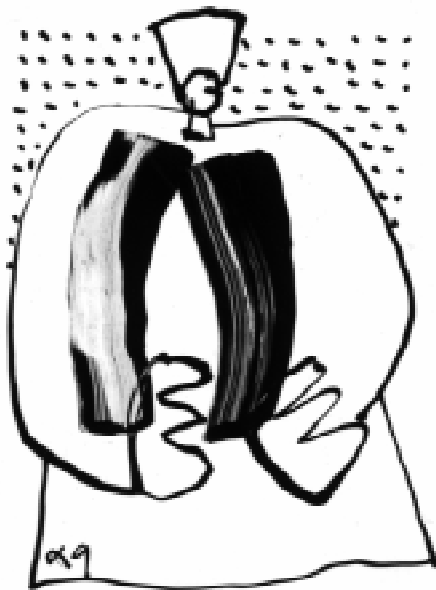


# MA-51

*Assembler for the 8051 family*

---



Reference Manual

January 2000

# RAISONANCE

### SOFTWARE LICENSE FOR MA-51

RAISONANCE grants to the CUSTOMER a license to use the MA-51 software (herein referred to as the SOFTWARE) subject to the following provisions :

- *The SOFTWARE can be only used on one computer. The CUSTOMER may use the SOFTWARE on as many computers as purchased licenses. The CUSTOMER may use the SOFTWARE on a multi-user or network system if one copy of the software is purchased for each node or terminal on which the SOFTWARE is to be simultaneously used.*
- *The SOFTWARE is copyrighted by and shall remain the property of RAISONANCE. The CUSTOMER is permitted to make a unique copy of the SOFTWARE only for backup purpose.*

*REFERENCE MANUAL: No part of this document may be copied or reproduced in any form or by any means without the prior written consent of RAISONANCE S.A. except for personal use by the CUSTOMER. The information contained herein is subject to change. RAISONANCE S.A. assumes no responsibility for the information contained in this document.*

*Copyright RAISONANCE S.A. 1998,2000*

---

## Table of Contents

---

<b>1. INTRODUCTION</b>	<b>7</b>
<b>1.1 Concerning this manual</b>	<b>8</b>
1.1.1 « Simplified syntax » and « Classical syntax »	8
1.1.2 Manual organisation	8
1.1.3 Conventions used in this manual	8
1.1.3.1 Syntax	8
1.1.3.2 Examples	8
1.1.3.3 Instructions	9
<b>1.2 General information on microcontrollers</b>	<b>9</b>
<b>1.3 The 8051 family</b>	<b>10</b>
1.3.1 Introduction	10
1.3.2 Internal organisation	10
<b>2. OVERVIEW OF AN ASSEMBLY PROGRAM</b>	<b>13</b>
<b>2.1 Assembly statements</b>	<b>14</b>
<b>2.2 Comments</b>	<b>14</b>
<b>2.3 Keywords</b>	<b>14</b>
<b>2.4 Symbols</b>	<b>14</b>
<b>2.5 Labels</b>	<b>15</b>
<b>2.6 Operands</b>	<b>15</b>
2.6.1 Numerical, character and string operands	15
2.6.2 Address pointer	16
2.6.3 Main registers	16
<b>2.7 Operators</b>	<b>16</b>
2.7.1 Binary operators	16
2.7.1.1 Arithmetic operators	16
2.7.1.2 Relational operators	17
2.7.1.3 Shifting operators	17
2.7.1.4 Binary operands operators	17
2.7.2 Unary operators	17
2.7.2.1 Arithmetic operators	17
2.7.2.2 Miscellaneous operators	17
2.7.3 Operator precedence	18
<b>2.8 Expressions</b>	<b>18</b>
<b>2.9 Memory areas</b>	<b>19</b>
<b>3. PROGRAM BASICS</b>	<b>21</b>
<b>3.1 Segment</b>	<b>23</b>
3.1.1 Definition	23
3.1.2 Classical Syntax	23
3.1.2.1 Relocatable segment	23
3.1.2.2 Absolute segment	25
3.1.3 Simplified Syntax	28
3.1.4 Segment location	31
<b>3.2 Memory</b>	<b>32</b>

