MA-51

Assembler for the 8051 family

Reference Manual

September 2005

RAISONANCE

Table of Contents

1 Introduction

1.1 Standard 8051 and 8051MX

The 8051MX (hereafter called MX) architecture is largely similar to the 8051 architecture from which it is derived. The main change, in addition to the accelerated execution of instructions, is the extension of the addressing space from 64kb to 8 Mbytes (that is, from 16 bit addresses to 23 bit addresses) for both Code and Data.
The code generated by the standard MA-51 Assembler can be used on MX microcontroller. Look at the “MA-MX Addendum” section (9) to see the specific features of the MA-MX Compiler. Moreover you can look at the MX Addendum documentation (“8051MX tools”) to have more information concerning the MX tools.

1.2 Concerning this manual

The Raisonance Macro Assemblers MA-51 Manual provides you with a syntactic description of MA-51 version 6.0.

1.2.1 « Simplified syntax » and « Classical syntax »

In order to remain compatible with former Raisonance assemblers (EMA-51) and with other similar assemblers, MA version 6.0 supports both the previous versions of Raisonance assembler syntax (herein referred as « Simplified Syntax ») and the Intel ASM-51 assembler syntax (herein referred as « Classical Syntax »). Only one form of syntax may be used in a file. In this manual, the « Classical » syntax is used to unify different examples.

1.2.2 Manual organization

This manual has been written to make as beneficial as possible to both beginners and experienced programmers. It gives, not only concise general information on 8051 microcontrollers, but also details all directives, controls, instructions and macro syntax.

Chapter 2	consists of a description of the components of an assembly program: directives, controls, instructions and symbols, labels, operands, operators.
Chapter 3	presents assembler-programming basics: segment declaration, program location, variable declaration, 8051 instructions and object file generation.
Chapter 4	is dedicated to additional facilities and enhancements to the basic assembler: conditional assembly, hardware specifications, linking directives and macro processor.
Appendix	includes program examples and keyword lists.

1.2.3 Conventions used in this manual

1.2.3.1 Syntax

The syntax of each keyword is presented as in the example below:

Syntax: **[$]RB(**num [**,**num...]**)**

[]	optional arguments in the command line and option fields are indicated by square brackets.
$, **RB**	bold capital text is used for directives, controls and instructions keywords.
num	simple text is used to specify information that has to be provided by the programmer.
...	dots indicate that the preceded items may be repeated zero or more times.

1.2.3.2 Examples

Examples always follow the format below:

```
Example_number:
                        ; comments concerning the example.

$PRAGMA
Labels:
   INSTRUCTIONS
                        ; comments
```

1.2.3.3 Instructions

All instructions are presented in the same way:

MNEMONIC destination_operand , source_operand

Some abbreviations are used to detail operand type, length... etc. The abbreviations are listed below:

@Ri	indirect internal or external RAM location addressed by Register R0 or R1.
A	Accumulator
addr11	destination address for LCALL and LJMP, can be anywhere within the same 2kbytes page of program memory as the 1st byte of the following instruction.
addr16	destination address for LCALL and LJMP, can be anywhere within the 64 Kbytes program memory address space.
bit	128 software flags, any bit-addressable I/O pin, control or status bit.
#data	8-bits constant included in the instruction.
direct	128 internal adjacent RAM locations, any I/O port, control or status register.
#data16	16-bit constant included in instruction.
rel	SJMP and all conditional jumps include an 8-bit 2's complement offset byte. The Range is -128 to +127 bytes relative to the 1st byte of the next instruction.
Rn	working registers R0-R7.

1.3 General information on microcontrollers

The object of a microcontroller is to integrate the maximum number of functions in one single component. A microcontroller should be capable of responding to input signals, applying command instructions, generating output signals, managing communication, while having the following qualities:

- Programmable options, allowing the microcontroller to adapt to the greatest number of industrial environments and command equations, communication modes.

- Economic both in production with low cost on long production runs and in development with fast development cycle.

1.4 The 8051 family

1.4.1 Introduction

Although many families of microcontrollers exist, the 8051 one has stood out due to its general qualities, and its evolutionary organization caused by the flexibility of the register segment.

This segment, which was under-exploited in the original version, allowed INTEL, and many other manufacturers, to produce a variety of enhanced derivatives. These components, which are all compatible with the 8051 due to their common core, are differentiated by the addition of supplementary internal peripherals such as bus controllers, enhanced timers, analogue-to-digital converter, etc. The common core allows the same development tool (Raisonance MA-51 assembler) to be used across the whole range of components.

1.4.2 Internal organization

The reference manuals for the 8051 and its derivatives provide complete diagrams of the internal organization of the microcontrollers. Figure 1 shows the organization of the addressable segments in a simplified form.

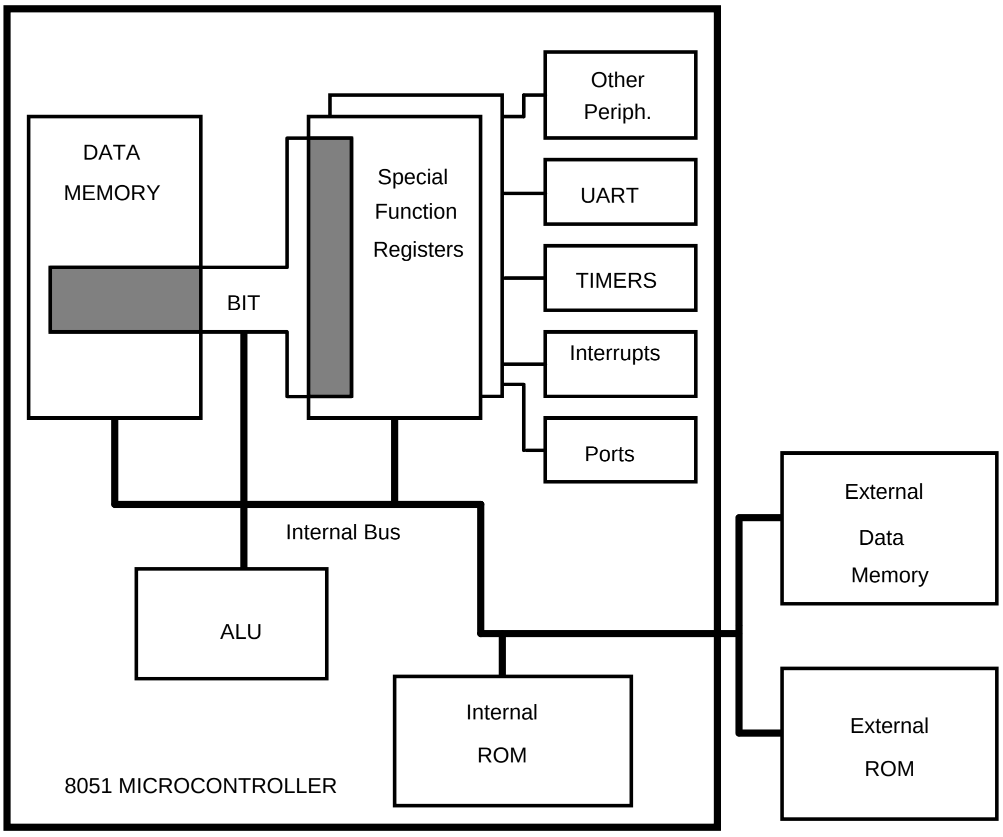

Figure 1: addressable segments of the 8051

This figure shows several physical spaces, which may be addressed using different types of instructions:

- The *CODE* is read by the internal sequencer, which loads an instruction to be executed every six clock pulses. Alternatively, the code bytes can be read by the programmer using the '**MOVC**' (move code) instruction. There are 64 Kbytes that can be addressed indirectly.

- The external RAM (XDATA) is read or written using the '**MOVX**', (move xdata) instruction. There are 64 Kbytes that can be addressed indirectly. Some devices (such as SAB8XC515A, 8XC592...) have an additional internal RAM (XRAM) which may be addressed as if it were the XDATA space. As far as the MA-51 is concerned, this space is considered as the XDATA space.
- The control or data registers (REG), of the internal peripherals (such as timers, parallel or serial ports, and interrupt controller), are accessible by using the direct transfer instructions on the internal bus, (128 bytes addressable from address 128 (80h) to 255 (0FFh). They may not be indirectly accessed, which makes their use safer. These registers never occupy all of the 128 bytes attributed space, and the number of registers varies from 23 for the 8051 to 90 for the 80517.
- The internal RAM (DATA) occupies 128 or 256 bytes depending on the microcontroller, and is also addressable on the internal bus. All the space is indirectly addressable, but the 128 bytes between addresses 0 and 127 (7Fh) are also directly addressable, as they do not overlap the register segment.
- There are 256 addressable bits (BIT), located within the 16 bytes of internal RAM and 16 bytes of internal registers. They may be addressed directly using the following instructions: SETB (set bit), CLR (clear bit), or JB, JNB, JBC (jump if bit group).
- This organization has been fully optimized to improve access to the variables, to increase programming options and to improve the overall program reliability.
- The core architecture has become more complex, and can surprise the inexperienced programmer. Hereafter are some examples of possible difficulties :
- The principal registers (such as **A**, **B**, **R0** to **R7**, and **DPTR**) which are implicitly addressed by numerous instructions, such as 'MUL AB', are also addressable directly within the REGISTER segment, or even in the DATA segment (for R0 to R7). Thus, "MOV R0, A" (implicit), is equivalent to "MOV AR0, ACC" (direct), but only requires one byte of code space instead of three.
- Some registers, such as PC (Program Counter), are not addressable. Others are addressable only, either in 'read', or in 'write'. The input/output ports, when read, may supply a value from either an internal latch or the port pin, depending on the instruction. Similarly, data written to SBUF affects the *transmit* buffer, and data read from SBUF affects the *receive* buffer.
- The addressable bits (which can be modified using the CLR and SETB instructions) are not a uniform and independent segment. 128 of them are taken from the internal RAM, (bytes 20h to 2Fh), and the other 128 are made up of the 16 bytes of the REGISTER segment which have addresses divisible by 8. Note that the address of the least significant bit of each of the registers corresponds exactly to the address of the register. The parallel ports, and most of the configuration bytes are bit addressable.
- The main registers, R0 to R7 are implicitly addressable and form a 'register bank'. One of four register banks can be selected. These are located in the lower part of the internal RAM, and have addresses from 0 to 1FH:

[0h to 07h]	for bank 0
[8h to 0Fh]	for bank 1
[10h to 17h]	for bank 2
[18h to 1Fh]	for bank 3

- The ability to change the register bank is useful whenever there is an interrupt. In practice, a bank is often allocated to an interrupt priority level, which avoids the need to save and then recall, the values of the registers of the bank active at each interrupt. The address of the first byte of the active bank is equal to the value (PSW ‘anded’ with 18h).
- Procedure calls use a stack contained in the internal RAM to save the return address. This stack is ascending, and is therefore generally placed above the last byte used for data. The stack pointer indicates the address most recently used and a stack overflow does not generate an interrupt.

2 Overview of an assembly program

2.1 Assembly statements

There are 3 assembly statements:

- Directives: are used to control the way the assembler processes assembly language instructions.
- Controls: are used to control the way the assembler generates the object file and the listing file. Controls may usually be used either on the command line or in the source module.
- Instructions: specify the source code to assemble.

2.2 Comments

Comments are used to make a program more readable. They are introduced with a ‘;’ and ignored by the assembler.

Syntax: **;** text

```
Example:
Buffer:    DS   3   ;The comment
                    ;begins after the semi-colon.
```

2.3 Keywords

Keywords are symbols used by the assembler that must not be redefined. A list of keywords is given in appendices.

2.4 Symbols

Symbols can be used before assembly instructions, but not before directives ($INCLUDE, etc.). In the variables segments (data, bit, xdata, code), symbols can be assigned to any address.
A symbol is equivalent to the numerical value of the address at which it is allocated:

```
Example:
CSEG   AT   100H
              ; selection of an absolute code segment
MESSAGE:   DB     ‘HELLO’, 0
LABEL:     MOV    DPTR,  #MESSAGE
```

A symbol may contain any of the following characters, digits ‘0..9’, letters ‘A..Z’ and ‘a..z’, dollar sign ($), underline (_), and question mark (?) but must not begin with a digit
.

Note:
1. Symbol length is unlimited.
2. Be careful not to use a keyword.
3. Declarations are visible from the beginning to the end of the program, including INCLUDE files.
4. The $ symbol alone represents the current address in the active segment, which is that of the first byte of the line.
5. Implicitly addressed registers such as A, C, B, DPTR, PC, and Rn are not considered as symbols, and do not need to be declared.

```
Example:
```

```
cu_ad:   JNB TI, $
             ;branches to the current address (which is
             ;cu_ad in this example) while TI is 0
```

2.5 Labels

A label is a symbol used to mark a particular place in an assembly program. It may refer to program code, to variable space in internal data memory, to variable space in external data memory, or to constant data stored in the code space.

Syntax: **label_name:**

```
Example:
                    ;  beginning of the initialization block
Init_start:    DB‘0’,’1’
Init_end:      JMP Start_prog
                    ;  end of the initializing block
                    ;  jump to the address of the beginning
                    ;  of the program
...
Start_prog:
```

2.6 Operands

Operands are arguments of instructions or directives. They may be divided into 3 groups:

2.6.1 Numerical, character and string operands

Numerical operands must be followed by a letter that indicates the base in which they are specified. Base types are hexadecimal, decimal, octal or binary, with H (or h), D (or d), O (or o or Q or q), B (or b) respectively. When a numerical value is specified, it must begin with a digit 0..9.
If no indicator is given the default base type is decimal.

```
Example1:
Hex:    DB    00fh    ; hexadecimal operand, value 15 decimal
Dec:    DB    003d    ; decimal operand
Oct:    DB    056o    ; octal operand, value 46 decimal
Bin:    DB    010b    ; binary operand, value 2 decimal
```

Character operands may be composed of one or two ASCII characters between two quotes and may be used wherever a numerical operand is valid.

```
Example2:
          Ch_A   EQU    ’AB’
```

String operands are composed of more than one character between single quotes and must be used with DB directive.

```
Example3:
Message:   DB ‘This is a message from Raisonance’
```

2.6.2 Address pointer

Each segment is controlled by an address pointer, which contains the offset of the instruction or data being assembled. Address pointers are initialized to 000h unless the ORG directive is used. The current counter location of the current segment may be obtained by employing dollar sign character ($).

```
Example:
RSEG   mycodseg
             ; supposes that mycodseg is a CODE segment
    Loc_count_cod      EQU     $
             ; Loc_count_cod symbol is assigned the mycodseg
             ; CODE segment address pointer value
DSEG   mydatseg
             ; absolute data segment specification
    Loc_count_dat      EQU     $
             ; Loc_count_dat symbol is assigned the mydatseg
             ; DATA segment address pointer value
             ; which is different from Loc_count_cod
```

2.6.3 Main registers

8051 (and 8051 derivatives) register names are pre-defined. They are listed below:

- A: accumulator (8-bit register)
- R0-R7: general purpose registers (8-bit registers)
- DPTR: 16-bit data pointer generally used to address data in external data memory or program memory
- DPL: 8-bit register containing the 8 lower bits of DPTR
- DPH: 8-bit register containing the 8 upper bits of DPTR
- PC: program counter (16-bit register)
- C: carry flag
- AB: represents the A and B register pair
- AR0-AR7: absolute data address of R0-R7 in the current register bank (RR0-RR7 are also accepted, for compatibility with previous versions of the assembler)

2.7 Operators

Binary operators must be used with 2 operands whereas unary operators are used with a single operand. They are listed in tables 1 to 7. All expressions use unsigned integers.

2.7.1 Binary operators

Binary operators are operators that have 2 operands.

