attribute has an effect, thus the resulting segment may contain unused gaps between the fragments. After binding is completed, the Linker knows the size and the alignment attribute of the segment.

Overlay segments. The Linker overlays all fragments with identical definitions instead of concatenating these fragments. The Linker takes the size of the largest fragment and sets it for the segment size. All overlay-type fragments with identical definitions are allocated starting from the same physical address.

See also: *Segment Declaration and Selection* in *Assembler Directives*, Chapter 2.

3.6. Segment Allocation

After binding of the relocatable segments is completed, the Linker starts allocating segments in the address spaces.
You can specify the left or right boundary for a segment using the Linker **-S** options.

The main principles of the segment allocation process are the following:

- The Linker checks whether address ranges of the address spaces CODE and DATA intersect. If they do, the Linker generates an error message.
- The segments specified in the Linker **-S** options are allocated first, and then the rest of the segments are allocated automatically.
- Absolute segments are automatically taken into account when relocatable and overlay segments are allocated. Address area occupied by the absolute segment with a particular allocation attribute is preserved in the address spaces with that allocation.
- The Linker always allocates relocatable segments so that they do not overlap with each other.
- When allocating relocatable and overlay segments, the Linker considers their alignment attributes (see also the .ALIGN directive description in *Address Control* section in Chapter 2).

If the left boundary (start address) or the right boundary (end address) is specified for a relocatable segment then the Linker pushes the segment as close to the boundary as possible. You may not specify boundaries for the absolute segments. A segment that is not specified in any of the **-S** options is automatically allocated in the appropriate address space. If such a segment is relocatable, it is automatically put as close to the lower address of the address space as possible.

After all segments have been allocated, all names (including public names) obtain absolute values.

Chapter 4. MCLIB Librarian

Library or a library file is a collection of object modules, which is created from the object files using MCLIB Library Management utility. Object file may contain one or several modules but the library is more than just a set of object modules. The main advantage of libraries compared to object files is a special header called *catalogue*, which is present in libraries and used by the Linker for scanning the libraries. This header enables the Linker to scan the libraries much faster, which is critical when linking large projects.

MCA-430 Assembler supports three types of modules: *program modules*, *library modules*, and *low-priority* library modules. After all of the individual input files (object or library files that are specified in the MCLINK command line) have been scanned and all found program modules have been linked unconditionally, the Linker tries to resolve any unresolved external references by including files from the libraries. This process is iterative. If the Linker has included a file from the library on one pass, it should make another pass to resolve any symbols required by the newly included files. If the Linker is able to resolve a reference with any of the usual libraries, it starts to scan low priority libraries. Low-priority library modules are not included in the program unless they contain any public names needed to resolve the external references yet unresolved with usual library modules.

In Assembler source file, module types are defined by the header directives .PMODULE, .LMODULE, or .LMODULE2. Module type can be temporarily redefined with the Linker command line filename prefixes **-p**, **-l**, and **-o**. These prefixes instruct the Linker to consider all modules in a specific object file to be program, library, or low-priority library modules respectively. To permanently change the type of a module residing in a library, use MCLIB.

The MCLIB Librarian is a tool for manipulating object files and creating, modifying, and servicing libraries. The Librarian can add, delete, extract, and replace modules in libraries as well as change attributes of the modules. Using the Librarian helps to convey the modular programming approach and easier coordinate large and extensive team projects.

See also: *Modules for Linking*, Chapter 3; *Module Declaration* in section *Assembler Directives*, Chapter 2.

4.1. Command Line Format

Usage:

```
MCLIB.EXE [option] [library_file] [object_file1] [object_file2] …
```

or

```
MCLIB.EXE [option] [library_file] [module1] [module2] …
```

The default library extension is .MCL.

Note, only one command line option can be specified for the Librarian at a time. In order to perform several operations with a library, launch the MCLIB several times. All options are case sensitive.

The following command line options are accepted by the MCLIB:

Option	Description
-a	Add modules to library
-d	Delete modules from library
-r	Replace modules in library
-x	Extract modules from library into object files
-X	Extract modules from library into a single object file
-m	Move modules from library into object files

-M	Move modules from library into a single object file
-l	Display library header on the console
-P	Assign “program” attribute to modules
-L	Assign “library” attribute to modules
-O	Assign “low-priority library” attribute to modules
-h or -?	Display the list of the Librarian’s options on the console
@*filename*	Append a response file to the command line

Chapter 5. MCDUMP Object-to-Text Converter

The MCDUMP Object-to-Text Conversion utility (text dumper) is a universal tool providing access to various source file formats: object files (.MCO), libraries (.MCL), and executable files (.MCE). The text data is directed to the standard output device (console).

5.1. Command Line Format

Usage:

```
MCDUMP.EXE [option] library_or_object_file
```

If you specify no options then the contents of the entire file will be displayed on the screen. If more than one option is specified, only the last one will have effect and the others will be ignored.

The following are the MCDUMP options:

Option	Description
-e	List external names and module names
-m	List module names only
-p	List public names and module names
-s	List segment names and module names
-H	List contents of library header
-r	Do not replace numbers with symbols (raw mode)
-h or -?	Display brief description of MCDUMP options on console
@*filename*	Append a response file to the command line

Appendix A. Assembler Directives Summary

Directive	Description	Section
=	Define an assembler variable	Symbol Definition
.ALIGN	Align program counter	Address Control
.ASEG	Select/declare an absolute segment	Segment Declaration and Selection
.BYTE	Assign BYTE operand type to an UNTYPED name	Assignment of Attributes to Name
.DB1	Initialize memory with byte values	Memory Initialization
.DB2	Initialize memory with word (2 byte) values	Memory Initialization
.DB4	Initialize memory with double word (4 byte) values	Memory Initialization
.DCx	Initialize memory with data of certain type	Memory Initialization
.DEFINE	Define a symbolic name for all modules in a file	Symbol Definition
.DS	Reserve the specified number of bytes	Memory Reservation without Initialization
.DSx	Define a name of certain type	Memory Reservation without Initialization
.DWORD	Assign DWORD operand type to a name of no type	Assignment of Attributes to Name
.ELSE	Continue a conditional assembly block – FALSE part	Conditional Assembling
.END	End module and source file	Module Declaration
.ENDF	End function definition	Function Declaration
.ENDIF	End a conditional assembly block	Conditional Assembling
.ENDMAC	End macro definition	Macro Tools: Defining a Macro
.ENDMOD	End module	Module Declaration
.ENDSEG	End segment	Segment Declaration and Selection
.EQU	Define a symbolic name for the current module	Symbol Definition
.ERROR	Display an error message on the console	Miscellaneous Directives
.EXITM	Terminate macro expansion	Macro Tools: Defining a Macro
.EXTRN	Declare an external name with type nothing	Program Linkage
.EXTRNF	Declares an external function	Program Linkage
.EXTRNx	Declares a typed external name	Program Linkage
.FUNC	Define an Assembler/C function	Function Declaration
.IF	Start a conditional assembly block – TRUE part	Conditional Assembling
.INCLUDE	Include a file into the source file	Directive .INCLUDE
.IRP	Repeating block with arguments	Macro Tools: Repeating Blocks
.IRPC	Repeating block with string of characters as arguments	Macro Tools: Repeating Blocks
.LABELx	Define a label of certain type	Symbol Definition
.LMODULE	Begin a library module	Module Declaration