3.2.1	<i>Classical Syntax</i>	32
3.2.1.1	Symbol definition	32
3.2.1.2	Memory reservation	36
3.2.1.3	Memory initialisation	37
3.2.1.4	Register bank reservation and selection	37
3.2.1.5	Register or numerical value assignment	39
3.2.2	<b>Simplified Syntax</b>	41
3.2.2.1	Symbol definition	41
3.2.2.2	Space reservation and initialisation	43
3.2.2.3	Register bank selection	43
3.2.2.4	Numerical value assignment	45
<b>3.3</b>	<b>Instructions</b>	<b>46</b>
3.3.1	Arithmetic instructions	47
3.3.2	Logical instructions	48
3.3.3	Data transfer instructions	49
3.3.4	Boolean instructions	50
3.3.5	Assembler Program Control instructions	51
3.3.6	Concerning JUMP and CALL instructions	52
<b>3.4</b>	<b>Object file</b>	<b>54</b>
3.4.1	Object file generation	54
3.4.2	Object file name	54
3.4.3	Debugging information	54
3.4.4	Line number	55
<b>4.</b>	<b>ADDITIONAL FACILITIES AND ENHANCEMENTS</b>	<b>57</b>
<b>4.1</b>	<b>Source file</b>	<b>58</b>
4.1.1	Definition	58
4.1.2	File inclusion	58
<b>4.2</b>	<b>8051 SFRs : (NO)MOD51</b>	<b>59</b>
<b>4.3</b>	<b>Conditional assembly</b>	<b>60</b>
<b>4.4</b>	<b>Linking directives</b>	<b>62</b>
4.4.1	Definition	62
4.4.2	Symbols used in several modules.	62
<b>4.5</b>	<b>Listing file</b>	<b>64</b>
4.5.1	Definition	64
4.5.2	Listing file generation	64
4.5.3	Listing file title	65
4.5.4	Number of characters per line in a listing file	65
4.5.5	Number of line in a listing file page	66
4.5.6	Unassembled parts of a conditional block	66
4.5.7	Date	67
4.5.8	Error messages	68
4.5.9	Macro assembly instruction	69
4.5.10	Source file inclusion	70
4.5.11	Form feed	70
4.5.12	Symbol table	71
4.5.13	Cross reference table	72
<b>4.6</b>	<b>Macro processor</b>	<b>73</b>
4.6.1	Definition	73
4.6.2	Repetition of blocks	75
4.6.2.1	Sequential block repetition	75
4.6.2.2	Nested block repetition	78

4.6.3	Macro operators	79
4.6.3.1	Determining if a macro argument is null	79
4.6.3.2	Concatenation of text and macro parameters	79
4.6.3.3	Keeping the literal text of an expression	79
4.6.3.4	Evaluating of an expression	80
4.6.3.5	Macro local comments	80
<b>5. APPENDICES</b>		<b>81</b>
<b>5.1 Example of program</b>		<b>82</b>
5.1.1	Example of program	82
5.1.1.1	<i>Simplified Syntax</i> version	82
5.1.1.2	8051 version in <i>Classical Syntax</i>	85
<b>5.2</b>	<b>Keywords</b>	<b>89</b>
<b>6. INDEX</b>		<b>91</b>
<b>7. TABLES &amp; FIGURES INDEX</b>		<b>95</b>
<b>7.1</b>	<b>Tables</b>	<b>95</b>
<b>7.2</b>	<b>Figures</b>	<b>95</b>
<b>8. BIBLIOGRAPHY</b>		<b>97</b>



---

# **1. Introduction**

---

**1.1 Concerning this manual**

**1.2 General information on microcontrollers**

**1.3 The 8051 family**

## 1.1 Concerning this manual

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

### 1.1.1 « 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.

### 1.1.2 Manual organisation

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.

### 1.1.3 Conventions used in this manual

#### 1.1.3.1 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.

#### 1.1.3.2 Examples

Examples always follow the format below :

```

Example_number:
                                ; comment concerning the example.

$PRAGMA
Labels:
    INSTRUCTIONS
                                ; comments
  
```

### 1.1.3.3 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 1 <sup>st</sup> 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 1 <sup>st</sup> byte of the next instruction.
Rn	working register R0-R7.

## 1.2 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.

## 1.3 The 8051 family

### 1.3.1 Introduction

Although many families of microcontrollers exist, the 8051 one has stood out due to its general qualities, and its evolutionary organisation 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.