2.7.1.1 Arithmetic operators

Operator	Syntax	Description
+	exp + exp	addition
-	exp - exp	Subtraction
*	exp * exp	Multiplication
/	exp / exp	integer division
MOD	exp MOD exp	remainder

Table 1: binary arithmetic operators

2.7.1.2 Relational operators

Operator	Syntax	Description
GTE	exp1 GTE exp2	true if exp1 is greater than or equal to exp2
>=	exp1 >= exp2	
GT	exp1 GT exp2	true is exp1 is greater than exp2
>	exp1 > exp2	
LTE	exp1 LTE exp2	true if exp1 is less than or equal to exp2
<=	exp1 <= exp2	
LT	exp1 LT exp2	true if exp1 is less than exp2
<	exp1 < exp2	
EQ	exp1 EQ exp2	true if exp1 is equal to exp2
=	exp1 = exp2	
NE	exp1 NE exp2	true if exp1 is not equal to exp2

Table 2: relational operators

2.7.1.3 Shifting operators

Operator	Syntax	Description
SHR	exp SHR num	shifts exp, num times to the right
SHL	exp SHL num	shifts exp, num times to the left

Table 3: shifting operators

2.7.1.4 Binary operands operators

Operator	Syntax	Description
AND	exp1 AND exp2	true if exp1 and exp2 are both true
OR	exp1 OR exp2	true if either exp1 is true or exp2 is true
XOR	exp1 XOR exp2	true if either exp1 is true, or exp2 is true, but not both at the same time.

Table 4: Boolean operators

2.7.2 Unary operators

Unary operators are operators that have a single operand.

2.7.2.1 Arithmetic operators

Operator	Syntax	Description
+	+ exp	returns exp
-	- exp	returns minus exp

Table 5: unary arithmetic operators

2.7.2.2 Miscellaneous operators

Operator	Syntax	Description
NOT	NOT exp	bit-wise complement
LOW	LOW exp	low-order byte of 16-bit expression
HIGH	HIGH exp	high-order byte of 16-bit expression

Table 6: miscellaneous unary operators

2.7.3 Operator precedence

Expression may include many operators with implicit precedence. The following table lists all operators with their respective precedence.

Operator	Priority
( )	1 (first operator to be evaluated)
HIGH, LOW, NOT	2
+ (unary), - (unary)	3
*, -, MOD	4
+, -	5
SHR, SHL	6
AND, OR, XOR	7
GTE, >=, LTE, <=, GT, >, LT, <, EQ, =, NE	8 (last operators to be evaluated)

Table 7: operator precedence

Note:
It is highly recommended to use parenthesis to encapsulate complex expressions (see example).

```
Example:
  IF( X=1 AND  Y=2) ...
  IF((X=1) AND (Y=2)) ...
            ; these expressions are not processed in the
            ; same way because the first one is seen as
  IF( (X= (1 AND Y)) =2)
```

2.8 Expressions

An expression is a combination of operands and operators and must be calculated by the assembler. It may become complex as symbols, used in that expression, may come from various segments.

Expressions containing a relocatable or an external symbol are called relocatable expression.

2.9 Memory areas

The 8051 and its derivatives use different addressing modes for different memory segments.

- *CODE*: internal or external ROM of up to 64K-bytes of read-only executable code or constants, accessible using the instruction MOVC.
- *XDATA*: external RAM, of up to 64KB of addressable variables accessible using the MOVX instructions.
- *DATA:* internal RAM (0-7FH) addressable directly or indirectly.
- *IDATA*: internal indirectly addressable RAM (0-0FFH), overlapping the DATA segment in the area 0 to 7FH.
- *BDATA*: internal bit addressable RAM (20H-2FH) overlapping the DATA segment.
- *BIT*: addressable bits of the BDATA segment (0-7FH).
- The two following are for compatibility with older syntax.
- *REG*: directly addressable internal registers (7Fh-0FFh).
- *RBIT*: register bits, addressable as bits 80h to 0FFh, specifically, all registers which have an address that is a multiple of eight (i.e. 80h, 88h, 90h, etc.).

3 Program basics

This chapter aims to present the basic elements of assembler programming:

- defining all segments, relocatable or absolute, data or code.
- defining and initializing symbols.
- writing the ‘core’ program.
- generating the object file.

All other directives and controls are secondary and will be dealt with in section 4.

GLOBAL is a general directive available for all the Raisonance tools. It allows to use a directive whatever the toolchain is. If the used tool implements the specified directive it will take it into account, if the used tool does not implement the directive, it will ignore it. It allows to keep the same code for different tools.
Syntax: global(directive)

Each paragraph is based upon the same principle. The first part (*Classical Syntax*) details INTEL syntax which is the syntax now adopted by the Raisonance MA assembler. The second one presents the *Simplified Syntax* which is appropriate for the previous Raisonance EMA-51 assembler and which is still supported by this version.

Note
With the DOS version, compilation with the assembler displays some additional information that are not necessary useful. These information may not be displayed thanks to the **QUIET** control.
Example:
MA51.exe C:\Ride\Examples\8051\MA51\ma51demo.a51 QUIET
This example compiles the ma51demo.a51file

3.1 Segment

3.1.1 Definition

With the exception of the REG and RBIT segments, a segment can be placed either in relocatable mode, or in absolute mode, (taking care to specify the address). This can be done regardless of the output format. The 'instructions', the initialization, and the addressing modes, differ from one physical space to another.

At each instruction, the current segment address can be re-assessed, and increased by the assembler to the size of memory space needed by the instruction. Therefore, the same number of address counters as types of segments are controlled in parallel.

Overlapping of absolute segments in the code segment generates an error.
Overlapping of absolute segments is allowed in the variables segment but a warning is generated.

The following sections aim to describe the directives that must be used to declare a relocatable segment, to select a relocatable segment or to select an absolute segment.

3.1.2 Classical Syntax

In order to remain compatible with former Raisonance assemblers (EMA-51) and with other similar assemblers, MA version 6 supports both the previous versions of Raisonance assembler syntax (herein referred as « *Simplified Syntax* ») and ASM51 syntax (herein referred as «*Classical Syntax* »). This section is dedicated to the *Classical Syntax*.

3.1.2.1 Relocatable segment

All relocatable segments must be declared by the SEGMENT directive. The name of the segment is specified by the symbol name immediately preceding SEGMENT directive.

Syntax: seg_name **SEGMENT** seg_type [reloc_type]

1. Its type is specified immediately after the SEGMENT directive and must be chosen from BIT, CODE, DATA, IDATA and XDATA.
2. The relocation type determines the relocation operations that may be performed by the assembler. This parameter must be chosen among BITADDRESSABLE, INBLOCK , INPAGE, OVERLAYABLE, PAGE and UNIT.

Relocation type (is an optional parameter that defaults to INPAGE):

- BITADDRESSABLE: the segment will be relocated within the bit addressable memory area.
- INBLOCK: specifies a segment of not more than 2048 bytes. This relocation type must only be used with CODE segments.
- INPAGE: specifies a segment of not more than 256 bytes. This relocation type must only be used with CODE or XDATA segments.
- OVERLAYABLE: specifies a segment that may share memory with other segments.
- PAGE: specifies a segment starting address must be a multiple of 256 bytes. This relocation type must only be used with CODE or XDATA segments.
- UNIT: specifies a segment whose starting address must be a unit of the segment type, i.e. a bit for BIT segments and a byte for CODE, DATA, IDATA or XDATA segments.

```
Example:     ;
mycodseg     SEGMENT    CODE   INPAGE
             ; this relocatable code segment has a size less
             ; than 256 bytes
```

Once declared a segment may be selected with the RSEG directive. It then remains the current segment until a new segment is specified.

Syntax: **RSEG** segment_name

```
Example:     ;

mycodseg     SEGMENT  CODE   INBLOCK
             ; this relocatable code segment has a size less
             ; than 2048 bytes
mydat1seg    SEGMENT  DATA
             ; mydat1seg DATA segment declaration
mydat2seg    SEGMENT  DATA
             ; SEGMENT statement allows several segment
             ; declaration even with the same type
RSEG    mycodseg
             ; mycodseg CODE segment selection
  MOV   TMOD,#20h
  MOV   TCON,#44h
             ; instructions within the mycodseg CODE segment
RSEG    mydat1seg
             ; mydat1seg DATA segment selection
  COUNTER:   DB  1
             ; memory allocation
RSEG    mycodseg
             ; mycodseg CODE segment may be reselected
             ; the address pointer in mycodseg is automatically
             ; increased depending on the instruction's length
  NOP
RSEG    mydat2seg
             ; mydat2seg DATA segment selection
```

3.1.2.2 Absolute segment

Absolute segments may be declared without employing the SEGMENT directive but only BSEG, CSEG, DSEG, ISEG or XSEG (only for 8051).

3.1.2.2.1 Selecting a segment in the bit address space

The BSEG directive is used to select an absolute segment within the bit address space. The segment's starting address may be specified using the keyword 'AT'. When not specified, the segment's starting address is either the physical segment's starting address or the address following the last address used in the previous segment of the same type. The segment name does not have to be specified and is implicitly generated.

Syntax: **BSEG [AT** addr**]**

```
Example:

mydatseg   SEGMENT    DATA   INPAGE
             ; DATA relocatable segment declaration

BSEG   AT   10h
             ; absolute BIT segment declaration

  bit1:    DBIT 1
             ; absolute bit declaration

RSEG   mydatseg
             ; mydatseg DATA relocatable segment selection
  COUNTER: DB1

BSEG
             ; absolute BIT segment declaration
             ; as no address is specified, the default
             ; address is 11h

  bit2:    DBIT   1
             ;declares an absolute bit
```

3.1.2.2.2 Selecting a segment in the code address space

The CSEG directive is used to select an absolute segment within the bit address space. The segment's starting address may be specified using the keyword 'AT'. When not specified, the segment's starting address is either the physical segment's starting address or the address following the last address used in the previous segment of the same type. The segment name does not have to be specified and is implicitly generated.

Syntax: **CSEG [AT** addr**]**

```
Example:
mycodseg   SEGMENT    CODE
             ; CODE relocatable segment declaration
CSEG   AT   800h
             ; absolute CODE segment declaration
             ; its starting address is 800h
  PUSH      ACC
             ; instruction length: 2 bytes
RSEG        mycodseg
             ; mycodseg CODE segment selection
  INC     DPTR
  MOV       A,B
CSEG
             ; absolute segment declaration
             ; as no address is specified, address is 802h
  ...
```

3.1.2.2.3 Selecting a segment in the data address space

The DSEG directive is used to select an absolute segment within the bit address space. The segment's starting address may be specified using the keyword 'AT'. When not specified, the segment's starting address is either the physical segment's starting address or the address following the last address used in the previous segment of the same type. The segment name does not have to be specified and is implicitly generated.

Syntax: **DSEG [AT** addr**]**

```
Example:
DSEG   AT     30h
             ; absolute DATA segment selection
             ; DSEG has the same syntax as CSEG
```

3.1.2.2.4 Selecting a segment in the indirectly addressable internal data address space

The ISEG directive is used to select an absolute segment within the bit address space. The segment’s starting address may be specified using the keyword ‘AT’. When not specified, the segment’s starting address is either the physical segment’s starting address or the address following the last address used in the previous segment of the same type. The segment name does not have to be specified and is implicitly generated.

Syntax: **ISEG [AT** addr**]**

```
Example:
ISEG AT 100h
             ; absolute indirect DATA segment selection
             ; ISEG has the same syntax as CSEG
```

3.1.2.2.5 Selecting a segment in the external data address space

The XSEG directive is used to select an absolute segment within the bit address space. The segment’s starting address may be specified using the keyword ‘AT’. When not specified, the segment’s starting address is either the physical segment’s starting address or the address following the last address used in the previous segment of the same type. The segment name does not have to be specified and is implicitly generated.

Syntax: **XSEG [AT** addr**]**

```
Example:
XSEG  AT  1000h
             ; absolute external DATA space selection
             ; XSEG has the same syntax as CSEG
```

Note:

If ‘addr’ is not specified, the final address of the previous segment is used. If ‘addr’ is not specified when a segment is referenced for the first time the address used will be zero.

3.1.3 Simplified Syntax

In order to remain compatible with former Raisonance assemblers (EMA-51) and with other similar assembler, MA version 6 supports both the previous versions of Raisonance assembler syntax (herein referred as « *Simplified Syntax* ») and ASM51 syntax (herein referred as «*Classical Syntax* »). This section is dedicated to the *Simplified Syntax* .

Before defining symbols or writing instructions, the segment in which you are working must be specified by one of the following directives: CODE, XDATA, DATA, REG, BDATA, BIT and RBIT (at the start of the program, the CODE segment is used default). The segment name does not have to be specified and is implicitly generated. For example, ?PR?Trial would be the name generated for the relocatable code segment of the program Trial. Broadly speaking, the segment name has the same name as the module, prefixed by its type (PR for CODE, XD for XDATA (8051 specific), DT for DATA, ID for IDATA, DT for bit-addressable DATA, BI for BIT) with one question mark before and one after.

When used alone, these directives select a relocatable segment. To select an absolute segment, prefix the segment name with the keyword ‘AT’ and append the required address.

This command can either be included on the same line as the directive (for changing the nature of the segment) or on a different line.

For relocatable segments, the current address is the last address used in the segment. At the beginning of the assembly, these addresses are initialized as follows:

- 0000H for CODE, XDATA and BIT.
- 20H for BDATA and IDATA.

Example 1: ; example with the *Simplified Syntax*

```
      LJMP   INIT
               ; Start of program
               ; Supposes address is 0, in the segment
               ; relocatable CODE, instruction has 3 bytes.

CODE  AT     3BH
               ; Absolute segment CODE
      LJMP  PROG_INT

CODE
               ; repositioning in the relocatable segment
               ; CODE relative address 3
      NOP

AT    100H
               ; absolute segment code selection
               ; the previous segment (code) has not
               ; been modified
      MOV    SP, #STACK
      MOV    SCON, #40H
      RET

DATA
               ; supposes address 20H.
  VAR1:  DB  2
               ; reserves 2 bytes
  VAR2:  DW  4
               ; reserves 8 bytes
STACK:

CODE
               ; repositioning in the relocatable
               ; CODE segment
      NOP
               ; current address at 4H

DATA
               ; current address at 0AH (10) on the STACK
               ; in the DATA segment
```

The directive IDATA allows you to reserve areas of internal RAM in the IDATA segment. These bytes can only be accessed by indirect register designation. It is advisable to use the IDATA directive (prior to the data directive) for reserving areas, which do not require data addressing (stack, data tables...).

With the *Classical Syntax*, example1 becomes example2.

```
Example 2:          ; example with the Classical Syntax
prg      SEGMENT    CODE
                  ; prg CODE segment declaration
RSEG   prg
                  ; prg CODE segment is selected and
                  ; becomes the current segment

        LJMP   init
                  ; Start of program
                  ; Supposes address is 00h, in the
                  ; relocatable segment prg
                  ; instruction has 3 bytes.

CSEG   AT   3BH
                  ; Absolute segment CODE declaration
        LJMP   PROG_INT

RSEG   prg
                  ; repositioning in the relocatable
                  ; segment prg relative address 3
        NOP

CSEG   AT   100H  ; selection of the absolute segment code,
                  ; without having modified the
                  ; previous segment (code)
        MOV  SP, #STACK
        MOV  SCON, #40H
        RET

DSEG
                  ; supposes address 20H.
  VAR1:  DB   2
                  ;reserves 2 bytes
  VAR2:  DW   4
                  ;reserves 8 bytes

STACK:

RSEG   prg
                  ; repositioning in the relocatable
                  ; CODE segment
      NOP
                  ;current address at 4h

DSEG              ; current address at 02Ah
                  ; in the DATA segment
```

With the *Classical Syntax* the relocation type (INBLOCK, PAGE, and INPAGE), is specified when declaring the segment with the SEGMENT statement.

Simplified syntax	*Classical syntax*

`CODE INBLOCK`	`mycodseg  SEGMENT  CODE  INBLOCK`
	`RSEG mycodseg`

`XDATA PAGE`	`myxdtseg  SEGMENT  XDATA  PAGE`
	`RSEG myxdtseg`

`XDATA INPAGE`	`myxdtinpage SEGMENT XDATA INPAGE`
	`RSEG myxdtinpage`

In the *Classical Syntax* column of the previous table, the line including the SEGMENT statement must not be re-specified each time the segment is selected.

3.1.4 Segment location

ORG is used to specify the offset for subsequent code or data. The address assigned after ORG directive must be specified by an absolute or simple relocatable expression without forward references.

Syntax: **ORG** exp

The ORG statement is often used to define the program's beginning.

Note:

ORG directive does not produce a new segment but changes the address pointer within it. When encountered in a relocatable segment, the offset is calculated from the segment's starting address.

