ICL51

Technical Reference Manual
release 4.0

Detail®

ELETTRONICA INDUSTRIALE

ICL51 - release 4.0
February 1997

Detail® retains the right to change , without notice, the software specification of this manual.

Every care is been taken in the collection and in the check of the documentation of this manual, anyway the property of trademark ***Detail***® can assume no responsibility for the use of its products.

Table of contents

Introduction 9

Updating of the release 4.0 11

General elements 13

Architecture of the language ICL51 14

System resources 17

The external resources of the system 19

The reserved bytes of the external resources 23

Table 1. Meaning of the reserved bytes 23

The internal resources of the system 25

Study of the internal resources 27

The COUNTER device 28

The PULSE GENERATOR device 30

Special bits of FLAG 31

The CONSTANTS in 1/2/4 bytes 32

SXS indicator of cycles per second 33

A complete WATCH/CALENDAR 34

The instructions set 37

Preliminary notions 39

Table 2. Fields of variation of the variables in 1/2/4 bytes 40

LD 41
LDNOT 42
AND 43
ANDNOT 44
OR 45
ORNOT 46
ANDLD 47
ORLD 48
OUT 49
OUTNOT 50
SET 51
RES 52
CPL 53
JMP 54
JME 55
GOTO 56
GOSUB 57
NOP 59
END 60
RET 61
TIM 62
CNT 64
SFR 66
ANDB 68
ORB 69
XORB 70
CPLB 71
MOV1 72
MOV2 73
MOV4 74

CMP1 75
CMP2 76
CMP4 77
ADD1 78
ADD2 79
ADD4 80
SUB1 81
SUB2 82
SUB4 83
MUL1 84
MUL2 85
MUL4 86
DIV1 87
DIV2 88
DIV4 89
INC1 90
INC2 91
INC4 92
DEC1 93
DEC2 94
DEC4 95
ABS1 96
ABS2 97
ABS4 98
NEG1 99
NEG2 100
NEG4 101
BINBCD1 102
BINBCD2 103
BINBCD4 104

BCDBIN1 105
BCDBIN2 106
BCDBIN4 107
SWAP 108
RCL1 109
RCL2 110
RCL4 111
STO1 112
STO2 113
STO4 114
ADD 115
SUB 116
MUL 117
DIV 118
CMP 119
MOVADD 120
MOVASC 121
MOVBLK 122
CMPBLK 123
RESMEM 124
IOREFR 125
RESWD 126
INCLUDE 127
PASSW 128
Indirect operands for bytes instructions 129
Mathematic expressions evaluation 130
Optimize the program performance 134
Summary table of ICL51 language 137
Table 3.a. Sistem resources 138
Table 4.a. Data RAM map 140
Table 5.a. Instructions set summary 143
Table 6. Validity groups of operands 152
Table 7.a. Instructions performance 153

External instructions 157

Add personalized instructions 159

Transferring operands to the external instruction 160

How create an external instruction 162

External instruction MUX 164

External instruction SQR 166

Figure 1. Flow diagram to calculate the square root 166

Develop environment 169

Programming by means PC 171

Installing the ICL51 software 172

ICL51 develop environment 174

Edit (F1) 174
Compile (F2) 174
View Error (F3) 174
Transfer (F4) 175
Monitor (F5) 175
Dos Shell (F6) 175
Configure (F7) 175
Help (F8) 175
Esc 175

Software configuration 176

Logic type 176
File 176
Logic at 177
Printer at 177
Automatic 177
Editor 177
Viewer 178
Password 178

Program writing 179

Compilation and error file 179

Transferring menu 180

DownLoad (F1) 180
UpLoad (F2) 180
View Text (F3) 181
Compare (F4) 181
Backup H/X bytes (F5) 181
Restore H/X bytes (F6) 182

View Holding file (F7) 182
Update Watch (F8) 182
Stop Logic (F9) 183

Program monitor 184
Start (F1) 185
Stop (F2) 185
Change (F3) 185
Force (F4) 186
Set (F5) 186
Res (F6) 186
Help (F7) 186
Search (F8) 186
Esc 186

Communication protocol 187

General informations 189

Opening the communication channel 190

Communication errors handling 191

Communication commands 193
Table 8.a. Data RAM map 195

STOP 197

RUN 198

STATUS 199

MONITOR1 200

MONITOR2 201

MONITOR4 202

FORCE1 203

FORCE2 204

FORCE4 205

RESBIT 206

SETBIT 207

BACKUP 208

RESTORE 209

UPLOAD 210

DOWNLOAD 211

Introduction

Updating of the release 4.0

The release 4.0 proposes principally the object to optimize the ICL51 software packet. For this purpose is completely controlled the global setting-up of the develop system taking into consideration especially its aspects of modular structure and expansibility.

The continuous introduction of new Logic Controllers required the study of a develop system with a completely generic setting-up and automatically adaptable and configurable: with the release 4.0 it is enough to add in the directory of the software the files corresponding to the new Logic and the system will be immediately ready to run also with this one.

The ICL51 software release 4.0 appears to the Programmer in a way much more practical and direct; the menu are shorter and easier and the phases of debugging of the program are suggested automatically by a flow diagram. All this not to force the Programmer to go round continuously in a “jungle” of menu and undermenu to perform also the most elementary operations. Simplicity and immediateness of use are the bases of the setting-up choices for the whole software packet: the Programmer will have only to concentrate on the program of the machine to automate and not on the programming software.

The different phases of program develop like editing, compilation, transferring and monitor are executed separately by proper programs optimized for every different Logic. In this way the principal menu, recalled by the ICL51 program, only works as coordinator of various programs loading in the PC memory only the rigorous necessary for the current phases. For this reason the new release 4.0 takes a RAM memory small space compared with the precedent versions, excluding in this way any possible saturation of the PC memory available for the program execution.

The release 4.0 is been particularly increased in the monitor program. The monitored variables list has become a single continuous list of 512 variables to which it is possible to join also an alphanumeric descriptive field. More advanced handling of the connections between the PC and the Logic permitted to remove every manual operation of resetting in case of fallen communication.

We don’t omit also that the new ICL51 software is born in a period of a big growth of Internet network. The new structure of the packet allows a easy updating by means Internet network; furthermore this manual is been completely written utilizing the potentiality of hypertextual connection and it is available as a file of type .PDF, got from now a in the diffusion of documentation.

General principles

The ICL51 language is been particularly studied to allow the high level programming of a complex electronic system, realized with connection, through a multipoint network, of several boards with a microprocessor of INTEL® 80C51 family.

The purpose of such electronic system is the control of the functions of an automatic machine or installation, independently of their dimensions. The word ICL51 comes from Industrial Control Language, whereas 51 denotes that the software is directed towards all the microprocessors of INTEL® 80C51 family, including all its extensions.

The basic characteristic of such electronic system, on which works this language, is to be composed of one or more electronic boards each other collected in accordance with a MASTER/SLAVE structure. The minimum system will be therefore built up of only one board, taking properly the function of MASTER. The maximum system instead built up of 32 boards, of which one works as a MASTER and the remaining 31 work as SLAVES.

Each board, both MASTER and SLAVE, must have aboard at least one microprocessor of the 80C51 family. The ICL51 language allows, by means the proper compilation, to produce the machine code running on the MASTER board of the system; the possible SLAVES boards have instead a proper defined program aboard, allowing to process in site its own resources and to communicate with the MASTER. However exists the possibility from some boards to work as a SLAVE of intelligent expansion, that is able to execute locally its own automation program, in according with the ICL51 language, in a parallel way to the MASTER. This functioning procedure allows to realize very complex systems formed by one principal MASTER controlling an ensemble of SLAVES in which some execute, in a parallel way, their own local program (on their own resources) with the same programming potentiality of the MASTER.

The MASTER handles therefore the real and peculiar management of the whole automatic process, in according with the program written in the ICL51 language, processing both its own resources and SLAVES's ones. Every way the MASTER has a particularized view of all resources of the whole system, whereas every SLAVES views only its own resources.

The resident software on the MASTER handles principally four things:

1) Handling of its own resources
2) Communication with the SLAVES for their resources
3) Communication with the connected COMPUTER
4) Execution of the user program compiled by ICL51

While the sections 1, 2, 3 are includes in the operative system of the MASTER and then always fixed, the section 4 is variable application by application, because it represents the part of object code generated by the compilation of the source ICL51 language, describing the cycle of machine operations.

ICL51 language architecture

The ICL51 language allows to describe the functioning of an automatic machine simply editing a source FILE, in according to the fixed syntactical rules. The source FILE (.PRG), after editing, will be submitted to a compilation process which will generate a object FILE (.OBJ), successively loaded on the MASTER board.

This language is a INSTRUCTION LIST type, that is a sequence of row in the source file, everyone of which gives an instruction of the language with some operands.

So an instruction will be formed by a single row, divided in a certain number of fields. The first field is always the mnemonic of the instruction, while the other fields mean the argument of the instruction that is the list of its possible operands. A possible last field, preceded by the character ‘ (quote), is considered comment, that is written text as note for the instruction:

MNEMONIC [OPERAND1] [OPERAND2] [OPERAND3] [‘COMMENT]

The different fields of a instruction row must be divided at least by a blank space (BLANK) or at least a tabulator (TAB).

The source file can contain, in any place, empty rows (CR), or text rows, at the purpose of COMMENT, preceding them with the character quote (‘).

The comments, preceded by the character quote, can stay on the right of the instruction, or fill by themselves all the row of the program file. These comments are completely ignored by the compiler and then they remain only notes of the edited source file.

An other type of comment useful inside the program file exists; differently from preceding this type of comment is elaborated by the compiler and it has become an integral part of the code loaded on the MASTER board. These comments cannot stay on the right of a instruction, but they must be alone in a row of the program file, as if they were executable instructions; in addition to distinguish them from the preceding ones, it is necessary that the character “ (double quote) precedes them.

The comments, preceded by the character double quote, are elaborated by the compiler by means an algorithm of secret code an then stored in an area of the program memory of the MASTER. The total memory available for those comments is up to 8176 bytes, for this reason you must pay attention to use those comments; every way the compiler will warn the Programmer when the space of comments memory will be finished.

All the comments of double quote type are stored in the program memory, till there is room enough; their recovery can happen simply loading the program from the MASTER board to the Computer and decoding such informations. The result is the creation of a file .TXT in ASCII format and then ready for printing; you can consider that this file will contain all the comment rows of double quote type consecutively and in the same order of the source list. The recover operation will be possible only knowing the password declared in the source program and then the informations will remain reserved only to the Programmer.

We consider now a further possibility given by the compiler, that is to associate with every operand a mnemonic string of maximum length up to 32 characters. This association is made editing in the source file a row like that:

```
LABEL = OPERAND [‘COMMENT]
```

With LABEL we mean a string (max. 32 alphanumeric characters) that generally remembers the meaning of that operand; placing this association in a row of the program, all the instruction of the following rows could have, as operands, indifferently their primitive words or the mnemonic LABEL. Generally it is convenient to place that associations in the top of the program, in this way they will be valid for all the following instructions. You have to take in consideration that many EDITOR programs of file allows the automatic replacement of a string with another; utilizing such a function, it is possible to replace, in the operand field of an instruction, the LABEL with the primitive identifier or vice versa. The compiler, in every way, will process the instruction equally.

It is possible also an other type of LABEL to use with conditioning jump instructions and to call subroutines: this type of LABEL identifies the exact point of the list where to jump with the instructions GOTO and GOSUB. It is formed by a string (max 32 alphanumerical characters) followed, without intermediate blanks, by the character : (colon) and it must occupy alone a whole row of the source file.

For example:

```
GOTO        POINT_WHERE_TO _GO

............
............

POINT_WHERE_TO _GO:
```

Concluding we remember that the last instruction of the list must be a END instruction to end the program, allowing to the operating system of the logic controller to know the scan of the user program has finished and then it allows the execution of the other phases described in introduction. If subroutines are present, other END instructions can appear because the instructions list of every subroutine must end with its own END instruction.

A detailed examination of all the instructions of the ICL51 language and of their syntax will be performed later.

System resources

System external resources

System RESOURCE means any element placed at disposal by the hardware and by the software and that it can functionally be utilized by the user program in order to operate the automatic process.

We define EXTERNAL resource, any resource of the total system (MASTER + SLAVES) which is in any way referred to a TERMINAL of the boards, or which has any communication function with the external world, by means devices on the boards. External resources are, for example, the digital and analogic Input/Output of both the MASTER and the SLAVES, the characters of a display, the keys of a terminal, etc.

An INTERNAL resource is instead placed at disposal only and exclusively by the MASTER and it has no reference to the external elements of the MASTER board; internal resources are the work memory, the counters, the timers and all that devices realized by the operating system of the Logic Controller. You consider that generally the internal resources are got through a software simulation by the operating system of the MASTER and so they have not a direct reference with hardware devices of the board.

The division in external and internal resources enables us to split in two parts the preliminary analysis ICL51 programming language.

In the preceding paragraph we talked of instructions and related operands. The operands are nothing but resources, because these are the ones to be processed by the instructions at the purpose to realize the automatic control of the machine.

Concentrating the attention only on the external resources, we can imagine all these ones are available in the RAM memory of the MASTER, in the form of operands indictable by the instructions of the user program.

Since we want to pass over, for generalization of the system, how many SLAVES will be present and their kind and performed functions, we consider that the portion of RAM of the MASTER, joined to the external resources, is equally divided in 32 areas of fixed dimension (128 bytes), each assigned to every board of the system.

Such areas numbered from 0 to 31 (where 0 is for external resources of the MASTER and 1÷31 for external resources of the SLAVES) allow to the user program to have access, in every moment, to all the external resources of the whole system, so as if they were materially present on the only MASTER board.

Each of such areas is representative of every board present in the distributed system; among 128 bytes, eight are reserved to the operating system, while the greater part (120 bytes) are potentially available for the external resources of the corresponding board.

For example if the MASTER is a board with the following external resources:

24	Digital input ON/OFF	(3 bytes)
16	Digital output ON/OFF	(2 bytes)
4	8 BITS analog input	(4 bytes)

the area number 0 will use 3+2+4=9 bytes to represent the external resources of the MASTER.
If the SLAVE number 3 is a board with the following external resources:

8	Digital input ON/OFF	(1 bytes)
8	Digital output ON/OFF	(1 bytes)
4	8 BITS analog input	(4 bytes)

the area number 3 will use 1+1+4=6 bytes to represent the external resources of such peripheral board.

Anyone of 128 bytes of all the 32 areas of such part of RAM of the MASTER, can be a valid operand for those instructions requiring one or more bytes, both in reading and in writing (bytes instructions); moreover, anyone of 8 bits of everyone of 128 bytes forming any area associate to a board, can be used, both in reading and in writing, by all those instructions operating with single bits (Boolean instructions).

This arrangement provides to the system a great potentiality and predisposition to future developments of SLAVES boards completely new, both in the function and in the dimension, without precluding the study of SLAVES boards designed properly for some applications, for which is not possible to use the standard peripheral boards since that moment developed.

It remains now to identify the simplest possible way to identify univocally, by means a code, anyone of 32*128=4096 bytes or anyone of 4096*8=32768 bits of the possible external resources of the whole system.

The simplest solution is to number the 32 areas, corresponding each to a possible board present in the system, with a number from 0 to 31; established the board, then you can number the 128 bytes in every area with a number from 0 to 127; at the end you can number the 8 bits in every byte beginning from 0 for the LSB (least significant bit) up to 7 for the MSB (most significant bit).

Finally separating the three identifier number with the character . (full stop), you can insert the operand for the external resources of the system:

Board.Byte.Bit (Board=0÷31, Byte=0÷127, Bit=0÷7)

For example we consider some byte type and bit type operand of a system formed by a MASTER (with 3*8=24 digital inputs addressed at the bytes of address 0÷2, plus 2*8=16 digital outputs addressed at the bytes 8÷9, plus 4 analog 8 bits inputs addressed at the bytes 4÷7) and a SLAVE with address 2 (with 2*8=16 digital inputs addressed at the bytes 0÷1, plus 1*8=8 digital output addressed at the byte 8):

0.0	input channel (byte) number 0 of the MASTER
0.8	output channel (byte) number 8 of the MASTER
2.1	input channel (byte) number 1 of the SLAVE 2
2.8	output channel (byte) number 8 of the SLAVE 2
0.4	first analog input (byte) of the MASTER
0.0.5	digital input 5 of channel 0 of the MASTER
0.9.6	digital output 6 of channel 9 of the MASTER
2.1.3	digital input 3 of channel 1 of the SLAVE 2
2.8.4	digital output 4 of channel 8 of the SLAVE 2

We have in this way explained the first step from which is born, in a completely natural way, the syntax definition of the ICL51 programming language; actually with these symbolic identifiers of the of the external resources we have introduced the first symbols of the language.

We give also basic indications, for the beginners in the use of this language, always regarding the external resources.

For every operand (byte type or bit type) the used identifier in the language must have, as first number, the board number. On every board of the system it is possible to define an address by means some switches of type Dip-Switch, Jumper or by means software addressing. The MASTER board will be always configured as board 0 whereas all the SLAVES will have a identifier number 1 through 31. It is obviously forbidden to configure two SLAVES with the same board number, whereas you can choose for each SLAVE any number 1 through 31.

At this purpose we have to do a further explanation: the resources update of the SLAVES happens in a dynamic way to the maximum address really used in a instruction of the program. Practically if defining the board numbers of the elements of the whole system, you use numeric progression from the address 0 of the MASTER, you can get a large increasing of speed in the execution of the machine program. This because the compiler automatically transmits to the operating system of the MASTER the maximum address of board found in the user program; in this way the MASTER will not try absolutely to communicate with the boards with higher address.