### 1.3.2 Internal organisation

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

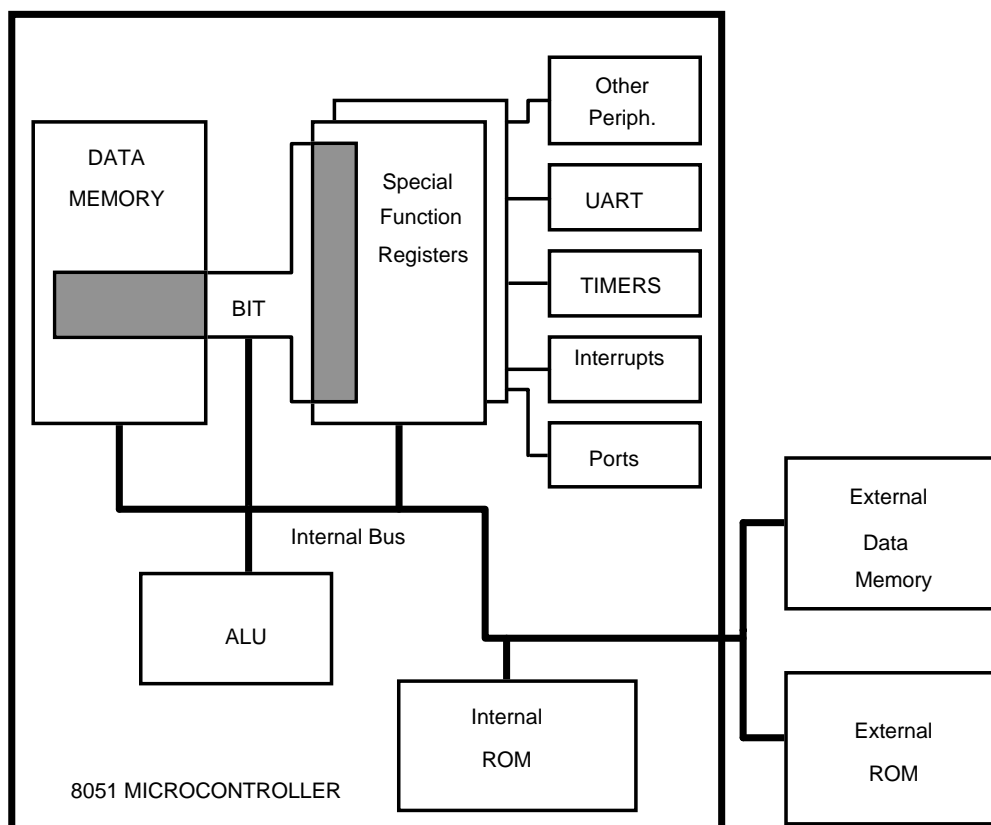


Figure 1: addressable segments of the 8051

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 organisation has been fully optimised 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**
- 2.2 Comments**
- 2.3 Keywords**
- 2.4 Symbols**
- 2.5 Labels**
- 2.6 Operands**
- 2.7 Operators**
- 2.8 Expressions**
- 2.9 Memory areas**

## 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 initialisation block
Init_start:  DB '0','1'
Init_end:    JMP Start_prog
          ; end of the initialising 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.

**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 initialised 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 if 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 that parenthesis are used 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**

---

**3.1 Segment**

**3.2 Memory**

**3.3 Instructions**

**3.4 Object file**

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

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

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

Each paragraph is based upon the same principle. The first part (*Classical Syntax*) details INTEL syntax which is the syntax recently 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.

## 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 initialisation, 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]

<b>Example:</b>	
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]

<b>Example:</b>	
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 initialised as follows:

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

```

Example1:      ; 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.

```

Example2:           ; 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.

<i>Simplified syntax</i>	<i>Classical syntax</i>
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   100h
           ; 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,
- initialise 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

<pre><b>Example:</b>      ; Counter      XDATA    40h               ; XDATA has the same syntax as CODE</pre>
---

---

**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 initialising 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

<b>Example:</b>	
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 internal data space or external data space. 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

<b>Example:</b>	
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 initialisation

---

**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 initialisation

Memory initialisation may be done either before the beginning of the program or within the program. Another distinction must be made between code memory space initialisation and data memory spaces initialisation. In the former case, you may use DB or DW, which initialise 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 initialise CODE memory with byte values. A label immediately preceding DB directive refers to the address of the initialised 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 initialise CODE memory with word values. A label immediately preceding the DW directive refers to the address of the initialised 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.