END directive specifies the end of the current module.

***Syntax:* END**

Note:

Any text following END directive is ignored.

```
Example:
DSEG     AT   30h
              ; absolute DATA segment declaration
ORG      127h
              ; data address pointer is now 127h
              ; space between 100h and 127h is empty
message:      DB'HELLO'
ORG      100h
              ; data address pointer is now 100h
table:   DB'Raisonance MA-51 assembler'
              ; be careful not to redefine the message
```

3.2 Memory

When developing an assembly program, you will have to :

- reserve data spaces,
- initialize those spaces when they are in CODE space,
- select the register bank you want to use,
- assign symbols to particular addresses, registers or values.

3.2.1 Classical Syntax

In order to remain compatible with former Raisonance assemblers (EMA-51) and with other similar assembler, MA version 6 supports both the previous versions of Raisonance assembler syntax (herein referred as « *Simplified Syntax* ») and ASM51 syntax (herein referred as «*Classical Syntax* »). This section is dedicated to the *Classical Syntax*.

3.2.1.1 Symbol definition

3.2.1.1.1 Defining a symbol referring to a bit address

The BIT directive is used to define a symbol that references a bit address. The name of the symbol to define is specified by the string immediately preceding the BIT directive. The address of the bit declared is controlled by the address following BIT directive.

Syntax: symb **BIT** bit-add

```
Example:
RSEG    dat
            ; select a relocatable DATA segment
  sw:         DS    1
            ; 1-byte is reserved for the status word sw
  file_full   BIT   sw.1
            ; file_full symbol now references bit one of sw
  file_state  BIT   sw.6
            ; file_state symbol now references bit six of sw
```

Note:

symb can not be redefined or changed.

3.2.1.1.2 Defining a symbol referring to a program address

The CODE directive is used to associate a program address with the specified symbol. The name of the symbol to be defined is specified by the string immediately preceding the CODE directive. The address of the code symbol is controlled by the expression following CODE directive.

Syntax: symb **CODE** addr

```
Example:
  int0   CODE     100h
              ; defines int0 as being address 100h
  int1   CODE     int0 + 10h
              ; relative definition may be processed
  int2   CODE     int1 + 20h
CSEG     AT       20h
              ; absolute CODE segment declaration
  LJMP   int0
              ; goto 100h
```

Note:

symb can not be redefined or changed.

3.2.1.1.3 Defining a symbol referring to an internal data address

The DATA directive is used to associate an internal data address with the specified symbol. The name of the symbol to be defined is specified by the string immediately preceding the DATA directive. The address of the data symbol is controlled by the expression following the DATA directive.

Syntax: symb **DATA** addr

```
Example:
  buff_st    DATA   30h
             ; buff_st symbol now represents address 30h
  buff_en    DATA   buf_st + 10
             ; relative assignment may be processed
```

Note:

symb can not be redefined or changed.

3.2.1.1.4 Defining a symbol referring to an indirectly addressable internal data address

The IDATA directive is used to assign an indirectly addressable internal data address to the specified symbol. The name of the symbol to be defined is specified by the string immediately preceding the IDATA directive. The address of the idata symbol is controlled by the expression following IDATA directive.

Syntax: symb **IDATA** ad

```
Example:
  clock1   IDATA  50h
             ; clock1 represents now address 50h
  clock2   IDATA    clock1   - 1
             ; clock2 represents now address 4Ah
```

Note:
symb can not be redefined or changed.

3.2.1.1.5 Defining a 16-bit long constant data

The NUMBER directive allows constant data to be defined as integer values, 16-bit long. The constant data name must be specified after the NUMBER declaration..

Syntax: **NUMBER** symbol_name expression

```
Example:
  CSEG AT 1234h
             ; absolute CODE segment selection
lab1 : DB 1
  NUMBER   NUM_TSKS   12
  NUMBER   SWP_NUM1  (lab1>>8)+(lab1<<8)
             ; this directive is directly issued from
             ; previous Raisonance assembler
```

Note:
1. NUMBER directive may be used with PUBLIC and EXTRN directives. Symbol_name can subsequently be used in several modules, the reference being resolved at the linking stage.
2. This directive stems from the Raisonance EMA51 assembler, but is still supported by the current version.

3.2.1.1.6 Defining a symbol referring to an external data address

The XDATA directive is used to assign an external data address to a specified symbol name. The string immediately preceding the XDATA directive specifies the name of the symbol to be defined. The address of the xdata symbol is controlled by the expression following the XDATA directive.

Syntax: symb **XDATA** addr

```
Example:     ;
  Counter    XDATA    40h
             ; XDATA has the same syntax as CODE
```

Note:
symb can not be redefined or changed.

3.2.1.2 Memory reservation

At anytime within a segment, you may reserve some space without initializing it. Space may be reserved in BIT, DATA, and IDATA spaces.

3.2.1.2.1 Bit segment

The DBIT directive is used to reserve space in the bit segment. A label preceding the DBIT directive refers to the address of the start of the reserved memory. The number of bits to be reserved is specified by an expression following the DBIT directive.

Syntax: [lab] **DBIT** exp

```
Example:
BSEG     AT       100h
             ;  absolute BIT segment selection
  sw:    DBIT     8
             ; 8 bits are reserved for the status word sw
             ; address pointer is now 101h
  buff_status:    BIT    $
             ; buff_status is 101h
```

Note:

Space reservation in bit segment is done at the current address. The address pointer of the bit segment is increased by the value of exp. Exp cannot contain forward references, relocatable symbols or external symbols.

3.2.1.2.2 Internal data space

The DS directive is used to reserve space in the memory space (including code from BN 7.26). A label preceding the DS directive refers to the address of the reserved memory. The number of bytes to be reserved is specified by a number following the DS directive, which cannot contain forward references, relocatable symbols or external symbols.

Syntax: [lab] **DS** exp

```
Example:
DSEG       AT     20h
             ; absolute DATA segment selection
  Table:   DS     16
             ; reserves 16 bytes for Table, which are
             ; located at addresses 20h to 2Fh
  Buff:    DS     10
             ; reserves 10 bytes for Buff
             ; which are located from 30h to 39h
CSEG
             ; absolute CODE segment selection
  MOV      Table, #00h
             ; first byte of Table initialization
```

Note:

Space reservation is done in the current segment, at the current address. The current address is increased by the value of expression. Be careful not to exceed current address space limitations.

3.2.1.3 Memory initialization

Memory initialization may be done either before the beginning of the program or within the program. Another distinction must be made between code memory space initialization and data memory spaces initialization. In the former case, you may use DB or DW, which initialize CODE memory space with bytes or words respectively. In the second case, you may use native 8051 instructions (see section 3.3).

3.2.1.3.1 Byte values

The DB directive is used to initialize CODE memory with byte values. A label immediately preceding DB directive refers to the address of the initialized memory. ‘exp’ may be a symbol, a string or arbitrary expressions.

Syntax: [lab] **DB** exp [**,** exp ...]

```
Example:
CSEG       AT   100h
             ; absolute CODE segment declaration
  list:    DB   0,1,2,3,’Raisonance’,Low(list)
             ; memory content    100h   -> 0
             ;                    101h  -> 1
             ;                    102h -> 2 ...
```

3.2.1.3.2 Word values

The DW directive is used to initialize CODE memory with word values. A label immediately preceding the DW directive refers to the address of the initialized memory. ‘exp’ may be a symbol, a string or arbitrary expressions.

Syntax: [lab] **DW** exp [**,** exp ...]

```
Example:
CSEG       AT   30h
  Buffer:  DW   24h,25h
  Timer:   DW   $
             ; memory content    30h -> 0024
             ;                   32h -> 0025
```

3.2.1.4 Register bank reservation and selection

In MA-51, the REGISTERBANK directive specifies which register banks are to be used in the source module (4 register banks are available for coding ARn registers).

Syntax: **[$]REGISTERBANK(**num [**,**num...]**)**
Abbreviation: **[$]RB(**num [**,**num...]**)**

Note:
REGISTERBANK directive is preceded by a dollar sign ($) only if used in the assembly file. The dollar sign is not necessary in the command line.

The NOREGISTERBANK directive specifies that the register bank to be used is bank 0, which is the default bank.

```
Example 1:
  $RB(0,2,3)
                ; reservation of the areas corresponding to
                ; bank 0, 2 and 3
                ; These 3 register banks will be used
```

The USING directive allows you to specify which register bank is to be the current one. The directive must be followed by a number from 0 to 3 inclusive.

Syntax: **USING** exp

```
Example 2:
  $RB(0,1,2)
  USING  0
              ; register bank 0 is used
  PUSH AR2
              ; register R2 of bank 0 is pushed
  USING 1
              ; register bank 1 is used
  CLR AR2
              ; register R2 of bank 1 is cleared
  USING  2
  POP AR2
              ; register R2 of bank 2 is popped
```

3.2.1.5 Register or numerical value assignment

Register or numerical value assignment may be performed using SET or EQU directives. Unlike EQU, SET directive allows the assigned symbol to be reassigned.

3.2.1.5.1 Assignment or re-assignment

SET exists as a directive and as a control.

1. The SET *directive* is used to assign a numerical value or a register to a specified symbol name. The string immediately preceding the BIT directive specifies the name of the symbol to be defined. If a numerical value is assigned, this must immediately follow the SET directive and must not contain a forward reference. If a register symbol is assigned, this must be chosen from A, R0-R7.

Syntax: symb **SET** exp
symb **SET** reg

```
Example1:
  timer      SET   R4
  ref        SET   20
  counter    SET   ref + 2
  CSEG
  MOV        R3,#counter
             ; the MOV instruction is processed as
  MOV        R3,#ref+2
             ; is ‘seen’ as 20+2
```

Note:
1. Each occurrence of the defined symbol is replaced in the assembly program by the specified numerical value or register symbol.
2. symb can be changed by another SET statement.

2. The SET *control* assigns values to the specified symbols. These symbols may be used in $IF $ELSIF control statements for conditional assembly and are only used by those controls. Symbols declared with the SET control do not interfere with CODE, DATA, BIT and XDATA symbols.

Syntax: **$SET(**symbol[=number] [**,**symbol[=number]...]**)**

Note:
The SET control is preceded by a dollar sign ($) only if used in the assembly file. The dollar sign is not necessary in the command line.

If no number is specified, the symbol is initialized with 0FFFFh.

```
Example2:
             ; command line
MA51 raison.a SET(type=10)

             ; content of raison.a
$IF(type=10)
  bw:  DB   16
             ; in this case bw is assigned 16
$ELSE
  bw:  DB   8
$ENDIF
```

The RESET control performs the same initialization as the SET *control* with 0 as the number.

Syntax: **RESET(**symbol**)**

```
Example3:
$RESET(sb)
           ;sb is initialized with 0
;the same initialization could have been done with $SET(sb=0)
```

3.2.1.5.2 Definitive assignment

The EQU directive is used to associate a numerical value or register symbol with a specified symbol name. The string immediately preceding the EQU directive specifies the name of the symbol to be defined. If a numerical value is assigned, this must immediately follow the EQU directive and must not contain a forward reference. If a register symbol is assigned, this must be chosen from A, R0-R7.

Syntax: symb **EQU** exp
symb **EQU** reg

```
Example1:
  nb_elem      EQU   130
  elem_size    EQU   3
  buff_ptr     EQU   R1
  buu_size     EQU   nb_elem * elem_size
```

Note:
1. Each occurrence of the defined symbol is replaced in the assembly program by the specified numerical value or register symbol.
2. symb can not be redefined or changed.

3.2.2 Simplified Syntax

In order to remain compatible with former Raisonance assemblers (EMA-51) and with other similar assembler, MA version 6 supports both the previous versions of Raisonance assembler syntax (herein referred as « *Simplified Syntax* ») and ASM51 syntax (herein referred as «*Classical Syntax* »). This section is dedicated to the *Simplified Syntax*.

3.2.2.1 Symbol definition

3.2.2.1.1 Defining a symbol referring to an address

This type of definition must be preceded by a reference to the segment in which the declaration must take place.

```
Example 1:   ; Simplified Syntax
XDATA  at 1000h
             ; absolute XDATA segment specification
CODE
             ; absolute CODE segment specification
             ; you must come back to a code segment
             ; to declare a code symbol
at 110h
  message:   db  1
```

With the *Classical Syntax*, example1 becomes example2.

```
Example 2:   ; Classical Syntax
XSEG   at  1000h
             ; absolute XDATA segment specification
  message  CODE   110h
             ; you must not come back to a code segment to
             ; declare a code symbol
```

3.2.2.1.2 Defining a 16-bit long constant data

The NUMBER directive allows constant data to be defined as integer values, 16-bit long. The constant data name must be specified after the NUMBER directive.

Syntax: **NUMBER** symbol_name expression

```
Example 1:   ; Simplified Syntax
CODE      AT        1234h
             ; absolute CODE segment declaration
  lab1:    DB        1
  NUMBER   NUM_TSKS   12
  NUMBER  SWP_NUM1  (lab1>>8)+(lab1<<8)
```

Note:

1. NUMBER directive may be used with PUBLIC and EXTRN directives. Symbol_name can subsequently be used in several modules, the reference being resolved at the linking stage.
2. This directive stems from Raisonance EMA51 version but is still supported by the current version.

For NUMBER type symbols (numerical fixed data), the linker does not impose any particular rules. The NUMBER attribute can replace any other attribute, such as : CODE, XDATA, DATA, BIT etc. The following three examples are all valid:

```
Example 2:   ; Simplified Syntax
             ; Module i
  PUBLIC NUMBER NBR_TSK0 2000h

             ; Module j
  EXTRN NUMBER NBR_TSK0
```

```
Example 3:   ; Simplified Syntax
             ; Module i
  PUBLIC NUMBER ADR_TAB0 100h

             ; Module j
  EXTRN XDATA ADR_TAB0

             ; this assignment will result in ADR_TAB0 being
             ; treated as a symbol (representing an address)
             ; in XDATA space extern CODE ADR_TAB0 would
             ; also be valid
```

```
Example 4:   ; Simplified Syntax
             ; Module i
  PUBLIC XDATA ADR_TAB0 : db 100
             ; declaration of the symbol ADR_TAB0 in XDATA
             ; space + allocation of 100 bytes.
             ; Module j
  EXTRN NUMBER ADR_TAB0
             ; valid syntax
```

With the *Classical Syntax*, example1 becomes example5.

```
Example 5:   ; Classical Syntax
CSEG      AT        1234h
             ; absolute CODE segment declaration
  lab1:   DB        1
  NUMBER  NUM_TSKS  12
  NUMBER  SWP_NUM1  (lab1>>8)+(lab1<<8)
```

3.2.2.2 Space reservation and initialization

Whereas in the previous Raisonance MA-51 assembler versions, the DB directive was used to reserve or initialize space in the internal data space, external data space or code address space, it is only used from the current version to initialize CODE memory space with byte values.

Syntax: [lab] **DB** exp [**,**exp ...]

Each expression following DB statement is stored in a singe byte. In the current version, DATA and XDATA space reservation is processed by the DS directive and bit segment reservation is processed due to DBIT directive.

The same remark may be applied to DW, which is from the current Raisonance MA assembler version used only to initialize CODE memory space with word values.

Syntax: **[lab] DW** exp [**,**exp ...]

```
Example:     ; Simplified Syntax
  Table:   DB   1,2,3,4
  Label:   DW   24h,25h
```

3.2.2.3 Register bank selection

The RB directive is equivalent to the definition of eight 'SETs'.AR0-AR7 at the following address ranges :

```
  RB0   0 to 07h    AR0 = 0  AR7 =7h
  RB1   8 to 0Fh    AR0 = 8  AR7 = 0Fh
  RB2  10 to 17h    AR0 = 10  AR7 = 17h
  RB3  18 to 1Fh    AR0 = 18  AR7 = 1Fh
```

By default, AR0 to AR7 refer to addresses 0 to 7.

In OBJ format, this directive allocates eight bytes at the corresponding address of the data segment.