.LMODULE2	Begin a low-priority library module	Module Declaration
.LNKCMD	Pass parameters to Linker	Miscellaneous Directives
.LSTCND	Control output of false conditionals	Listing Control
.LSTEXP	Control output of macro expansions	Listing Control
.LSTMAC	Control output of macro definitions	Listing Control
.LSTOUT	Control listing output	Listing Control
.LSTWID	Enable "wide" listing format	Listing Control
.LSTXRF	Control output of cross-references	Listing Control
.MACRO	Start macro definition	Macro Tools: Defining a Macro
.MESSAGE	Display a text message on the console	Miscellaneous Directives
.OBJREC	Write sequence of bytes in the output object file	Miscellaneous Directives
.ORG	Set program counter	Address Control
.OSEG	Select/declare an overlay segment	Segment Declaration and Selection
.PAGE	Insert a page break	Listing Control
.PMODULE	Begin a program module	Module Declaration
.PUBLIC	Declare a public name	Program Linkage
.REPT	Start a repeating block	Macro Tools: Repeating Blocks
.RSEG	Select/declare a relocatable segment	Segment Declaration and Selection
.SET	Define an assembler variable	Symbol Definition
.STITL	Specify a subtitle	Listing Control
.TITL	Specify a title	Listing Control
.TYPE	Assign type to an untyped name	Assignment of Attributes to Name
.WARNING	Display a warning message on the console	Miscellaneous Directives
.WORD	Assign WORD operand type to a name of no type	Assignment of Attributes to Name

Appendix B. Assembler Operators and Variables Summary

Operator	Description	Section
-	Arithmetic subtraction	Addition and Subtraction
-	Two's complement	Bitwise Operators
~ or .INV	One's complement (bitwise inversion)	Bitwise Operators
& or .AND	Bitwise logical AND	Bitwise Operators
*	Multiplication	Multiplication and Division
/	Unsigned division	Multiplication and Division
^ or .XOR	Bitwise logical XOR	Bitwise Operators
| or .OR	Bitwise logical OR	Bitwise Operators
+	Arithmetic addition	Addition and Subtraction
<, .LT	Signed LESS THAN	Relational Operators
<=, .LE	Signed LESS THAN OR EQUAL	Relational Operators
>, .GT	Signed GREATER THAN	Relational Operators
>=, .GE	Signed GREATER THAN OR EQUAL	Relational Operators
<>, .NE	NOT EQUAL	Relational Operators
==, .EQ	EQUAL	Relational Operators
.ALLOCATION	Allocation of the operand	Miscellaneous Operators
.BLANK	Macro argument is empty (true/false)	Macro Tools: Defining a Macro
.BYTE	BYTE type assignment to operand	Setting Operand Type of Expression
.BYTE3	Lower byte of the operand’s higher word	Byte/Word Extraction Operators
.BYTE4	Higher byte of the operand’s higher word	Byte/Word Extraction Operators
.DATE	Current date and time	Miscellaneous Operators
.DEFINED	Checks if the name supplied as operand is defined	Miscellaneous Operators
.DWORD	DWORD type assignment to operand	Setting Operand Type of Expression
.HIGH	Higher byte of the operand	Byte/Word Extraction Operators
.HWRD	Higher word of the operand	Byte/Word Extraction Operators
.IDIV	Signed division	Multiplication and Division
.IMOD	Signed remainder selection	Multiplication and Division
.LOW	Lower byte of the operand	Byte/Word Extraction Operators
.LWRD	Lower word of the operand	Byte/Word Extraction Operators
.MOD	Unsigned remainder selection	Multiplication and Division
.NOT	Logical negation	Bitwise Operators
.OFFSET	Displacement of the operand from segment's origin	Miscellaneous Operators
.OPTYPE	Operand type of expression supplied as operand	Miscellaneous Operators
.PARM	Text of actual macro argument specified by operand	Macro Tools: Defining a Macro

.SFB	Start address of the segment	Miscellaneous Operators
.SFE	End address of the segment	Miscellaneous Operators
.SHL	Arithmetic left shift	Shift Operators
.SHR	Arithmetic right shift	Shift Operators
.SHRL	Logical right shift	Shift Operators
.TYPE	Type of the name specified as operand	Miscellaneous Operators
.UGT	Unsigned GREATER THAN	Relational Operators
.ULT	Unsigned LESS THAN	Relational Operators
.UNTYPED	UNTYPED type assignment to operand	Setting Operand Type of Expression
.WORD	WORD type assignment to operand	Setting Operand Type of Expression

Assembler predefined variables:

Variable	Returned Value	Section
.NPARMS	The number of actual arguments in a macro call	Macro Tools
.UPPERCASEONLYMODE	True/False depending on the current case sensitivity mode	Predefined Variables

Appendix C. Assembler Command Line Interface

@filename

Include the contents of the file filename (response file) when processing the command line. There are no size limits for the response file. This makes it possible to specify any number of command line options in it.

There should be no spaces between @ and filename. The <CR> (carriage return) and <LF> (line feed) characters are ignored in the response file. Comments start with the "#" symbol and end with the end of line. If not specified, the extension is expected to be .OPL.

-I*path*: Search for include files in the specified directories

Specifies the search paths for include files that are specified without path in the source text. The files are first searched in the current directory and then in the directories listed in this option. The directories are searched in the order listed, consequently until the file is found. Directory names are separated by semicolons.

For example, if the "`-I..\include;c:\mca430\inc;loc`" option is specified, the .INCLUDE directives in the source text will have the following effect:

```
.INCLUDE 'mem.inc'         ;MEM.INC will be searched in the following
                           ;directories: current, then
                           ;"..\INCLUDE" ("INCLUDE" in the parent
                           ;directory), then
                           ;"C:\MCA430\INC", then
                           ;"LOC" subdirectory of the current directory.

.INCLUDE 'lib\config.inc' ;CONFIG.INC will be searched in the following
                           ;directories:
                           ;"LIB" subdirectory of current directory, then
                           ;"..\INCLUDE\LIB", then
                           ;"C:\MCA430\INC\LIB", then
                           ;"LOC\LIB" (two levels below current).
```

-u: Ignore character case

If this option is specified in the command line, the Assembler converts all user identifiers (labels, segment names, macro names) to upper case symbols. This option eases migration from the case-insensitive assemblers provided by other vendors to the MCA-430 Assembler.

If the option is not specified, the Assembler will work in a case-sensitive mode and distinguish upper and lower case letters in symbolic names (for example, `BUFFER` and `BuFFer` will be different identifiers).

See also: .UPPERCASEONLYMODE in *Predefined Variables*, Chapter 2.

-d: Generate debugging information

Include debugging information in the object file. This information is used for the source-level debugging and allows, for example, the Linker to refer in error messages to a position in the source file. By default, the debug information is not included in the object file.

-a: Disable instruction operand type checking

When processing instructions, the Assembler checks if the instruction operand attributes match the instructions. For example, when processing a jump instruction, the Assembler checks if the operand (jump destination address) has **code** allocation. Attribute mismatch results in generating warnings. Most of these warnings can be disabled with the **-a** option.

It is not recommend using this option when assembling generic assembler programs since operand attribute checking helps to detect a large number of accidental and logical errors.

On the other hand, using this option is recommended when assembling programs that were generated by some intelligent software. An example is a disassembled program text where numbers (addresses of variables) are used instead of symbolic names of variables. This option allows disabling most of the warnings which are irrelevant in this case. See also description of the MCC-430 **-g** option.

-r: Disable detection of jumps made to the data memory

This option disables warnings about passing control to data memory. This warning is produced when the operand in a jump (or in CALL or BR instruction with immediate addressing) has allocation **data**, except when this operand is a label in a segment declared in the current module using the extended segment declaration format.