We suggest therefore to choose the SLAVES boards addresses consecutively starting from the number 1, paying attention not to leave empty spaces in the numbering; this will allow the operating system to reach the maximum speed.

The second number of the identifier of any operand (byte or bit) representing an external resource, must be always the byte number of the area formed by the 128 bytes corresponding to every board. The specified functions of the board (both MASTER and SLAVE) will define in which numbers you can find the offered resources. For every board you can refer to the related documentation to identify the address numbers of the bytes corresponding to the various resources, both in input (reading the byte) and in output (writing the byte).

By definition in every single area of the board, all the bytes corresponding to resources in input are contiguous into the system, so that all the bytes corresponding to resources in output from the system are contiguous. Both the blocks of bytes of input and output are not overlapped and they can be separated by not used bytes, as long as such blocks are included within the area of the bytes 0÷119. The remaining 8 bytes (120÷127) are reserved to the operating system at the purpose to store some informations, as the identification of the SLAVE found in that address, or the starting points and the length of the input and output blocks of the external resources.

All the bytes (within 0÷119) of the areas of the board, not used by the external resources, are available as internal not retentive memories and reset at every power on of the system; anyway pay attention to use writing instructions on bytes reserved to the operating system (120÷127), because it is possible to cause problems to itself. The use of the 8 bytes reserved to the operating system allows, for example, to suspend temporarily , under certain conditions, the communications with the SLAVE (resetting to 0 the number of transmitted or received bytes), reducing in this way the execution time of the program.

The third and possible number of the identifier of a external resource, recognizes one of the 8 bits of the channel (i.e. of the byte). The compiler will notice, communicating an error, if a certain instruction requires, as operands, some byte identifiers or bit identifiers.

We emphasize that all the bits and all the bytes of all the 32 areas of the boards, can be always in the same way either written or read by the user program , whatever the kind of the boards is. The duty of the operating system of the MASTER is the one to write, every program cycle, all the output bytes of the MASTER’S RAM to its own output resources and those of the SLAVES; in addition, always in a invisible way for the user program, the operating system of the MASTER will make a reading of all the own input resources and of the SLAVES, writing them in the proper input bytes of the user RAM.

Reserved bytes of external resources

All the external resources of the system are stored in the data RAM memory of the MASTER; each board, for its own resources, fills a 128 contiguous bytes area. Depending on the kind of the considered, such area will be filled in part by the bytes of the input resources of the system (for example the bytes corresponding to the sensors of the machine) and in part by the bytes of the output resources of the system (for example the bytes corresponding to the digital output ports connected to the peripheral devices of the machine).

The first 120 bytes associate to each board (from byte 0 up to byte 119) are used for INPUT/OUTPUT operations of the external resources (according to a characteristic map for each kind of board explained in the specific manual), while the last 8 (from byte 120 up to byte 127) are reserved to the operating system; this does not mean, for advanced and expert programmer, it is not possible to read and write such bytes (with care obviously).

Leaving the details to the specifics documentation for each single board, we analyse in detail the meaning of the dedicated bytes to the operating system. The Table 1 summarizes such informations:

RESERVED BYTES OF THE EXTERNAL RESOURCES		
RESERVED BYTE	DESCRIPTION	VALUE
*.120	Total number of output bytes of the SLAVE board	0 - 120
*.121	Total number of input bytes of the SLAVE board	0 - 120
*.122	Beginning address of the output bytes of the SLAVE board	0 - 119
*.123	Beginning address of the input bytes of the SLAVE board	0 - 119
*.124	Byte 0 identification code SLAVE	0 - 255
*.125	Byte 1 identification code SLAVE	0 - 255
*.126	Byte 2 identification code SLAVE	0 - 255
*.127	Byte 3 identification code SLAVE	0 - 255

Table 1. Meaning of the reserved bytes

Those bytes, used by the operating system of the MASTER for the communication operations with the SLAVES, can be modified by the user program with the normal instructions provided by the language.

On power-on of the whole system, the MASTER makes a call to all the SLAVES starting from the address 1 up to 31, independently from the material presence of them. With such a call the MASTER establishes the system configuration, so that it can manage in an efficient way all the external resources.

The external resources are communicated between the MASTER and the SLAVES through the net line connecting all the boards. Obviously the faster way to execute the updating operations of the current value of the resources, is that to transmit the minimum necessary number of bytes; the greater part of the SLAVES will have at disposal resources consisting of few bytes, so it is not absolutely necessary to communicate all the available 120 bytes of the area associate to each board. In addition it is completely vain to communicate with those addresses of SLAVES, on which is not material present any board.

For this reason, on power on, the MASTER tries to communicate with all the possible 31 SLAVES boards; each SLAVE board, found present on the net, will give to the MASTER the 8 bytes containing all the informations of the area reserved to the operating system. Making reference to the Table 1, each SLAVE present will transmit to the MASTER the values to store in the bytes addressed in the positions 120÷127.

Those default values will characterize all the following updating of the SLAVES' S resources. This preset is done automatically by the system on power-on; like already said, an expert user programmer can change those communications parameters, for example to stop, under certain conditions, the updating of some SLAVES' S resources, with the purpose of reducing the execution time of the program.

The reading by the MASTER of the bytes 124 e 127 allows to have, in the user program, a clear configuration of the hardware conditions, which allows to implement control and alarm functions on the state of the plant.

Between these last four reserved bytes, one of them has a special role; the byte of address 127 holds indeed, in the position 2^0 (LSB), the communication error bit of the related SLAVE. Normally the bit *.127.0 has a logic value “0”; this means a correct communication between MASTER and SLAVE.

In the case the operating system suddenly detects a communication error, this bit will automatically set to the logic state “1”, corresponding to the communication alarm signal for the particular SLAVE. This bit is stored in such a state even if the communication will begin again his normal activity; the user program will have to use this bit, indicating the anomaly (when required) and later reset it. This last operation will be possible obviously only if the communication has already started again its regular functions.

System internal resources

The internal resources of the system are all on the MASTER and they practically consist of devices recalling particular physical elements present in a plant of automation, but not material present on the MASTER. These are realized by means software simulation and so they are often very powerful too.

Internal resources are for example the work memories of the user program, both considered as bytes and as bits; internal resources are also counters, timers, pulse generators on falling or rising edge, etc.

Like for the external resources, it is necessary, also for the internal resources, to identify a symbolic terminology to classify them in an univocal way.

This time is not necessary to specify the board number because the internal resources are all on the MASTER and then we consider useless the number 0 board identifier.

To avoid confusion inside the ensemble of several devices of the internal resources, it is convenient to use, as first field of the identifier, a CHARACTER (or a CHARACTERS STRING), typical of the resource type:

M	for not retentive memory (reset on POWER ON)
H	for retentive memory (with battery)
C	for 16 bits counters
P	for pulse generators on OFF/ON and ON/OFF edge
T	for the periodic oscillating bits
F	for special flags
K	for the byte constants
SXS	for the cycles per second counter
W	for the watch/calendar in real time
X	for the extended retentive memory (available only on some Logic Controllers)

For analogy with the external resources, following that character, it is convenient to put, separated by the character . (full stop) a second field, because the system will place at Programmer’s disposal several devices for each type.

For example:

M.0	not retentive memory byte number 0
M.25	not retentive memory byte number 25
M.1023	not retentive memory byte number 1023
H.0	retentive memory byte number 0
H.89	retentive memory byte number 89
H.1023	retentive memory byte number 1023

C.0	counter number 0
C.127	counter number 127
P.0	pulse generator number 0
P.127	pulse generator number 127
W.HOUR	byte for actual value hours of the day
W.DAY	byte for actual value day of the week
W.YEAR	byte for actual value of the year
X.0	extended retentive memory byte number 0
X.24567	extended retentive memory byte number 24567

At the end, in according to the device type, a possible following field will be able to identify a particular resource. For example, to identify the single bit inside a byte of type M, H and X it is enough to add a further numeric field from 0 to 7 (separating it with the character full stop), following so the same procedure used to identify the single bit inside the byte of a external resource. To identify the bytes or the bits, specific of the other resources, you have to add a proper identifier field; details will be developed in the next sections.

Analysis of the internal resources

The internal resources are peculiar only to the MASTER and they are formed by the ensemble of the devices and elements that the software makes available to the Programmer. These devices are very seldom material present on the MASTER board, because they are realized by means SOFTWARE simulation by the operating system.

Previously we have seen how to classify such elements by means the first two fields of the identifier. For each of the elements of the internal resources, we can define an ensemble of byte type and bit type operands, which represent its proper functions; so each type of internal resource will be represented by a identifier formed by two or three fields, separated by the full stop character, studied in the most appropriate way for each single type of device.

For the memories of M, H and X type (available only on some Logic Controllers) a representation in two fields always means a byte type operand, while a representation in three fields always means a bit type operand: for such resources the way of identification is the same used for the external resources.

Keep in mind that the numeric fields of the identifiers must not necessary have always the same number of digits, putting before some zeroes; this means, for example, for the byte of memory number 3 we can write indifferently M.3, M.03, M.003 or M.0003.

The COUNTER device

Regarding the COUNTER type devices (the first field is C), the second field is the considered device number; the third field, always present, allows to address the single bits or single bytes characterizing the counter device. Each of the possible 128 counters (operating at 16 bits) is implemented by means the use of 5 bytes of memory, the first of which (byte CB “control byte”) represents the byte of CONTROL of the counter, while the other 4, two pairs of byte, are the CURRENT value of the count (CL, CH) and the FINAL value of the count (FL, FH).

Indicating generically with * the considered device number, we summarize the meaning of the single bytes of the counter device:

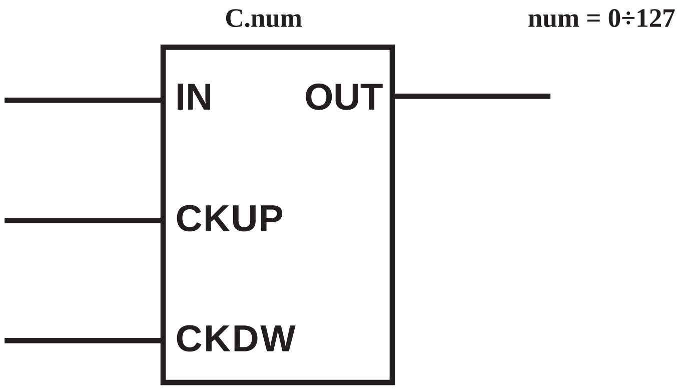
C.*.CB	control byte of the counter
C.*.CL	byte LOW of the current value of the count
C.*.CH	byte HIGH of the current value of the count
C.*.FL	byte LOW of the final value of preset of the count
C.*.FH	byte HIGH of the final value of preset of the count

Inside the control byte C.*.CB the meaning of the single bits is the following:

C.*.IN	count enable bit
C.*.OUT	count stop bit (CURRENT = PRESET)
C.*.CKUP	counter clock up bit
C.*.CKDW	counter clock down bit

The counter, when finds the input C.*.IN in the logic state 1, starts to count the number of pulses detected in C.*.CKUP and in C.*.CKDW, increasing and reducing respectively the 16 bits counter register (CL, CH). When this count reach the preset final value of the 16 bits register (FL, FH), the output C.*.OUT goes ON (logic state 1); only when the input C.*.IN goes OFF (logic state 0), the counter is reset and also the output goes OFF.

The counter device can be represented as an element with 3 inputs (IN, CKUP, CKDW) and 1 output (OUT):



FL = PRESET LOW
FH = PRESET HIGH

The input terminals are supplied with normal output bit instructions (OUT, OUTNOT, SET, RES, CPL); if the count is only UP or only DOWN, you can leave not defined the input CKDW or CKUP respectively.

The output terminal OUT (stop counting) can be used to supply the other parts of the net, using it as operator of normal boolean instructions to read the bit (LD, LDNOT, AND, ANDNOT, OR, ORNOT).

The preset value (FH for the byte High, FL for the byte Low) must be instead defined by the user program by means any instruction forcing the byte value or a pair of bytes (for example MOV1, MOV2).

The counter allows also to realize a TIMER device, simply connecting the input CKUP to one of the periodic oscillator bits with predetermined period. The input IN allows to supply the TIMER with function “ EXCITATION DELAY” function, while the output OUT is the signal of elapsed time (the input CKDW must be open).

To do those things, among the internal resources of the system, is available for the Programmer an ensemble of 6 bits with fixed periodic oscillations, belonging to the same byte (identified with the character T):

T.50	oscillator bit with period 50 ms
T.100	oscillator bit with period 100 ms
T.200	oscillator bit with period 200 ms
T.500	oscillator bit with period 500 ms
T.1000	oscillator bit with period 1000 ms
T.2000	oscillator bit with period 2000 ms

Those 6 different TIME BASES allow to realize TIMERS with times starting from 0" to 2184.5 h.

The bits with fixed periodical oscillation can be used to make the outputs of the system blink, like for example in the case of light signals; to do this it is enough to connect in series (by means the instruction AND) one of those bits in the line supplying the output.

We suggest, for the identifiers of the counters, to use increasing numbers starting from the number 0, avoiding jumps in the numeration; this allows to get by the operating system the best performance in the execution speed of the user program. The compiler indeed transmit to the operating system the maximum number of identifiers found in the program; in this way all the devices with higher number are not handled, with advantage for the execution time of the machine cycle obviously. You can consider that this advice has only the purpose of training the Programmer to get the maximum performances, if this rule does not limit his way of working, but also without this standard the system warrants a very fast updating of the counters.

In the ICL51 language are present also two particular instructions that allows to use in a easy and quick way the counter device; they are the CNT and TIM instructions respectively to manage such devices as real counters and to use them as timers with a fixed 100 ms time base. For details we refer to the specific description of the instructions CNT and TIM.

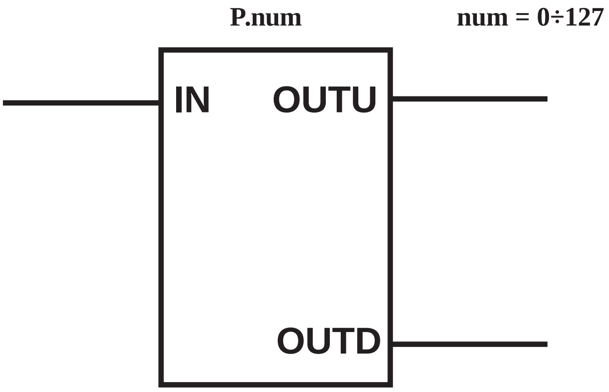
PULSE GENERATOR device

The pulse generator device on rising or falling edge of the input signal are other useful devices for programming.

While the first field is the character P, the second field of the identifier means the generator device number (the maximum number of devices is 128); the third field identifies the input bit or the output bit.

So the identifiers are only three bits (one for input P.*.IN and two for output P.*.OUTU and P.*.OUTD) and with them it is possible to detect the transition of a signal from OFF to ON value (on the output OUTU) or from ON to OFF value (on the output OUTD). When the input bit changes its state, on the proper output bit we have a ON signal for all the following program cycle.

The pulse generator on change of input signal can be represented with the following element:



The input IN is supplied by means a normal output instruction (OUT, OUTNOT, SET, RES, CPL); the outputs OUTU and OUTD can be read by the normal boolean operations (LD, LDNOT, AND, ANDNOT, OR, ORNOT).

Every time in the input IN a change from OFF to ON state is detected, the output OUTU goes ON for a program cycle; if on the contrary on the input you have a transition from ON to OFF, the output OUTD goes ON for a program cycle. In this way the same device can be used both to detect rising edge and to detect falling edge of a certain input signal.

We suggest, for the identifiers of pulse generators, to use increasing numbers starting from the number 0, avoiding jumps in the numeration; this allows to get from the operating system the maximum performances in execution speed of the user program. The compiler actually transfers to the operating system the maximum number of identifier found in the program; in this way all the device with a higher number are not handled, with advantage obviously for the execution time of the machine cycle. You can consider that this advice has only the purpose of training the Programmer to get the maximum performances, if this rule does not limit his way of working, but also without this standard the system warrants a very fast updating of the pulse generators.

Special bits of FLAG

There are other internal resources with quite particular characteristics; we mean the type F.* resources made by a single byte F whose has a particular function.

The type F.* resources correspond to an ensemble of type bit operands (signal flag).

The resources are the following:

F.0	bit always OFF
F.1	bit always ON
F.P	pulse ON in the first program cycle
F.<	bit ON if compare result is <
F.=	bit ON if compare result is =
F.>	bit ON if compare result is >
F.C	carry bit
F.E	error bit

The bits F.0 and F.1 have the meaning of boolean constant; particularly F.1 can be loaded by an instruction LD before bytes operations if you want the Logic Controller executes them every program cycle and not conditionally as would be the rule.

The F.P bit correspond to a pulse ON during all the first program cycle: this bit allows to initialize the program when you give power to the machine.

The F.<, F.= ed F.> allow to test the result of a compare instruction among bytes; those bits must be read immediately after the compare instruction (CMP1, CMP2, CMP4, CMP, ?, CMPBLK), to know the its result: actually the following compare instruction will affect those bits.

The F.C bit corresponds to a possible carry after an add or subtraction operation. It is also useful as input and output for the SFR (shift) operation, realizing, in this way, SHIFT REGISTER of any length (multiple of a byte).

The F.E bit is ON if after a multiplication instruction there is a overflow, or if the divisor is zero before a division instruction; this flag is also set to a logic “1” if there are errors during the calculation of an mathematic expression with the method of the reverse polish notation.