**Example1:**

```
$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

**Example2:**

```
$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 SET 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

#### Example 1:

```

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 \$ELIF 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 initialised 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 initialisation as the SET *control* with 0 as the number.

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

**Example3:**

```

$RESET(sb)
;sb is initialised with 0
;the same initialisation 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.

```

Example1:      ; 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.

```

Example2:      ; 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

```

Example1:      ; 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 EXTERN 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:

```
Example2:      ; Simplified Syntax
                ; Module i
PUBLIC NUMBER NBR_TSK0 2000h

                ; Module j
EXTERN NUMBER NBR_TSK0
```

```
Example3:      ; Simplified Syntax
                ; Module i
PUBLIC NUMBER ADR_TAB0 100h

                ; Module j
EXTERN 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
```

```
Example4:      ; 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
EXTERN NUMBER ADR_TAB0
                ; valid syntax
```

With the *Classical Syntax*, example1 becomes example5.

```
Example5:      ; 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 initialisation

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

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

Each expression following DB statement is stored in a single 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 initialise CODE memory space with word values.

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

<b>Example:</b>	<i>; Simplified Syntax</i>
Table:	DB 1,2,3,4
eti9:	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.

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

```
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').

Summary:

- 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
ADD	A,Rn	Add register to Accumulator	1	1
ADD	A,direct	Add direct byte to Accumulator	2	1
ADD	A,@Ri	Add indirect RAM to Accumulator	1	1
ADD	A,#data	Add immediate data to Accumulator	2	1
ADDC	A,Rn	Add register to Accumulator with Carry	1	1
ADDC	A,direct	Add direct byte to A with Carry flag	2	1
ADDC	A,@Ri	Add indirect RAM to A with Carry flag	1	1
ADDC	A,#data	Add immediate data to A with Carry flag	2	1
SUBB	A,Rn	Subtract register from A with Borrow	1	1
SUBB	A,direct	Subtract direct byte from A with Borrow	2	1
SUBB	A,@Ri	Subtract indirect RAM from A with Borrow	1	1
SUBB	A,#data	Subtract immediate data from A with Borrow	2	1
INC	A	Increment Accumulator	1	1
INC	Rn	Increment register	1	1
INC	direct	Increment direct byte	2	1
INC	@Ri	Increment indirect RAM	1	1
INC	DPTR	Increment Data Pointer	1	2
DEC	A	Decrement Accumulator	1	1
DEC	Rn	Decrement register	1	1
DEC	direct	Decrement direct byte	2	1
DEC	@Ri	Decrement indirect RAM	1	1
MUL	AB	Multiply A by B	1	4
DIV	AB	Divide A by B	1	4
DA	A	Decimal Adjust Accumulator	1	1

### 3.3.2 Logical instructions

Mnemonic	Operand(s)	Action	Byte(s)	Cycles
ANL	A,Rn	AND register to Accumulator	1	1
ANL	A,direct	AND direct byte to Accumulator	2	1
ANL	A,@Ri	AND indirect RAM to Accumulator	1	1
ANL	A,#data	AND immediate data to Accumulator	2	1
ANL	direct,A	AND Accumulator to direct byte	2	1
ANL	direct,#data	AND immediate data to direct byte	3	2
ORL	A,Rn	OR register to Accumulator	1	1
ORL	A,direct	OR direct byte to Accumulator	2	1
ORL	A,@Ri	OR indirect RAM to Accumulator	1	1
ORL	A,#data	OR immediate data to Accumulator	2	1
ORL	direct,A	OR Accumulator to direct byte	2	1
ORL	direct,#data	OR immediate data to direct byte	3	2
XRL	A,Rn	Exclusive-OR register to Accumulator	1	1