-l: Generate listing file

Generate a listing file with the same name as the source file and the extension .LST. By default, the listing file is not generated. Example:

```
MCA430.EXE –l MYPROG.MCA   ;MYPROG.LST will be generated
```

See also: the .LSTOUT directive under *Listing Control* in *Assembler Directives*, Chapter 2.

-J*path*: Place object file in the specified directory

By default, the Assembler places the output object file(s) in the current directory. This option instructs the Assembler to use a different directory.

Example:

```
-Jobj
```

Assembler will place object file(s) in the OBJ subdirectory in the current directory. This subdirectory must exist.

-L*path*: Place listing file in the specified directory

By default, the Assembler places listing files in the current directory. This option instructs the Assembler to use a different directory.

Example:

```
-Lc:\tmp\listings
```

-x: Include cross-reference table in the listing

See the .LSTXRF directive under *Listing Control* in *Assembler Directives*, Chapter 2.

-c: Include false conditionals in the listing

See the .LSTCND directive under *Listing Control* in *Assembler Directives*, Chapter 2.

-g: Include macro definitions in the listing

See the .LSTMAC directive under *Listing Control* in *Assembler Directives*, Chapter 2.

-e: Include macro expansions in the listing

Generate listing for the text of macro expansions. False conditionals located in macro expansions are included in the listing only if listing generation is enabled for both macro expansions and false conditionals. See also the .LSTEXP directive under *Listing Control* in *Assembler Directives*, Chapter 2.

-w: “Wide” listing output

See the .LSTWID directive under *Listing Control* in *Assembler Directives*, Chapter 2.

-p: Split listing into 40 line pages

Split the listing file into pages of 40 lines in length. By default, the listing file is not split into pages.

-P*nn:* Split listing into pages of the specified length

Split the listing file into pages of lengths set by *nn*. By default, the listing file is not split into pages.

-E*nn:* Terminate assembling after *nn* errors

Terminate assembling after *nn* errors have been detected. By default, all error messages are displayed until the end of source file is reached.

-W*nn:* Display not more than *nn* warnings

Stop displaying warnings after *nn* warnings have been given. If *n* is set to 0, the warnings are disabled entirely. By default, warnings are enabled.

-b: Produce beep if error is detected

Produce a beep when an error is detected.

-s: Display the number of processed lines

Display the number of lines assembled. Only line numbers multiple of 8 are displayed. By default, the line numbers are not shown. Note that on many computers the speed of assembling reduces significantly when this option is enabled.

-h or -?: Display the list of options on the console

Display a brief description of options on the console. This information is also displayed if the Assembler is run without parameters.

Appendix D. MCA-430 Command Line Interface

@filename

Include the contents of the file filename (response file) when processing the command line. There are no size limits for the response file. This makes it possible to specify any number of command line options in it.

There should be no spaces between @ and filename. The <CR> (carriage return) and <LF> (line feed) characters are ignored in the response file. Comments start with the "#" symbol and end with the end of line. If not specified, the extension is expected to be .OPL.

-A: Define an address space

Usage:

```
-A (allocation)addr_space_name(range_list)
```

Addr_space_name is the name of the address space being defined. This name is defined by the user. If you specify the name of a standard address space then the size of this address space will be redefined.

Allocation is the allocation attribute of the address space being defined. *Range_list* is the list of address ranges allowed for using in this address space; the Linker will allocate segments only within these ranges. The address ranges must be within the limits set for the address spaces with the specified allocation. For details on address spaces limits, see *Address Space Allocation Attributes* in *Chapter 1. Basic Conceptions.*

Range list syntax:

```
start_addr-end_addr[,start_addr-end_addr] …
```

Start_addr and *end_addr* can be specified in decimal, binary, octal, or hexadecimal format; binary values should begin with 0 and end with "B" or "b"; octal values should begin with 0 and end with "Q" or "O" (or "q" and "o", respectively); hexadecimal values should begin with 0 and end with "H" or "h".

Example:

```
-A (code)CODE(0F000H-0FFFFH)
```

In the above example, the Linker will redefine the size of the standard address space CODE. This address space will have the address range 0F000H-0FFFFH (in bytes).

-C: Enable code generating in address space

Usage:

```
-C address_space_name
```

The Linker enables allocating code in the specified address space, in other words, enables code generating in this space. The user should specify this option for the address spaces where executable code, constant tables, etc. will be allocated. Code generating may not be enabled for the register address spaces.

If code generating is not enabled for a particular address space then only the segments that do not contain code can be allocated in this address space, otherwise the Linker will generate an error message.

-K: Reserve address ranges in address space with the specified allocation

Usage:

```
-K allocation(range_list)
```

Allocation is the allocation attribute of the address space. *Range_list* is the list of address ranges. See the **-A** option for *range_list* syntax.

This option instructs the Linker to exclude the list of address ranges specified in the *range_list* from the list of ranges that can accommodate relocatable segments. The **-K** option is mainly used to preserve space for the absolute segments and avoid overlapping with the relocatable segments without the need to set absolute addresses with the **-S** option.

Note, when an absolute segment is used in the source text the Assembler automatically passes this option and corresponding address ranges to the Linker, therefore the user does not need to be careful about relocatable segments overlapping with absolute segments.

-N: Reserve address ranges in the address space

Usage:

```
-N address_space_name(range_list)
```

Address_space_name is the name of the address space. *Range_list* is the list of address ranges. See the **-A** option for *range_list* syntax.

The Linker excludes the address ranges specified in the *range_list* from address ranges that can accommodate the relocatable segments.

-S: Segment allocation

Usage:

```
-S address_space_name(segment_list)
```

Address_space_name is the name of the address space. *Segment_list* is the list of segments to be allocated in this address space; for relocatable segments you may specify a boundary, and the Linker will put a segment as close to this boundary as possible.

Segment list syntax:

```
segname [{>,<}address][%pagesize] [,segname[{>,<}address][%pagesize]]…
```

The ">" character between *segname* and *address* instructs the Linker to allocate this segment starting from the *address*. If it is not possible then the Linker will allocate the segment starting from the nearest suitable address greater than the one specified.

The "<" character between *segname* and *address* instructs the Linker to allocate this segment ending at the *address*. If it is not possible then the Linker will allocate the segment to end at the nearest suitable address less than the one specified.

Note, start or end addresses can be specified for relocatable segments only.

The *address* can be specified in decimal, binary, octal, or hexadecimal format; binary values should begin with "0" (zero) and end with "B" or "b"; octal values should begin with "0" (zero) and end with "Q" or "O" (or "q" and "o", respectively); hexadecimal values should begin with "0" (zero) and end with "H" or "h".

Pagesize must be a numeric value multiple of the power of 2. If the *pagesize* is specified, the Linker allocates the segment so that the whole segment fits on the page with the size specified. Ultimately, this means that the segment must not cross the boundary of the page with the size *pagesize*, i.e. the segment should be within the *pagesize***n*-*pagesize**(*n*+1) range, where n=0.

Examples:

```
-S CODE(CODESEG1>0F000H,CODESEG2<0FF00H)
-S CODE(CODESEG3%512,CODESEG4%2048)
```

You can specify segment start/end address and page size at the same time:

```
-S CODE(CODESEG>0F0E0H%1024,CODESEG1<0FFA0H%2048)
```

-E: Specify output file name and target directory

Usage:

```
-E [path]filename
```

This option is used to specify the common name and the path for the output file(s). The filename extension is automatically selected by the Linker according to the type of file being created. If you specify the extension it will be ignored. If the *path* is not specified, output files will be created in the current directory.