1/2/4 bytes CONSTANTS

The K.* type resources are constants operands; a classic use is to preset bytes with a final value of the counters. The second field of the identifier represent directly the integer constant value, expressed in three possible bases of representation: decimal, hexadecimal and binary.

For the decimal constants you must put only the symbol K. before the number. For the binary constants, the character B must follow the binary value (characters 0 and 1). Finally for the hexadecimal constants (characters 0-9, A, B, C, D, E, F) we use as suffix the character H. All the constants are preceded by the symbol K.

For the negative integer numbers, the character - (minus) must precede the decimal value; the negative decimal constant will automatically converted in the two-complement binary representation. For the negative constant directly in binary form you must give the binary digits 0 and 1 with the two-complement representation of the number.

Some examples of constant operands are the following:

K.0	decimal constant 0
K.-128	decimal constant -128
K.125	decimal constant 125
K.-31627	decimal constant -31627
K.2312634	decimal constant 2312634
K.-452312634	decimal constant -452312634
K.10010011B	binary constant 10010011
K.001111101010B	binary constant 001111101010
K.3EFH	exadecimal constant 3EF
K.E34FA4C2H	exadecimal constant E34FA4C2

The constant value expressed in the second field, independently from the representation base, must not exceed the limits allowed by the used instruction.

All the instructions for handling bytes are divided in three classes, according to the size of the used entity. For example to transfer bytes are present 3 instructions: MOV1 allows to transfer from a single byte to another byte, MOV2 transfers two consecutive bytes, MOV4 transfers four consecutive bytes (in the operations requiring more than one byte, the operands after the instruction, represent the least significant byte of the variables).

The instructions with constant operands cannot distinguish if they are absolute unsigned or two-complement representation of a negative constant and consequently the validity fields are the following:

1 BYTE:	-128.....0.....127.....255
2 BYTES:	-32768.....0.....32767.....65535
4 BYTES:	-2147483648.....0.....2147483647.....4294967295

The preceding validity fields are expressed in decimal base; in binary representation the constants can have respectively at most 8, 16, 32 binary digits, while in exadecimal representation 2, 4, 8 exadecimal digits.

In any case a violation of these limits, according to the dimensions of the instruction, is indicated by the compiler.

SXS indicator of the cycles per second

The internal resource of type SXS consists in a two bytes variable of the same name whose value is updated by the operating system every second. This variable stores, in this period of time, the copy of the preceding final counting value of a counter increased of a unit every program cycle.

The use of this variable is generally limited to a exclusively informative function and of control of the processing speed of the developed program; higher is this value, better the program is optimized in the reply speed.

To visualize this value you can use the MONITOR on-line function of the develop environment, recalling the monitoring of 2 bytes from SXS; bear in mind, in this connection, that the on-line link of the Computer to the MASTER board, because of the data exchange between the units, makes slower the execution process of the user program, reducing the value of SXS. In this way the reading of this value is a particularly pessimistic evaluation of the program cycle speed: for certain the number of program cycles per second, during the normal functioning of the MASTER without Computer connected, will be higher.

At this point you can ask what is the utility of SXS if, to read its value, you have to modify it: this variable can be read correctly sending its value to I/O channels of the system, like for example a terminal with DISPLAY connected in a net. In this way it is possible to visualize this value, without connecting to the Computer.

A use of this variable inside the user program, could be that to control the maximum time of cycle, generating some alarms, in the case of getting over certain fixed limits, before the occurrence of the circuit of WATCHDOG hardware of the MASTER board.

The WATCH/CALENDAR

The internal resources of type W places at Programmer's disposal a complete and exact watch/calendar in real time, quartz controlled.

This is the only example of internal resource connected to the presence of specific hardware devices on the MASTER board and not realized by means software simulation.

We specify that this type of resource is an OPTION of the MASTER board, because it can be installed easily later (if a MASTER board with this option already installed was not required). The presence of this resource causes an extra charge of the MASTER board, because it needs the RAM memory of a ZERO-POWER TIMEKEEPER type, whose cost is higher, instead of a normal ZERO-POWER type.

The type W resources can be resumed in a ensemble of 8 bytes of the user RAM memory. To identify these bytes you must insert, after the common full-stop separator, the following fields:

W.CB	control byte of the watch/calendar
W.SEC	byte decimal value of the seconds (0-59)
W.MIN	byte decimal value of the minutes (0-59)
W.HOUR	byte decimal value of the hours (0-23)
W.DAY	byte decimal value of the day of the week (1-7)
W.DATE	byte decimal value of the day of the month (1-31)
W.MTH	byte decimal value of the month (1-12)
W.YEAR	byte decimal value of the year (0-99)

The meaning of the bytes from W.SEC to W.YEAR is clear: they store the current value of the hour and the date in decimal notation. We want to specify that the MSB (most significant bit) of the byte of the seconds is the STOP bit of the quartz oscillator of the watch; so pay attention to not preset this byte with overflow values (0-59), otherwise it is possible this bit is forced to the logic value 1 with the following stop of the watch/calendar.

Analyse the details of the control byte W.CB; this byte uses only one bit (the least significant bit 2^0), to preset the hour and the date:

W.CB.ADJ adjustment bit of the watch/calendar

This bit allows to preset the value of all 7 bytes of the watch/calendar. The operating system, every program cycle, controls the value of this bit; in accordance to its value, it works consequently.

If the value of W.CB.ADJ is a logic "0", the watch/calendar is in a normal working mode, it corresponds to the reading possibility, by the user program, of the current hour and date; in this case the system updates this area of memory with the current value present in the watch/calendar (contained in the in the RAM TIMEKEEPER device).

If the value of W.CB.ADJ is a logic “1”, the watch/calendar is in the setting mode of the hour and the date. The system finding this bit ON stops the updating of the user RAM area, because it is waiting the user to force the values in those bytes at the purpose to preset the watch/calendar. When the adjustment bit is again OFF, the operating system transfers the forced value from the user RAM to the RAM TIMEKEEPER device, returning in the normal mode of reading the hour/date.

Normally the bit W.CB.ADJ must be in the OFF state, as it is, every time the MASTER board is powered.

The preset handling of a new time is left to the Programmer by means the use of the bit W.CB.ADJ; if the whole system does not consider particular SLAVE boards, like a terminal with display (which, with appropriate sections of the user program, could realize the man/machine interface with the watch/calendar), the only way to preset the time is to use the develop environment on Personal Computer, which places at disposal a proper utility to preset the watch/calendar.

The instruction set

Preliminary concepts

In the following paragraphs will be done a detailed analysis of the available instructions of the language.

As we have already mentioned, a program is made by the sequence of a certain number of instructions; each instruction fills a row of the edited source file and it will produce with the compilation process a portion of object code executable by the MASTER board. A large number of instructions, to be complete, needs a certain number of operands: describing the resources of the system we have already introduced the operands and we have defined a manner to identify them.

Excluding some of them of particular or special use, most of the instructions can be divided in two principal classes: the boolean instructions (which work on the single bit) and the bytes handling instructions.

The boolean instructions are very much used because they allow to describe an equivalent electromechanical net which correspond to the functioning requirements of the automatic machine. These instructions realize boolean operations on the resources (external and internal) of the system. The calculation method of the value of the net node uses an accumulator register, transparent for the Programmer, allowing the temporary storage of the intermediate values of the calculation.

The accumulator register is like a list of single bits of LIFO type (Last In First Out), in which enter the temporary values of the calculation of the net every time you use the LD or LDNOT instruction. The maximum nesting of the list is 8 bits, that allows to store the intermediate result of 8 lines of the net, before of possible series and parallels between the same lines by means the instructions ANDLD and ORLD. These instructions realize respectively the series and parallel function of two intermediate results (between the first and the second position to exit from the list) and, after positioning the result in the first position of the list, they shift of a place all the other possible intermediate results (from the third to the second, from the fourth to the third, etc.).

The first bit to exit from the list LIFO, is used to supply the output instructions; this bit is not affected by the output operations, therefore it is possible to supply, with the same calculated line, any number of outputs and devices.

The instructions of bytes handling work instead on sizes of 1, 2 e 4 bytes; they allow to do calculations on variables and parameters of the system. Through the comparing instruction it is possible to control the assumed value of these data and consequently to make the program goes ahead. A frequent use of these instructions is to preset final values of the counters and of the timers and to process analogic data of the system.

The bytes variables can be of three types in according to their size; the 1 byte variables are of the smallest size and they are indicated by the name or the Label of the byte. The variables of 2 and 4 bytes are instead made by di 2 e 4 consecutive bytes and their identification occurs by means the Label of the first byte (lower address) which is the least significant byte of the variable. In the Table 2 are reported the modification field of the variables in the three different sizes.

TYPE	MODIFICATION FIELD	
	POSITIVE INTEGER	SIGNED INTEGER
1 BYTE	0........255	-128........+127
2 BYTES	0........65535	-32768........+32767
4 BYTES	0........4294967295	-2147483648...+2147483647

Table 2. Variation fields of the 1/2/4 bytes variables

Between bytes instructions some merit a particular attention. They are the instructions RCL1, RCL2, RCL4, ADD, SUB, MUL, DIV, CMP, STO1, STO2 and STO4 for the evaluation of mathematic expressions on variables of 1, 2, 4 bytes (also in a mixed way) in according with the "Reverse Polish Notation". These instructions represent an efficient option to the mathematic instructions of the automation languages. We can advance that this evaluation technique of the expressions is very similar to that just described for the boolean expression evaluation on the single bits; also in this case it is present a list LIFO in which signed 32 bits variables are temporary stored as intermediate results of the expression (up to 4 nesting levels). We remand to the proper paragraph for details of this technique for evaluation of mathematic expression.

Generally we consider that the output instructions (i.e. the instruction forcing with writing the bits or the bytes, cannot be used for all the operands; all the "constant", the special operands (oscillators, some flags) and the outputs already supplied by the internal devices are actually excluded. In any case will be the compiler to analyze the validity of application of the instruction and to signal to the Programmer the possible inconsistency of some instructions with some type of operands.

The instructions more frequently used, are provided of a dual shortened mnemonic, for a faster writing of the source program; this alternative mnemonic will be showed at the voice "shortened mnemonic" of the specific instruction.

LD

Load the ON/OFF value of the normally open contact

Syntax

```
LD          Bit Operand                 ‘comment
```

Argument

Bit Operand is the address or the related label of any bit in the ram memory.

Description

LD load in the bits accumulator register the ON/OFF value of the normally open contact indicated by the argument; the bits list of the accumulator register automatically increases of a level. This is generally the first of the list instructions which describe a line of the electromechanical net.

Example

The following example explains the loading of the bit 0.0.0 and the copy on the bit 0.8.0:

```
LD          0.0.0
OUT         0.8.0
```

Shortened mnemonic

```
L           Bit Operand                 ‘comment
```

See also

LDNOT

LDNOT

Load the ON/OFF value of the normally closed contact

Syntax

```
LDNOT     Bit Operand                  ‘comment
```

Argument

Bit Operand is the address or the related label of any bit in the ram memory.

Description

LDNOT load in the bits accumulator register the ON/OFF value of the normally closed contact indicated by the argument; the bits list of the accumulator register automatically increases of a level. This is generally the first of the list instructions which describe a line of the electromechanical net.

Example

The following example explains the inverted loading of the bit 0.0.0 and the copy on the bit 0.8.0:

```
LDNOT     0.0.0
OUT       0.8.0
```

Shortened mnemonic

```
LN        Bit Operand                  ‘comment
```

See also

LD

AND

Logical and with the value ON/OFF of the normally open contact

Syntax

```
AND      Bit Operand         ‘comment
```

Argument

Bit Operand is the address or the related label of any bit in the ram memory.

Description

AND performs the logical and between the last bit inserted in the bits accumulator register and the value ON/OFF of the normally open contact indicated in the argument. The result is overwritten on the last inserted bit of the list in the accumulator register. A bit in the result is set if the corresponding original bits are set, otherwise the bit is cleared.

Example

The following example performs the and of the bit 0.0.0 with the bit 0.0.1. The result is then copied in the bit 0.8.0:

```
LD       0.0.0
AND      0.0.1
OUT      0.8.0
```

Shortened mnemonic

```
A        Bit Operand              ‘comment
```

See also

ANDNOT

ANDNOT

Logical and with the value ON/OFF of the normally closed contact

Syntax

```
ANDNOT   Bit Operand          ‘comment
```

Argument

Bit Operand is the address or the related label of any bit in the ram memory.

Description

ANDNOT performs the logical and between the last bit inserted in the bits accumulator register and the value ON/OFF of the normally closed contact indicated in the argument. The result is overwritten on the last inserted bit of the list in the accumulator register. A bit in the result is set if the corresponding original bit in the list is set and the operand bit is not set, otherwise the bit is cleared.

Example

The following example performs the and of the bit 0.0.0 with the inverted bit 0.0.1. The result is then copied in the bit 0.8.0:

```
LD       0.0.0
ANDNOT   0.0.1
OUT      0.8.0
```

Shortened mnemonic

```
AN       Bit Operand               ‘comment
```

See also

AND

OR

Logical or with the value ON/OFF of the normally open contact

Syntax

```
OR          Bit Operand            ‘comment
```

Argument

Bit Operand is the address or the related label of any bit in the ram memory.

Description

OR performs the logical or between the last bit inserted in the bits accumulator register and the value ON/OFF of the normally open contact indicated in the argument. The result is overwritten on the last inserted bit of the list in the accumulator register. A bit in the result is set if either or both corresponding original bits are set, otherwise the bit is cleared.

Example

The following example performs the or of the bit 0.0.0 with the bit 0.0.1. The result is then copied in the bit 0.8.0:

```
LD          0.0.0
OR          0.0.1
OUT         0.8.0
```

Shortened mnemonic

```
O           Bit Operand                  ‘comment
```

See also

ORNOT

ORNOT

Logical or with the value ON/OFF of the normally closed contact

Syntax

```
ORNOT    Bit Operand          ‘comment
```

Argument

Bit Operand is the address or the related label of any bit in the ram memory.

Description

ORNOT performs the logical or between the last bit inserted in the bits accumulator register and the value ON/OFF of the normally closed contact indicated in the argument. The result is overwritten on the last inserted bit of the list in the accumulator register. A bit in the result is set if or the accumulator bit is set, or the operand bit is not set, otherwise the bit is cleared.

Example

The following example performs the or of the bit 0.0.0 with the inverted bit 0.0.1. The result is then copied in the bit 0.8.0:

```
LD       0.0.0
ORNOT    0.0.1
OUT      0.8.0
```

Shortened mnemonic

```
ON       Bit Operand               ‘comment
```

See also

ORNOT

ANDLD

Logical and between two intermediate results

Syntax

```
ANDLD                         ‘comment
```

Argument

None.

Description

ANDLD performs the logical and between the last two bits inserted in the bit accumulator register. The bits list present in the accumulator register decreases automatically of one level and the result is in the last bit in the list of the accumulator register.

Example

The following example performs the logical or between the bit 0.0.0 and the bit 0.0.1 and the logical or between the bit 0.0.2 and the 0.0.3. The corresponding results are “ANDed” and the final result is copied in the bit 0.8.0:

```
LD       0.0.0              ‘first or
OR       0.0.1
LD       0.0.2              ‘second or
OR       0.0.3
ANDLD                       ‘and of or
OUT      0.8.0
```

Shortened mnemonic

```
AL                            ‘comment
```

See also

ORLD

ORLD

Logical or between two intermediate results

Syntax

```
ORLD                          ‘comment
```

Argument

None.

Description

ORLD performs the logical or between the last two bits inserted in the bit accumulator register. The bits list present in the accumulator register decreases automatically of one level and the result is in the last bit in the list of the accumulator register.

Example

The following example performs the logical and between the bit 0.0.0 and the bit 0.0.1 and the logical and between the bit 0.0.2 and the 0.0.3. The corresponding results are “ORed” and the final result is copied in the bit 0.8.0:

```
LD      0.0.0                 ‘first or
AND     0.0.1
LD      0.0.2                 ‘second or
AND     0.0.3
ORLD                          ‘or of and
OUT     0.8.0
```

Shortened mnemonic

```
AL                            ‘comment
```

See also

ANDLD

OUT

Transfers a bit to an output port

Syntax

```
OUT         Bit Operand         ‘comment
```

Argument

Bit Operand is the address or the related label of any bit in the ram memory.

Description

OUT copies the value ON/OFF of the last bit inserted in the accumulator register on the bit specified by the argument.
The bits list of the accumulator register is not affected and then it is possible to transfer the same bit to several bits of an output port.

Example

The following example copies the bit 0.0.0 on both the bits 0.8.0 and 0.8.1:

```
LD          0.0.0
OUT         0.8.0
OUT         0.8.1
```

Shortened mnemonic

```
=           Bit Operand         ‘comment
```

See also

OUTNOT

OUTNOT

Transfers an inverted bit to an output port

Syntax

```
OUTNOT          Bit Operand          ‘comment
```

Argument

Bit Operand is the address or the related label of any bit in the ram memory.

Description

OUTNOT copies the inverted value ON/OFF of the last bit inserted in the accumulator register on the bit specified by the argument.
The bits list of the accumulator register is not affected and then it is possible to transfer the same bit to several bits of an output port.

Example

The following example copies the inverted bit 0.0.0 on both the bits 0.8.0 e 0.8.1:

```
LD      0.0.0
OUTNOT  0.8.0
OUTNOT  0.8.1
```

Shortened mnemonic

```
=N          Bit Operand          ‘comment
```

See also

OUT

SET

Sets a bit of an output port

Syntax

```
SET     Bit Operand        ‘comment
```

Argument

Bit Operand is the address or the related label of any bit in the ram memory.