Note:

When working with the OBJ format and using several register banks (which is very common) use of the RB directive to reserve the corresponding address ranges is highly recommended.

```
Example 1:   ; Simplified Syntax
             ;(for the OBJ format)
             ;
  RB3
  RB2
  RB0
             ; reservation of the areas corresponding
             ; to bank 0, 2 and 3
```

```
Example 2:   ; Simplified Syntax
  RB2
             ; defines the addresses of AR0,...AR7
             ; respectively at 10H,...17H
             ; corresponding to the register bank  NUMBER 2
  MOV R4, AR5
             ; MOV R4,R5 is forbidden.
             ; AR5 is equivalent to 15H.
```

This is particularly useful for making direct transfers between 'registers', by using the assembler MOV ARi,Rj instruction.

The implicit relationship that is set up by the RB directive between ‘ARx’ and a memory location can lead to confusion if the programmer uses a different location explicitly defined using the ‘SET’ directive. It is advisable to use only the RB directive.

For instructions involving only one register R0 to R7, it is best to use implicit addressing, (for example, ‘mov Ri,num’ rather than ‘mov ARi,num’).

<u>Summary</u>:

- the 'names’ AR0 to AR7 are dealt with as EQUs and never treated as symbols.
- they simplify transfers within a register bank
- RB can also be used for memory allocation when working with the OBJ format.

With the *Classical Syntax*, example2 becomes example3.

```
Example 3:   ; Classical Syntax
  RB(2)
             ; specifies that register bank 2 will be used
             ; in the assembly module
...
  USING 2
             ; specifies that we now use bank 2
  MOV R4,AR5
```

3.2.2.4 Numerical value assignment

The EQU directive assigns a name to a character string. Each time the name occurs, the string will be substituted in its place. The assembler will only process the character string after the substitution. A symbol declared by an EQU cannot be defined elsewhere, in any form (Symbol, Macro, Mnemonic, etc.).

```
Example 1:   ; Simplified Syntax
  FLAGS_STATE:     DS   1
             ; 1-byte is reserved for FLAGS_STATE
  EQU   ANOMALY   FLAGS_STATE.4
             ; ANOMALY is the fifth bit of FLAGS_STATE
```

With the *Classical Syntax*, example1 becomes example2.

```
Example 2:   ; Classical Syntax
  FLAGS_STATE:     DS   1
             ; 1-byte is reserved for FLAGS_STATE
  ANOMALY  EQU  FLAGS_STATE.4
             ; ANOMALY is the fifth bit of FLAGS_STATE
```

3.3 Instructions

This section is a syntactic summary of all mnemonics with their description, their coding length and duration in machine cycles (a machine cycle is 12 oscillator periods, except for special 8051 derivatives).

For a detailed description of each instruction, please refer to the manufacturer’s manual for the particular microcontroller of interest.

3.3.1 Arithmetic instructions

Mnemonic	Operand(s)	Action	Byte(s)	Cycles	Opcode
ADD	A,Rn	Add register to Accumulator	1	1	28->2F
ADD	A,direct	Add direct byte to Accumulator	2	1	25
ADD	A,@Ri	Add indirect RAM to Accumulator	1	1	26, 27
ADD	A,#data	Add immediate data to Accumulator	2	1	24
ADDC	A,Rn	Add register to Accumulator with Carry	1	1	38->3F
ADDC	A,direct	Add direct byte to A with Carry flag	2	1	35
ADDC	A,@Ri	Add indirect RAM to A with Carry flag	1	1	36, 37
ADDC	A,#data	Add immediate data to A with Carry flag	2	1	34
SUBB	A,Rn	Subtract register from A with Borrow	1	1	98->9F
SUBB	A,direct	Subtract direct byte from A with Borrow	2	1	95
SUBB	A,@Ri	Subtract indirect RAM from A with Borrow	1	1	96, 97
SUBB	A,#data	Subtract immediate data from A with Borrow	2	1	94
INC	A	Increment Accumulator	1	1	04
INC	Rn	Increment register	1	1	08->0F
INC	direct	Increment direct byte	2	1	05
INC	@Ri	Increment indirect RAM	1	1	06, 07
INC	DPTR	Increment Data Pointer	1	2	A3
DEC	A	Decrement Accumulator	1	1	14
DEC	Rn	Decrement register	1	1	18->1F
DEC	direct	Decrement direct byte	2	1	15
DEC	@Ri	Decrement indirect RAM	1	1	16, 17
MUL	AB	Multiply A by B	1	4	A4
DIV	AB	Divide A by B	1	4	84
DA	A	Decimal Adjust Accumulator	1	1	D4

3.3.2 Logical instructions

Mnemonic	Operand(s)	Action	Byte(s)	Cycles	Opcode
ANL	A,Rn	AND register to Accumulator	1	1	58->5F
ANL	A,direct	AND direct byte to Accumulator	2	1	55
ANL	A,@Ri	AND indirect RAM to Accumulator	1	1	56, 57
ANL	A,#data	AND immediate data to Accumulator	2	1	54
ANL	direct,A	AND Accumulator to direct byte	2	1	52
ANL	direct,#data	AND immediate data to direct byte	3	2	53
ORL	A,Rn	OR register to Accumulator	1	1	48->4F
ORL	A,direct	OR direct byte to Accumulator	2	1	45
ORL	A,@Ri	OR indirect RAM to Accumulator	1	1	46, 47
ORL	A,#data	OR immediate data to Accumulator	2	1	44
ORL	direct,A	OR Accumulator to direct byte	2	1	42
ORL	direct,#data	OR immediate data to direct byte	3	2	43
XRL	A,Rn	Exclusive-OR register to Accumulator	1	1	68->6F
XRL	A,direct	Exclusive-OR direct byte to Accumulator	2	1	65
XRL	A,@Ri	Exclusive-OR indirect RAM to A	1	1	66, 67
XRL	A,#data	Exclusive-OR immediate data to A	2	1	64
XRL	direct,A	Exclusive-OR Accumulator to direct byte	2	1	62
XRL	direct,#data	Exclusive-OR immediate data to direct	3	2	63
CLR	A	Clear Accumulator	1	1	E4
CPL	A	Complement Accumulator	1	1	F4
RL	A	Rotate Accumulator Left	1	1	24
RLC	A	Rotate A Left through the Carry flag	1	1	33
RR	A	Rotate Accumulator Right	1	1	03
RRC	A	Rotate A Right through Carry flag	1	1	13
SWAP	A	Swap nibbles within the Accumulator	1	1	C4

3.3.3 Data transfer instructions

Mnemonic	Operand(s)	Action	Byte(s)	Cycles	Opcode
MOV	A,Rn	Move register to Accumulator	1	1	E8-EF
MOV	A,direct	Move direct byte to Accumulator	2	1	E5
MOV	A,@Ri	Move indirect RAM to Accumulator	1	1	E6, E7
MOV	A,#data	Move immediate data to Accumulator	2	1	74
MOV	Rn,A	Move Accumulator to register	1	1	F8-FF
MOV	Rn,direct	Move direct byte to register	2	2	A8->AF
MOV	Rn,#data	Move immediate data to register	2	1	78->7F
MOV	direct,A	Move Accumulator to direct byte	2	1	F5
MOV	direct,Rn	Move register to direct byte	2	2	88->8F
MOV	direct,direct	Move direct byte to direct	3	2	85
MOV	direct,@Ri	Move indirect RAM to direct byte	2	2	86, 87
MOV	direct,#data	Move immediate data to direct byte	3	2	75
MOV	@Ri,A	Move Accumulator to indirect RAM	1	1	F6, F7
MOV	@Ri,direct	Move direct byte to indirect RAM	2	2	A6, A7
MOV	@Ri,#data	Move immediate data to indirect RAM	2	1	76, 77
MOV	DPTR,#data16	Load Data Pointer with a 16-bit constant	3	2	90
MOVC	A,@A + DPTR	Move Code byte relative to DPTR to A	1	2	93
MOVC	A,@A + PC	Move Code byte relative to PC to A	1	2	83
MOVX	A,@Ri	Move External RAM (8-bit addr) to A	1	2	E2, E3
MOVX	A,@DPTR	Move External RAM (16-bit addr) to A	1	2	E0
MOVX	@Ri,A	Move A to External RAM (8-bit addr)	1	2	F2, F3
MOVX	@DPTR,A	Move A to External RAM (16-bit addr)	1	2	F0
PUSH	direct	Push direct byte onto stack	2	2	C0
POP	direct	Pop direct byte from stack	2	2	D0
XCH	A,Rn	Exchange register with Accumulator	1	1	C8->CF
XCH	A,direct	Exchange direct byte with Accumulator	2	1	C5
XCH	A,@Ri	Exchange indirect RAM with A	1	1	C6, C7
XCHD	A,@Ri	Exchange low-order nibble in RAM with A	1	1	D6, D7

3.3.4 Boolean instructions

Mnemonic	Operand(s)	Action	Byte(s)	Cycles	Opcode
CLR	C	Clear Carry flag	1	1	C3
CLR	bit	Clear direct bit	2	1	C2
SETB	C	Set Carry flag	1	1	D3
SETB	bit	Set direct Bit	2	1	D2
CPL	C	Complement Carry flag	1	1	B3
CPL	bit	Complement direct bit	2	1	B2
ANL	C,bit	AND direct bit to Carry flag	2	2	82
ANL	C,/bit	AND complement of direct bit to Carry	2	2	B0
ORL	C,bit	OR direct bit to Carry flag	2	2	72
ORL	C,/bit	OR complement of direct bit to Carry	2	2	A0
MOV	C,bit	Move direct bit to Carry flag	2	1	A2
MOV	bit,C	Move Carry flag to direct bit	2	2	92

3.3.5 Assembler Program Control instructions

Mnemonic	Operand(s)	Action	Byte(s)	Cycles	Opcode
ACALL	addrll	Absolute Subroutine Call	2	2	11, 31, 51 71, 91, B1 D1, F1
LCALL	addrl6	Long Subroutine Call	3	2	12
RET		Return from subroutine	1	2	22
RETI		Return from interrupt	1	2	32
AJMP	addrll	Absolute Jump	2	2	01, 21, 41 61, 81, A1 C1, E1
LJMP	addrl6	Long Jump	3	2	02
SJMP	rel	Short Jump (relative addr)	2	2	80
JMP	@A + DPTR	Jump indirect relative to the DPTR	1	2	73
JZ	rel	Jump if Accumulator is Zero	2	2	60
JNZ	rel	Jump if Accumulator is Not Zero	2	2	70
JC	rel	Jump if Carry flag is set	2	2	40
JNC	rel	Jump if No Carry flag	2	2	50
JB	bit,rel	Jump if direct Bit set	3	2	20
JNB	bit,rel	Jump if direct Bit Not set	3	2	30
JBC	bit,rel	Jump if direct Bit is set & Clear bit	3	2	10
CJNE	A,direct,rel	Compare direct to A & Jump if Not Equal	3	2	B5
CJNE	A,#data,rel	Comp. imm. to A & Jump if Not Equal	3	2	B4
CJNE	Rn,#data,rel	Comp. imm. to reg & Jump if Not Equal	3	2	B8->BF
CJNE	@Ri,#data. rel	Comp. imm. to ind. & Jump if Not Equal	3	2	B6, B7
DJNZ	Rn,rel	Decrement register & Jump if Not Zero	2	2	D8->DF
DJNZ	direct. rel	Decrement direct & Jump if Not Zero	3	2	D5
NOP		No operation	1	1	00

3.3.6 Concerning JUMP and CALL instructions

These instructions include the jump instructions (SJMP, AJMP, and LJMP), the conditional jump instructions (JZ, JNZ, CJNE etc.) and the call instructions (ACALL, LCALL).

- When writing CALL instructions in an assembly file, the assembler determines whether to use ACALL or LCALL Similarly, JMP will be replaced by AJMP or LJMP.
- Although there are no general rules concerning the implementation of these instructions, it may be useful to take into account the following advice: relative jumps (SJMP, JB, CJNE etc.) must refer to a single module. The 'jump instruction' and the 'destination' address must belong to the same module.

```
Example 1:
  EXTRN CODE TRANSMIT
  JB TI, TRANSMIT; incorrect syntax
```

There are three types of jump instructions, which are: SJMP, LJMP and AJMP. It is not possible to select automatically the optimum operation. SJMP must be reserved for 'local' calls. Similarly, the selection between LCALL and ACALL must be made manually. Both AJMP and ACALL are 2-byte instructions, in contrast to LCALL and LJMP, which are 3-byte instructions. The use of ACALL and AJMP allows some optimization, such as that shown in the example below. When the instructions AJMP or ACALL are used within a module, the output relocatable code segment is relocated within a 2K-byte page. The size of this segment is restricted to less than 2K, as with the INBLOCK directive.

```
Example 2:
JMP_TABLE:   MOV    A,R7
                  ; R7 holds the index of the function to run.
             RL     A
                  ; calculation of  the offset
                  ; (= instruction size * 2)
             MOV    DPTR, #TABLE
                  ; DPTR holds the start address of the table
             JMP    @A+DPTR
TABLE:       AJMP   FCT0
             AJMP   FCT1
             AJMP   FCT2
             AJMP   FCT3
```

- The instructions AJMP, LJMP, ACALL and LCALL are flexible in their use: They can be used between a relocatable segment and an absolute segment, and to allow redirection to an 'External' symbol, which is resolved by the linking stage.

```
Example 3:
  EXTRN CODE TRT_INT_1
  MOV A,R7
  ACALL TRT_INT_1  ; valid syntax
```

Note:

If you use AJMP or ACALL instructions, which refer to external symbols (as in the previous example), the linker, tries to resolve these references. However, this will not impose any extra constraints on the Linker. Thus, for "big applications" (>2K of code), it is not possible to know in advance if these references can be resolved.

3.4 Object file

Object directives controls object file generation, its name, debugging information and line numbers.

3.4.1 Object file generation

The default name for the object file is the source file name with the *.obj extension, however the OBJECT directive allows any desired name to be used. By default an object file is generated.

Syntax: **OBJECT(**object_file_name.obj**)**

***Abbreviation**:* **OJ**

```
Example 1:
  MA51 raison.a OBJECT(raison.obj)
```

```
Example 2:
  $OBJECT(raison.obj)
```

The NOOBJECT directive prevents generation of the object file.

Syntax: **NOOBJECT**

Abbreviation: **NOOJ**

```
Example 3:
  MA51 raison.a NOOJ
```

3.4.2 Object file name

When assembling the current module an object file is generated. Its name may be controlled by the string following the NAME directive.

Syntax: **NAME** str

```
Example:
  NAME rais_mod
```

Note:

If the object module name is not explicitly specified, it will be the name of the source file without the extension.

The NOPATH directive allows to not include the path of the source file in the object file (.obj). Only the name of the source will appear.

Syntax: **NOPATH**

3.4.3 Debugging information

The DEBUG directive controls the inclusion of debugging information in the object file.

Syntax: **DEBUG**

Abbreviation: **DB**

```
Example 1:
  MA51 raison.a DEBUG
```

```
Example 2:
  $DB
```

The NODEBUG prevents the debugging information from appearing in the object file.

Syntax: **NODEBUG**

Abbreviation: **NODB**

```
Example 3:
  MA51 raison.a NODB
```

The OBJECTEXTEND directive directs the assembler to include additional debugging information in the object file, which is used by Raisonance's debugging tools.

Syntax: **OBJECTEXTEND**

Abbreviation: **OE**

```
Example:
  MA51 raison.a OBJECTEXTEND
```

```
Example:
  $OE
```

3.4.4 Line number

The LINES directive allows the definition of line numbers in an assembly source file, similar to those generated by compilers. The text following the LINES directive is regarded as a comment, and does not need to be preceded by a semi-colon. The line number generated will correspond to the absolute line number of the current file.

Syntax: **LINES** text_comment

Abbreviation: **LN**

The NOLINES directive prevents the definition of line numbers in an assembly source file.

Syntax: **NOLINES**

Abbreviation: **NOLN**

Note:
The following restrictions must be taken into account:
1. The LINES directive generates a line number only if the current segment is code and is ignored in all other segments (data, bit, xdata, etc.).
2. The LINES directive is ignored when used in ‘INCLUDE’ files.

3.4.5 CodeCompressor directives

Hereafter are listed some directives used with the CodeCompressor. To get more information on these directives look at the CodeCompressor51 manual.

The POSTOPTIMIZE directive enables the generation of the post-optimization information inside the OMF-51 object file.

The MULTITARGET directive allows to define a multitarget instruction.