If the **-E** option is not specified, the output file name will be the same as the name of the first file in the list of files to be linked. The same applies to path.

Example:

```
-E C:\Myproj\Project1
```

Output files will be created in the "C:\Myproj" directory the base filename will be "Project1".

-O: Specify the search paths for object files

Usage:

```
-Opath
```

This option specifies the directories where the Linker will search for the object and library files if they are not found in the current directory and their names do not include paths. The directory names should be separated by semicolons, for example:

```
-OC:\MCA430\lib;C:\Myproj\my_obj
```

First, the Linker will search for the files in the directory "C:\MCA430\lib" and then in the directory "C:\Myproj\my_obj".

-F: Specify the output file format

Usage:

```
-F format [format] …
```

This option specifies the output file format. If you specify more than one output formats then more than one output files in appropriate formats will be generated. Below is the list of allowable formats:

`M` Phyton/MicroCOSM-ST executable file format
`I` Intel HEX-format
`Z` Debugging information in ZAX-format

Executable files in Phyton/MicroCOSM-ST format have the extension .MCE. HEX-files, by default, have the extension .HEX; use the **-H** option to explicitly specify the extension for HEX-files.

Example:

```
-F MI    #The Linker will create files in the MCE-format (for debugging)
         #and HEX-format (HEX-files can be used with any PROM programmer)
```

-H: Define the filename extension for HEX-file

Usage:

```
-H address_space_name(HEX-extension[,ZAX-extension])
```

This option allows you specify directly what extensions (instead of the default extensions) will be used for the output HEX-files and ZAX-files containing debugging information.

Address_space_name is the address space name. *HEX-extension* sets the extension for the HEX-file corresponding to the specified address space. *ZAX-extension* is an optional parameter that can be used to specify the extension of the ZAX-file corresponding to the specified address space.

-Z: Increase segment size

Usage:

```
-Z segment_name(new_size)
```

This option makes it possible specifying the segment sizes at link time rather than in the source text, which can be helpful in certain situations.

Segment_name is the segment name. The segment must be relocatable and contain no code. *New_size* is the new segment size (in bytes). It should be greater than the original size of the segment (calculated as a sum of sizes of all fragments of this segment from every module where this segment is used). Segment size is specified in the same way as in the **-A** option.

Example:

```
-Z STACK(0100h)      # sets stack size to 256 bytes
```

-m: Create a MAP-file

This option instructs the Linker to generate a MAP-file (Linker's report). A MAP-file has the same name as the output file but with the .MAP extension.

-M: Omit specified section in MAP-file

Usage:

```
-M nn
```

This option can be used to skip putting unnecessary information in the MAP-file to make it better suitable for further analysis of the Linker's work. This option can be specified multiple times with different parameters. The list of available parameters is the following:

`0` original list of used Linker options
`1` process of linking program modules
`10` public/external names in program modules
`2` process of adding library modules
`20` public/external names in the linked library modules
`21` list of unlinked library modules
`3` complete set of Linker options
`4` report on segment building
`5` list of address spaces
`6` report on the results of relocatable segments allocation
`60` process of relocatable segments allocation
`7` allocation map of address spaces (memory map)
`8` table of public and external names
`80` table of reserved/internal public and external names
`9` report on output files

Recommended typical usage for most situations:

-M 10 # skip public/external names in program modules
-M 20 # skip public/external names in library modules
-M 21 # skip a list of unlinked library modules
-M 60 # skip the process of relocatable segments allocation
-M 80 # skip the list of reserved and internal names

-t: Disable type checking

Disables type checking while resolving external references.

-w: Linker warnings control

This option prevents the Linker from displaying the messages on the console about the automatic segment allocation in the standard address spaces. Generally, automatic segment allocation is satisfactorily, therefore usage of the **-w** option is advisable.

When two or more address spaces with the same allocation attribute are declared, the Linker displays such warnings for all segments with that allocation, which are not specified in the Linker **-S** options.

-h or -?: Display the list of options on the console

These options instruct the Linker to display brief description of options on the console. The same occurs when the Linker is launched without parameters.

-p, -l, -o: Prefixes changing module type

Usage:

```
{-p,-l,-o} object_or_library_file
```

Any of the **-p**, **-l**, or **-o** prefixes can be inserted before any of the files linked. These prefixes serve as module type modifiers.

`-p` consider all modules in the file to be program modules
`-l` consider all modules in the file to be library modules
`-o` consider all modules in the file to be low priority modules

Appendix E. MCLIB Command Line Interface

@filename

Include the contents of the file filename (response file) when processing the command line. There are no size limits for the response file. This makes it possible to specify any number of command line options in it.

There should be no spaces between @ and filename. The <CR> (carriage return) and <LF> (line feed) characters are ignored in the response file. Comments start with the "#" symbol and end with the end of line. If not specified, the extension is expected to be .OPL.

-a: Add modules to library

Usage:

```
-a library_file obj_file1 [obj_file2] …
```

Adds all modules from the specified object file(s) to the library with *library_file* name. If the library *library_name* does not exist, it is created. If there is a module in existing library with the same name as the name of one of the modules being added then the "Duplicate module" error occurs.

-d: Delete modules from library

Usage:

```
-d library_file module1 [module2] …
```

Deletes the specified module(s) from the library with *library_file* name. If the modules are not found, then the "Unknown module" error occurs. The *library_file* must exist.

-r: Replace modules in library

Usage:

```
-r library_file obj_file1 [obj_file2] …
```

Updates the library replacing the modules in the *library_file* library with the modules that have the same names from the specified object file(s). If the library *library_name* does not exist, it is created. If any object file contains a module that is not in the library being updated, it is added to the library.

-x: Extract modules from library to object files

Usage:

```
-x library_file module1 [module2] …
```

Extracts modules from the library into object files. Modules are not deleted from the library. Each extracted module is placed into a separate object file with extension .MCO. The name of the object file is the same as the module name if it is 8 characters or less. Otherwise, the module name is truncated to 8 characters and set as the object filename.

If the specified modules are not found, the "Unknown module" error occurs. It is assumed that the specified *library_file* exists.

-X: Extract modules from library into a single object file

Usage:

```
-X object_file library_file module1 [module2] …
```

Extracts modules form the library and places them into one object file with the name *object_file* and the extension .MCO (or other extension, if specified). Modules are not deleted from the library. If the specified modules are not found in the library, the "Unknown module" error occurs. The specified *library_file* must exist.

-m: Move modules from library to object files

Usage:

```
-m library_file module1 [module2] …
```

Extracts modules from the library to object files and deletes the modules from the library *library_file*. Each specified module is placed in a separate object file with the .MCO extension and the file name coincident with the module name. If the module name is longer than 8 characters then it is truncated to 8 characters and set as the object file name. If the specified modules are not found in the library, the "Unknown module" error occurs. The specified *library_file* must exist.

-M: Move modules from library to single object file

Usage:

```
-M object_file library_file module1 [module2] …
```

Extracts modules from the library into one object file with the name *object_file* and deletes the modules from the library *library_file*. The object file is given the .MCO extension if not specified otherwise. If the specified modules are not found in the library, the "Unknown module" error occurs. The specified *library_file* must exist.

-l: Display library header on the console

Usage:

```
-l library_file
```

Displays the header (catalogue) of the specified library on the console. If the header is too long and cannot fit on one screen, you can redirect the standard output to a file using the ">" character or the MS-DOS MORE utility. Example:

```
mclib -l mylib >mylib.txt
```

The Librarian directs the header listing of the MYLIB.MCL library to the standard output; DOS captures this data and puts it in the MYLIB.TXT file.