XRL	A,direct	Exclusive-OR direct byte to Accumulator	2	1
XRL	A,@Ri	Exclusive-OR indirect RAM to A	1	1
XRL	A,#data	Exclusive-OR immediate data to A	2	1
XRL	direct,A	Exclusive-OR Accumulator to direct byte	2	1
XRL	direct,#data	Exclusive-OR immediate data to direct	3	2
CLR	A	Clear Accumulator	1	1
CPL	A	Complement Accumulator	1	1
RL	A	Rotate Accumulator Left	1	1
RLC	A	Rotate A Left through the Carry flag	1	1
RR	A	Rotate Accumulator Right	1	1
RRC	A	Rotate A Right through Carry flag	1	1
SWAP	A	Swap nibbles within the Accumulator	1	1

### 3.3.3 Data transfer instructions

Mnemonic	Operand(s)	Action	Byte(s)	Cycles
MOV	A,Rn	Move register to Accumulator	1	1
MOV	A,direct	Move direct byte to Accumulator	2	1
MOV	A,@Ri	Move indirect RAM to Accumulator	1	1
MOV	A,#data	Move immediate data to Accumulator	2	1
MOV	Rn,A	Move Accumulator to register	1	1
MOV	Rn,direct	Move direct byte to register	2	2
MOV	Rn,#data	Move immediate data to register	2	1
MOV	direct,A	Move Accumulator to direct byte	2	1
MOV	direct,Rn	Move register to direct byte	2	2
MOV	direct,direct	Move direct byte to direct	3	2
MOV	direct,@Ri	Move indirect RAM to direct byte	2	2
MOV	direct,#data	Move immediate data to direct byte	3	2
MOV	@Ri,A	Move Accumulator to indirect RAM	1	1
MOV	@Ri,direct	Move direct byte to indirect RAM	2	2
MOV	@Ri,#data	Move immediate data to indirect RAM	2	1
MOV	DPTR,#data16	Load Data Pointer with a 16-bit constant	3	2
MOVC	A,@A + DPTR	Move Code byte relative to DPTR to A	1	2
MOVC	A,@A + PC	Move Code byte relative to PC to A	1	2
MOVX	A,@Ri	Move External RAM (8-bit addr) to A	1	2
MOVX	A,@DPTR	Move External RAM (16-bit addr) to A	1	2
MOVX	@Ri,A	Move A to External RAM (8-bit addr)	1	2
MOVX	@DPTR,A	Move A to External RAM (16-bit addr)	1	2
PUSH	direct	Push direct byte onto stack	2	2
POP	direct	Pop direct byte from stack	2	2
XCH	A,Rn	Exchange register with Accumulator	1	1
XCH	A,direct	Exchange direct byte with Accumulator	2	1
XCH	A,@Ri	Exchange indirect RAM with A	1	1
XCHD	A,@Ri	Exchange low-order nibble in RAM with A	1	1

**3.3.4 Boolean instructions**

<u>Mnemonic</u>	<u>Operand(s)</u>	<u>Action</u>	<u>Byte(s) Cycles</u>	
CLR	C	Clear Carry flag	1	1
CLR	bit	Clear direct bit	2	1
SETB	C	Set Carry flag	1	1
SETB	bit	Set direct Bit	2	1
CPL	C	Complement Carry flag	1	1
CPL	bit	Complement direct bit	2	1
ANL	C,bit	AND direct bit to Carry flag	2	2
ANL	C,/bit	AND complement of direct bit to Carry	2	2
ORL	C,bit	OR direct bit to Carry flag	2	2
ORL	C,/bit	OR complement of direct bit to Carry	2	2
MOV	C,bit	Move direct bit to Carry flag	2	1
MOV	bit,C	Move Carry flag to direct bit	2	2

### 3.3.5 Assembler Program Control instructions

Mnemonic	Operand(s)	Action	Byte(s)	Cycles
ACALL	addr11	Absolute Subroutine Call	2	2
LCALL	addr16	Long Subroutine Call	3	2
RET		Return from subroutine	1	2
RETI		Return from interrupt	1	2
AJMP	addr11	Absolute Jump	2	2
LJMP	addr16	Long Jump	3	2
SJMP	rel	Short Jump (relative addr)	2	2
JMP	@A + DPTR	Jump indirect relative to the DPTR	1	2
JZ	rel	Jump if Accumulator is Zero	2	2
JNZ	rel	Jump if Accumulator is Not Zero	2	2
JC	rel	Jump if Carry flag is set	2	2
JNC	rel	Jump if No Carry flag	2	2
JB	bit,rel	Jump if direct Bit set	3	2
JNB	bit,rel	Jump if direct Bit Not set	3	2
JBC	bit,rel	Jump if direct Bit is set & Clear bit	3	2
CJNE	A,direct,rel	Compare direct to A & Jump if Not Equal	3	2
CJNE	A,#data,rel	Comp. imm. to A & Jump if Not Equal	3	2
CJNE	Rn,#data,rel	Comp. imm. to reg & Jump if Not Equal	3	2
CJNE	@Ri,#data. rel	Comp. imm. to ind. & Jump if Not Equal	3	2
DJNZ	Rn,rel	Decrement register & Jump if Not Zero	2	2
DJNZ	direct. rel	Decrement direct & Jump if Not Zero	3	2
NOP		No operation	1	1

### 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 optimisation, 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 'Extern' 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 EXTERN 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.

#### 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
```