Syntax: **[$]MULTITARGET**(StartLabel,Type,NbTarget[,TargetLabel]+)

Where:

- **StartLabel** is the label where the multitarget instruction starts,
- **Type** defines the type of the multitarget instruction. For the moment it can be CALL_TYPE or JMP_TYPE
- **NbTargets** defines the number of targets for this multitarget instruction
- **TargetLabel** is a target label repeated NbTargets times.

Restriction:
The labels used in this control must be local (no external labels).

The MACROINSTRUCTION directive allows to determine MacroInstructions without any stack modification in terms of code compressor. This means that it is a group of instructions that you do not want to be separated by the code compressor in any way.

Syntax: **[$]MACROINSTRUCTION**(StartLabel,NbBytes)

Where:

- **StartLabel** is the label where the MacroInstruction starts,
- **NbBytes** defines the length in bytes of the macroinstruction.

Restriction:
The label used in this control must be local (no external labels).

Abbreviation: **MACROINST**

The MACROINSTSTK directive allows to determine MacroInstructions which affect the stack during their execution. This means that it is a group of instructions that you do not want to be separated by the code compressor in any way.

Syntax: **[$]MACROINSTRUCTIONSTACK**(StartLabel,NbBytes,StackModif)

Where:

- **StartLabel** is the label where the MacroInstruction starts,
- **NbBytes** defines the length in bytes of the macroinstruction.
- **StackModif** [-128,+127] is the affection of the stack during macroinstruction execution.

Restriction:
The label used in this control must be local (no external labels).

Abbreviation: **MACROINSTSTK**

The ACCESSTHROUGHCALL directive allows to define a new fixup type that is not generated by the assembler. When a fixup of this type is generated, each time the code is inlined, the value (referenced in the offset part) must be decremented by 2 by the code compressor. And each time the code is factorized, the value must be decremented by 2 by the code compressor.

Syntax: **[$]ACCESSTHROUGHCALL**(StartLabel, StackOffset)

Where:

- **StartLabel** is the position in the current code segment where the fixup must be resolved,
- **StackOffset** [-128,+127] is the initial value coded in the instruction.

Restriction:
The label used in this control must be local (no external labels).
This control must take place inside an OPEN code segment.

The ACCESSWITHOUTCALL directive allows to define a new fixup type that is not generated by the assembler. When a fixup of this type is generated, each time the code is inlined, the value (referenced in the offset part) must not be modified by the code compressor. But each time the code is factorized, the value must be decremented by 2 by the code compressor.

Syntax: **[$]ACCESSWITHOUTCALL**(StartLabel, StackOffset)

Where:

- **StartLabel** is the position in the current code segment where the fixup must be resolved,
- **StackOffset** [-128,+127] is the initial value coded in the instruction.

Restriction:
The label used in this control must be local (no external labels).
This control must take place inside an OPEN code segment.

The DONOTFACTORIZE directive allows to define an instruction that will not be factorize in anyway during the code compressor process.

Syntax: **[$]DONOTFACTORIZE** (Label[, Label]*)

Where:

- **Label** is the position in the current code segment of the instruction that must not be factorized.

Restriction:
The label used in this control must be local (no external labels).
This control must take place inside an OPEN code segment.

Abbreviation: **DNF**

The LOCKEDAREA directive has the same meaning has the MACROINST control except that the macro instruction defined here cannot have any of its instructions modified. This means that its length must be strictly respected.

Syntax: **[$]LOCKEDAREA (Start, ByteNr)**

Where:

- **Start** is the address where the macroinstruction starts,
- **ByteNr** is the number of bytes involved in the macroinstruction.

Abbreviation: **LA**

For example, this control must be used in case of a macroinstruction whose length is involved explicitly in an offset calculation to access a table in code:

```
Example:
  $LOCKEDAREA(TABLE-2,12)

    ;TABLE-2 is the address, from which size must not be
    ;modified,
    ;12 is the size of the area (2 bytes from index plus 10
    ;which is ;the ‘TABLE’ size)
```

3.5 High Level Language Syntax

MA51 supports the combination PROC/ENDP.

Syntax:	MyProcedure	**PROC**
	……;	My code
	ENDP	

```
Example:
  C51_START:

  MOV R6, #0

  CALL  MYPROCEDURE
  JMP   C51_START

PUBLIC MYPROCEDURE

MYPROCEDURE PROC

  MOV R4, #1
  RET

ENDP
```

4 Additional Facilities and Enhancements

4.1 Source file

4.1.1 Definition

In many projects, using assembly level coding, it is convenient to partition various functions and procedures into different files. This permits efficient testing and re-use in other projects. MA51 allows the use of INCLUDE files and other features to aid in file and project management.

4.1.2 File inclusion

The INCLUDE control allows an external file to be read as if it had appeared in the main text of a source file. There may be multiple INCLUDE statements which refer to different files and if the path is not specified the assembler searches the current directory.

The INCLUDE control is recursive and may be used within another INCLUDEd file. However, care should be taken to avoid cyclic references to files. Any changes to the active segment (or any others) persist after the inclusion of the file.

Syntax : **[$]INCLUDE (file_name)**
where file_name is the name of the file to include.

***Abbreviation**:* **[$]IC**

```
Example:
  $INCLUDE (c:\RIDE\INC\REG51.INC)     ;SFR assignments
```

4.1.3 Headers inclusion

The PATHINCLUDE control allows to define the directory where the headers are placed.

Syntax : **[$]PATHINCLUDE (directory_name)**
where directory_name is the name of the directory where the headers are placed.

***Abbreviation**:* **[$]PIN**

```
Example:
  $PATHINCLUDE (c:\RIDE\INC)  ;
```

4.1.4 Case sensitive

The CASESENSITIVE control allows to define variables as case sensitive, which means that two variables with the same spelling but with upper-case or lower-case letters will be different. NOCASENSITIVE is the opposite control.

Syntax: **CASESENSITIVE**
NOCASESENTIVE

***Abbreviation**:* **[$]CS**
[$]NOCS

This control can be used only in the command line

4.2 8051 SFRs : (NO)MOD51

The MA51 assembler allows you not only to use 8051 microcontroller but also its derivatives. Special Function Registers (SFRs) of 8051 derivatives are generally different from the original 8051 SFRs. MOD51 and NOMOD51 determine if the assembler uses the original 8051 definitions or not.

If the MOD51 control is specified, default definition of 8051 SFRs will be used.

Syntax: **[$]MOD51**

***Abbreviation**:* **[$]MO**

```
Example 1:
  MA51 raison.a MOD51
```

```
Example 2:
  $MOD51
        ;$include (REG51.INC) has the same effect.
```

The NOMOD51 control prevents definition of default 8051 special function registers.

Syntax: **[$]NOMOD51**

Abbreviation: **[$]NOMO**

```
Example 3:
  MA51 raison.a NOMO
```

Note:
dollar sign ($) is used when the control is specified in the assembly file and must not be used when the control is used in the command line.

4.3 Conditional assembly

Some sections of an assembly program may be ignored by conditional assembly controls presented in this section: They are IF, ELSEIF, ELSE and ENDIF.

Note:

Those controls may be used with or without a dollar sign ($). When prefixed by a dollar sign, they only access symbols defined with $SET or $RESET controls. When not prefixed by a dollar sign they first access symbols defined with SET and EQU directives and then those defined with $SET and $RESET controls.

For tests on variables, you can use the keyword **defined** after if (i.e.; $if defined (var) is equivalent to $if (var),

- The IF control begins a block which must be terminated by the ENDIF control. The block constructed with these 2 controls is assembled provided the expression following IF control is true, otherwise it is not assembled.

Syntax: **IF** expression
...
ENDIF

```
Example 1:
  $IF (version>200)
    lcall  INIT_SERIAL_PORT
            ; if version is greater than 200 the function
            ; INIT_SERIAL_PORT will be called
  $ENDIF
```

- IF control can also be used in blocks constructed with ELSE control.

Syntax: **IF** expression
block1
ELSE
block2
ENDIF

In this case, if the expression is true, only block1 is assembled, otherwise only block2 is assembled.

```
Example 2:
  $IF (version>200)
      lcall    INIT_SERIAL_PORT
            ; if version is greater than 200,
            ; function INIT_SERIAL_PORT will be called
  $ELSE
      lcall    INIT_PARALLEL_PORT
            ; function INIT_PARALLEL_PORT is called
            ; if version is less or equal to 200
  $ENDIF
```

- The IF control can be used in blocks constructed with ELSEIF control.

Syntax: **IF** expression1
block1
ELSIF expression2
block2
ENDIF

In this case, block1 will be assembled if expression1 is true, and block2 will be assembled only if expression1 is false and expression2 is true. It is therefore possible that neither block is assembled.

Note:
The ELSEIF control may be used any number of times

```
Example:
  $IF (bdt==1)
       lcall  lcall func1
              ; if bdt is equal to 1, func1 is called
  $ELSEIF (bdt==2)
       lcall func2
              ; if bdt is equal to 2, func2 is called
  $ELSEIF (bdt==3)
       lcall func3
              ; if bdt is equal to 3, func3 is called
  $ENDIF
```

- Finally, the sequence IF, ELSEIF, ELSE and ENDIF may be used in the same block.

Syntax: **IF** expression1
block1
ELSEIF expression2
block2
ELSE
block3
ENDIF

```
Example:
  $IF (bdt==1)
       lcall  lcall func1
  $ELSEIF (bdt==2)
       lcall func2
  $ELSEIF (bdt==3)
       lcall func3
  $ELSE
       lcall func4
              ; func4 is called only if bdt is not
              ; 1, 2 or 3
  $ENDIF
```

In the example, lcall func4 is assembled only if bdt is less than 1 or greater than 3.

4.4 Linking directives

4.4.1 Definition

In projects which use several files it is useful for symbols declared in one file to be available for use in other files. To allow for this facility PUBLIC and EXTRN directives may be used.

4.4.2 Symbols used in several modules.

The PUBLIC directive is used to specify which symbols declared in the current module may be used in other modules. The symbols following the PUBLIC directive must be declared in the current module though not necessarily by the time the directive is encountered, as forward references are allowed.

Syntax: **PUBLIC** symb [**,** symb ...]

Note:

1. registers and segment symbols can not be declared as public symbols.
2. PUBLIC directive cannot be used in the command line.

The EXTRN directive is used to identify symbols that are to be used in the current module but are declared in others. The segment type must be specified and must be one among the following: CODE, DATA, IDATA, XDATA, BIT, NUMBER.

Syntax: **EXTRN** seg_typ **(**symb [**,** symb ...]**)**

Note:

The EXTRN directive cannot be used in the command line.

```
Example 1:
             ; module1
$TITLE Module1
             ; module1 title definition
  PUBLIC Buffer
             ; Buffer is declared as public symbol and thus
             ; may be used in module in which it is declared
             ; as external
  Buffer DATA 40h

             ; module2
$TITLE Module2
  EXTRN DATA Buffer
             ; specifies that Buffer has been declared in an
             ; other module
```

The implementation of the PUBLIC and EXTRN symbols may become tedious for large applications and it is advisable to:

- Group all the external declarations in the same file, independently of the modules in which they are actually used.

```
Example 2:
            ;Declaration file EX_DCL.a
  EXTRN CODE (TOTO, MESUR1, MESUR2, RESULAT)
  EXTRN CODE ....
  EXTRN CODE ...
  EXTRN DATA (NUMBER, COUNTER)
  EXTRN XDATA TABLE, TASKS
```

- INCLUDE this declaration file at the beginning of each dependent.

```
Example 3:
            ; Module i
  $INCLUDE(EX_DCL.a)
  NAME MAIN
            ; specifies which name to use for the
            ; object file
            ; Module j
  $INCLUDE(EX_DCL.a)
  NAME SERIE
            ; Module k
  $INCLUDE(EX_DCL.a)
  NAME ACQUISITION
```

Conditions to be satisfied for correct linking are listed below:

1. A symbol must NOT be declared with the PUBLIC attribute in different modules.

2. Whenever a module refers to an external symbol (via a jmp or call instruction) the corresponding symbol must be declared with the PUBLIC attribute in another module.

3. A module cannot refer to more then 255 external symbols (defined with the PUBLIC attribute in other modules). This limitation is due to OMF-51.

4. If a symbol is declared as both EXTRN and PUBLIC, the EXTRN declaration will be ignored.

5. The assembler ignores EXTRN symbols that are not used in a module. It is therefore possible to define a 'declaration file' containing more than 255 EXTRN symbols, providing that none of the modules actually uses more than 255.

```
Example 4:
  EXTRN CODE (TRT_INT1)
            ; this declaration will be ignored
  CODE
  PUBLIC TRT_INT1
            ; the PUBLIC attribute will be used
```

4.5 Listing file

4.5.1 Definition

A file listing information concerning the assembly operation may be generated using the PRINT control. This listing file may include some information such as the unassembled parts of the source program, error messages, current date, the program source listing.. etc. The selection of the included information, is controlled by listing controls presented in this section: TITLE, PAGEWIDTH, PAGELENGTH, COND, NOCOND, DATE, EJECT, ERRORPRINT, NOERRORPRINT, GEN, NOGEN, LIST, NOLIST, SYMBOLS, NOSYMBOLS, XREF, NOXREF.

4.5.2 Listing file generation

A listing file, which contains details of the assembly process, may be generated with the PRINT control. The string following the PRINT control may specify the listing file name. If no file name is specified, the listing file name will be the source file name with the *.lst extension.

Syntax: **[$]PRINT (**file_name.lst**)**
[$]PRINT

Abbreviation: **[$]PR**

```
Example 1:
  MA51 raison.a PRINT
```

```
Example 2:
  MA51 raison.a PRINT(listing.lst)
```

```
Example 3:
  $PRINT
```

The NOPRINT control prevents the assembler from generating a listing file.

Syntax: **[$]NOPRINT**

Abbreviation: **[$]NOPR**

```
Example 4:
  $NOPR
```

```
Example 5:
  MA51 raison.a NOPRINT
```

Note:

$PRINT (or $PR, $NOPRINT, $NOPR) is used when specified in the assembly file whereas PRINT (or PR, NOPRINT, NOPR) is used in the command line.

4.5.3 Listing file title

You may include a title in the listing file by employing the TITLE control. The title is specified by the string immediately following this control.

Syntax: **TITLE(**raisonance**)**

Abbreviation: **TT**

```
Example:
  $TITLE(Raisonance benchmark)
```

Note:
TITLE cannot be used in the command line.

4.5.4 Number of characters per line in a listing file

The maximum number of characters per line in a listing file line may be controlled with the PAGEWIDTH control. Lines that are longer than the specified value automatically wrap around the next line. The default value is 120.

Syntax: **[$]PAGEWIDTH(**number**)**

Abbreviation: **[$]PW**

```
Example 1:
  MA51 raison.a PW(50)
```

```
Example 2:
  $PAGEWIDTH(130)
```

Note:
$PAGEWIDTH (or $PW) is used when specified in the assembly file whereas PAGEWIDTH (or PW) is specified in the command line.

4.5.5 Number of line in a listing file page

The number of lines per page printed in the listing file may be controlled with the PAGELENGTH control. This number must be chosen between 10 and 65535. The default value is 60.

Syntax: **[$]PAGELENGTH(**number**)**

Abbreviation: **[$]PL**

```
Example 1:
  MA51 raison.a PL(120)
```

```
Example 2:
  $PAGELENGTH(20)
```

PAGELENGTH(0) is also accepted and means that the listing file is to contain no page breaks. This may be useful when exporting the listing file to other software.

Note:
$PAGELENGTH (or $PL) is used when specified in the assembly file whereas PAGELENGTH (or PL) is specified in the command line.

4.5.6 Unassembled parts of a conditional block

Sometimes it is useful to include the unassembled parts of a conditional IF, ELSIF, ELSE, ENDIF block in the listing file. This is possible by specifying the control COND either on the command line, or at the top of the source file.

Syntax: **[$]COND**

```
Example 1:
  MA51 raison.a COND
```

```
Example 2:
  $COND
```

NOCOND prevents unassembled portions of a conditional IF, ELSIF, ELSE, ENDIF block in the source file from appearing in the listing file.

Syntax: **[$]NOCOND**

```
Example 1:
  MA51 raison.a NOCOND
```

```
Example 2:
  $NOCOND
```

Note:
$COND (or $NOCOND) is used when specified in the assembly file whereas COND (or NOCOND) is specified in the command line.