-P: Assign ‘program’ attribute to modules

Usage:

```
-P library_file module1 [module2] …
```

Assigns the 'program' attribute to the specified modules in library *library_file*. If the specified modules are not found in the library, the "Unknown module" error occurs. The specified *library_file* must exist.

-L: Assign ‘library’ attribute to modules

Usage:

```
-L library_file module1 [module2] …
```

Assigns the 'library' attribute to the specified modules in library *library_file*. If the specified modules are not found in the library, the "Unknown module" error occurs. The specified *library_file* must exist.

-O: Assign ‘low-priority library’ attribute to modules

Usage:

```
-O library_file module1 [module2] …
```

Assigns the 'low-priority library' attribute to the specified modules in library *library_file*. If the specified modules are not found in the library, the "Unknown module" error occurs. The specified *library_file* must exist.

-h or -?: Display the list of options on the console

Displays a brief description of options on the console. The same occurs if the MCLIB is started without parameters.

Appendix F. MCDUMP Command Line Interface

-e: List external names and module names

Displays the records for external names used in the modules of an object/library file. The **-e** option is not valid for using with the MCE-files.

-m: List module names

Displays only the records describing modules in an object or library file. The **-m** option is not valid for the MCE-files.

-p: List public names and module names

Displays the records for public and module names defined in the modules of an object/library file. The **-p** option is not valid for the MCE-files.

-s: List segment names and module names

Displays the records for segments declared in the modules of an object/library file. The **-s** option is not valid for the MCE-files.

-H: List contents of library header

Display a library header. The **-H** option is ignored for files in the MCO and MCE formats.

-r: Do not replace numbers with symbols

Preceding records are referenced without symbolic information (i.e. only indexes are used).

-h or -?: Display the list of options on the console

Displays a brief description of options on the console. The same occurs if the MCDUMP is started without any parameters.

Appendix G. Diagnostics

MCA-430 Diagnostic Messages

Warning A2: <*user-defined warning message*>
This is a warning message defined by the user. It is displayed with the .WARNING directive.

Warning A53: Operand type mismatch
The 'operand type' attribute of the operand is not the same as the op-type expected for the instruction. This warning can be caused by an error in the program; for example, a missing "#" while the immediate addressing mode was intended. This warning can be suppressed by using the **-a** option. A better approach is to use the operand type reassignment operators and explicitly specify the op-type of expression, since if the operand type checking is disabled completely, there is a risk of overlooking some errors. See *Setting Operand Type of Expression*, Chapter 2. Example:

```
.RSEG MyData,data
  MyByte .dsb              ;MyByte obtains BYTE operand type
  MyArr  .ds 5
.RSEG _MyCode,code
mov.b MyByte,MyArr         ;op-type mismatch for MyArr
.BYTE MyArr                ;op-type of MyArr becomes BYTE
mov.b MyByte, MyArr        ;ok
```

Warning A72: Too many warnings
Number of warnings is more than specified by the user in the **-W** option. No warnings will be displayed further.

Warning A84: Operand allocation required
An instruction operand should be an address of the memory location, but the operand has no allocation attribute (i.e. it is just a number). Typically, this warning indicates a logical error in the program. It can also occur if the program is written in obsolete programming style or if the source text has been obtained as a result of disassembling. This warning can be disabled with the **-a** option. Example:

```
.RSEG MyCode,code
JMP 0FC00h                 ;warning
.ASEG ACODE,code           ;workaround for this problem
.org 0FC00h
L0FC00H:
.RSEG MyCode
JMP L0FC00H                ;ok
.END
```

Error A1: <*user-defined error message*>
This is an error message defined by user. It is displayed by the .ERROR directive.

Error A3: '(' required
An opening bracket is possibly missing in one of the Assembler directives.

Error A4: ')' required
A closing bracket is possibly missing in one of the Assembler directives.

Error A5: '+', '-' or '.' required
The listing control command was not followed by the "+", "–", or "." symbol.

Error A6: ',' required
A comma is required here but is missing.

Error A8: .ENDF address is less than address of function
As a result of some "tricky" sequence of instructions (probably, using the .ORG directives), the Assembler program counter (PC) value at the .ENDF directive is less then PC value at the corresponding .FUNC directive. Example:

```
.ASEG ACODE,code
.ORG 0FC10h
.FUNC Foo
    nop
.ORG 0FC00h
    ret
.ENDF
.END
```

Error A9: ASCII constant too long
A character string in an expression is too long. Only a string of up to 4 characters can be used as an operand in arithmetic expressions (i.e. as a number).

Error A10: Absolute expression required
Absolute expression was expected here, but it is either not specified or specified incorrectly.

Error A11: Allocation required
Allocation attribute was expected here, but it is either no specified or specified incorrectly.

Error A12: Bad allocation
Allocation attribute was expected here, but it was specified incorrectly. Example:

```
.extrnb(xdata) Var          ;error: no such allocation 'xdata'
```

Error A13: Bad number of arguments
Wrong number of arguments is specified in assembler instruction or directive. Such an error is often caused by another error ("induced" error).

Error A14: Division by zero
When evaluating an arithmetic expression, the Assembler has encountered the "division by zero" operation.

Error A15: Duplicate label <*lname*>
Name *lname* was declared more than once in the current scope. The same name may not be declared in a module more than once. The only exception is local labels in the functions which should be unique between a .FUNC/ .ENDF directive pair, and not within a module. Example:

```
.RSEG MyData,data
X  .dsb
.RSEG MyCode,code
X:                          ; error here
    dec R7
    jnz X
```

Error A16: Duplicate macro <*mname*>
The Assembler has encountered more than one macro definition with *mname*. There can not be two macros with the same name in a module.

Error A17: Expression <> current relocation
.ORG directive contains the expression, 'relocatability' attribute of which does not match the current segment 'relocatability'. Expression should be absolute in the absolute segment and relocatable in the relocatable or overlay segment. Example:

```
.extrnn Start
Init .equ 5
.ASEG AbsSeg, data
.ORG Init                    ;ok, program counter=5
.ORG $+5                     ;ok, program counter=10
;...
.ORG Start                   ;error, external expression
.RSEG RelSeg, data
;...
.ORG $+10                    ;ok
;...
.ORG 5                       ;error, absolute expression
```

Error A18: Expression not absolute
Only absolute expression can be used here. Example:

```
.EXTRNN SerNo
.EXTRNB(data) Size
.RSEG MyCode, code
.if SerNo='1234'              ;error: external numeric
                              ;constant SerNo is not absolute
    mov.b #128, Size
.else
    mov.b #255, Size
.ENDIF
.END
```

Error A19: Expression required
An expression is expected here but is missing.

Error A20: Extra characters in line
Extra characters are found at the end of the line. Such an error is often caused by a missing comma or another syntax error ("induced" error).

Error A21: Identifier required
Identifier is missing in an assembler directive. Commonly, this error occurs in one of the following directives: .PUBLIC, .EXTRNx, .BYTE, .WORD, .DWORD, .TYPE. Such an error can be also caused by an extra comma (right after a closing bracket) in the .EXTRNx directive.

Error A22: Illegal instruction
An illegal instruction or assembler directive is specified. This error is often caused by a missing colon after a label name or by a missing dot before an assembler directive. This error can also be caused by another error ("induced" error).

Error A23: Illegal use of local variable <*_name*>
Illegal use of local (in function) identifier *_name*.

Error A24: Illegal use of macro name <*mname*>
Illegal use of the *mname* macro. Macro name can be used for macro call (expansion) only. For example, a macro can not be an operand in instruction or Assembler directive.