#### 3.4.4 Line number

The LINE directive allows the definition of line numbers in an assembly source file, similar to those generated by compilers. The text following the LINE 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:*            **LINE** text\_comment

---

**Note:**

The following restrictions must be taken into account:

1. The LINE directive generates a line number only if the current segment is and is ignored in all other segments (data, bit, xdata, etc.).
  2. The LINE directive is ignored when used in 'INCLUDE' files.
-



---

## **4. Additional Facilities and Enhancements**

---

- 4.1 Source file**
- 4.2 8051 SFRs : (NO)MOD51**
- 4.3 Conditional assembly**
- 4.4 Linking directives**
- 4.5 Listing file**
- 4.6 Macro processor**

## 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.

**Example :**

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

---

**Note:**

\$INCLUDE is used in assembly file whereas INCLUDE is used 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.

- 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  
                   **ELSEIF** 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 EXTERN 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 EXTERN 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 extern 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, RESULTAT)
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 EXTERN 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 than 255 EXTERN symbols (defined with the PUBLIC attribute in other modules). This limitation is due to OMF-51.
4. If a symbol is declared as both EXTERN and PUBLIC, the EXTERN declaration will be ignored.
5. The assembler ignores EXTERN symbols that are not used in a module. It is therefore possible to define a 'declaration file' containing more than 255 EXTERN 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 what information is included, is controlled by listing controls presented in this section: TITLE, PAGESWIDTH, 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 PAGEDLENGTH control. This number must be chosen between 10 and 65535. The default value is 60.

**Syntax:**            [**\$**]PAGEDLENGTH(number)

**Abbreviation:**    [**\$**]PL

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

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

PAGEDLENGTH(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:**

\$PAGEDLENGTH (or \$PL) is used when specified in the assembly file whereas PAGEDLENGTH (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 visualised 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.

---

#### 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

<b>Example 1:</b> MA51 raison.a SYMBOLS
--

<b>Example 2:</b> \$SB
---------------------------

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

**Syntax:**            [**\$**]NOSYMBOLS

**Abbreviation:**    [**\$**]NOSB

<b>Example 3:</b> 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 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

**Example 3:**

```
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.

---

## 4.6 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 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.

### 4.6.2.1 Sequential block repetition

#### 4.6.2.1.1 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
```

#### 4.6.2.1.2 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
```



#### 4.6.2.2 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
```

### 4.6.3 Macro operators

Some operators may be used in macros.

#### 4.6.3.1 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

```

#### 4.6.3.2 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

```

#### 4.6.3.3 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.

#### 4.6.3.4 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
```

#### 4.6.3.5 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 initialisation
    MOVE R1, #00h ;; R1 initialisation
ENDM
```

---

## **5. Appendices**

---

### **5.1 Example of a program**

### **5.2 Keywords**

## 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 initialisation

;Initialisation 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
TRT_INT1:   ;Relocatable.
             ;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 initialisation
;Initialisation 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	CJNE	CLR	CODE	CPL
COND	CSEG			
DA	DATA	DATE	DB	DBIT
DEC	DEBUG	DIV	DIRECT	DJNZ
DPL	DPH	DPTR	DS	DSEG
DW				
EJ	EJECT	ELSE	ELSEIF	END
ENDIF	ENDM	ENDP	EP	EQ
EQU	ERRORPRINT	EXTERN	EXTRN	
GEN	GT	GTE		
HIGH				
IDATA	IF	INBLOCK	INCLUDE	INC
INPAGE	IRP	IRPC	ISEG	
JB	JBC	JC	JMP	JNB
JNC	JNZ	JZ		
LCALL	LI	LINE	LIST	LJMP
LOCAL	LT	LTE		
MACRO	MO	MOD	MOV	MOVC
MOVX	MUL			
NAME	NE	NOCOND	NODB	NODEBUG
NOEP	NOERRORPRINT	NOGEN	NOLIST	NOLI
NOMO	NOOJ	NOOBJECT	NOP	NOPRINT
NOSB	NOSYMBOLS	NOT	NOXR	NOXREF
NUL	NUMBER			
OBJECT	OJ	OR	ORG	ORL
OVERLAYABLE				
PAGE	PAGELNGTH	PAGEWIDTH	PC	PL
POP	PR	PRINT	PUBLIC	PUSH
PW				
R0	R1	R2	R3	R4
R5	R6	R7	RBIT	REG