Description

SET forces the logic value “1” on the bit specified by the argument only if also the last bit inserted in the accumulator register is “1”. The instruction SET sets the bit, you must use the instruction RES to reset the same bit; in this case the second instruction in the program list will have the priority on the other.
The bits list of the accumulator register is not affected and then it is possible to set several bits with the same tested bit.

Example

The following example sets the bit 0.8.0 if the bit 0.0.0 is ON:

```
LD      0.0.0
SET     0.8.0
```

Shortened mnemonic

```
S       Bit Operand        ‘comment
```

See also

RES, CPL

RES

Resets a bit of an output port

Syntax

```
RES         Bit Operand            ‘comment
```

Argument

Bit Operand is the address or the related label of any bit in the ram memory.

Description

RES forces the logic value “0” on the bit specified by the argument only if also the last bit inserted in the accumulator register is “1”. The instruction RES resets the bit, you must use the instruction SET to set the same bit; in this case the second instruction in the program list will have the priority on the other.
The bits list of the accumulator register is not affected and then it is possible to set several bits with the same tested bit.

Example

The following example sets the bit 0.8.0 if the bit 0.0.0 is ON:

```
LD          0.0.0
SET         0.8.0
```

Shortened mnemonic

```
R           Bit Operand            ‘comment
```

See also

SET, CPL

CPL

Complements a bit of an output port

Syntax

```
CPL      Bit Operand        ‘comment
```

Argument

Bit Operand is the address or the related label of any bit in the ram memory.

Description

CPL complements (inversion of state) the logic value of the bit specified by the argument only if also the last bit inserted in the accumulator register is “1”. With the instruction CPL the bit changes its state every time the instruction is executed, in this case you must execute the instruction testing a pulse generator during a program cycle otherwise the bit will change state every program cycle.

The bits list of the accumulator register is not affected and then it is possible to complement several bits with the same tested bit.

Example

The following example complements the bit 0.8.0 on the rising edge of the bit 0.0.0 is ON:

```
LD       0.0.0          ‘set the pulse generator
OUT      P.0.IN

LD       P.0.OUTU       ‘CPL is executed on the rising edge
CPL      0.8.0
```

Shortened mnemonic

```
C        Bit Operand        ‘comment
```

See also

SET, RES

JMP

Jump to the next JME if the condition tested is ON

Syntax

```
JMP                              ‘comment
```

Argument

None.

Description

JMP jump to the next instruction JME if the last bit inserted in the accumulator register is “1”. This instruction allows to skip pieces of program on certain conditions, for example to divide the whole program in parts to execute alternately or to accelerate the program cycle excluding not active parts of program in the current phase.

Example

The following example excludes the indicated piece of program as long as the bit 0.0.0 is “1”:

```
LD      0.0.0                    ‘condition enabling the jump
JMP

LD      T.500                    ‘piece of program excluded by the jump
OUT     0.8.0

JME                              ‘target location
```

See also

JME

JME

Target location of the conditionally jump instruction JMP

Syntax

```
JME                                ‘comment
```

Argument

None

Description

JME is the target location of the jump of the preceding instruction JMP. The pair of instructions JMP/JME allows to skip whole parts of program on certain conditions. You must remember the instructions JMP and JME can be used in pair in any number inside the program, but those jumps cannot be nested: an instruction JMP requires later its JME, before beginning an other block of jump.

Example

The following example excludes the indicated piece of program as long as the bit 0.0.0 is “1”:

```
LD      0.0.0              ‘condition enabling the jump
JMP

LD      T.500              ‘piece of program excluded by the jump
OUT     0.8.0

JME                        ‘target location
```

See also

JMP

GOTO

Jump to the Label if the condition is ON

Syntax

```
GOTO        Label                       ‘comment
```

Argument

Label is a text string with 32 characters maximum and without blanks inside.

Description

GOTO jumps to the Label indicated in the argument if the last bit inserted in the accumulator register is “1”. The instruction GOTO is very similar to the instruction JMP; the only difference is that the JMP does not require to specify a particular Label where to jump, because the jump is always to the following instruction JME, while the instruction GOTO requires always as argument Label name where to jump. The Label of the target location can be placed everywhere in the program list (also before the GOTO) provided always followed by the character “:” (colon) and must be the only instruction in the line. GOTO can be nested without limits because GOTO jumps always in the indicated place.

Example

The following example excludes the dashed piece of program as long as the bit 0.0.0 is “1”:

```
LD          0.0.0                       ‘condition enabling the jump
GOTO        Label_where_to_jump

-----------------------
-----------------------

Label_where_to_jump:                    ‘Label of the target location of the jump
```

See also

JMP

GOSUB

Call a subroutine if the condition is ON

Syntax

```
GOSUB    Label                ‘comment
```

Argument

Label is a text string with 32 characters maximum and without blanks inside.

Description

GOSUB executes the subroutine indicated in the argument Label if the last bit inserted in the accumulator register is “1”. The beginning Label of the subroutine must be placed after the instruction END of the main program, in a empty line and it must be always followed by the character “:” (colon). After the Label you can put the instruction list of the subroutine ending it with a new instruction END or with the instruction RET. Inside the subroutine you can use all the instruction of the language; it is possible to use also GOTO and GOSUB and their Labels. You can have up to 16 nesting levels of the subroutines.

Example

The following example executes the subroutine “Handling_Alarm_0” as long as the bit 0.0.0 is “1”, while it executes the subroutine “Handling_Alarm_1” as long as the bit 0.0.1 is “1”:

```
----------------------
main program
----------------------

LD       0.0.0                ‘condition enabling the subroutine
GOSUB    Handling_Alarm_0

LD       0.0.1                ‘condition enabling the subroutine
GOSUB    Handling_Alarm_1

----------------------
main program
----------------------
END
```

```
Handling_Alarm_0:               ‘beginning Label of the subroutine

LD        M.0.0.0               ‘enabling another nested subroutine
GOSUB     Alarm_Verify
------------------------
subroutine list
------------------------
END



Handling_Alarm_1:               ‘beginning Label of the subroutine

LD        M.0.0.1               ‘enabling another nested subroutine
GOSUB     Alarm_Verify
------------------------
subroutine list
------------------------
END



Alarm_Verify:                   ‘beginning label of the nested subroutine
------------------------
nested subroutine list
------------------------
END
```

See also

END, RET

NOP

No operation instruction

Syntax

```
NOP                                        ‘comment
```

Argument

None.

Description

NOP is not a really proper instruction because it does nothing inside the program. It is a fictitious instruction you can insert, for example, temporary in the program to remember a deleted instruction. This instruction inserts a fixed delay of 1 µs in the cycle.

Example

The following example shows the insertion of the instruction NOP between other two instructions:

```
LD          0.0.0
NOP
OUT         0.8.0
```

END

End of the program list

Syntax

```
END                                   ‘comment
```

Argument

None.

Description

END is the last instruction of the program list. This one allows the operating system to update both the internal and the external resources and to begin again another cycle of program.
If you use subroutines this instruction must be used to end the single parts of them; the subroutines must be placed at the end of the main program after the instruction END. Each subroutine must begin with a Label (ending with the character “:”) and finish with its own instruction END.

Example

The following example shows the use of the instruction END in the subroutines:

```
--------------------------
main program list
--------------------------
END

Subroutine1:
--------------------------
subroutine 1 list
--------------------------
END

Subroutine2:
--------------------------
subroutine 2 list
--------------------------
END
```

See also

GOSUB, RET

RET

End of the program list of a subroutine

Syntax

```
RET                              ‘comment
```

Argument

None.

Description

RET is the last instruction of the program list of a subroutine. Actually the instruction RET is perfectly equivalent and replaceable to the instruction END, therefore you might use it to end the main program. You can think also the main program as a subroutine called this time by the operating system every scan cycle. But we suggest to use the instruction RET only for the subroutine to distinguish them from the main program.

Example

The following example shows the use of the instruction RET:

```
--------------------------
main program list
--------------------------
END

Subroutine1:
--------------------------
subroutine 1 list
--------------------------
RET

Subroutine2:
--------------------------
subroutine 2 list
--------------------------
RET
```

See also

GOSUB, END

TIM

Timer with time base of 0.1”

Syntax

```
TIM        C.*.IN     Constant        ‘comment
```

Argument

C.*.IN is the start bit of the generic counter (* = 0÷127).
Constant is a numeric constant of 2 bytes maximum dimension (K.0 ÷ K.65535).

Description

The device TIM realizes timers with function delayed on excitation and times from 0 to 6553.5 seconds.

TIM is a easy and fast way to use the counters devices as timers devices. To use a counter as a timer you need to connect the clock input UP to one of the bits of periodic oscillation (type T bits); additionally you need to load the final value of the counter with the wanted constant of time.
The instruction TIM develops all these functions automatically changing a counter device in a timer. The instruction TIM connects automatically inside the value of the last bit inserted in the accumulator register to the start bit of the counter, the periodic oscillation bit T.100 (100 milliseconds) to the clock UP and load the final value of the counter with the provided constant of time (number of required tenths of seconds). The expiration time bit of the timer is then the expiration time bit of the counter.
The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: we suggest, where possible, to use the instruction TIM instead of managing entirely the counter; in this way you can save memory and the development speed is higher.

Example

The following example compares the whole use of the counter to realize a timer with its compact handling by means the instruction TIM:

```
LD      0.0.0                   ‘enabling timer signal
OUT     C.0.IN                  ‘counter device enable
MOV2    C.0.FL      K.30        ‘preset of the final value
LD      T.100                   ‘0.1” periodic oscillation bit
OUT     C.0.CKUP                ‘clock of counting increment

LD      C.0.OUT                 ‘load the compare value
OUT     0.8.0                   ‘if expired time set the bit 0.8.0
```

using the instruction TIM you can write the preceding part of program in this way:

```
LD      0.0.0                   ‘enabling timer signal
TIM     C.0.IN      K.30        ‘timer device enable

LD      C.0.OUT                 ‘load the timer input
OUT     0.8.0                   ‘if expired time set the bit 0.8.0
```

See also

CNT

CNT

Increment counter

Syntax

CNT	C.*.IN	Bit Operand	Constant	‘comment

Argument

C.*.IN is the start bit of the generic counter (* = 0÷127).
Bit Operand is the address or the related label of the bit to use as clock Up.
Constant is a numeric constant of maximum dimension 2 bytes (K.0 ÷ K.65535).

Description

The device CNT realizes counters with only increment of its value.

CNT is a easy and fast way to use the counters devices when you need only to increment the current value. The CNT instruction connects automatically inside, the value of the last bit inserted in the accumulator register to the start bit of the counter, the bit indicated in the argument to the clock UP and load the final value of the counter with the provided numeric constant.
If you need to decrement the counter, it is possible to use the clock Down input, not used by the instruction CNT.
The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: we suggest, where possible, to use the instruction CNT instead of managing entirely the counter; in this way you can save memory and the development speed is higher.

Example

The following example compares the whole use of the counter with its compact handling by means the instruction CNT:

```
LD      0.0.0                       ‘enabling counter signal
OUT     C.0.IN                      ‘counter device enable
MOV2    C.0.FL     K.30             ‘preset of the final value
LD      0.0.1                       ‘bit edge to count
OUT     C.0.CKUP                    ‘clock of counting increment

LD      C.0.OUT                     ‘load the compare value
OUT     0.8.0                       ‘if expired count set the bit 0.8.0
```

using the instruction CNT you can write the preceding part of program in this way:

```
LD      0.0.0                       ‘enabling counter signal
CNT     C.0.IN     0.0.1   K.30     ‘counter device enable

LD      C.0.OUT                     ‘load the counter input
OUT     0.8.0                       ‘if expired count set the bit 0.8.0
```

See also

TIM

SFR

Shift left of a bit inside a byte

Syntax

SFR Byte Operand ‘comment

Argument

Byte Operand is the address or the related label of any byte of the ram memory.

Description

SFR performs the shift left by a position of the bits in the byte specified in the argument only if the last bit inserted in the accumulator register is “1”. The shift operation affects the carry flag (F.C) because the carry flag goes into the low order bit and the high order bit goes into the carry flag. This allows the cascade connection of more instruction SFR realizing shifts of very long chains also (integer multiples of a byte). The carry flag bit F.C takes the meaning of “data input” of the SHIFT-REGISTER; the execution of the instruction SFR is performed on the “clock ” signal while to “reset” the byte, you must force the constant 0 on the byte.
The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: the instruction SFR performs the shift by a position every program cycle if the condition is on. To perform only one shift corresponding to a certain condition, you must use the pulse generator device as enable condition.

Example

The following example shows the use of the instruction SFR to realize a 16 bits shift-register device:

```
Data_Shift      =    0.0.0        'data input signal in the shift-register
Clock_Shift     =    0.0.1        'clock signal for the shift
Reset_Shift     =    0.0.2        'reset to "0" signal of the shift-register

LD       Data_Shift               'load the carry flag
OUT      F.C

LD       Clock_Shift              'pulse generator on rising edge
OUT      P.0.IN

LD       P.0.OUTU                 'condition for the shift instruction
SFR      M.100
SFR      M.101

LD       Reset_Shift              'reset of the shift  forcing "0"
MOV2     M.100       K.0
```

ANDB

Logical and of the single bits of two bytes

Syntax

```
ANDB      Byte1Operand   Byte2Operand   Byte3Operand         ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a byte constant.
Byte3Operand is the address or label of any byte or a byte constant.

Description

ANDB performs the logical AND between the single bits of the bytes Byte2Operand and Byte3Operand; the byte result is returned in the address of the destination Byte1Operand. A bit in the result is set if both corresponding bits of the original operands are set; otherwise the result bit is cleared.
The instruction ANDB is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example executes the logical and bit by bit between the byte M.101 and M.102 and returns the result in the M.100. As example, the and bit by bit between the variable M.201 and a binary constant is also executed; in this way you can realize multiple RES instructions on the bits of the same byte with a defined mask:

```
LD      0.0.0                           ‘enabling condition
ANDB    M.100    M.101    M.102         ‘instruction with variables
ANDB    M.200    M.201    K.11101011B   ‘between variable and constant
```

See also

ORB, XORB, CPLB

ORB

Logical or of the single bits of two bytes

Syntax

```
ORB  Byte1Operand     Byte2Operand     Byte3Operand              ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a byte constant.
Byte3Operand is the address or label of any byte or a byte constant.

Description

ORB performs the logical OR between the single bits of the bytes Byte2Operand and Byte3Operand; the byte result is returned in the address of the destination Byte1Operand. A bit in the result is set if either or both corresponding bits in the original operands are set; otherwise the result bit is cleared.
The instruction ORB is executed only if the last bit inserted in the accumulator register is "1". The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example executes the logical or bit by bit between the byte M.101 and M.102 and returns the result in the M.100. As example, the or bit by bit between the variable M.201 and a binary constant is also executed; in this way you can realize multiple SET instructions on the bits of the same byte with a defined mask:

```
LD      0.0.0                          ‘enabling condition
ORB     M.100    M.101    M.102        ‘instruction with variables
ORB     M.200    M.201    K.11101011B  ‘between variable and constant
```

See also

ANDB, XORB, CPLB

XORB

Logical exclusive or of the single bits of two bytes

Syntax

```
XORB     Byte1Operand     Byte2Operand     Byte3Operand          ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a byte constant.
Byte3Operand is the address or label of any byte or a byte constant.

Description

XORB performs the logical exclusive or (XOR) between the single bits of the bytes Byte2Operand and Byte3Operand; the byte result is returned in the address of the destination Byte1Operand. A bit in the result is set if the corresponding bits of the original operands contain opposite values; otherwise the result bit is cleared.
The instruction XORB is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example executes the logical exclusive or, bit by bit, between the byte M.101 and M.102 and returns the result in the M.100. As example, the xor bit by bit between the variable M.201 and a binary constant is also executed; in this way you can realize multiple CPL instructions on the bits of the same byte with a defined mask:

```
LD       0.0.0                            ‘enabling condition
XORB     M.100    M.101    M.102          ‘instruction with variables
XORB     M.200    M.201    K.11101011B    ‘between variable and constant
```

See also

ANDB, ORB, CPLB

CPLB

Complement of all the bits of a byte

Syntax

```
CPLB     Byte Operand               ‘comment
```

Argument

Byte Operand is the address or the related label of any byte of the ram memory.

Description

CPLB performs the complement of all the bits of the argument byte, i.e. it changes the single bit into its opposite state.
The instruction CPLB is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: the instruction CPLB performs the complement of the byte every program cycle if the condition is on. To perform only one complement corresponding to a certain condition, you must use the pulse generator device as enable condition.

Example

The following example performs the complement of all the bits of the byte M.100 every time a rising edge of the bit 0.0.0 is detected:

```
LD       0.0.0                      ‘signal to enable the complement and input
OUT      P.0.IN                     ‘for the pulse generator

LD       P.0.OUTU                   ‘rising edge pulse for a program cycle
CPLB     M.100                      ‘execution of the complement instruction
```

See also

ANDB, ORB, XORB

MOV1

Move 1 byte variable or constant to 1 byte variable

Syntax

```
MOV1      Byte1Operand    Byte2Operand              ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a byte constant.