4.5.7 Date

The DATE control allows you to specify the current date in the header of each page of the listing file.

Syntax: **[$]DATE(**dd/mm/yy**)**

***Abbreviation**:* **[$]DA**

```
Example 1:
  MA51 raison.a DATE(25/12/95)
```

```
Example 2:
  $DATE(25/12/95)
```

Note:

$DATE (or $DA) is used when specified in the assembly file whereas DATE (or DA) is specified in the command line.

4.5.8 Error messages

Error messages may be visualized by employing the ERRORPRINT control. Error messages will either be output to the console or written in a specified file depending on the presence of a file name after the ERRORPRINT control. If no file name is specified, all messages errors are output to the console.

Syntax: **[$]ERRORPRINT**
[$]ERRORPRINT(file_name**)**

Abbreviation: **[$]EP**

```
Example 1:
  MA51 raison.err  ERRORPRINT(no_error.err)
```

```
Example 2:
  MA51 raison.err  EP
```

```
Example 3:
  $ERRORPRINT(no_error.err)
```

No error file is generated if NOERRORPRINT is used.

Syntax: **[$]NOERRORPRINT**

Abbreviation: **[$]NOEP**

```
Example 4:
  MA51 raison.err  NOEP
```

Note:
$ERRORPRINT (or $EP, $NOERRORPRINT, $NOEP) is used when specified in the assembly file whereas ERRORPRINT (or EP, NOERRORPRINT, NOEP) is specified in the command line.

4.5.9 Macro assembly instruction

When the GEN control is specified the assembly instruction within a macro definition will appear in the listing file.

Syntax: **[$]GEN**

```
Example 1:
  MA51 raison.a GEN
```

```
Example 2:
  $GEN
```

NOGEN prevents the Raisonance MA-51 assembler from listing macro text in the listing file.

Syntax: **[$]NOGEN**

```
Example 3:
  MA51 raison.a NOGEN
```

Note:

$GEN (or $NOGEN) is used when specified in the assembly file whereas GEN (or NOGEN) is specified in the command line.

The MACRO control allows the assembler MA-51 to recognize and process Intel ASM51 compatible macro definitions and invocations.

Syntax: **[$]MACRO** (MPL)

```
Example:
  MA51 raison.a MACRO (MPL)
```

The NOMACRO control prevents the recognition and process of Intel ASM51 compatible macro definitions and invocations.

Syntax: **[$]NOMACRO**

```
Example:
  MA51 raison.a NOMACRO
```

4.5.10 Source file inclusion

Sometimes it is useful to see lines from the source file in the listing file and this is possible with the LIST control.

Syntax:	**[$]LIST**
Abbreviation:	**[$]LI**

```
Example 1:
  MA51 raison.a LIST
```

```
Example 2:
  $LI
```

The NOLIST control prevents the source program from appearing in the listing file unless an error occurs at that line.

Syntax:	**[$]NOLIST**
Abbreviation:	**[$]NOLI**

Note:

$LIST (or $LI, $NOLIST, $NOLI) is used when specified in the assembly file whereas LIST (or LI, NOLIST, NOLI) is specified in the command line.

4.5.11 Form feed

The EJECT control may be used to insert a form feed into the listing file after the line containing this control.

Syntax:	**EJECT**
Abbreviation:	**EJ**

Note:
1. This control is ignored if NOLIST or NOPRINT control has been previously specified.
2. This control cannot be specified in the command line.

```
Example:
  $EJECT
```

4.5.12 Symbol table

The SYMBOL control allows you to make the assembler write all symbols used in and by the assembly program to the listing file.

Syntax: **[$]SYMBOLS**

Abbreviation: **[$]SB**

```
Example 1:
  MA51 raison.a SYMBOLS
```

```
Example 2:
  $SB
```

To prevent the assembler from including this symbol table in the listing file, you must use the NOSYMBOLS control.

Syntax: **[$]NOSYMBOLS**

Abbreviation: **[$]NOSB**

```
Example 3:
  MA51 raison.a NOSB
```

Note:
$SYMBOLS (or $SB, $NOSYMBOLS, $NOSB) is used when specified in the assembly file whereas SYMBOLS (or SB, NOSYMBOLS, NOSB) is specified in the command line.

4.5.13 Include files

The LISTINCLUDE control is used to add the content of the include files to the listing file. By default the listing file does not incorporate these files, corresponding to the NOLISTINCLUDE control

Syntax: **LISTINCLUDE**
NOLISTINCLUDE

Abbreviation: **LC**
NOLC

```
Example 1:
  MA51 raison.c LISTINCLUDE
```

```
Example 2:
  $LISTINCLUDE
```

4.5.14 Cross reference table

You can specify that you want a cross-reference table of all symbols used in the source module to be written in the listing file by employing the XREF control.

Syntax: **[$]XREF**

Abbreviation: **[$]XR**

```
Example 1:
  MA51 raison.a XREF
```

```
Example 2:
  $XREF
```

The NOXREF control prevents the assembler from generating this cross-reference table.

Syntax: **[$]NOXREF**

Abbreviation: **[$]NOXR**

```
Example3:
  MA51 raison.a NOXR
```

Note:
$XREF (or $XR, $NOXREF, $NOXR) is used when specified in the assembly file whereas XREF (or XR, NOXREF, NOXR) is specified in the command line.

4.5.15 Settings

The SAVE control stores the current settings of the LIST and GEN controls. The RESTORE control restores previously saved settings.

Syntax: **[$]SAVE**
[$]RESTORE

Abbreviation: **[$]SA**
[$]RE

4.6 Macro processor

The macro syntax developed in this chapter is the macro syntax used by default by Raisonance MA version 6.0. However the macro syntax used in former versions of the Raisonance EMA-51 assembler are still supported, such as MPL and ASM51.

4.6.1 Definition

A macro is an amalgamation of one or more instruction lines and is declared using the MACRO directive. The macro can then be used several times within the file by using the macro name, which is equivalent to a mnemonic instruction. At assembly, the corresponding lines of code between the keywords MACRO and ENDM, are substituted, each time the name of the macro is encountered. The corresponding instructions are then assembled, as if they had been encountered directly.

Syntax: macro_name **MACRO** par1**,**par2**,**..park
{instruction lines}
ENDM

Example1:

```
  SAVE MACRO
             ; declaration of a macro whose name is SAVE
        PUSH    ACC
        PUSH    B
        PUSH    PSW
        PUSH    DPH
        PUSH    DPL
  ENDM
             ; end of macro declaration
  restore MACRO
        POP     DPL
        POP     DPH
        POP     PSW
        POP     B
        POP     ACC
  ENDM
```

```
Example2:
  Exchange   MACRO reg1 reg2
        PUSH   reg1
        PUSH   reg2
        POP    reg1
        POP    reg2
  ENDM
```

Note:

The former macro declaration syntax in which the macro name is specified after MACRO statement is still supported. Example2 may therefore be written as in example3.

```
Example2:
    MACRO Exchange reg1 reg2
        PUSH   reg1
        PUSH   reg2
        POP    reg1
        POP    reg2
    ENDM
```

4.6.2 Recursion

Recursion is supported for the Macro, up to 32 times for the same Macro.

4.6.3 Repetition of blocks

A block contained in a macro may be repeated a given number of times by using the macro repetition directives: REPT, WHILE, IRP and IRPC. Repeated blocks may be nested using the LOCAL directive.

4.6.3.1 Sequential block repetition

4.6.3.1.1 Repetition of blocks controlled by a number

The REPT directive is used to repeat a block of text. The number of repetitions must be specified after the REPT statement and ENDM directive specifies the end of the block to repeat.

Syntax: **REPT** number_of_repetitions
...
ENDM

```
Example:
  delay    MACRO
           REPT 3
           NOP
           ENDM
             ; end of repeated block
  ENDM
             ; end of the macro
```

```
; When invoked the delay macro will generate the following
; code:
           NOP
           NOP
           NOP
```

4.6.3.1.2 Repetition of a block of text controlled by an argument list

The IRP directive is used to repeat a block of text. The number of repetitions is not explicitly specified but is implicitly calculated depending on the number of arguments specified in the list following the IRP statement. This directive must be used with ENDM directive, which specifies the end of the block to repeat.

Syntax: **IRP** parameter**,**<arg [**,** arg ...]>
...
ENDM

```
Example:
  init_tab   MACRO port
        IRP    table_element, <55h,0aah,55h>
        MOV    port,#table_element
        ENDM
  ENDM

; When invoked as init_tab P1 the macro will generate:
; code:
      MOV P1, #55H
      MOV P1, #0AAH
      MOV P1, #55H
```

4.6.3.1.3 Repetition of a block of text controlled by a set of characters

The IRPC directive is used to repeat a block of text. The repetition number is not explicitly specified by a number but is implicitly calculated depending on the number of characters in the argument. This directive must be used with the ENDM directive, which specifies the end of the block to repeat.

Syntax: **IRPC** parameter**,** <string>
...
ENDM

```
Example:
  stack_filling MACRO
        IRPC   param, <Raisonance>
        MOV      A,#‘param’
        PUSH     A
        ENDM
  ENDM
```

```
; When invoked the stack_filling macro will fill the stack as
; follows:
        MOV      A,#‘R’
        PUSH     A
        MOV      A,#‘a’
        PUSH     A
        ...
```

4.6.3.2 Nested block repetition

The previous macro repetition directives (REPT, WHILE, IRP and IRPC) may be nested up to 9 levels deep.

```
Example 1:
  fast_stack_filling MACRO
        REPT 8
              IRPC   param, <initvalue>
              MOV      A,#‘param’
              PUSH     A
              ENDM
                ; end of IRPC block
        ENDM
                ; end of REPT block
  ENDM
                ; end of fast_stack_filling macro
```

Using the LOCAL directive, you can declare up to 16 local symbols (labels, data declarations...) to be used inside the macro.

Syntax: macro_name **MACRO** par1**,**par2**,**..par j
LOCAL symb [,symb ...]
{instruction lines}
ENDM

```
Example 2:
  slow_stack_filling MACRO
        LOCAL  delay
        delay MACRO
              REPT 3
              NOP
              ENDM
                    ; end of repeated block
        ENDM
                    ; end of delay macro
        IRPC   param, <initvalue>
              MOV      A,#‘param’
              PUSH     A
              delay
        ENDM
                    ; end of IRPC block
  ENDM
                    ; end of slow_stack_filling macro
```

4.6.4 Macro operators

Some operators may be used in macros.

4.6.4.1 Determining if a macro argument is null

The NUL operator is used to determine if its argument is null. Its cannot be replaced by ‘ ‘, which is not equivalent.

Syntax: **NUL** argument

```
Example:
  raison MACRO argu
          IF NUL argu
          ...
                      ; this block will be treated provided
                      ; argu is NUL
          ENDIF
          ...
  ENDM
```

4.6.4.2 Concatenation of text and macro parameters

‘&’ is used as an operator to concatenate text and macro parameters.

```
Example:
  routine_title MACRO title
          title&1: NOP
  ENDM
```

```
; When invoked by routine_title RAISON the routine_title macro ;
will generate the following code:
  Raison1: NOP
```

4.6.4.3 Keeping the literal text of an expression

The angle bracket operators (< and >) are used to enclose text that should remain the same until macro code generation

```
Example1:
  init_R1_R2 MACRO argu
        IRP table_name,<argu>
              MOV     R1,#table_name&1
              MOV     R2,#table_name&2
        ENDM
  ENDM
```

```
; When invoked by « init_R1_R2 10 » the macro
; will generate the following code:
              MOV     R1,#101
              MOV     R2,#102
```

The exclamation mark operator may be used to literally pass characters such as comma.

4.6.4.4 Evaluating of an expression

The percent character operator (%) is used to evaluate an expression and thus to pass its value rather than the literal expression.

```
Example:
  init_Rn MACRO argu
        IRPC register_number,<0123>
              MOVE R&register_number,#argu
        ENDM
  ENDM
  ten      EQU 0ah
        init_Rn  %ten
```

```
; When invoked the init_Rn macro will generate the following
; code:
            MOVE R0,#0ah
            MOVE R1,#0ah
            MOVE R2,#0ah
            MOVE R3,#0ah
```

4.6.4.5 Macro local comments

The double semicolon operator (;;) is used to comment macros without generating the text when the macro's code is expanded.

```
Example:
  registers01_init MACRO
          MOVE R0,#00h  ;; R0 initialization
          MOVE R1,#00h  ;; R1 initialization
  ENDM
```

5 Appendices

5.1 Example of program

This section presents a program first written in *Simplified Syntax* and also in *Classical Syntax* for MA-51.

5.1.1 Example of program

5.1.1.1 Simplified Syntax version

```
;=======================================
;Test program for demonstration of MA-51
; (Simplified Syntax )
;=======================================
;Object: To output a warning message on the serial line
;whenever a signal is detected at port P3.3 (INT1).
;---------------------------------------------------------------
name DEMO

$include   (REG_51.PDF)
             ;Declaration of 8051 SFRs.

;---------------------------------------------------------------

;Declaration of a macro-instruction to transmit message at serial
port.
SEND_MESSAGE   MACRO  MESSAGE
LOCAL  STRING:
LOCAL  OUT:
      mov      DPTR, #STRING
      clr      MES_SEG
             ;The message is to be read in CODE
      lcall    PUTS
      sjmp     OUT
      DB MESSAGE,0dh,0ah,0  ;Content of the message
ENDM

BIT
      MES_SEG: DB 1
             ;if 1, the message is to be read in external RAM
           ;and if 0, in CODE.

CODE         ;Relocatable segment
  PUTS:
             ;procedure for string writing.
  NEXT:
      jb     MES_SEG, PUTX
      clr    A
      movc   A, @A+DPTR ;message read in CODE
      sjmp   TESTC
  PUTX:
      movx   A,@DPTR
             ;message read in XDATA
  TESTC:
      jz     END_PUTS
      mov    C,P
             ;Determine parity
      mov    ACC.7,C
```

```
             ;Mode 7 bit + even Parity.
      jnb    TI,$
      clr    TI
      mov    SBUF,A
      inc    DPTR
             ;next character
      sjmp   NEXT
  END_PUTS:
      ret

;----------------------------------------------------------------

;Main Program
CODE AT 0
             ;Absolute segment : RESET vector
      ljmp   START
CODE         ;Relocatable
  START:
      mov    SP, #40h
      lcall  INIT_HARD
      mov    COUNTER,#0
             ;WARNING Counter
      SEND_MESSAGE 'INIT''  ;use of the MACRO SEND_MESSAGE
      setb   EA
      sjmp   $
             ;infinite loop

;----------------------------------------------------------------

  INIT_HARD:
             ;general initialization

;Initialization of serial port, with timer1 as baud generator
      mov    TMOD, #20h
             ;Timer 1 in mode 2
      mov    TCON, #44H
      mov    TH1,  #0E6H
             ;1200 bauds per 12 MHz
      mov    PCON, #0
             ;SMOD at 0 (pre-divisor)
      mov    SCON, #52H
             ;Serial Port : mode 1

;Then the interrupts :
      setb   EX1
             ;Enable INT1
      ret
             ;Push out 'setb EA''

;----------------------------------------------------------------

;Interrupt vector of outside interrupt 1.
CODE AT 13H
             ;Absolute
      ljmp   TRT_INT1

DATA
  COUNTER:   db  1
             ;Number of warnings already transmitted
```

```
XDATA
  XMES:       db   3
              ;Space for receiving number of the
              ;warning in ASCII.

CODE          ;Relocatable.
  TRT_INT1:
              ;Processing interrupt 1.
      push    PSW
      push    ACC
      push    DPH
      push    DPL
  SEND_MESSAGE 'WARNING AT P3.3 :'"
              ;output of warning number
              ;in ASCII (between 01 and 99)
      inc     COUNTER
              ;Incrementation of number
      mov     A,COUNTER
      cjne    A,#99,DIVISION
              ;Limited to 99
      mov     COUNTER, #0
  DIVISION:
      mov     B,#10
      div     AB
              ;A = tens, B = units.
      add     A,#30H
              ;Put into ASCII form (string in XDATA)
      mov     DPTR,  #XMES
      movx    @DPTR,A
      inc     DPTR
      mov     A,B
      add     A,#30H
              ;Units in ASCII
      movx    @DPTR,A
      inc     DPTR
      clr     A
      movx    @DPTR,A
              ;End of string.
      setb    MES_SEG
              ;the message will be read in XDATA
      mov     DPTR,#XMES
      lcall   PUTS

  SEND_MESSAGE '""
              ;Skip a line

      pop     DPL
      pop     DPH
      pop     ACC
      pop     PSW
      reti
```