Error A25: Inappropriate use of variable symbol
Incorrect use of a variable identifier. Identifier defined with the .SET directive or the equation symbol "=" may be redefined by subsequent .SET/= statements anywhere in the source text. This imposes restrictions on the use of such identifier. For example, it may not be declared as public with the .PUBLIC directive. Use the .EQU directive to define identifiers that can be declared as public. Such identifiers may not be redefined. Example:

```
     Var1 = 10
.PUBLIC Var1                ;error
     Var2 .EQU 10
.PUBLIC Var2                ;correct
```

Error A26: Instruction operand required
One or more operands are expected in the instruction but are missing.

Error A28: Invalid operand combination
Invalid operand combination is specified in instruction.

Error A29: Invalid syntax
Invalid syntax. Example:

```
.rseg RCODE,code
zzz .MACRO abc
4&abc:
.ENDMAC
zzz def
.END
```

Error A30: Directive cannot be used with this allocation
The directive can not be used with the specified allocation attribute.

Error A31: Label <*ident*> already typified
The identifier *ident* is used as operand in a directive that assigns the operand type and the operand type of *ident* is not UNTYPED. If you need to use a typed name (or an expression) as operand in instruction then use the .BYTE, .WORD, .DWORD operators to specify the necessary operand type, rather than using the operand type assignment directives. Example:

```
Arr1 .ds 10                 ;untyped array
.byte Arr1                  ;ok: Arr1 becomes array of 10 bytes

Arr2 .dsb 10                ;array of 5 BYTE's
.word Arr2                  ;error: Arr2 already has op-type WORD

Arr3 .labelw
.ds 10                      ;array of 5 words
Arr3 .dsd                   ;error: Arr3 already has op-type WORD
Var1 .db2                   ;Var1 is 2 byte UNTYPED name
.dword Var1                 ;Var1 obtains op-type DWORD
```

Error A32: Label required for this directive
Current Assembler directive must be labeled, i.e. such directive always defines an identifier. Example:

```
.DEFINE 10                  ;error: identifier requied
.RSEG MyData, data
.DSB 10                     ;error: identifier requied
.DS  10                     ;ok
.RSEG MyCode, code
```

```
.DCB 10                    ;ok
.END
```

Error A34: Misplaced .ELSE
The .ELSE directive can be placed inside a conditional block only, i.e. between the .IF and .ENDIF directives.

Error A35: Misplaced .ENDIF
The .ENDIF directive can be placed at the end of a conditional block only, i.e. after .IF or .ELSE.

Error A36: Misplaced .ENDMAC
Inappropriate use of the .ENDMAC directive. The directive .ENDMAC can be placed only at the end of a macro definition or repeat block.

Error A37: Misplaced .EXITM
Inappropriate use of the .EXITM directive. Directive .EXITM can be placed inside a macro definition or repeat block only.

Error A38: Misplaced operator .PARM
Inappropriate use of the .PARM operator. The .PARM operator can be used inside macro definitions only.

Error A39: Module name required
A module name is not specified (or specified incorrectly) in the .PMODULE, .LMODULE, or the .LMODULE2 directive.

Error A40: Nested macro definitions
The Assembler has encountered a nested macro definition. In this version of the Assembler, macro definitions cannot be nested in other macro definitions or repeating blocks.

Error A41: No .END directive
There is no .END directive at the end of the source file. Source file is probably damaged.

Error A43: No .ENDIF directive
A conditional block .IF does not have the .ENDIF directive at the end.

Error A44: No .ENDMAC directive
A macro definition or repeat block does not have the .ENDMAC directive at the end.

Error A45: No .FUNC directive
The Assembler has encountered an .ENDF directive without a preceding .FUNC directive.

Error A46: Relocatability conflict
This occurs when operands in arithmetic expression have incompatible 'relocatability' attributes.

Error A47: No module prologue
Extra .END or .ENDMOD directive was encountered.

Error A48: No segment declaration
The error usually occurs in the beginning of a source file. The Assembler has encountered code, label or data definition outside of any segment. Code and data must be only placed in the declared segments.

Insert the .RSEG, .OSEG, or .ASEG directive before the line that has caused the error, in order to declare corresponding segment.

Error A51: Operand required
Operand is required in the arithmetic expression but is missing.

Error A52: Operand syntax
Incorrect operand syntax in an arithmetic expression.

Error A55: Operator not allowed here
Illegal use of the part extraction operator .HWRD, .LWRD, .HIGH, .LOW, .BYTE3, or .BYTE4.

Error A56: Out of range
Some value is outside of the allowed range. Whether the value of relocatable expression falls inside the allowed range is checked at link time only. If it does not, the Linker generates the same "Out of range" error message.

Error A57: Segment mismatch for function <*fname*>
The .ENDF directive is not in the same segment as the previous directive .FUNC *fname* which declares the *fname* function.

Error A58: Segment name required
Illegal segment name or no name is specified in the .RSEG, .OSEG, or .ASEG directive. For example, names of allocation attributes and other reserved words may not be used as segment names.

Error A59: Something after .END directive
There is some text following the .END directive. The .END directive indicates the end of file and the end of the last module in the file, so no text is allowed beyond it.

Error A61: Module should be closed with .ENDMOD
Before opening the next module with the .PMODULE, .LMODULE, or .LMODULE2 directive, the previous module should be closed with the .ENDMOD directive.

Error A62: Segment <*sname*> must be declared before using
You can not use a segment that has not been previously declared in the module. Specify allocation attribute for the segment *sname* in the corresponding .RSEG, .OSEG, or .ASEG directive.

Error A63: String required
A string is expected here but is missing. This error usually occurs in such directives as .WARNING, .ERROR, .MESSAGE, .LNKCMD. Note that character strings in MCA-430 should be enclosed in single quotes (').

Error A64: String too long
Character string length exceeds 255 characters.

Error A65: Superfluous label <*lname*>
An extra label. Directives .BYTE, .PUBLIC, .EXTRN, .TYPE, .FUNC can not be labeled.

Error A68: Operand cannot be the result of part extraction operator
Operand cannot be a result of part extraction operator.

Error A69: Too many macro arguments
Too many macro arguments are specified in the .MACRO or .IRP directive. Not more than 33 arguments can be used in a macro definition (declared with .MACRO) and not more than 32 arguments can be used in a repeat block with parameter scanning (.IRP).

Error A73: Type conflict
An attempt to change the 'type' atribute of an identifier. If the type attribute (different from 0 or nothing) has already been assigned to an identifier earlier in the module, the attribute may not be changed with the .TYPE directive. Example:

```
aa .EQU 16
.TYPE aa(.NOCHECK)
. TYPE aa(.UCHAR)      ;error
.RSEG MyData,data
dd .DSI                ;INT type is assigned to dd
…
. TYPE dd(.UINT)       ;error, type attribute has been already assigned
.END
```

Error A74: Unclosed function <*fname*>
Function *fname* is not ended with .ENDF directive.

Error A75: Expression not relocatable
Only relocatable expression can be the operand of the .OFFSET operator.

Error A78: Unknown label <*id*>
The Assembler has encountered an unknown name *id* (either label or identifier).

Error A79: Instruction is illegal for this CPU type
Machine instruction is not supported by selected processor type or is inaccessible in selected memory model.