Description

MOV1 copies the value of the 1 byte variable or constant of the Byte2Operand on the 1 byte variable of the Byte1Operand.
The instruction MOV1 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example moves in the byte M.100 the constant 123 and then it copies in the byte M.200 the value of the byte M.300:

```
LD        0.0.0                                     ‘enable condition
MOV1      M.100     K.123                           ‘copies in M.100 the constant 123
MOV1      M.200     M.300                           ‘copies in M.200 the value of M.300
```

See also

MOV2, MOV4, MOVBLK

MOV2

Move a 2 bytes variable or constant to a 2 bytes variable

Syntax

```
MOV2     Byte1Operand    Byte2Operand           ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 2 bytes constant.

Description

MOV2 copies the value of the 2 bytes variable or constant of the Byte2Operand on the 2 bytes variable of the Byte1Operand.
The instruction MOV2 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example moves in the 2 bytes variable M.100 the constant 12345 and then it copies in the 2 bytes variable M.200 the value of the 2 bytes variable M.300:

```
LD       0.0.0                          ‘enable condition
MOV2     M.100     K.12345              ‘M.101÷100 <--- 12345
MOV2     M.200     M.300                ‘M.201÷200 <--- M.301÷300
```

See also

MOV1, MOV4, MOVBLK

MOV4

Move a 4 bytes variable or constant to a 4 bytes variable

Syntax

```
MOV4     Byte1Operand   Byte2Operand        ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 4 bytes constant.

Description

MOV4 copies the value of the 4 bytes variable or constant of the Byte2Operand on the 4 bytes variable of the Byte1Operand.
The instruction MOV4 is executed only if the last bit inserted in the accumulator register is "1". The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example moves in the 4 bytes variable M.100 the constant 123456789 and then it copies in the 4 bytes variable M.200 the value of the 4 bytes variable M.300:

```
LD       0.0.0                              ‘condition of habilitation
MOV4     M.100     K.123456789              ‘M.103÷100 <--- 123456789
MOV4     M.200     M.300                    ‘M.203÷200 <--- M.303÷300
```

See also

MOV1, MOV2, MOVBLK

CMP1

Compare the value of two variables or constants which are bytes

Syntax

```
CMP1      Byte1Operand    Byte2Operand         ‘comment
```

Argument

Byte1Operand is the address or label of any byte or one byte constant.
Byte2Operand is the address or label of any byte or one byte constant.

Description

CMP1 compares the value of the 1 byte variable or constant of the Byte1Operand with the value of the 1 byte variable or constant of the Byte2Operand. The result of the compare is available, immediately after this instruction, in the special bits of Flag (F.<, F.=, F.>), which can be tested by any boolean instruction.
The instruction CMP1 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: the forced value in the result flags will be unchanged in the executed list until another possible compare instruction.

Example

The following example compares the value in the byte M.100 with the value in the byte M.200:

```
LD        0.0.0                                 ‘enable condition
CMP1      M.100     M.200                       ‘compare of M.100 with M.200
```

RESULT	F.<	F.=	F.>
M.100 < M.200	ON	OFF	OFF
M.100 = M.200	OFF	ON	OFF
M.100 > M.200	OFF	OFF	ON

See also

CMP2, CMP4

CMP2

Compare the value of two variables or constants which are words (2 bytes)

Syntax

```
CMP2       Byte1Operand     Byte2Operand          ‘comment
```

Argument

Byte1Operand is the address or label of any byte or 2 bytes constant.
Byte2Operand is the address or label of any byte or 2 bytes constant.

Description

CMP2 compares the value of the 2 bytes variable or constant of the Byte1Operand with the value of the 2 bytes variable or constant of the Byte2Operand. The result of the compare is available, immediately after this instruction, in the special bits of Flag (F.<, F.=, F.>), which can be tested by any boolean instruction.
The instruction CMP2 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: the forced value in the result flags will be unchanged in the executed list until another possible compare instruction.

Example

The following example compares the value in the variable M.100 with the value in the variable M.200:

```
LD         0.0.0                                ‘enable condition
CMP2       M.100      M.200                     ‘compare M.101÷100 with M.201÷200
```

RESULT	F.<	F.=	F.>
M.101,M.100 < M.201,M.200	ON	OFF	OFF
M.101,M.100 = M.201,M.200	OFF	ON	OFF
M.101,M.100 > M.201,M.200	OFF	OFF	ON

See also

CMP1, CMP4

CMP4

Compare the value of two variables or constants which are double words (4 bytes)

Syntax

```
CMP4      Byte1Operand    Byte2Operand          ‘comment
```

Argument

Byte1Operand is the address or label of any byte or 4 bytes constant.
Byte2Operand is the address or label of any byte or 4 bytes constant.

Description

CMP4 compares the value of the 4 bytes variable or constant of the Byte1Operand with the value of the 4 bytes variable or constant of the Byte2Operand. The result of the compare is available, immediately after this instruction, in the special bits of Flag (F.<, F.=, F.>), which can be tested by any boolean instruction.
The instruction CMP4 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: the forced value in the result flags will be unchanged in the executed list until another possible compare instruction.

Example

The following example compares the value in the variable M.100 with the value in the variable M.200:

```
LD        0.0.0                                 ‘enable condition
CMP4      M.100     M.200                       ‘compare M.103÷100 with M.203÷200
```

RESULT	F.<	F.=	F.>
M.103,,,M.100 < M.203,,,M.200	ON	OFF	OFF
M.103,,,M.100 = M.203,,,M.200	OFF	ON	OFF
M.103,,,M.100 > M.203,,,M.200	OFF	OFF	ON

See also

CMP1, CMP2

ADD1

Binary sum of two 1 byte variables or constants

Syntax

```
ADD1      Byte1Operand    Byte2Operand    Byte3Operand          ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a byte constant.
Byte3Operand is the address or label of any byte or a byte constant.

Description

ADD1 sums the value of the 1 byte variable or constant of the Byte2Operand with the value of the 1 byte variable or constant of the Byte3Operand and returns the result to the 1 byte variable of the Byte1Operand. The carry flag value is affected and returned to F.C.
The instruction ADD1 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example sums the byte value M.200 with the constant 123 and returns the result to the byte M.100:

```
LD        0.0.0                                        ‘enable condition
ADD1      M.100     M.200     K.123                    ‘M.100 <--- M.200 + 123
```

See also

ADD2, ADD4

ADD2

Binary sum of two variables or constants. The operands are words

Syntax

```
ADD2      Byte1Operand    Byte2Operand    Byte3Operand          ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 2 bytes constant.
Byte3Operand is the address or label of any byte or a 2 bytes constant.

Description

ADD2 sums the value of the 2 byte variable or constant of the Byte2Operand with the value of the 2 bytes variable or constant of the Byte3Operand and returns the result to the 2 bytes variable of the Byte1Operand. The carry flag value is affected and returned to F.C.
The instruction ADD2 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example sums the 2 bytes variable M.200 with the constant 12345 and returns the result to the 2 bytes variable M.100:

```
LD        0.0.0                                  ‘enable condition
ADD2      M.100      M.200      K.12345          ‘M.101÷100 <--- M.201÷200 + 12345
```

See also

ADD1, ADD4

ADD4

Binary sum of two variables or constants. The operands are 4 bytes

Syntax

```
ADD4      Byte1Operand    Byte2Operand    Byte3Operand          ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 4 bytes constant.
Byte3Operand is the address or label of any byte or a 4 bytes constant.

Description

ADD4 sums the value of the 4 bytes variable or constant of the Byte2Operand with the value of the 4 bytes variable or constant of the Byte3Operand and returns the result to the 4 byte variable of the Byte1Operand. The carry flag value is affected and returned to F.C.
The instruction ADD4 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example sums the 4 bytes variable M.200 with the constant 123456789 and returns the result to the 4 bytes variable M.100:

```
LD      0.0.0                            ‘enable condition
ADD4    M.100   M.200   K.123456789      ‘M.103÷100 <--- M203÷200 + 123456789
```

See also

ADD1, ADD2

SUB1

Binary subtraction between two 1 byte variables or constants

Syntax

```
SUB1      Byte1Operand   Byte2Operand   Byte3Operand        ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a byte constant.
Byte3Operand is the address or label of any byte or a byte constant.

Description

SUB1 subtracts from the value of the 1 byte variable or constant of the Byte2Operand the value of the 1 byte variable or constant of the Byte3Operand and returns the result to the 1 byte variable of the Byte1Operand. The borrow bit value is returned to F.C.
The instruction SUB1 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example subtracts from the byte value M.200 the constant 123 and returns the result to the byte M.100:

```
LD        0.0.0                                 ‘enable condition
SUB1      M.100     M.200     K.123             ‘M.100 <--- M.200 - 123
```

See also

SUB2, SUB4

SUB2

Binary subtraction between two variables or constants. The operands are words

Syntax

```
SUB2      Byte1Operand    Byte2Operand    Byte3Operand          ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 2 bytes constant.
Byte3Operand is the address or label of any byte or a 2 bytes constant.

Description

SUB2 subtracts from the value of the 2 bytes variable or constant of the Byte2Operand the value of the 2 bytes variable or constant of the Byte3Operand and returns the result to the 2 bytes variable of the Byte1Operand. The borrow bit value is returned to F.C.
The instruction SUB2 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example subtracts from the 2 bytes variable M.200 the constant 12345 and returns the result to the 2 bytes variable M.100:

```
LD        0.0.0                                 ‘enable condition
SUB2      M.100     M.200     K.12345           ‘M.101÷100 <--- M.201÷200 - 12345
```

See also

SUB1, SUB4

SUB4

Binary subtraction between two variables or constants. The operands are 4 bytes

Syntax

```
SUB4      Byte1Operand    Byte2Operand    Byte3Operand          ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 4 bytes constant.
Byte3Operand is the address or label of any byte or a 4 bytes constant.

Description

SUB4 subtracts from the value of the 4 bytes variable or constant of the Byte2Operand the value of the 4 bytes variable or constant of the Byte3Operand and returns the result to the 4 bytes variable of the Byte1Operand. The borrow bit value is returned to F.C.
The instruction SUB4 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example subtracts from the 4 bytes variable M.200 the constant 123456789 and returns the result to the 4 bytes variable M.100:

```
LD      0.0.0                               ‘condition of habilitation
SUB4    M.100   M.200   K.123456789         ‘M.103÷100 <--- M203÷200 - 123456789
```

See also

SUB1, SUB2

MUL1

Multiplication of two 1 byte variables or constants

Syntax

```
MUL1      Byte1Operand    Byte2Operand    Byte3Operand          ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a byte constant.
Byte3Operand is the address or label of any byte or a byte constant.

Description

MUL1 multiplies the value of the 1 byte variable or constant of the Byte2Operand with the value of the 1 byte variable or constant of the Byte3Operand and returns the result to the 2 byte variable of the Byte1Operand.
The multiplication performed by MUL1 is binary unsigned; the result has always double length because “8 bits” multiplied by “8 bits” can require up to 16 bits for the result. Any way the necessity of the additional byte to contain the result is indicated by the flag bit F.E (Overflow over 1 byte).
The instruction MUL1 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example multiplies the byte value M.200 with the constant 123 and returns the result to the 2 bytes variable M.100:

```
LD        0.0.0                                   ‘enable condition
MUL1      M.100     M.200     K.123               ‘M.101÷100 <--- M.200 * 123
```

See also

MUL2, MUL4

MUL2

Multiplication of two 2 bytes variables or constants

Syntax

```
MUL2      Byte1Operand    Byte2Operand    Byte3Operand            ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 2 bytes constant.
Byte3Operand is the address or label of any byte or a 2 bytes constant.

Description

MUL2 multiplies the value of the 2 bytes variable or constant of the Byte2Operand with the value of the 2 bytes variable or constant of the Byte3Operand and returns the result to the 4 bytes variable of the Byte1Operand.
The multiplication performed by MUL2 is binary unsigned; the result has always double length compared with the multiplied variables because “16 bits” multiplied by “16 bits” can require up to 32 bits for the result. Any way the necessity of the additional 2 bytes to contain the result is indicated by the flag bit F.E (Overflow over 2 bytes).
The instruction MUL2 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example multiplies the 2 bytes variable M.200 with the constant 12345 and returns the result to the 4 bytes variable M.100:

```
LD        0.0.0                                  ‘enable condition
MUL2      M.100      M.200      K.12345          ‘M.103÷100 <--- M.201÷200 * 12345
```

See also

MUL1, MUL4

MUL4

Multiplication of two 4 bytes variables or constants

Syntax

```
MUL4      Byte1Operand    Byte2Operand    Byte3Operand          ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 2 bytes constant.
Byte3Operand is the address or label of any byte or a 2 bytes constant.

Description

MUL4 multiplies the value of the 4 bytes variable or constant of the Byte2Operand with the value of the 4 bytes variable or constant of the Byte3Operand and returns the result to the 8 bytes variable of the Byte1Operand.
The multiplication performed by MUL4 is binary unsigned; the result has always double length compared with the multiplied variables because “32 bits” multiplied by “32 bits” can require up to 64 bits for the result. Any way the necessity of the additional 4 bytes to contain the result is indicated by the flag bit F.E (Overflow over 4 bytes).
The instruction MUL4 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example multiplies the 4 bytes variable M.200 with the constant 123456789 and returns the result to the 8 bytes variable M.100:

```
LD      0.0.0                              ‘enable condition
MUL4    M.100   M.200   K.123456789        ‘M.107÷100 <--- M.203÷200 * 123456789
```

See also

MUL1, MUL2

DIV1

Division of two 1 byte variables or constants

Syntax

```
DIV1 Byte1Operand     Byte2Operand     Byte3Operand            ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a byte constant.
Byte3Operand is the address or label of any byte or a byte constant.

Description

DIV1 divides the value of the 1 byte variable or constant of the Byte2Operand with the value of the 1 byte variable or constant of the Byte3Operand and returns the quotient to the 1 byte variable of the Byte1Operand and the remainder to the 1 byte variable in the following address.
The division performed by DIV1 is binary unsigned; the result has always double length compared with the divided variables, because “8 bits” divided by “8 bits” can require 8 bits for the quotient and 8 bits for the remainder.
When a division by 0 is attempted, the error flag F.E goes to “1” and the quotient and the remainder are undefined, i.e. the instruction execution is cancelled.
The instruction DIV1 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example divides the byte value M.200 with the constant 123 and returns the quotient to the byte M.100 and the remainder to the byte M.101:

```
LD       0.0.0                                  ‘enable condition
DIV1     M.100     M.200     K.123              ‘M.100 <--- Q(M.200 / 123)
                                                ‘M.101 <--- R(M.200 / 123)
```

See also

DIV2, DIV4

DIV2

Division of two 2 bytes variables or constants

Syntax

```
DIV2        Byte1Operand     Byte2Operand     Byte3Operand          'comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 2 bytes constant.
Byte3Operand is the address or label of any byte or a 2 bytes constant.

Description

DIV2 divides the value of the 2 bytes variable or constant of the Byte2Operand with the value of the 2 bytes variable or constant of the Byte3Operand and returns the quotient to the 2 bytes variable of the Byte1Operand and the remainder to the 2 bytes variable in the following address.
The division performed by DIV2 is binary unsigned; the result has always double length compared with the divided variables, because "16 bits" divided by "16 bits" can require 16 bits for the quotient and 16 bits for the remainder.
When a division by 0 is attempted, the error flag F.E goes to "1" and the quotient and the remainder are undefined, i.e. the instruction execution is cancelled.
The instruction DIV2 is executed only if the last bit inserted in the accumulator register is "1".
The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example divides the 2 bytes value M.200 with the constant 12345 and returns the quotient to the 2 bytes variable M.100 and the remainder to the 2 bytes variable M.102:

```
LD       0.0.0                          'enable condition
DIV2   M.100   M.200   K.12345          'M.101÷100 <--- Q(M.201÷200 / 12345)
                                        'M.103÷102 <--- R(M.201÷200 / 12345)
```

See also

DIV1, DIV4

DIV4

Division of two 4 bytes variables or constants

Syntax

```
DIV4      Byte1Operand    Byte2Operand    Byte3Operand          'comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 4 bytes constant.
Byte3Operand is the address or label of any byte or a 4 bytes constant.

Description

DIV4 divides the value of the 4 bytes variable or constant of the Byte2Operand with the value of the 4 bytes variable or constant of the Byte3Operand and returns the quotient to the 4 bytes variable of the Byte1Operand and the remainder to the 4 bytes variable in the following address. The division performed by DIV4 is binary unsigned; the result has always double length compared with the divided variables, because "32 bits" divided by "32 bits" can require 32 bits for the quotient and 32 bits for the remainder.
When a division by 0 is attempted, the error flag F.E goes to "1" and the quotient and the remainder are undefined, i.e. the instruction execution is cancelled.
The instruction DIV4 is executed only if the last bit inserted in the accumulator register is "1". The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example divides the 4 bytes variable M.200 with the constant 123456789 and returns the quotient to the 4 bytes variable M.100 and the remainder to the 4 bytes variable M.104:

```
LD     0.0.0                           'condition of habilitation
DIV4   M.100   M.200   K.123456789     'M.103÷100 <--- Q(M.203÷200 / 123456789)
                                       'M.107÷104 <--- R(M.203÷200 / 123456789)
```

See also

DIV1, DIV2

INC1

Adds one to the 1 byte variable

```
INC1      ByteOperand               ‘comment
```

Argument

ByteOperand is the address or label of any byte in the memory.

Description

INC1 adds one to the value in the 1 byte variable of the argument ByteOperand.
The carry bit value is returned in the carry flag F.C.
The instruction INC1 is executed only if the last bit inserted in the accumulator register is “1”.
The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: the instruction INC1 performs an increment of the variable every program cycle if the condition is on. To perform only one increment corresponding to a certain condition, you must use the pulse generator device as enable condition.