5.1.1.2 8051 version in *Classical Syntax*

```
;==================================================
;Test program for demonstration of Raisonance MA-51
; (Classical Syntax)
;==================================================
;Object: To output a warning message on the serial line
;whenever a signal is detected at port P3.3 (INT1).
;----------------------------------------------------------

name DEMO

$include   (REG_51.INC)
                   ;Declaration of 8051 SFRs.

;---------------------------------------------------------------
;Declaration of a macro-instruction to transmit message at ;serial
port.

SEND_MESSAGE      MACRO   MESSAGE
LOCAL  STRING:
LOCAL  OUT:
         mov      DPTR, #STRING
         clr      MES_SEG
                   ;The message is to be read in CODE
         lcall    PUTS
         sjmp     OUT
         DB       MESSAGE,0dh,0ah,0
                   ;Content of the message
ENDM
                   ;end of the macro

mybitseg   SEGMENT  BIT
                   ;relocatable bit segment declaration
RSEG       mybitseg
                   ;relocatable segment selection
  MES_SEG:   DBIT   1
                   ;if 1, the message is to be read in
                 ;external RAM and if 0, in CODE.

mycodseg   SEGMENT CODE
                   ;relocatable code segment declaration
RSEG       mycodseg
                   ;Relocatable code segment selection
  PUTS:
                   ;procedure for string writing.
  NEXT:
         jb     MES_SEG, PUTX
         clr    A
         movc   A, @A+DPTR
                 ;message read in CODE
         sjmp   TESTC

  PUTX:
         movx A,@DPTR
                 ;message read in XDATA

  TESTC:
         jz     END_PUTS
         mov    C,P
                  ;Determine parity
         mov    ACC.7,C
                  ;Mode 7 bit + even Parity.
         jnb    TI,$
         clr    TI
         mov    SBUF,A
         inc    DPTR
                  ;next character
         sjmp   NEXT
   END_PUTS:
         ret
;----------------------------------------------------------------
;Main Program
CSEG   AT   0
                  ;Absolute code segment selection
                  ; -> RESET vector
         ljmp START

RSEG   mycodseg
                  ;Relocatable code segment selection

START:
         mov    SP, #40h
         lcall  INIT_HARD
         mov    COUNTER, #0
                  ;WARNING Counter
         SEND_MESSAGE 'INIT''
                  ;use of the MACRO SEND_MESSAGE
         setb   EA
         sjmp   $
                  ;infinite loop

;----------------------------------------------------------------
INIT_HARD:
                  ;general initialization
;Initialization of serial port, with timer1 as baud generator
         mov    TMOD,#20h
                  ;Timer 1 in mode 2
         mov    TCON,#44H
         mov    TH1,#0E6H
                  ;1200 bauds per 12 MHz
         mov    PCON,#0
                  ;SMOD at 0 (pre-divisor)
         mov    SCON,#52H
                  ;Serial Port : mode 1

;Then the interrupts :
         setb   EX1
                  ;Enable INT1
         ret
                  ;Push out 'setb EA''

;----------------------------------------------------------------
;Interrupt vector of outside interrupt 1.
CSEG AT 13H
                    ;Absolute code segment selection
         ljmp   TRT_INT1
```

```
mydatseg   SEGMENT  DATA
                    ;relocatable data segment declaration
RSEG       mydatseg
                    ;relocatable data segment selection

  COUNTER: ds  1
                    ;Number of warnings already transmitted

myxdatseg  SEGMENT  XDATA
                    ;relocatable XDATA segment declaration
RSEG       myxdatseg
                    ;relocatable XDATA segment selection

XMES:    ds  3
                    ;Space for receiving number of the
                    ;warning in ASCII.

RSEG       mycodseg
                    ;Relocatable code segment selection
         TRT_INT1:
                    ;Processing interrupt 1.
         push  PSW
         push  ACC
         push  DPH
         push  DPL
         SEND_MESSAGE 'WARNING AT P3.3 :'"
                    ;output of warning number
                    ;in ASCII (between 01 and 99)
         inc   COUNTER
                    ;Incrementation of number
         mov   A,COUNTER
         cjne  A,#99,DIVISION
                    ;Limited to 99
         mov   COUNTER,#0
DIVISION:
         mov   B,#10
         div   AB
                    ;A = tens, B = units.
         add   A,#30H
                    ;Put into ASCII form (string in XDATA)
         mov   DPTR,#XMES
         movx  @DPTR,A
         inc   DPTR
         mov   A,B
         add   A,#30H
                    ;Units in ASCII
         movx@DPTR,A
         inc   DPTR
         clr   A
         movx  @DPTR,A
                    ;End of string.
         setb  MES_SEG
                    ;the message will be read in XDATA
         mov   DPTR,#XMES
         lcall PUTS

         SEND_MESSAGE '""
                    ;Skip a line
```

```
        pop    DPL
        pop    DPH
        pop    ACC
        pop    PSW
        reti
```

5.2 Keywords

Keywords are symbols that cannot be redefined by the programmer.

A	AB	ACALL	ADD	ADDC
AND	ANL	AR0	AR1	AR2
AR3	AR4	AR5	AR6	AR7
ASEG				
BDATA	BIT	BITADDREASSABLE	BSEG	
C	CASESENSITIVE	CJNE	CLR	CODE
CPL	COND	CS	CSEG	
DA	DATA	DATE	DB	DBIT
DEC	DEBUG	DEFINED	DIV	DIRECT
DJNZ	DNF	DONOTFACTORIZE	DPL	DPH
DPTR	DS	DSEG	DW	
EJ	EJECT	ELSE	ELSEIF	END
ENDIF	ENDM	ENDP	EP	EQ
EQU	ER	ERROR	ERRORPRINT	EXTERN
EXTRN				
FLIP8051				
GB	GEN	GLOBAL	GT	GTE
HIGH				
IC	IDATA	IF	INBLOCK	INCLUDE
INC	INPAGE	IRP	IRPC	ISEG
JB	JBC	JC	JMP	JNB
JNC	JNZ	JZ		
LA	LC	LCALL	LI	LINES
LIST	LISTINCLUDE	LJMP	LN	LOCAL
LOCKEDAREA	LT	LTE		
MACRO	MACROINST	MACROINSTRUCTION	MO	MOD51
MOV	MOVC	MOVX	MUL	MULTITARGET
NAME	NB	NE	NOCOND	NOCS
NOCASESENSITIVE	NODB	NODEBUG	NOEP	NOERRORPRINT
NOGEN	NOLC	NOLI	NOLINES	NOLIST
NOLISTINCLUDE	NOLN	NOMACRO	NOMO	NOMOD51
NOOJ	NOOBJECT	NOP	NOPATH	NOPR
NOPRINT	NOREGISTERBANK	NORB	NOSB	NOSYMBOLS
NOT	NOXR	NOXREF	NUL	NUMBER
OBJECT	OBJECTEXTEND	OE	OJ	OR
ORG	ORL	OVERLAYABLE		

PAGE	PAGELENGTH	PAGEWIDTH	PATHINCLUDE	PC
PIN	PL	POP	POSTOPT	POSTOPTIMZE
PR	PRINT	PUBLIC	PUSH	PW
QUIET				
R0	R1	R2	R3	R4
R5	R6	R7	RB	RBIT
REG	REGISTERBANK	REL	REPT	RESET
RESTORE	RET	RETI	RL	RLC
RR	RRC	RS	RSEG	
SA	SAVE	SB	SEGMENT	SET
SETB	SETS	SHL	SHR	SJMP
STANDARD	SUBB	SWAP	SYMBOLS	
TITLE	TT			
UNIT	USING			
WHILE	WORD			
XCH	XCHD	XDATA	XOR	XR
XREF	XRL	XSEG		

5.3 Error Messages

5.3.1 Error types

A warning is generated if any abnormalities are detected in the program that may have been caused by a programming error. A warning indicates source code that is allowed by the standard but either with a style that does not conform to good programming rules or with a highly probable programming error.

An error does not usually abort the assembling process, but suspends code generation.

A fatal error causes immediate termination of the compilation. The most likely cause is generally a system problem such as a full disk or memory space.

The ERROR control allows to generate an error. This is useful to generate an error in specific conditions and to test a program.

Syntax: **[$]ERROR**

Abbreviation: **[$]ER**

5.3.2 List of errors

Error 1: ILLEGAL CHARACTER IN NUMERIC CONSTANT
Numeric constants must begin with a decimal digit. Dollar signs are allowed in constants to improve readability. This error occurs if the dollar sign is the last character in a numeric constant.

Error 2: INVALID CHARACTER IN NUMERIC CONSTANT
Indicates that an invalid character was found in a numeric constant.

Error 3: MISSING STRING TERMINATOR
The ending string terminator is missing.

Error 4: BAD (OR MISUSED) SPECIAL CHARACTER
A special character as %, &, and ! is not used correctly.

Error 5: BAD INDIRECT REGISTER
Combined registers are entered incorrectly, e.g. @R2, @R3, @DPTR, @A+PC, @A+DPTR

Error 6: ‘symb’ ILLEGAL USE OF A RESERVED WORD
A label has the same name as a reserved keyword (see the MA-51 user manual).

Error 8: LABEL NOT PERMITTED
The use of this label is not permitted in this context.

Error 10: SYNTAX ERROR
The MA-51 assembler encountered an error of syntax when processing the specified line.

Error 11: ATTEMPT TO REDEFINE SYMBOL 'symb’

The symbol “symb” has been defined more than once. You must keep only one definition of the symbol “symb”.

Error 14: ')' EXPECTED

A closing parenthesis is expected. This error usually occurs for the definition of external symbols:
Example: EXTRN DATA (i,j,buf
The ‘)’ is missing at the end of the expression.

Error 15: BAD RELOCATABLE EXPRESSION

This error is generated, for example, when operands of an instruction are not of one of the following types:

1. constant_numeric_value
2. relocatable_symbolic_value
3. relocatable_ symbolic_value + constant_numeric_value
4. relocatable_symbolic_value - relocatable_symbolic_value

NOTES:
1. In case 4, both symbols must be defined in the file and in the same segment. In that case, the difference is equivalent to a constant numeric value (i.e. "gap" between these two symbols in the segment).
2. The symbolic operators (HIGH-LOW) are allowed on the symbolic values.
3. A constant numeric value can be made of every expression established by combination of the operators described in the manual.
Then, 2*3+4 is also a constant_numeric_value because the expression will be evaluated during assembling.
However, replacement of "2" and "3" by relocatable symbols will produce an error:
extrn code (lab1,lab2)
toto set lab1 + Lab2 ; << error

Error 17: DIVIDE BY ZERO ERROR

Division by zero is forbidden. A division by zero is attempted while calculating an expression. The value calculated is undefined.

Error 18: INVALID BYTE BASE IN BIT ADDRESS EXPRESSION

In the expression EXP.BITOFS, EXP is not a correct symbolic expression
Example:
setb DPTR.1 ;error..

Error 19: OUT OF RANGE OR NON-TYPELESS BIT-OFFSET

In the expression EXP.BITOFS, BITOFS is not a numerical constant included in the interval [0,7].
Example:
setb P1.8 ;error

Error 21: INVALID SIMPLE RELOCATABLE EXPRESSION

Example:
DEMOSEG segment BIT
rseg DEMOSEG
LABS: DBIT 1
CSEG at 0
org LABS ; a simple expression is required here!!

Error 23: EXPRESSION TYPE DOES NOT MATCH INSTRUCTION
The expression is not complete. A #, /Bit, register or numeric expression was expected.
Example: DIV A
Division needs two elements

Error 24: NUMERIC EXPRESSION EXPECTED
A numeric expression is expected but an expression of another type is found.

Error 25: SEGMENT-TYPE EXPECTED
demoxdataseg SEGMENT XDATA
demoxxx SEGMENT XXX ; XXX is not a valid segment type.
Segment type is among CODE, DATA, XDATA, BIT

Error 26: RELOCATION-TYPE EXPECTED
Example:
democodeseg SEGMENT CODE OUTPAGE
The relocation type is among INPAGE, PAGE, INBLOCK but OUTPAGE is not a valid relocation page.

Error 27: BAD RELOCATION-TYPE
Example:
democodeseg SEGMENT CODE BITADDRESSABLE
The relocation type is not allowed for the segment type (here, ibtaddressable can be used only for DATA segments).

Error 29: ABSOLUTE EXPRESSION REQUIRED
The operand of an RSEG directive must be a segment symbol. The operand of the RSEG directive has not been defined or not as a segment symbol.
Check that this operand is defined as a segment symbol.

Error 30: SEGMENT-LIMIT EXCEEDED
dataseg segment DATA
rseg DATASEG
org 400H ; DATA segment is up to 256 bytes

Error 31: SEGMENT-SYMBOL EXPECTED
The operand of an RSEG directive must be a segment symbol
The following example is valid:
fct3CD SEGMENT CODE
RSG fct3CD

Error 32: PUBLIC-ATTRIBUTE NOT PERMITTED
The PUBLIC attribute is not permitted for the specified symbol

Error 33: ATTEMPT TO RESPECIFY MODULE NAME
The NAME directive is used more than once. The NAME directive allows which name to use for the object module generated for the current program. So, only one name can be defined.

Error 35: ',' EXPECTED
A coma ‘,’ is expected in a list of expressions or symbols.

Error 36: (' EXPECTED

A left parenthesis is expected at the specified line.

Example: EXTRN DATA i,j,buf) \\ INVALID
EXTRN DATA (i,j,buf) \\ VALID

Error 38: OPERATION INVALID IN THIS SEGMENT

The operation is valid only in a CODE segment

The following example is not valid:
fct3CD SEGMENT **DATA**
RSEG fct3CD
It should be
fct3CD SEGMENT **CODE**
RSEG fct3CD

Error 39: NUMBER OF OPERANDS DOES NOT MATCH INSTRUCTION

NOP A ; NOP does not take any operand
JC A,$; JC takes only one operand

Error 43: 'LABEL' PHASE ERROR (PASS-2)

This occurs if a symbol in PASS 2 contains a value different that in PASS 1.

Error 45: MISPLACED PRIMARY CONTROL, LINE IGNORED

The primary control is misplaced. Primary controls may be entered in the invocation line or at the beginning of the source file (as $ instruction). If no primary control is define in the first line, the other defined primary controls are ignored.

Error 46: UNDEFINED SYMBOL (PASS-2) 'symb’

The symbol that is used here is not defined. You must define it before using it.

Error 47: CODE-ADDRESS EXPECTED

A CODE or typeless expression is expected

Error 48: XDATA-ADDRESS EXPECTED

An XDATA or typeless expression is expected

Error 49: DATA-ADDRESS EXPECTED

A DATA or typeless expression is expected

Error 50: IDATA-ADDRESS EXPECTED

An IDATA or typeless expression is expected

Error 51: BIT-ADDRESS EXPECTED

A BIT or typeless expression is expected

Error 52: TARGET OUT OF RANGE

org 100h
sjmp label ; label is too far for a sjmp/ Use a LJMP
org 200h
label: nop

Error 53: VALUE HAS BEEN TRUNCATED TO 8 BITS
MOV A,#1234H

Error 56: BAD CONDITIONAL EXPRESSION
An IF or ELSEIF control is invalid. An ELSEIF may be defined without IF before.
These expressions must be absolute and may not contain relocatable symbols.

Error 57: UNBALANCED IF-ENDIF-CONTROLS
Each IF block has to be terminated with an ENDIF control. There should be the same number of IF and ENDIF controls.

Error 58: SAVE STACK UNDERFLOW
You can restore only what has been saved before. A $RESTORE control instruction is valid only if a $SAVE control was previously executed.

Error 59: SAVE STACK OVERFLOW
The context of the GEN, COND and LIST controls may be stored by the $SAVE control a maximum of 9 times.