Error A81: Segment <*seg_name*> type conflict: <*type1*>/<*type2*>
Different directives (.RSEG, .OSEG, or .ASEG) are used for declaration of the *seg_name* segment and for switching to that segment. For example, if the segment *seg_name* was declared with the .RSEG directive, you can not switch to it with .ASEG. Example:

```
.RSEG RCODE,code ; RCODE segment declaration
.RSEG RCODE      ; switch to RCODE segment
…
.ASEG RCODE      ; switch to RCODE - ERROR:
                 ;"Segment RCODE type conflict: relocatable/absolute"
…
```

Error A82: Incompatible segment allocation
The error usually occurs when instruction mnemonics are used in segments with **data** allocaion. Instructions can be used in **code** segments only.

Error A83: Illegal forward reference
Illegal forward reference. For example, the .ALLOCATION and .OPTYPE operators do not allow using forward references in their operand specification.

Error A86: Too many nested macro calls
Too many nested macro calls. Generally, such an error occurs in case of macro calls cycling.

Error A87: Incorrect segment allocation specified
When switching to a previously declared segment, an incorrect allocation attribute was specified. If the segment is already declared, you do not need to specify the allocation attribute. However, if the allocation is still specified, it should be the same as in the segment declaration. Delete the allocation attribute and preceding comma from the segment switching directive to avoid this error. Example:

```
.RSEG RCODE,code      ;declare RCODE segment - OK
…
.RSEG RCODE           ;switch to RCODE segment- OK
…
.RSEG RCODE,code      ;switch to RCODE segment- OK
…
.RSEG RCODE,data      ;ERROR
…
.END
```

Error A88: Segment must have allocation <*alloc*>
Incorrect allocation is specified for a segment declared using the extended format.**See also**: *Segment Declaration and Selection* in *Assembler Directives*, Chapter 2.

Example:

```
.RSEG ExtSeg,code(CodeSeg)  ;error: ExtSeg should have allocation 'data'
.RSEG ExtSeg,data(CodeSeg)  ;correct

.RSEG MySeg, data           ;declaraing a 'data' segment
.RSEG ExtSeg, data (MySeg)  ;error: MySeg must be a 'code' segment
```

Error A89: Segment must be relocatable/overlay
The Assembler has detected an attempt to declare an absolute segment in extended format. Only relocatable and overlay data segment can be declared using the extended format of segment declaration. **See also**: *Segment Declaration and Selection* in *Assembler Directives*, Chapter 2.

Error A92: Misplaced .ENDSEG
The Assembler has encountered the .ENDSEG directive without a matching .RSEG, .OSEG, and .ASEG directive. The .RSEG, .OSEG, and .ASEG directives not only switch the Assembler to the specified segment. They save in memory, structured as stack, the name of the previous segment which has been used before switching. The .ENDSEG directive, in turn, pops the last stored segment name from the stack and switches the Assembler to that segment. This error occurs when the Assembler encounters .ENDSEG directive and the segment stack is empty. For example, if the Assembler has not yet encountered any .RSEG, .OSEG, or .ASEG directives. Example:

```
.ENDSEG             ;error
.RSEG MyCode,code
.RSEG MyData,data
.ENDSEG             ;switches to MyCode segment
.ENDSEG             ;switches to the 'no segment' state
.ENDSEG             ;error
.END
```

Fatal A0: No such file or directory (<*filename*>)
The Assembler can not find the include file *filename*. The assembling is terminated.

Fatal A49: No source file name
The source file name was not specified to the Assembler. Assembling is immediately terminated.

Fatal A60: Source line too long
Source line is too long and the Assembler can not process the source file. Maximum line length, 1000 characters, should be more than sufficient. Possibly, there is something wrong with the file.

Fatal A66: Too many errors
Assembler has detected more errors than the maximum number of errors specified by the user (in the **-E** option).

Fatal A67: Too many externals
Too many external names were declared. Not more than 1023 external names can be declared in the module.

Fatal A70: Too many segments
Too many segments were declared. Not more than 1023 segments can be declared in the module.

Fatal A71: Too many source files
Too many files need to be included in the source file (with .INCLUDE directive). Not more than 1023 include files can be used in the source file.

Fatal A77: Unknown command line option <*bad_option*>
Unknown option *bad_option* is specified to the Assembler. Assembling will be immediately terminated.

Fatal A80: Command line too long
Specified command line is too long. Assembling will be immediately terminated. Use the response files to avoid this problem.

Fatal A90: Invalid argument in option <*opt*>
Wrong argument is specified in the *opt* option. The assembling is terminated. This error can occur with the **-E**, **-W**, and **-P** options.

Fatal A91: Include file cannot contain more than one module
If the include file contains .PMODULE, .LMODULE, or .LMODULE2 directive then it must consist of a single module, i.e. the file must begin with a module declaring directive and end with the module ending directive .ENDMOD. The assembling is terminated.

MCLINK Diagnostic Messages

Warning L2: Address area <*addr_space*> file extension already defined
File extension has been already defined with the Linker -H option for the address space *addr_space*. Only one -H option may be specified for an address space.

Warning L15: Duplicate label <*name*> in module <*mod*>
Public *name* is defined in more than one module. Public names must be unique in the program.

Warning L30: Segment <*seg*> auto-placed in the <*addr_space*>
The address space in which the segment *seg* is to be placed was not specified to the Linker. The Linker has allocated it in the default address space *addr_space*.

Warning L31: Operand type conflict <*var*>: <*optype1*>!=<*optype2*>
The Linker checks the operand type attributes of public and external names. For example, if in one module the variable *var* is declared as public and defined with the .DSB directive (with BYTE operand type), but another module refers to *var* by the .EXTRNW directive, in the latter module the variable's operand type will be WORD and the Linker will generate the warning message.

Warning L32: Overlapping range 0xxxxh-0yyyyh in segment <*seg*>
The specified range of the relocatable segment *seg* is occupied by more than one fragment of code or initialized with data more than one time.

Warning L45: Type conflict <*var*>: <*type1*>!=<*type2*>
The Linker checks the type attributes of public and external names. For example, if in one module the variable *var* is of the .CHAR type and declared with the .PUBLIC directive, and another module refers to *var* by the .EXTRNB directive, in the latter module the variable is of the .UCHAR type and the Linker generates the warning message.

Warning L52: Unknown segment <*seg*>
The segment *seg* specified in one of the **-S** Linker option cannot be found in modules for linking. The segment is ignored.

Warning L58: Different address areas specified in options -S for segment <*seg*>, address area <*addr_space*> is used
The *seg* segment is specified in two or more **-S** Linker option with different address spaces. The segment is placed in the address space *addr_space* (specified in the last of such options).

Error L5: Allocation conflict for segment <*seg*> and address area <*addr_space*>
The segment *seg* is listed in the **-S** Linker option option for the address space *addr_space*, but allocations of *addr_space* and *seg* are different. Check **-S** and **-A** options for *addr_space*. Assume the following source text is assembled.

Error L11: Bad version of MCL format in <*filename*>
The version of the object library file format is not compatible with the current Linker version.

Error L12: Bad version of MCO format in <*filename*>
The version of the relocatable object file format is not compatible with the current Linker version.

Error L46: Undefined extern name <*name*> in module <*mod*>
The Linker has determined that the *name* is declared as external and used in the module *mod*, but is not declared as public in any other module available for linking. If an external name is used in a module, it should be defined and declared as public (using the .PUBLIC directive) in another module. This error can be possibly caused by assembling files containing program/library modules, with different case sensitivity setting.

Note, if you need to include an object or a library module assembled in the case insensitive mode (containing all names in upper case) in a mixed Assembler/C project, it is convenient to use the **`#define`** directive to redefine such names in C source text. For example, assume you have defined the function `MyFunc` in an assembler module. If the module was assembled in the case insensitive mode, put the following line in the C header file and include this file in all C modules where `MyFunc` is called:

```
#define MyFunc MYFUNC
```

Fatal L1: Address area <*addr_space*> exceeds predefined limits
Address range specified for the address space *addr_space* is beyond the allowed limits: see *Address Space Allocation Attributes* in *Chapter 1. Basic Conceptions* for information on the address space upper and lower boundaries.