Example

The following example adds one to the value of the 1 byte variable M.100 every time a rising edge of the bit 0.0.0 occurs:

```
LD        0.0.0                     ‘signal to enable the pulse generator
OUT       P.0.IN

LD        P.0.OUTU                  ‘rising edge pulse for a program cycle
INC1      M.100                     ‘M.100 <--- M.100 + 1
```

See also

INC2, INC4

INC2

Adds one to the 2 bytes variable

```
INC2        ByteOperand                 ‘comment
```

Argument

ByteOperand is the address or label of any byte in the memory.

Description

INC2 adds one to the value in the 2 bytes variable of the argument ByteOperand.
The carry bit value is returned in the carry flag F.C.
The instruction INC2 is executed only if the last bit inserted in the accumulator register is “1”.
The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: the instruction INC2 performs an increment of the variable every program cycle if the condition is on. To perform only one increment corresponding to a certain condition, you must use the pulse generator device as enable condition.

Example

The following example adds one to the value of the 2 bytes variable M.100 every time a rising edge of the bit 0.0.0 occurs:

```
LD          0.0.0                       ‘signal to enable the pulse generator
OUT         P.0.IN

LD          P.0.OUTU                    ‘rising edge pulse for a program cycle
INC2        M.100                       ‘M.101÷100 <--- M.101÷100 + 1
```

See also

INC1, INC4

INC4

Adds one to the 4 bytes variable

```
INC4      Byte Operand              ‘comment
```

Argument

ByteOperand is the address or label of any byte in the memory.

Description

INC4 adds one to the value in the 4 bytes variable of the argument ByteOperand.
The carry bit value is returned in the carry flag F.C.
The instruction INC4 is executed only if the last bit inserted in the accumulator register is “1”.
The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: the instruction INC4 performs an increment of the variable every program cycle if the condition is on. To perform only one increment corresponding to a certain condition, you must use the pulse generator device as enable condition.

Example

The following example adds one to the value of the 4 bytes variable M.100 every time a rising edge of the bit 0.0.0 occurs:

```
LD        0.0.0                     ‘signal to enable the pulse generator
OUT       P.0.IN

LD        P.0.OUTU                  ‘rising edge pulse for a program cycle
INC4      M.100                     ‘M.103÷100 <--- M.103÷100 + 1
```

See also

INC1, INC2

DEC1

Subtracts one to the 1 byte variable

```
DEC1            ByteOperand               ‘comment
```

Argument

ByteOperand is the address or label of any byte in the memory.

Description

DEC1 subtracts one to the value in the 1 byte variable of the argument ByteOperand.
The borrow bit value is returned in the carry flag F.C.
The instruction DEC1 is executed only if the last bit inserted in the accumulator register is “1”.
The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: the instruction DEC1 performs a decrement of the variable every program cycle if the condition is on. To perform only one decrement corresponding to a certain condition, you must use the pulse generator device as enable condition.

Example

The following example subtracts one to the value of the 1 byte variable M.100 every time a rising edge of the bit 0.0.0 occurs:

```
LD      0.0.0                   ‘signal to enable the pulse generator
OUT     P.0.IN

LD      P.0.OUTU                ‘rising edge pulse for a program cycle
DEC1    M.100                   ‘M.100 <--- M.100 - 1
```

See also

DEC2, DEC4

DEC2

Subtracts one to the 2 bytes variable

```
DEC2            Byte Operand            ‘comment
```

Argument

ByteOperand is the address or label of any byte in the memory.

Description

DEC2 subtracts one to the value in the 2 bytes variable of the argument ByteOperand.
The borrow bit value is returned in the carry flag F.C.
The instruction DEC2 is executed only if the last bit inserted in the accumulator register is “1”.
The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: the instruction DEC2 performs a decrement of the variable every program cycle if the condition is on. To perform only one increment corresponding to a certain condition, you must use the pulse generator device as enable condition.

Example

The following example subtracts one to the value of the 2 bytes variable M.100 every time a rising edge of the bit 0.0.0 occurs:

```
LD       0.0.0                  ‘signal to enable the pulse generator
OUT      P.0.IN

LD       P.0.OUTU               ‘rising edge pulse for a program cycle
DEC2     M.100                  ‘M.101÷100 <--- M.101÷100 - 1
```

See also

DEC1, DEC4

DEC4

Subtracts one to the 4 bytes variable

```
DEC4          ByteOperand          ‘comment
```

Argument

ByteOperand is the address or label of any byte in the memory.

Description

DEC4 subtracts one to the value in the 4 bytes variable of the argument ByteOperand.
The borrow bit value is returned in the carry flag F.C.
The instruction DEC4 is executed only if the last bit inserted in the accumulator register is “1”.
The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: the instruction DEC4 performs a decrement of the variable every program cycle if the condition is on. To perform only one increment corresponding to a certain condition, you must use the pulse generator device as enable condition.

Example

The following example subtracts one to the value of the 4 bytes variable M.100 every time a rising edge of the bit 0.0.0 occurs:

```
LD      0.0.0                  ‘signal to enable the pulse generator
OUT     P.0.IN

LD      P.0.OUTU               ‘rising edge pulse for a program cycle
DEC4    M.100                  ‘M.103÷100 <--- M.103÷100 - 1
```

See also

DEC1, DEC2

ABS1

Absolute value of a 1 byte variable or constant

Syntax

```
ABS1        Byte1Operand    Byte2Operand            ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 1 byte constant.

Description

ABS1 performs the absolute value of the 1 byte variable or constant of the Byte2Operand and returns the result to the 1 byte variable of the Byte1Operand.
If the variable is negative, this instruction is equivalent to one reversing the sign of the integer; in this case also the carry flag F.C is reversed to indicate the occurred change of sign.
The flag F.C can be used to test the resulting sign of unsigned multiplications or divisions. Resetting the flag F.C and executing the absolute value of the operands, you have in F.C the value “1” if the negative operands are odd: this flag will then enable the sign reversion of the result.
The instruction ABS1 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example returns to the 1 byte variable M.100 the absolute value of the 1 byte variable M.200, reversing the flag F.C if the value in M.200 is negative:

```
LD          0.0.0                                   ‘enable condition
ABS1        M.100       M.200                       ‘M.100 <--- |M.200|
```

See also

ABS2, ABS4, NEG1

ABS2

Absolute value of a 2 bytes variable or constant

Syntax

```
ABS2      Byte1Operand    Byte2Operand           ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 2 bytes constant.

Description

ABS2 performs the absolute value of the 2 bytes variable or constant of the Byte2Operand and returns the result to the 2 bytes variable of the Byte1Operand.
If the variable is negative, this instruction is equivalent to one reversing the sign of the integer; in this case also the carry flag F.C is reversed to indicate the occurred change of sign.
The flag F.C can be used to test the resulting sign of unsigned multiplications or divisions. Resetting the flag F.C and executing the absolute value of the operands, you have in F.C the value “1” if the negative operands are odd: this flag will then enable the sign reversion of the result.
The instruction ABS2 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example returns to the 2 bytes variable M.100 the absolute value of the 2 byte variable M.200, reversing the flag F.C if the value in M.200 is negative:

```
LD        0.0.0                                ‘enable condition
ABS2      M.100      M.200                     ‘M.101÷100 <--- |M.201÷200|
```

See also

ABS1, ABS4, NEG2

ABS4

Absolute value of a 4 bytes variable or constant

Syntax

```
ABS4        Byte1Operand     Byte2Operand            'comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 4 bytes constant.

Description

ABS4 performs the absolute value of the 4 bytes variable or constant of the Byte2Operand and returns the result to the 4 bytes variable of the Byte1Operand.
If the variable is negative, this instruction is equivalent to one reversing the sign of the integer; in this case also the carry flag F.C is reversed to indicate the occurred change of sign.
The flag F.C can be used to test the resulting sign of unsigned multiplications or divisions. Resetting the flag F.C and executing the absolute value of the operands, you have in F.C the value "1" if the negative operands are odd: this flag will then enable the sign reversion of the result.
The instruction ABS4 is executed only if the last bit inserted in the accumulator register is "1". The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example returns to the 4 bytes variable M.100 the absolute value of the 4 bytes variable M.200, reversing the flag F.C if the value in M.200 is negative:

```
LD          0.0.0                                    'enable condition
ABS4        M.100        M.200                       'M.103÷100 <--- |M.203÷200|
```

See also

ABS1, ABS2, NEG4

NEG1

Sign reversion of a 1 byte variable

Syntax

```
NEG1          ByteOperand          ‘comment
```

Argument

ByteOperand is the address or label of any byte in the memory.

Description

NEG1 reverses the sign of the value existing in the 1 byte variable of the argument ByteOperand. This instruction allows to “adjust” the resulting sign of an arithmetic operation, like unsigned multiplication and division, after executing the operation on the absolute values of the operands. The instruction NEG1 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: the instruction NEG1 performs a sign change of the variable every program cycle if the condition is on. To perform only one reversion corresponding to a certain condition, you must use the pulse generator device as enable condition.

Example

The following example negates the value of the 1 byte variable M.100 every time a rising edge of the bit 0.0.0 occurs:

```
LD      0.0.0               ‘signal to enable the pulse generator
OUT     P.0.IN

LD      P.0.OUTU            ‘rising edge pulse for a program cycle
NEG1    M.100               ‘M.100 <---  - M.100
```

See also

NEG2, NEG4, ABS1

NEG2

Sign reversion of a 2 bytes variable

Syntax

```
NEG2            ByteOperand         ‘comment
```

Argument

ByteOperand is the address or label of any byte in the memory.

Description

NEG2 reverses the sign of the value existing in the 2 bytes variable of the argument ByteOperand.
This instruction allows to “adjust” the resulting sign of an arithmetic operation, like unsigned multiplication and division, after executing the operation on the absolute values of the operands.
The instruction NEG2 is executed only if the last bit inserted in the accumulator register is “1”.
The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: the instruction NEG2 performs a sign change of the variable every program cycle if the condition is on. To perform only one reversion corresponding to a certain condition, you must use the pulse generator device as enable condition.

Example

The following example negates the value of the 2 bytes variable M.100 every time a rising edge of the bit 0.0.0 occurs:

```
LD      0.0.0                   ‘signal to enable the pulse generator
OUT     P.0.IN

LD      P.0.OUTU                ‘rising edge pulse for a program cycle
NEG2    M.100                   ‘M.101÷100 <---  - M.101÷100
```

See also

NEG1, NEG4, ABS2

NEG4

Sign reversion of a 4 bytes variable

Syntax

```
NEG4          ByteOperand          ‘comment
```

Argument

ByteOperand is the address or label of any byte in the memory.

Description

NEG4 reverses the sign of the value existing in the 4 bytes variable of the argument ByteOperand.
This instruction allows to “adjust” the resulting sign of an arithmetic operation, like unsigned multiplication and division, after executing the operation on the absolute values of the operands.
The instruction NEG4 is executed only if the last bit inserted in the accumulator register is “1”.
The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: the instruction NEG2 performs a sign change of the variable every program cycle if the condition is on. To perform only one reversion corresponding to a certain condition, you must use the pulse generator device as enable condition.

Example

The following example negates the value of the 4 bytes variable M.100 every time a rising edge of the bit 0.0.0 occurs:

```
LD      0.0.0                    ‘signal to enable the pulse generator
OUT     P.0.IN

LD      P.0.OUTU                 ‘rising edge pulse for a program cycle
NEG4    M.100                    ‘M.103÷100 <---  - M.103÷100
```

See also

NEG1, NEG2, ABS4

BINBCD1

Conversion from binary to BCD of a 1 byte variable

Syntax

```
BINBCD1  Byte1Operand    Byte2Operand           ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 1 byte constant.

Description

BINBCD1 converts the binary unsigned value of the 1 byte variable or constant of the Byte2Operand to BCD format with two digits returning it to the 1 byte variable of the Byte1Operand.
The result of the BINBCD1 being expressed by means two decimal digits, the validity field of the binary value to convert is 0÷99.
The instruction BINBCD1 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: conversion instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example converts the binary value of the 1 byte variable M.200 and returns the result in BCD with 2 digits in the 1 byte variable M.100:

```
LD       0.0.0                          ‘enable condition
MOV1     M.200    K.53                  ‘value 53 in M.200
BINBCD1  M.100    M.200                 ‘M.100 <--- 0101 0011B
```

See also

BINBCD2, BINBCD4

BINBCD2

Conversion from binary to BCD of a 2 bytes variable

Syntax

```
BINBCD2  Byte1Operand     Byte2Operand              ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 2 bytes constant.

Description

BINBCD2 converts the binary unsigned value of the 2 bytes variable or constant of the Byte2Operand to BCD format with four digits returning it to the 2 bytes variable of the Byte1Operand.
The result of the BINBCD2 being expressed by means four decimal digits, the validity field of the binary value to convert is 0÷9999.
The instruction BINBCD2 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: conversion instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example converts the binary value of the 2 bytes variable M.200 and returns the result in BCD with 4 digits in the 2 bytes variable M.100:

```
LD       0.0.0                         ‘enable condition
MOV2     M.200     K.3567              ‘value 3567 in M.201÷200
BINBCD2  M.100     M.200               ‘M.101÷100 <--- 0011 0101 0110 0111B
```

See also

BINBCD1, BINBCD4

BINBCD4

Conversion from binary to BCD of a 4 bytes variable

Syntax

```
BINBCD4  Byte1Operand    Byte2Operand          ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 4 bytes constant.

Description

BINBCD4 converts the binary unsigned value of the 4 bytes variable or constant of the Byte2Operand to BCD format with eight digits returning it to the 4 bytes variable of the Byte1Operand.
The result of the BINBCD4 being expressed by means eight decimal digits, the validity field of the binary value to convert is 0÷99999999.
The instruction BINBCD4 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: conversion instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example converts the binary value of the 4 bytes variable M.200 and returns the result in BCD with 8 digits in the 4 bytes variable M.100:

```
LD        0.0.0                        ‘enable condition
MOV4      M.200     K.85463567         ‘value 85463567 in M.203÷200
BINBCD4   M.100     M.200              ‘M.103÷102 <--- 1000 0101 0100 0110B
                                       ‘M.101÷100 <--- 0011 0101 0110 0111B
```

See also

BINBCD1, BINBCD2

BCDBIN1

Conversion from BCD to binary of a 1 byte variable

Syntax

```
BCDBIN1  Byte1Operand     Byte2Operand            ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 1 byte constant.

Description

BCDBIN1 converts the BCD value with two digits of the 1 byte variable or constant of the Byte2Operand to a binary unsigned format returning it to the 1 byte variable of the Byte1Operand.
The value of the byte, to be correctly converted, must have every nibble in the field 0÷9.
The instruction BCDBIN1 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: conversion instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example converts the BCD value with 2 digits of the 1 byte variable M.200 and returns the binary result in the 1 byte variable M.100:

```
LD        0.0.0                         ‘enable condition
MOV1      M.200     K.53                ‘value 53 in M.200
BCDBIN1   M.100     M.200               ‘M.100 <--- 53
```

See also

BCDBIN2, BCDBIN4

BCDBIN2

Conversion from BCD to binary of a 2 bytes variable

Syntax

```
BCDBIN2  Byte1Operand     Byte2Operand            ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 2 bytes constant.