REGISTERBANK	REL	REPT	RESET	RET
RETI	RL	RLC	RR	RRC
RSEG				
SB	SEGMENT	SET	SETB	SETS
SHL	SHR	SJMP	SUBB	SWAP
SYMBOLS				
TITLE	TT			
UNIT	USING			
WHILE	WORD			
XCH	XCHD	XDATA	XOR	XR
XREF	XRL	XSEG		

## 6. Index

-	16, 17, 18	Address pointer	16
\$		AJMP	51
\$	14	alignment	23
%		<b>AND</b>	17, 18
%		ANL	48, 50
%	macros	Appendices	81
		AR0-AR7	16
<b>&amp;</b>		AT	28
<b>&amp;</b>			
<b>&amp;</b>	macros	<b>B</b>	
		<i>BDATA</i>	19, 28
(		Bibliography	97
( )	18	binary	15
*		BIT	11, 19, 28, 32
*	16, 18	BITADDRESSABLE	23
/		Boolean	50
/	16	BSEG	25
;			
::		<b>C</b>	
::	macros	C	16
		CALL	52
+		Character	15
+	16, 17, 18	CJNE	51
<		CLR	48, 50
<	17, 18	Code	
<	macros	directive	28
<=		<i>CODE</i>	10, 19, 33
=		<b>CODE INBLOCK</b>	31
=	17, 18	Comments	14
>		<b>COND</b>	66
>	17, 18	Conditional assembly	60
>	macros	Controls	14
>=		CPL	48, 50
<b>8</b>		CSEG	26
8051	10		
<b>A</b>		<b>D</b>	
A	16	DA	47, 67
AB	16	DATA	11, 19, 28, 33
absolute	23	<b>DATE</b>	67
ACALL	51	DB	37, 43, 54
accumulator	16	DBIT	36
ADD	47	DEBUG	54
ADDC	47	DEC	47
		decimal	15
		Directives	14
		distance	23
		DIV	47
		DJNZ	51
		DPH	16
		DPL	16
		DPTR	16
		DS	36
		DSEG	26

DW .....	37	Listing file .....	64
<b>E</b>		LJMP .....	51
<b>EJ</b> .....	70	<b>LOW</b> .....	17, 18
<b>EJECT</b> .....	70	<b>LT</b> .....	17, 18
ELSE .....	60	<b>LTE</b> .....	17, 18
ELSEIF .....	60	<b>M</b>	
END .....	31	<b>MACRO</b> .....	73, 78
ENDIF .....	60	Macro processor .....	73
<b>ENDM</b> .....	73, 78	Memory .....	19, 32
<b>EP</b> .....	68	microcontrollers .....	9
<b>EQ</b> .....	17, 18	<b>MO</b> .....	59
EQU .....	40, 45	<b>MOD</b> .....	16, 18
<b>ERRORPRINT</b> .....	68	MOD51 .....	59
Expressions .....	18	MOV .....	49, 50
EXTERN .....	62	MOVC .....	49
external RAM .....	11	MOVX .....	49
<b>G</b>		MUL .....	47
<b>GEN</b> .....	69	<b>N</b>	
<b>GT</b> .....	17, 18	Name	
<b>GTE</b> .....	17, 18	directive .....	82, 85
<b>H</b>		<b>NE</b> .....	17, 18
hexadecimal .....	15	<b>NOCOND</b> .....	66
<b>HIGH</b> .....	18	<b>NODB</b> .....	55
<b>I</b>		NODEBUG .....	55
<b>IDATA</b> .....	19, 34	<b>NOEP</b> .....	68
IF 60 .....		<b>NOERRORPRINT</b> .....	68
INC .....	47	<b>NOGEN</b> .....	69
INCLUDE .....	58	NOLIST .....	70
index		<b>NOMO</b> .....	59
figures .....	95	<b>NOMOD51</b> .....	59
tables .....	95	NOOBJECT .....	54
Index .....	91	<b>NOOJ</b> .....	54
Instructions .....	14, 19, 46	NOP .....	51
<b>IRP</b> .....	76	<b>NOPR</b> .....	64
<b>IRPC</b> .....	77	<b>NOPRINT</b> .....	64
ISEG .....	27	<b>NOSB</b> .....	71
<b>J</b>		<b>NOSYMBOLS</b> .....	71
JB51 .....		<b>NOT</b> .....	17, 18
JBC .....	51	<b>NOXR</b> .....	72
JC51 .....		<b>NOXREF</b> .....	72
JMP .....	51	<b>NUL</b> .....	79
JNB .....	51	NUMBER .....	34, 41
JNC .....	51	<b>O</b>	
JNZ .....	51	OBJECT .....	54
JUMP .....	52	Object file .....	54
JZ51 .....		octal .....	15
<b>L</b>		<b>OJ</b> .....	54
label .....	15	Operands .....	15
LCALL .....	51	operators .....	16
<b>LI70</b> .....		<b>OR</b> .....	17, 18
LINE .....	55	ORG .....	16, 31
Linkage directives .....	62	ORL .....	48, 50
<b>LIST</b> .....	70	Overlapping .....	23
		<b>P</b>	
		<b>PAGELength</b> .....	66