Fatal L3: Alignment error
The address assigned to segment or variable is not compatible with the required alignment. Such an error may occur, for instance, when an external variable located at an odd address is accessed by a word instruction, in the current module. If the Linker generates this error, assemble/compile the corresponding source file with the **-d** command line option (to include debugging information in object file). The Linker can then refer to line number in the source file where the out-of-range error occurs. See also the .ALIGN directive under *Segment Alignment* in *Assembler Directives*, Chapter 2.

Fatal L4: Allocation conflict for <*name*>: <*alloc1*>!=<*alloc2*>
The Linker generates this error when different allocations are specified for a name, for example:

- Range of a predefined address space is changed, but wrong allocation is specified in the corresponding **-A** Linker option.
- In different modules, segments with the same name are declared with different allocation.
- External name is declared with allocation which is not the same as allocation of corresponding public name.

Fatal L6: Bad number <*num*>
No or invalid number *num* is specified in one of the Linker options.

Fatal L7: Bad object file <*filename*>
The input file *filename* is not in the correct object format or the file is corrupt.

Fatal L8: Bad option format <*string*>
String specified in one of the Linker options has wrong syntax.

Fatal L9: Bad range <*string*>
The Linker expected an address range in one of the Linker options. The range is not specified or is misspelled.

Fatal L10: Code size exceeds demo version limit
This error can only be encountered with the demo version of the product.

Fatal L20: Illegal type number 0xxh
The type number specified for the name is not valid. Use MCA-430 predefined constants that designate name types.

Fatal L23: Invalid object file <*filename*>
The input object file filename does not have a valid object code file format.

Fatal L24: Linker stack empty
The most possible cause for this error is a failure in the Assembler or compiler.

Fatal L25: Linker stack full
The most possible cause for this error is a failure in the Assembler or compiler.

Fatal L27: No free room in address space <*addr_space*> for segment <*seg*>
The Linker is not able to link the segment *seg* within the valid range for the address space *addr_space*. For example, the available space has been used for placing other relocatable segments, so the relocatable segment can not be placed; or, addresses of the absolute segment are out of range for the corresponding address space. To solve this problem in mixed Assembler/C projects use less static variables, which occupy space in memory during the entire program execution time. Instead, use function parameters to pass variables from one function to another.

Fatal L28: No modules to linking
There are no modules in the list of modules to be processed, after reading all input files. This is probably because there are only library modules in the input files.

Fatal L29: No object file specified
No object files are specified in the Linker command line.

Fatal L36: Segment <*seg*> type conflict: <*rel_1*>!=<*rel_2*>
In different modules the segment *seg* is declared with different relocatability attributes. See also *Segment Declaration and Selection* in *Assembler Directives*, Chapter 2.

Fatal L37: Segment <*seg*> is not relocatable
The segment *seg* specified in the **-Z** Linker option can not be absolute.

Fatal L38: Too many address areas
The Linker has encountered too many address spaces. Maximum number of address spaces is 255.

Fatal L39: Too many extern names
The Linker has encountered too many external names. Maximum number of external names in a program is 4096.

Fatal L40: Too many names
The Linker has encountered too many public names. Maximum number of public names in a program is 4096.

Fatal L42: Too many scopes
The Linker has encountered too many scopes.

Fatal L43: Too many segments
The Linker has encounters too many segments. Maximum number of segments per program is 1024.

Fatal L44: Too many types
The Linker has encountered too many types.

Fatal L48: Unknown address area <*addr_space*>
The address space *addr_space* specified in one of the MCLINK options is not valid.

Fatal L49: Unknown allocation <*alloc*>
The allocation *alloc* specified in one of the MCLINK options is not valid.

Fatal L50: Unknown option <*opt*>
Opt is not a valid MCLINK option.

Fatal L51: Unknown output format
The output format specified in the **-F** Linker option is not valid.

Fatal L53: Banking disabled for allocation <*alloc*>

Fatal L54: Unresolved externals encountered
One or more external names have no corresponding public names in any of the input files. This error is preceded by the error(s) specifying which external name is unresolved.

Fatal L55: Value 0xxxxh out of range
If the Linker generates this error, assemble/compile the corresponding source file with the **-d** command line option (to include debugging information into that object file). The Linker can then refer to line number in the source file where the out-of-range error occurs.

Fatal L56: Too long command line. Please use @filename option
The Linker command line is too long. Use the response file for storing most frequently used options and linked file names (such as libraries). Then, you can specify one or more response files in the command line using the "@" Linker option along with other options and linked file names.

Fatal L57: Invalid pagesize <*num*>
The page size specified in the Linker **-S** command line option is not valid. The page size must be a numeric value multiple of the power of 2.

Fatal L60: Segment <*seg*> too long to be placed on <*num*>-byte page
Size of the *seg* segment is greater than the page size where the segment is to be placed.

MCLIB Diagnostic Messages

Error B3: Bad object file <*object_file*>
The Librarian has detected an attempt to add modules to the library from the object file *object_file* that has invalid format. Object file may be corrupt.

Error B4: Duplicate module <*module_name*>
The Librarian has detected that the name *module_name* of the module being added to the library is already in use by another module existing in the library. Use either the **-d** option to delete the module from the library prior to adding another one, or use the **-r** option which will replace the module with duplicate name.

Error B5: Duplicate public <*name*>

Error B6: Invalid object file <*filename*>
The file *filename* specified to the Librarian as a source of object module(s) is not an object file.

Error B7: No library file specified
Library file is not specified or is accidentally omitted in one of the options.

Error B8: Unknown module <*module_name*>
The Librarian was unable to find the specified module *module_name* in the library file.

Error B9: Unknown option <*option*>
The command line option *option* specified to the Librarian is not supported.

Error B10: Too long command line. Please use \"@filename\" option
The command line is too long. Create a response file and move some parameters into it.

MCDUMP Diagnostic Messages

Fatal D1: Bad version of MCE format in <filename>
The format version of the MCE-file specified is not compatible with the version of MCDUMP used or the file is corrupt.

Fatal D2: Bad version of MCL format in <filename>
The format version of the MCL-file specified is not compatible with the version of MCDUMP used or the file is corrupt.

Fatal D3: Bad version of MCO format in <filename>
The format version of the MCO-file specified is not compatible with the version of MCDUMP used or the file is corrupt.

Fatal D4: Bad object file <filename>
MCDUMP has detected that the object file or library file format is not valid. The specified file is not an object, library, or executable file, or the file is corrupt.

Fatal D5: Incompatible options
The option specified in the MCDUMP command line is not compatible with the format of the specified file. **Note** that some of the options are not compatible with the MCE-files.

Fatal D6: No object file specified
No file was specified as the input file for MCDUMP.

Fatal D7: Too many extern names
MCDUMP has encountered too many external names. The maximum number of external names is 4096.

Fatal D8: Too many names
MCDUMP has encountered too many public names. The maximum number of public names is 4096.

Fatal D9: Too many source files
MCDUMP has encountered too many references to source files. The maximum number of references to source files is 1023.

Fatal D10: Too many types
MCDUMP has encountered too many records describing types.

Fatal D11: Unknown option <option>
The option specified in the MCDUMP command line is not supported.

Fatal D12: Too long command line. Please use \"@filename\" option
The command line is too long. Create a response file and move some parameters into this file.