Description

BCDBIN2 converts the BCD value with four digits of the 2 bytes variable or constant of the Byte2Operand to a binary unsigned format returning it to the 2 bytes variable of the Byte1Operand.
The values of the bytes, to be correctly converted, must have every nibble in the field 0÷9.
The instruction BCDBIN2 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: conversion instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example converts the BCD value with 4 digits of the 2 bytes variable M.200 and returns the binary result in the 2 bytes variable M.100:

```
LD        0.0.0                           ‘enable condition
MOV2      M.200     K.13671               ‘value 13671 in M.201÷200
BCDBIN2   M.100     M.200                 ‘M.101÷100 <--- 3567
```

See also

BCDBIN1, BCDBIN4

BCDBIN4

Conversion from BCD to binary of a 4 bytes variable

Syntax

```
BCDBIN4  Byte1Operand      Byte2Operand              ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 4 bytes constant.

Description

BCDBIN4 converts the BCD value with four digits of the 4 bytes variable or constant of the Byte2Operand to a binary unsigned format returning it to the 4 bytes variable of the Byte1Operand.
The values of the bytes, to be correctly converted, must have every nibble in the field 0÷9.
The instruction BCDBIN4 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: conversion instructions can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example converts the BCD value with 8 digits of the 4 bytes variable M.200 and returns the binary result in the 4 bytes variable M.100:

```
LD        0.0.0                               ‘enable condition
MOV4      M.200     K.87453567H               ‘value 87453567H in M.203÷200
BCDBIN4   M.100     M.200                     ‘M.103÷100 <--- 87453567
```

See also

BCDBIN1, BCDBIN2

SWAP

Switches the contents of the nibbles of a 1 byte variable

Syntax

```
SWAP            ByteOperand         'comment
```

Argument

ByteOperand is the address or label of any byte in the memory.

Description

SWAP performs the switch between the nibbles of the byte of the ByteOperand.
The instruction SWAP is executed only if the last bit inserted in the accumulator register is "1". The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: the instruction SWAP performs the nibbles switch of the byte every program cycle if the condition is on. To perform only one switch corresponding to a certain condition, you must use the pulse generator device as enable condition.

Example

The following example exchanges the nibbles of the byte M.100 every time a rising edge of the bit 0.0.0 occurs:

```
LD      0.0.0                   ‘signal to enable the pulse generator
OUT     P.0.IN

LD      P.0.OUTU                ‘rising edge to enable the instruction
SWAP    M.100                   ‘M.100.7 <---> M.100.3
                                ‘M.100.6 <---> M.100.2
                                ‘M.100.5 <---> M.100.1
                                ‘M.100.4 <---> M.100.0
```

RCL1

Loads a 1 byte variable or constant in the stack

Syntax

```
RCL1        ByteOperand             ‘comment
```

Argument

ByteOperand is the address or label of any byte in the memory or 1 byte constant.

Description

RCL1 loads the 1 byte variable or constant of the ByteOperand in the first position (level 0) of the stack; the values already present in the stack are previously shifted up by a position, with resultant loss of the current value in the level 3. The value of the new loaded variable is a signed 1 byte value and so it is automatically converted in a signed 4 bytes format.
The instruction RCL1 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example loads in the stack the value of the signed 1 byte variable M.100 if the bit 0.0.0 is ON:

```
LD          0.0.0
RCL1        M.100
```

```
The effect on the stack is the following:    STACK(3) <--- STACK(2)
                                              STACK(2)<--- STACK(1)
                                              STACK(1) <--- STACK(0)
                                              STACK(0) <--- M.100
```

Shortened mnemonic

```
R1          ByteOperand             ‘comment
```

See also

RCL2, RCL4, STO1, STO2, STO4, ADD, SUB, MUL, DIV, CMP

RCL2

Loads a 2 bytes variable or constant in the stack

Syntax

```
RCL2        ByteOperand           ‘comment
```

Argument

Byte Operand is the address or label of any byte in the memory or a 2 bytes constant.

Description

RCL2 loads the 2 bytes variable or constant of the ByteOperand in the first position (level 0) of the stack; the values already present in the stack are previously shifted up by a position, with resultant loss of the current value in the level 3. The value of the new loaded variable is a signed 2 bytes value and so it is automatically converted in a signed 4 bytes format.
The instruction RCL2 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example loads in the stack the value of the signed 2 byte variable M.100 if the bit 0.0.0 is ON:

```
LD          0.0.0
RCL2        M.100
```

The effect on the stack is the following:

```
STACK(3) <--- STACK(2)
STACK(2) <--- STACK(1)
STACK(1) <--- STACK(0)
STACK(0) <--- M.101÷100
```

Shortened mnemonic

```
R2          ByteOperand           ‘comment
```

See also

RCL1, RCL4, STO1, STO2, STO4, ADD, SUB, MUL, DIV, CMP

RCL4

Loads a 4 bytes variable or constant in the stack

Syntax

```
RCL4      ByteOperand          'comment
```

Argument

ByteOperand is the address or label of any byte in the memory or a 4 bytes constant.

Description

RCL4 loads the 4 bytes variable or constant of the ByteOperand in the first position (level 0) of the stack; the values already present in the stack are previously shifted up by a position, with resultant loss of the current value in the level 3. The value of the new loaded variable is a signed 4 bytes value and it is not converted because it is already in the right format.
The instruction RCL4 is executed only if the last bit inserted in the accumulator register is "1". The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example loads in the stack the value of the signed 4 bytes variable M.100 if the bit 0.0.0 is ON:

```
LD        0.0.0
RCL4      M.100
```

```
The effect on the stack is the following:   STACK(3) <--- STACK(2)
                                            STACK(2) <--- STACK(1)
                                            STACK(1) <--- STACK(0)
                                            STACK(0) <--- M.103÷100
```

Shortened mnemonic

```
R4        ByteOperand          'comment
```

See also

RCL1, RCL2, STO1, STO2, STO4, ADD, SUB, MUL, DIV, CMP

STO1

Copies the level 0 value of the stack to a 1 byte variable

Syntax

STO1	Byte Operand	‘comment

Argument

ByteOperand is the address or label of any byte in the memory.

Description

STO1 copies the value of the first position (level 0) of the stack to a 1 byte variable indicated by the ByteOperand. The value present in the level 0 of the stack is previously converted from the signed 4 bytes format to the signed 1 byte format, before copying in the memory; however all the values present in the four levels of the stack are not changed or shifted. If the value present in the stack cannot be converted in the signed 1 byte format, the error flag F.E is set.
The instruction STO1 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example copies the value of the stack onto the signed 1 byte variable M.100 if the bit 0.0.0 is ON:

```
LD          0.0.0
STO1        M.100
```

The effect on the stack is the following: M.100 <--- STACK(0)

Shortened mnemonic

S1	ByteOperand	‘comment

See also

RCL1, RCL2, RCL4, STO2, STO4, ADD, SUB, MUL, DIV, CMP

STO2

Copies the level 0 value of the stack to a 2 bytes variable

Syntax

```
STO2        Byte Operand          ‘comment
```

Argument

Byte Operand is the address or label of any byte in the memory.

Description

STO2 copies the value of the first position (level 0) of the stack to a 2 bytes variable indicated by the ByteOperand. The value present in the level 0 of the stack is previously converted from the signed 4 bytes format to the signed 2 byte format, before copying in the memory; however all the values present in the four levels of the stack are not changed or shifted. If the value present in the stack cannot be converted in the signed 2 bytes format, the error flag F.E is set.
The instruction STO2 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example copies the value of the stack onto the signed 2 bytes variable M.100 if the bit 0.0.0 is ON:

```
LD          0.0.0
STO2        M.100
```

The effect on the stack is the following: M.101÷100 <--- STACK(0)

Shortened mnemonic

```
S2          ByteOperand           ‘comment
```

See also

RCL1, RCL2, RCL4, STO1, STO4, ADD, SUB, MUL, DIV, CMP

STO4

Copies the level 0 value of the stack to a 4 bytes variable

Syntax

STO4	Byte Operand	‘comment

Argument

ByteOperand is the address or label of any byte in the memory.

Description

STO4 copies the value of the first position (level 0) of the stack to a 4 bytes variable indicated by the ByteOperand. The value present in the level 0 of the stack is not converted because it is already in the right format; all the values present in the four levels of the stack are not changed or shifted.

The instruction STO4 is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example copies the value of the stack onto the signed 4 bytes variable M.100 if the bit 0.0.0 is ON:

```
LD          0.0.0
STO2        M.100
```

The effect on the stack is the following: M.103÷100 <--- STACK(0)

Shortened mnemonic

S4	ByteOperand	‘comment

See also

RCL1, RCL2, RCL4, STO1, STO2, ADD, SUB, MUL, DIV, CMP

ADD

Adds the level 1 value to the level 0 value of the stack

Syntax

ADD	‘comment

Argument

None.

Description

ADD performs the sum of the value in the second position (level 1) with the value in the first position (level 0) of the stack, returning the result in the level 0. The values already present in the other levels of the stack are in successive order shifted toward decreasing levels by a position. If the signed 4 bytes operation of sum causes a overflow, the error flag F.E is set.
The instruction ADD is executed only if the last bit inserted in the accumulator register is “1”.
The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions executed on the stack can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example sums the values of the levels 1 e 0 of the stack if the bit 0.0.0 is ON:

```
LD          0.0.0
ADD
```

The effect on the stack is the following:

```
STACK(0) <--- STACK(1) + STACK(0)
STACK(1) <--- STACK(2)
STACK(2) <--- STACK(3)
```

Shortened mnemonic

+	‘comment

See also

RCL1, RCL2, RCL4, STO1, STO2, STO4, SUB, MUL, DIV, CMP

SUB

Subtracts to the level 1 value the level 0 value of the stack

Syntax

```
SUB                     ‘comment
```

Argument

None.

Description

SUB performs the subtraction from the value in the second position (level 1) of the value in the first position (level 0) of the stack, returning the result in the level 0. The values already present in the other levels of the stack are in successive order shifted toward decreasing levels by a position. If the signed 4 bytes operation of subtraction causes a overflow (borrow), the error flag F.E is set.
The instruction SUB is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions executed on the stack can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example subtracts the values of the levels 1 e 0 of the stack if the bit 0.0.0 is ON:

```
LD          0.0.0
SUB
```

The effect on the stack is the following:

```
STACK(0) <--- STACK(1) - STACK(0)
STACK(1) <--- STACK(2)
STACK(2) <--- STACK(3)
```

Shortened mnemonic

```
-                              ‘comment
```

See also

RCL1, RCL2, RCL4, STO1, STO2, STO4, ADD, MUL, DIV, CMP

MUL

Multiplies the level 1 value with the level 0 value of the stack

Syntax

```
MUL                    ‘comment
```

Argument

None.

Description

MUL performs the multiplication of the value in the second position (level 1) and the value in the first position (level 0) of the stack, returning the result in the level 0. The values already present in the other levels of the stack are in successive order shifted toward decreasing levels by a position. If the signed 4 bytes operation of multiplication causes a overflow, the error flag F.E is set.
The instruction MUL is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions executed on the stack can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example multiplies the values of the levels 1 e 0 of the stack if the bit 0.0.0 is ON:

```
LD          0.0.0
MUL
```

The effect on the stack is the following:

```
STACK(0) <--- STACK(1) * STACK(0)
STACK(1) <--- STACK(2)
STACK(2) <--- STACK(3)
```

Shortened mnemonic

```
*                      ‘comment
```

See also

RCL1, RCL2, RCL4, STO1, STO2, STO4, ADD, SUB, DIV, CMP

DIV

Divides the level 1 value by the level 0 value of the stack

Syntax

```
DIV                         ‘comment
```

Argument

None.

Description

DIV performs the division of the value in the second position (level 1) by the value in the first position (level 0) of the stack, returning the result (integer signed quotient) in the level 0. The values already present in the other levels of the stack are in successive order shifted toward decreasing levels by a position. If in the level 0 is present a null divisor, the operation is not executed and the error flag F.E is set.
The instruction DIV is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Attention: arithmetic instructions executed on the stack can require high execution times. Verify in the proper tables the required time for their execution and, if advisable, execute the instructions only for one program cycle or in not critical program states for the looping time.

Example

The following example divides the values of the levels 1 e 0 of the stack if the bit 0.0.0 is ON:

```
LD          0.0.0
DIV
```

The effect on the stack is the following: STACK(0) <--- STACK(1) / STACK(0)
STACK(1) <--- STACK(2)
STACK(2) <--- STACK(3)

Shortened mnemonic

```
/                               ‘comment
```

See also

RCL1, RCL2, RCL4, STO1, STO2, STO4, ADD, SUB, MUL, CMP

CMP

Compares the level 1 value with the level 0 value of the stack

Syntax

```
CMP                 ‘comment
```

Argument

None.

Description

CMP compares the value in the second position (level 1) with the value in the first position (level 0) of the stack. The result is available, immediately after this instruction, in the special Flag bits (F.<, F.=, F.>), which can be tested with any boolean instruction. The values already present in the other levels of the stack are not changed or shifted.

The instruction CMP is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example compares the values of the levels 1 e 0 of the stack if the bit 0.0.0 is ON:

```
LD          0.0.0
CMP
```

RESULT	F.<	F.=	F.>
STACK(1) < STACK(0)	ON	OFF	OFF
STACK(1) = STACK(0)	OFF	ON	OFF
STACK(1) > STACK(0)	OFF	OFF	ON

Shortened mnemonic

```
?                             ‘comment
```

See also

RCL1, RCL2, RCL4, STO1, STO2, STO4, ADD, SUB, MUL, DIV

MOVADD

Moves the absolute address of a variable to a 2 bytes variable

Syntax

```
MOVADD      Byte1Operand   Byte2Operand        'comment
```

Argument

Byte1Operand (pointer) is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte in the memory.

Description

MOVADD transfers the value of the RAM absolute address of the variable indicated by Byte2Operand to the 2 bytes variable indicated by Byte1Operand which becomes pointer of Byte2Operand.
To identify any variable by means its absolute address you must use a two bytes variable pointer containing the value of the 16 bits address; if the variable to point is 2 or 4 bytes wide (word or double word), the Byte2Operand points to the last significant byte.
The instruction MOVADD is essential in the use of the indirect addressing mode. In this mode modifying the pointer value it is possible to make the same operations to blocks of variables instead of a specified name of variable. The MOVADD allows to initialize the pointer to the address of a specific variable of the block; further increments or decrements of the pointer value will allow to work with the same piece of program on all the other variables.
The instruction MOVADD is executed only if the last bit inserted in the accumulator register is "1". The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example moves in the 2 bytes variable M.100 the absolute address of the variable H.0:

```
LD       F.1                                  'always enabled condition
MOVADD   M.100      H.0                       'M101÷100 <--- 37888
```

MOVASC

Moves a text string to a memory block

Syntax

```
MOVASC          ByteOperand       |String|              ‘comment
```

Argument

ByteOperand (destination) is the address or label of any byte in the memory.
String is a sequence of ASCII characters of maximum length 100 characters.

Description

MOVASC fills the RAM memory block, starting from the byte indicated by ByteOperand, with the sequence of ASCII code corresponding to the characters of the provided String.
The text string must be bounded between two characters “ | “ (vertical slash) and it can contain up to 100 characters excluding the delimiter.
Inside the string three characters @ (commercial a), \ (reversed slash) and ^ (quote) have a particular meaning; actually these characters are automatically replaced by the compiler respectively with the ASCII code 13, 10, 12 corresponding to the characters CARRIAGE RETURN, LINE FEED and FORM FEED. This allows to position inside the string, these three special characters, often necessary to communicate the strings towards the outside. If you want to store in the bytes of the RAM the real characters @, \, ^ without replacing them with the codes 13, 10, 12, you need to move in any byte, by means the instruction MOV1, the decimal value 64, 92, 94 corresponding to the real ASCII code of the desired character.
The instruction MOVASC is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example moves in the ram space, starting from the M.100, the sequence of ASCII codes corresponding to the indicated string:

```
LD          F.1
MOVASC   M.100        |EXAMPLE OF STRING@|
```

the effect of this instruction is:

```
M.100÷118 <--- 69,115,101,109,112,105,111,32,100,105,32,115,116,114,105,110,103,97,13
```

MOVBLK

Moves a memory block onto another memory block

Syntax

```
MOVBLK          Byte1Operand    Byte2Operand    Byte3Operand            ‘comment
```

Argument

Byte1Operand (destination) is the address or label of any byte in the memory.
Byte2Operand (source) is the address or label of any byte in the memory.
Byte3Operand is the address or label of any byte or a 1 byte constant.

Description

MOVBLK copies the RAM memory block beginning from Byte2Operand onto the RAM memory block beginning from Byte1Operand; the number of transferred bytes is contained in the value of Byte3Operand.
This operation of multiple copy is equivalent to the reiteration of the instruction MOV1 for the indicated number of bytes. The value of Byte3Operand can be a constant or a 1 byte variable and its value must be included in the field 0÷255.
The instruction MOVBLK is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example moves to the ram block, starting from the byte M.100, the ram block, starting from M.200, the number of transferred bytes is 18:

```
LD          0.0.0                                   ‘enable condition
MOVBLK      M.100       M.200       K.18            ‘M100÷117 <--- M200÷217
```

See also

MOV1, MOV2, MOV4

CMPBLK

Compare a memory block with another memory block

Syntax

```
CMPBLK        Byte1Operand   Byte2Operand   Byte3Operand          ‘comment
```

Argument

Byte1Operand is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte in the memory.
Byte3Operand is the address or label of any byte or a 1 byte constant.

Description

CMPBLK compares the RAM memory block beginning from Byte1Operand with the RAM memory block beginning from Byte2Operand; the number of compared bytes is contained in the value of Byte3Operand. The value of Byte3Operand can be a constant or a 1 byte variable and its value must be included in the field 0÷255.
The bytes of the memory blocks are compared two by two for all the block starting from the byte with lower address. If all the bytes of the same position in the two blocks are equal, the flag F.= is set, otherwise the compare is suspended at the first difference; in this case the flag F.< or F.> is set if the result of the compare of the two current bytes is less or more.
The instruction CMPBLK is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example compares the 18 bytes ram block, starting from the byte M.100, with the ram block, starting from M.200:

```
LD        0.0.0                              ‘enable condition
CMPLK     M.100     M.200     K.18           ‘compares M100÷117 with M200÷217
```

RESULT	F.<	F.=	F.>
M.100...M.117 < M.200...M.217	ON	OFF	OFF
M.100...M.117 = M.200...M.217	OFF	ON	OFF
M.100...M.117 > M.200...M.217	OFF	OFF	ON

RESMEM

Reset a memory block

Syntax

```
RESMEM          Byte1Operand    Byte2Operand                    ‘comment
```

Argument

Byte1Operand is the address or label of any byte in the memory.
Byte2Operand is the address or label of any byte or a 1 byte constant.

Description

RESMEM resets a memory RAM block starting from Byte1Operand; the number of zeroed bytes is the value of Byte2Operand.
The value of Byte2Operand can be a constant or a 1 byte and in any case its value must be in the field 0÷255.
The instruction RESMEM is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example moves the constant value 0 in all the 18 bytes of the ram block starting from the M.100:

```
LD        0.0.0                                       ‘enable condition
RESMEM    M.100      K.18                             ‘M100÷117 <--- 0,,,0
```

IOREFR

Updates the data ram corresponding to the external resources of the MASTER

Syntax

```
IOREFR                                         ‘comment
```

Argument

None.

Description

IOREFR allows to update the data ram corresponding to the process image of the MASTER local Input/Output.
This instruction is automatically executed every program cycle by the operating system; for this reason its use is limited only at the case you need to update the local I/O more times per cycle, in intermediate points of the user program. The forces updating of the I/O occurs only and exclusively for the external local resources of the MASTER board, without affecting the communication with the SLAVES.
The instruction IOREFR is executed only if the last bit inserted in the accumulator register is “1”.

Attention: the bits list of the accumulator register can be affected by the execution of this instruction.

Example

The following example updates the local I/O of the MASTER if the bit M.0.0 is ON:

```
LD          M.0.0                              ‘enable condition
IOREFR                                         ‘forced updating of the I/O
```

RESWD

Reset of the watchdog timer

Syntax