<b>PAGEWIDTH</b> .....	65	<b>SET</b> .....	39
<b>PC</b> .....	11, 16	<b>SETB</b> .....	50
<b>PL</b> .....	66	<b>SETS</b> .....	50
<b>POP</b> .....	49	<b>SHL</b> .....	17, 18
<b>PR</b> .....	64	<b>SHR</b> .....	17, 18
precedence .....	18	<b>SJMP</b> .....	51
<b>PRINT</b> .....	64	Source file .....	58
program counter.....	16	String.....	15
Program Counter.....	11	<b>SUBB</b> .....	47
<b>PUBLIC</b> .....	62	<b>SWAP</b> .....	48
<b>PUSH</b> .....	49	Symbols.....	14
<b>PW</b> .....	65	<b>SYMBOLS</b> .....	71
<b>R</b>		<b>T</b>	
<b>R0-R7</b> .....	16	the NAME.....	54
<b>RB</b> .....	37, 43	transfer .....	49
<b><i>RBIT</i></b> .....	19, 28	<b>TT</b> .....	65
<b><i>REG</i></b> .....	19, 28	<b>U</b>	
register.....	10, 11, 19, 44	<b>USING</b> .....	38
relocatable .....	23	<b>X</b>	
<b>REPT</b> .....	75	<b>XCH</b> .....	49
<b>RET</b> .....	51	<b>XCHD</b> .....	49
<b>RETI</b> .....	51	<b>XDATA</b> .....	11, 19, 35
<b>RL</b> .....	48	<b>XDATA INPAGE</b> .....	31
<b>RLC</b> .....	48	<b>XDATA PAGE</b> .....	31
<b>RR</b> .....	48	<b>XOR</b> .....	17, 18
<b>RRC</b> .....	48	<b>XR</b> .....	72
<b>RSEG</b> .....	24	<b>XREF</b> .....	72
<b>S</b>		<b>XRL</b> .....	48
<b>SB</b> .....	71	<b>XSEG</b> .....	27
<b>SEGMENT</b> .....	23		
segments .....	10		



---

## 7. Tables & figures index

---

### 7.1 Tables

Table 1: binary arithmetic operators

Table 2: relational operators

Table 3: shifting operators

Table 4: Boolean operators

Table 5: unary arithmetic operators

Table 6: miscellaneous unary operators

Table 7: operator precedence

### 7.2 Figures

Figure 1: addressable segments of the 8051



---

## 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 Philips devices:

**80C51-Based 8-bit Microcontrollers**

Philips

Ref: IC20

- for Siemens 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**

Siemens

Semiconductor Group

Best.-Nr. B193-H6900-X-X-7400

**Technical Product Information  
for Siemens Semiconductors**

Siemens

Semiconductor Group

Best.-Nr. B192-H6641-X3-X-7400