Fatal 63: NON-NULL ARGUMENT EXPECTED
The name of the source file to assemble or a file name in a command line option was omitted

Fatal 64: CONFLICTING CONTROL
An option on the command line conflicts with another option.
Example:
MA51.EXE EXAMPLE.A51 **NOOBECT OBJECT**(EXAMPLE.OBJ)

Fatal 65: CANNOT HAVE GENERAL CONTROL IN INVOCATION-LINE
Some controls can be used only in the source file but cannot be used in the command line.
See the specified control to have more information on its use;

Fatal 66: ARGUMENT TOO LONG
An Argument contains too many characters.

Fatal 69: UNDEFINED CONTROL
The specified control in the command line is not recognized by the MA-51 Assembler

Fatal 71: OUT OF RANGE NUMERIC VALUE
A numeric argument is out of range. This occurs for example, when the PAGELENGTH option has an argument that is smaller than 10 or greater than 65535

Fatal 72: SYMBOL-TABLE SPACE EXHAUSTED
There is no more room for symbol information in the symbol table. The symbol table can accommodate approximately 40Kbytes of symbols.

Error 73: TOO MANY WARNINGS
Too many warnings have been found when translating the file.

Error 75: 'DB/DW' ALLOWED ONLY IN CODE SEGMENT
The use of DB and DW directives are allowed only in segment that have been defined as CODE type.
Check that you have not tried to use them in another space segment.

Error 76: 'DS' NOT ALLOWED IN BIT SEGMENT

The DS directive is used to reserve space in the internal data space or external data space. So, the DS directive cannot be used in BIT segment.

```
NOT VALID
demobitseg      SEGMENT     BIT
RSEG demobitseg
COUNTER:        DS      1

VALID
demodataseg     SEGMENT     DATA
RSEG demodataseg
COUNTER:        DS      1
```

Error 77: 'DBIT' ALLOWED ONLY IN BIT SEGMENT

The use of DBIT is allowed only in segment that have been defined as BIT type.
Example:

```
demobitseg              SEGMENT     CODE  //NOT VALID
demobitseg              SEGMENT     BIT     //VALID

RSEG        demobitseg
MES_SEG:        DBIT    1
```

Error 79: UNABLE TO CREATE LISTING FILE '%s'

'lstfilename' cannot be opened: the disk is full, or a protected (read-only) file already exists.

Fatal 80: UNABLE TO CREATE OBJECT FILE '%s'

'objfilename' cannot be opened: the disk is full, or a protected (read-only) file already exists.

Fatal 81: UNABLE TO CREATE TEMPORARY FILE '%s'

'tmpfilename' cannot be opened: the disk is full, or a protected (read-only) file already exists

Warning 82: BAD PARAM NUMBER IN MACRO INVOCATION

The macro is invoked with less or more parameters as declared in the definition.

Fatal 84: CANNOT OPEN FILE '%s'

The specified file cannot be open; Check that this file exists and that it is well placed.

Error 85: INTERNAL ERROR

This error should not occur, ... contact the technical support.

Warning 87: MISSING 'END' STATEMENT

The program must be terminated by an END statement. This END statement is missing. Put an END statement at the end of your program.

Fatal 88: UNEXPECTED END OF FILE

This error appears when an assembling is ended while a macro is still running (the endm instruction is missing) or a conditional assembling block ($if) is not closed.

Fatal 89: TOO MANY ERRORS
Too many errors have been found. The translation process has been stopped.

Fatal 90: TOO MANY SEGMENTS
Too many segments have been defined.

Fatal 91: UNABLE TO OPEN INCLUDE FILE '%s'
At least one of the include file (.inc) that are called from your source file cannot be open. Check that they exist and that they are placed in the right directory.

Warning 92: UNABLE TO OPEN SERIAL NUMBER FILE
The files concerning the tool serialization have not been opened successfully. Contact Raisonance.

Warning 93: BAD SERIAL NUMBER
The serial number is not correct. Contact Raisonance.

Error 94: POSITIVE VALUE EXPECTED
The value that defines the expression or the segment must be a positive value.

Warning 95: VALUE OUT OF RANGE, IT HAS BEEN TRUNCATED
The specified numerical value is outside the allowed interval.
Example:
iseg at 1000H ; the address of an idata segment has to be less than 256.

Error 96: INTERNAL ERROR
This error should not occur, contact the technical support.

Fatal 97: INCLUDE LOOP
The use of the file inclusion with the INCLUDE directive leads to loop: the included file includes itself either directly or indirectly.

Error 98: ELSE WITHOUT IF
The ELSE control is used for a block program without the IF control for the previous block program.

Error 99: ELSE IN ELSE STATEMENT
On a conditional assembling, two “else” blocks follow each other. Syntax allows only one.

Error 100: ELSE TYPE DOESN'T MATCH IF TYPE
An "else" block is encountered, but no current "if" condition.

Error 101: ELSEIF WITHOUT IF
An ELSEIF control has been used without the use of a previous IF control.

Error 102: ELSEIF IN ELSE STATEMENT
On a conditional assembling, an else block can follow an elseif block, but the reverse is not allowed (an else block, if it exists, should be always in last position.

Error 103: ELSEIF TYPE DOESN'T MATCH IF TYPE
An “elseif” block" is encountered, without a current “if” condition

Error 104: ENDIF WITHOUT IF

The ENDIF control is used without any IF control. If several ENDIF and IF controls are used, check that there is the same number of IF as ENDIF controls.

Error 105: ENDIF TYPE DOESN'T MATCH IF TYPE
An "endif" directive is encountered, without a current "if" condition.

Error 106: ILLEGAL OBJECT FILE EXTENSION
Not all the files are possible for the generation of an object file. For example, the LST suffix is reserved for the listing files and cannot be used simultaneously for the object file.

Error 107: ILLEGAL LISTING FILE EXTENSION
Not all the files are possible for the generation of a listing file. For example, the OBJ suffix is reserved for the object files and cannot be used simultaneously for the listing file.

Fatal 110: EVALUATION VERSION: MAXIMUM SIZE OF OBJECT FILE REACHED
You are using an evaluation version of the assembler. In the evaluation version, MA-51 is limited to 4Kb of generated code. To get a full version of the Assembler, please contact us at info@raisonance.com

Error 111: INVALID NUMBER
The defined number is not a valid one.

Fatal 112: USER BREAK
The user stopped assembling during the processing (with the CANCEL button for example)

Warning 113: UNKNOWN CONTROL, LINE IGNORED
An unknown directive is placed on the command line.

Error 114: INVALID SFR ADDRESS
A SFR register has been specified with an incorrect address. The SFR directive is present only in the former Raisonance EMA51 syntax and must not appear in a ASM-51 compliant file.

Error 115: INVALID SBIT ADDRESS
An SBIT register has been specified with an incorrect address. The SBIT directive is present only in the former Raisonance EMA-51 syntax and must not appear in an ASM-51 compliant file.

Error 116: TOO MANY INITIALIZERS",
When a table has been initialized in code, too many initializers were present (number greater than the table size).

Warning 117: ORG ASSUMED, PLEASE USE NEW SYNTAX
This warning appears when the former syntax is used for the statement of absolute segments.

Error 118: INVALID ABSOLUTE BDATA ADDRESS
The DATA BITADDRESSABLE (BDATA) space is included between 20H and 2FH. Every address outside this interval is not valid.

Error 122: '%s' INVALID OUTSIDE OF A MACRO
A macro operator (EXITM for example) has been encountered outside every macro definition.

Warning 123: LISTING LINE HAS BEEN TRUNCATED

The lines exited in the listing file are truncated, because they are tool long.

Error 124: MPL-PREPROCESSOR ERROR

An error has been encountered during the MPL preprocessing of the file. The preprocessor usually gives some details on the encountered error.

Error 125: 'DS' NOT ALLOWED IN CODE SEGMENT

The DS directive is used to reserve space in the internal data space or external data space. So, the DS directive cannot be used in CODE segment.

```
NOT VALID
democodeseg      SEGMENT     BIT
RSEG democodeseg
COUNTER:         DS     1

VALID
demodataseg      SEGMENT     DATA
RSEG demodataseg
COUNTER:         DS     1
```

Error 126: ‘DBIT' NOT ALLOWED IN CODE SEGMENT

The use of DBIT is allowed only in segment that have been defined as BIT type.

```
Example:
demobitseg                SEGMENT     CODE  //NOT VALID
demobitseg                SEGMENT     BIT     //VALID

RSEG      demobitseg
MES_SEG:         DBIT   1
```

Error 127: ATTEMPT TO DEFINE AN ALREADY DEFINED MACRO

An attempt was made to redefine an already defined macro with the same name.

Fatal 128: LITE VERSION: MAXIMUM SIZE OF OBJECT FILE REACHED

You are using an limited version of the assembler. In the limited version, MA-51 is limited to 32Kb of generated code. To get a full version of the Assembler, please contact us at info@raisonance.com

WARNING 129: 'END' FOUND IN INCLUDE FILE '%s' IGNORED

The END directive has been found in an include file. This control is ignored.

5.4 Revision History

DATE	SECTION	DESCRIPTION
2001/12	1.1	Comparison between 8051 and MX
	11	Addition of addendum for MA-MX
2002/01	3.3	Addition of Opcodes in the tables of Mnemonics
2002/07	4.6.2	Addition of Recursion in Macros
2003/02	4.1.4	Addition of a new control CS (Case sensitive)
2005/09	3.5	New section about PROC/ENDP

6 Index

7 Tables & figures index

7.1 Tables

7.2 Figures

8 Bibliography

Two books dedicated to the 8051 and its derivatives are published by DUNOD. They aim at describing these microcontrollers and discuss their implementation.

Microcontrôleurs 8051 et 8052
B. Odant
ISBN: 2-10-001764-0

Microcontrôleurs 80C535, 80C537 et 80C552
B. Odant
ISBN: 2-10-001921-X

For more details on a particular device, refer to data books (generally free) distributed by device designers.
For example:

- for Intel devices:

Embedded Microcontrollers
Intel
Semiconductor Products
ISBN: 1-55512-230-2

- for NXP devices:

80C51-Based 8-bit Microcontrollers
NXP
Ref: IC20

- for Infineon devices: CD-Roms describing devices are furnished. Data book on each of those devices are as well available.

Application Notes and
User Manuals for Semiconductors
Infineon
Semiconductor Group
Best.-Nr. B193-H6900-X-X-7400

Technical Product Information
for Siemens Semiconductors
Infineon
Semiconductor Group
Best.-Nr. B192-H6641-X3-X-7400

9 MA-MX Addendum

9.1 New Directives

9.2 New Mnemonics and addressing modes

The MA-MX assembler is a superset of the MA51 assembler.

New mnemonics and addressing modes have been added to represent the new instructions, namely:
- Move and call instructions with 23 bits addresses (EMOV, ECALL, ERET)
- Instructions that work with EPTR
- Instructions that work with PR0 / PR1

9.1 New directives

New directives include:
- Directives to assign an MX-specific address space to a specified symbol,
- Directives to select an absolute segment within a MX-specific memory space.
- Extended possibility for Segment definition
- New Operators

9.1.1 Directives to assign a MX-specific address space to a specified symbol

ESFR

The ESFR directive assigns a ESFR address to the specified symbol. The declaration is as follows:

symbol ESFR esfr-address
where esfr-address is in the range [0,0FFH].

when a symbol assigned to ESFR is met within an instruction (in direct addressing mode), the prefix A5H will be added to this instruction.

EBIT

The EBIT directive assigns a EBIT address to the specified symbol. The declaration is as follows:

symbol EBIT ebit-address
where ebit-address is in the range [0,0FFH].

when a symbol assigned to EBIT is met within an instruction (in bit-direct addressing mode), the prefix A5H will be added to this instruction.

HDATA

The HDATA directive assigns an HDATA address to the specified symbol. The declaration is as follows:

symbol HDATA hdata-address
where hdata-address is in the range [0,07FFFFFH].

EDATA

The EDATA directive assigns an EDATA address to the specified symbol. The declaration is as follows:

symbol EDATA edata-address
where edata-address is in the range [0,0FFFFH].

9.1.2 Directives to select an absolute segment within a MX-specific memory space

HSEG

The HSEG directive selects an absolute segment within the HDATA address space. The format of a statement including this directive is as follows:
HSEG [AT hdata-address]
where edata-address is in the range [0,07FFFFFH].

ESEG

The ESEG directive selects an absolute segment within the EDATA address space. The format of a statement including this directive is as follows:
ESEG [AT edata-address]
where edata-address is in the range [0,0FFFFH].

9.1.3 Segment definition

In MA-MX, the SEGMENT directive is used to declare a segment in the same way as in MA51:

segname SEGMENT segtype [reloctype]

but both "segtype" and "reloctype" can take additional values in MA-MX. Those values are:

Relocation type
The new relocation types concern the extension of the range of addresses (to 8MBytes) for both the CODE space and the HDATA space.

INSEG

INSEG specifies a segment which must be within a 64KB block.
It is useful for applications larger than 64KB, when a code segments contains some "LCALL/LJMP" instructions.
This relocation type is valid only for CODE and HDATA segment.

INSEG0

INSEG0 specifies a segment which must be contained in the FIRST 64KB. This relocation type is valid only for CODE segment.

Segment type

Segment type can be:

CODE	(8051 and MX)
BIT	(8051 and MX)
DATA	(8051 and MX)
IDATA	(8051 and MX)
XDATA	(8051 and MX)
EDATA	(MX only)
HDATA	(MX only)

9.1.4 New operators

MA51 provide the operators HIGH and LOW to extract the high and low bytes from a 16-bit expression.
Because the MX architecture can handle 24-bit addresses, the following operators have been added:

High24

HIGH24 returns the higher byte of a 24-bit expression,

Example:

```
LABH HDATA 01ABCDH

MOV EPH, #HIGH24 LABH; load 01H into EPH
```

High24U

HIGH24U returns also the High order by of a 24-bit expression, but the bit 23 (MSB) is set when the expression is a symbol located in code. It makes the management of the Universal Pointers easier.

Example:

```
LABC CODE 123456H

MOV R1, #LOW LABC; load 56H into R1
MOV R2, #HIGH LABC; load 34H into R2
MOV R3, #HIGH24U LABC; load 92H into R3

MOV R4, #HIGH24 LABC; load 12H into R4
```

9.2 New mnemonics and addressing modes

All new mnemonics and addressing modes specified in the MX datasheet are supported.
MX-specific mnemonics and addressing modes (that is, mnemonics that are understood by MA-MX, but not by MA51) are:

Mnemonic	Operand(s)	Opcode	Action
ECALL	addr23	12	Load a 23-bit address into the Program Counter.
EJMP	addr23	02	Load a 23-bit address into the Program Counter.
JMP	@A+EPTR	73	Program Counter = Accumulator + EPTR.
MOV	EPTR,#data23	90	Load EPTR with a 23-bit constant
MOVC	A,@A+EPTR	93	The accumulator is loaded with the content of address A+EPTR
MOVX	@EPTR,A	F0	EPTR points to an address anywhere in HDATA memory (not DATA, IDATA, or EDATA).
MOVX	A,@EPTR	E0	EPTR points to an address anywhere in HDATA memory (not DATA, IDATA, or EDATA).
INC	EPTR	A3	Increment the 23-bit EPTR.
ERET		22	Load a 23-bit address in the Program Counter from the stack.
EMOV	A,@PRi+disp	48->4F	Load A with the value from the Universal Memory Map at the Address formed by PR0 or PR1 + disp (a value from 0 to 3).
EMOV	@PRi+disp,A	58->5F	Load the Universal Memory Map address formed by PR0 or PR1 + disp (a value from 0 to 3) with the Accumulator.
ADD	PRi, #data2	68->6F	Add an immediate data value from 1 to 4 to the specified Universal Pointer. This is a 24-bit addition.

Note that the specific MX Opcodes must be preceded by the A5 prefix.

10 Flip8051 Addendum

10.1 Specific Directives

10.1 Specific directives

$FLIP8051

This directive enables the DSP instructions (and disables the AJMP/ACALL).

$STANDARD

This directive disables the DSP instruction (and enables the AJMP/ACALL).

Note that these two directives can be replaced by the option Options|Project|MA51|Source|Flip 8051, by checking or not the box.

The very last set of DSP instructions (new BSH and MAC16 with a parameter) are now supported by the assembler and simulated.