```
RESWD                                   ‘comment
```

Argument

None.

Description

RESWD allows to reset the current value of the watchdog timer of the MASTER board.
The watchdog is an hardware device built-in the Logic Controller which controls its correct functioning. A counter is incremented at regular intervals automatically and it cannot be disabled, while the operating system of the Logic Controller provides, normally in the cycle, to reset it. If this thing does not happen, the counter, ending its count, will reset in a hardware way the Logic Controller, restoring its functions as after a power failing.
The forced reset of the watchdog circuit inside the program list can be necessary in programs with very long cycles. However we suggest to never arrive to this limit organizing with much care the programs and optimizing the instructions use.
The instruction RESWD is executed only if the last bit inserted in the accumulator register is “1”. The bits list of the accumulator register is not affected and then it is possible to execute several instruction with the same tested bit.

Example

The following example resets the value of the watchdog timer of the del MASTER if the bit M.0.0 is ON:

```
LD      M.0.0                           ‘enable condition
RESWD                                   ‘forced reset of the watchdog circuit
```

INCLUDE

Include in the program the instruction list of another file

Syntax

```
INCLUDE        FileName                    ‘comment
```

Argument

FileName is the name of a file without extension .PRG and containing program instructions.

Description

INCLUDE allows to insert in a particular point of the program list the instructions contained in a second file present in the compilation directory.
The name of the second file must be declared by means the maximum allowed 8 characters and without extension .PRG. The compiler in the presence of this instruction suspends temporarily the reading of the instructions from the principal file and starts to compile all the instructions found in the second file; when all the lines of the file end, the compilation starts again in the principal file from the following line of the instruction INCLUDE. In this way it is possible to divide the source file in several files, positioning in the secondary files, for example, common or recurrent parts of program or directly library subroutines.
The compiler can handle any number of include instructions provided that no secondary source file contains include instructions. Possible syntax errors found in the instructions of a include file are reported in the principal error file .ERR with indication of the file and the line containing the error.
The instruction INCLUDE is not conditioned for its execution; it is always executed because allows to insert instructions lists in the program.

Example

The following example shows how you can insert in the program all the instructions present in the source file AUXPROG.PRG:

```
INCLUDE        AUXPROG          ‘include the instructions of AUXPROG
```

PASSW

Declaration of the password of the specific program

Syntax

```
PASSW          String               ‘comment
```

Argument

String any character ASCII sequence up to 8 characters.

Description

PASSW allows to declare, inside the program list, the key word with which it is possible to read later program from the memory of the MASTER.
The instruction PASSW can appear in any point of the program list; the compiler executes this a instruction storing the provided string with a secret code and in hidden points of the object code. At this point any transferring operation of the object code from the MASTER to the Computer is forbidden, if you don’t specify, inside the programming environment of the Computer, the same password of the stored one in the program.
You can consider that without this instruction, the compiler automatically will declare a default password corresponding to the string PASSWORD; in this case you must select in the programming environment of the Computer a password equal to the string PASSWORD.
The execution of the instruction PASSW is not conditioned.

Example

The following example shows how to declare in the program the password “ELISA”:

```
PASSW          ELISA                    ‘declaration of the  password
```

Indirect operands for the instructions on bytes

All the instructions operating on bytes variables can use indirect operands.

An operand is indirect when not the value elaborated or modified by the instruction but its absolute address is contained directly in the bytes corresponding to the identifier.

The indirect operands are very important to execute repetitive operations on a data ensemble; parameterizing all the operations you can avoid to repeat more times the same piece of program for each data.

An operand becomes indirect preceding its identifier with the character @ (commercial a) without blanks. Obviously the used symbol to identify the operand must be considered a 2 bytes variable, such as to contain the absolute address pointing to the real operand.

In each instruction one or more indirect operands can exist. For example:

```
LD        M.0.0
MOVADD    M.100     M.110
MOVADD    M.200     M.210
MOVADD    M.300     M.310
MUL4      @M.100    @M.200    @M.300
```

represents a multiply instruction in which both the variables to multiply and the result variable, are identified by pointers (2 bytes variables M.100, M.200, M.300). Obviously, before using an instruction with one or more indirect operands, you must load in the pointer operands, the absolute address of the real operands (4 bytes variables M.110, M.210, M.310).

The instruction MOVADD allows to initialize the pointers; possible increments or decrements of the pointers allow to perform the same operations on data ensemble.

Using indirect operands on transferring bytes instructions, it is possible to generate output sequences by means values loaded in proper tables. For example:

```
LD        M.0.0
MOV1      M.100     K.11001010B
MOV1      M.101     K.00011010B
MOV1      M.102     K.11010101B
MOV1      M.103     K.01010111B
MOVADD    M.200     M.100
ADD2      M.200     M.200     M.300
MOV1      0.8       @M.200
```

the value transferred to the output byte 0.8 corresponds to the line of the table which number is contained in the variable M.300 (line address).

Evaluation of the arithmetic expressions

The ICL51 language provides a instructions set allowing a fast and easy evaluation of the expressions, without resorting to the classic arithmetic expressions of the programmable controllers.

We talk over a execution technic of the arithmetic expressions called " POLISH REVERSE NOTATION ". This technic is very efficient because allows to evaluate very complicated expressions without using brackets; you can evaluate fractional expressions with many terms in the numerator and denominator without worrying to store temporarily the intermediate result.

The reverse polish notation technic refers to the presence of a OPERATING STACK; it is a LIFO (Last In First Out) type list of signed 32 bits variables. The stack allows to store automatically and in a completely transparent to the Programmer up to 4 intermediate results in so many registers each with dimension 4 bytes:

STACK(3)	level 3
STACK(2)	level 2
STACK(1)	level 1
STACK(0)	level 0

You have to imagine the variables enter in the list (by means the instructions RCL1, RCL2, RCL4) from the bottom, that is from the level 0, shifting up the values already present:

STACK(3) <— STACK(2)
STACK(2) <— STACK(1)
STACK(1) <— STACK(0)
STACK(0) <— NEW VALUE

Before executing any arithmetic instruction you need therefore to load in the stack the input operands of the instruction.

For example to calculate the sum, indicated with S, of two signed 4 bytes variables indicated with A and B, you need to use twice the instruction RCL4:

```
RCL4     A          STACK(3) <— STACK(2)
                    STACK(2) <— STACK(1)
                    STACK(1) <— STACK(0)
                    STACK(0) <— A

RCL4     B          STACK(3) <—STACK(2)
                    STACK(2) <— STACK(1)
                    STACK(1) <— A
                    STACK(0) <— B
```

At this point it is possible to recall the sum instruction ADD (shortened mnemonic +) with which is calculated the sum A+B; the result is returned again in the stack in the level 0 position, and automatically the values present in the upper positions automatically go down of one level:

```
ADD                 STACK(2) <— STACK(3)
                    STACK(1) <— STACK(2)
                    STACK(0) <— A + B
```

When you have calculated the sum you need to transfer the result from the stack to the desired variable:

```
STO4     S          S <— STACK(0)
```

The stack system allows to evaluate very complex expressions and the intermediate result are automatically stored and recalled at the proper moment. At the first approach the system can seem complex; any way, once learned the mechanism, you will realize the immense possibilities offers in the evaluation of expressions also very complex. A Programmer of Logic Controllers will not find difficulties to learn this method because it is the extension to the numeric variables of the technic used for the boolean instructions; for example the instruction LD corresponds to the RCL1/2/4, the instruction OUT corresponds to the STO1/2/4, while the instructions ANDLD, ORLD work in a equivalent way to the instructions ADD, SUB, MUL, DIV.

We make another example a little more complex. We evaluate the expression:

$$R = A(B - C) + D$$

```
RCL4    B          STACK(0) <— B


RCL4    C          STACK(1) <— B
                   STACK(0) <— C

SUB                STACK(0) <— B - C

RCL4    A          STACK(1) <— B - C
                   STACK(0) <— A

MUL                STACK(0) <— A(B - C)

RCL4    D          STACK(1) <— A(B - C)
                   STACK(0) <— D

ADD                STACK(0) <— A(B - C) + D

STO4    R          R <— STACK(0)
```

All the arithmetic instructions available to process data in the stack work with signed 32 bits variables; this allows to have the maximum possible precision of the results because you can work with -2147483648 through +2147483647 data sizes.

To avoid to cut off the decimal part with the instruction DIV you need to multiply the dividend for a proper power of 10 before performing the division; the integer type result obviously will be multiplicated by the same factor (fixed point representation).

The evaluation of the arithmetic expressions is possible also for variables of size different by signed 4 bytes; indeed can be indifferently mixed signed 1 byte, signed 2 bytes, signed 4 bytes variables. At this purpose recalling and storing instructions are been provided in three forms: RCL1 and STO1 interface the stack with signed 1 byte variables, RCL2 and STO2 with signed 2 byte variables, RCL4 and STO4 with signed 4 byte variables.

Remember that a set of instructions for a whole evaluation of an expression must be active only for a cycle, at the purpose to not extend the looping time; the tested bit can be the same because these instructions do not affect the accumulator register.

At the end we make a last example to understand better the functioning of evaluation expression technic. When the bit M.0.0 is ON, we want to evaluate an inequality and, only if it is true, execute the subroutine MINOR:

$$\frac{\dfrac{A(B - 24C) - D}{256 + E} + 455}{\dfrac{F - G}{H} + 3455} < \frac{I - L}{M - 223}$$

The corresponding set of instructions is the following:

```
LD          M.0.0
RCL4        B
RCL4        K.24
RCL4        C
*
-
RCL4        A
*
RCL4        D
-
RCL4        K.256
RCL4        E
+
/
RCL4        K.455
+
RCL4        F
RCL4        G
-
RCL4        H
/
RCL4        K.3455
+
/
RCL4        I
RCL4        L
-
RCL4        M
RCL4        K.223
-
/
?
AND         F.<
GOSUB       MINOR
```

Optimize the program performance

This paragraph shows some easy programming artifices suggested to get the maximum performance by the system; particularly we are referring to performances regarding the processing speed of the user program and program memory occupation.

You can get considerable increments of performances, in the boolean instructions, using in following instructions, bits of the same byte. For example, the following list:

```
LD      M.0.0
AND     M.1.0
AND     M.2.0
AND     M.3.0
AND     M.4.0
AND     M.5.0
AND     M.6.0
OUT     M.7.0
```

occupies 49 bytes of memory and the looping time is 49 µs. Choosing the memory in the bits of the same byte you can write the same instructions in this way:

```
LD      M.0.0
AND     M.0.1
AND     M.0.2
AND     M.0.3
AND     M.0.4
AND     M.0.5
AND     M.0.6
OUT     M.0.7
```

This part of program occupies 21 bytes of memory and the looping time is 21µs.

If we consider the average values per instruction, in the first case we have 6.125 bytes/instruction and 6.125µs/instruction, while in the second 2.625 bytes/instruction and 2.625µs/instruction; the increment of performances with the second choice is 233%.

Obviously these choices are not always possible, but generally the habit to get following bits of memory will bring you to a better performance.

Another programming technic allowing you to increase the system performance is to use to identify the SLAVE boards, the COUNTER devices and the PULSE GENERATOR devices, numbers lower as possible without jumps in the numeration. This because the compiler recognizes, for each of the three ensemble, the maximum number of device recalled in the program list and transfers to the operating system of the Logic Controller these informations; during the updating of the resources the operating system automatically excludes to process all the devices with identifier number higher than the maximum. It is good practice to compact as more as possible the identifier numbers of the SLAVE boards, of the counters and of the pulse generator, towards numbers lower as possible.

For the using of arithmetic instructions, conversion functions and bytes or bytes blocks processing functions, you need to provide a pulse generator to enable them or a memory bit enabled only for a cycle. Actually these instructions take processing times considerably longer than the boolean instructions; for this reason it is suitable enable them for only one cycle.

We recommend to use jump instruction (JMP e JME) and jump to Label (GOTO) to exclude from the cyclic process program blocks temporarily not necessary. For example, if the program provides for several operating modes selectable by the operator, it is not necessary to process the instructions of all the modes, but it is suitable to jump the not selected part with an instruction.

Also using often subroutines inside the program allows to improve the performance; furthermore the use of indexed instructions allows to execute operations on data ensembles in a way certainly more efficient than the repetition of the same instructions for each data.

Summary tables of the ICL51 language

The following pages present the summary tables of the ICL51 language.

The Table 3 lists all the possible external and internal resources of the system with the proper terminology of identification.

The Table 4 reports the “map” of the data RAM memory of the MASTER, in which are placed all the system resources.

The Table 5 reports a whole list of all the instructions recognized by the language, with the syntax to use them and a summary description. For each instruction are provided validity fields of the operands referring to groups showed in the Table 6.

At the end the Table 7 gives you a brief scheme of the performance of the single instructions; the occupied bytes in the program code and the execution time of each instruction are reported. These performances depend on particular way of using of the instructions and for this reason you need to verify the corresponding notes.

Remember that the listed instructions are only the basic ones of the language and then always recognized by the compiler; i.e. all the possible added instructions to the compiler by means the creation of external routines in assembler language are excluded.

OPERAND = FIELD1.[FIELD2].[FIELD3]

FIELD1	FIELD2	FIELD3
EXTERNAL RESOURCES		
0 - 31 BOARD NUMBER	0 - 127 BYTE NUMBER	0 - 7 BIT NUMBER
INTERNAL RESOURCES		
M NOT RETENTIVE MEMORY	0 - 1023 BYTE NUMBER	0 - 7 BIT NUMBER
H RETENTIVE MEMORY	0 - 1023 BYTE NUMBER	0 - 7 BIT NUMBER
C 16 BITS COUNTER	0 - 127 DEVICE NUMBER	IN START COUNTING BIT OUT END COUNTING BIT CKUP COUNTING UP BIT CKDW COUNTING DOWN BIT CB CONTROL BYTE CL BYTE LOW CURRENT VALUE CH BYTE HIGH CURRENT VALUE FL BYTE LOW END VALUE FH BYTE HIGH END VALUE BIT / BYTE TYPE
P PULSE GENERATOR	0 - 127 DEVICE NUMBER	IN INPUT EDGE BIT OUTU RISING EDGE OUTPUT BIT OUTD FALLING EDGE OUTPUT BIT BIT / BYTE TYPE

Table 3.a. System resources

FIELD1	FIELD2	FIELD3
INTERNAL RESOURCES		
T TIME PERIODS	50 PERIOD 50 ms 100 PERIOD 100 ms 200 PERIOD 200 ms 500 PERIOD 500 ms 1000 PERIOD 1 s 2000 PERIOD 2 s BIT TYPE	
F SPECIAL FLAGS	0 BIT ALWAYS OFF 1 BIT ALWAYS ON P POWER ON PULSE < RESULT < = RESULT = > RESULT > C CARRY BIT E ERROR BIT BIT TYPE	
K CONSTANT	-2^{31} -2^{32} -1 DEC. 0H - F...FH ESA. 0B - 1...1B BIN. CONSTANT VALUE	
SXS SCAN x SEC. COUNTER		
W WATCH / CALENDAR	CB CONTROL BYTE SEC SECONDS MIN MINUTES HOUR HOURS DAY DAY OF WEEK DATE DAY OF MONTH MTH MONTH YEAR YEAR BIT TYPE	ADJ ADJUSTMENT BIT TYPE
X EXTENDED RETENTIVE MEMORY	0 - 24567 BYTE NUMBER	0 - 7 BIT NUMBER

CAUTION: EXTENDED RETENTIVE MEMORY IS NOT AVAILABLE ON EVERY PLC

Table 3.b. System resources

RAM (8000H - FFF7H)

	0.0	.7	.6	.5	.4	.3	.2	.1	.0	8000H
	0.1	.7	.6	.5	.4	.3	.2	.1	.0	8001H
MASTER (BOARD 0)										
	0.126	.7	.6	.5	.4	.3	.2	.1	.0	807EH
	0.127	.7	.6	.5	.4	.3	.2	.1	.0	807FH
	1.0	.7	.6	.5	.4	.3	.2	.1	.0	8080H
	1.1	.7	.6	.5	.4	.3	.2	.1	.0	8081H
SLAVE 1 (BOARD 1)										
	1.126	.7	.6	.5	.4	.3	.2	.1	.0	80FEH
	1.127	.7	.6	.5	.4	.3	.2	.1	.0	80FFH
	31.0	.7	.6	.5	.4	.3	.2	.1	.0	8F80H
	31.1	.7	.6	.5	.4	.3	.2	.1	.0	8F81H
SLAVE 31 (BOARD 31)										
	31.126	.7	.6	.5	.4	.3	.2	.1	.0	8FFEH
	31.127	.7	.6	.5	.4	.3	.2	.1	.0	8FFFH
	M.0	.7	.6	.5	.4	.3	.2	.1	.0	9000H
	M.1	.7	.6	.5	.4	.3	.2	.1	.0	9001H
NOT RETENTIVE MEMORY										
	M.1022	.7	.6	.5	.4	.3	.2	.1	.0	93FEH
	M.1023	.7	.6	.5	.4	.3	.2	.1	.0	93FFH
	H.0	.7	.6	.5	.4	.3	.2	.1	.0	9400H
	H.1	.7	.6	.5	.4	.3	.2	.1	.0	9401H
RETENTIVE MEMORY										
	H.1022	.7	.6	.5	.4	.3	.2	.1	.0	97FEH
	H.1023	.7	.6	.5	.4	.3	.2	.1	.0	97FFH
	C.0.CB					.CKDW	.CKUP	.OUT	.IN	9800H
	C.0.CL	BYTE LOW CURRENT VALUE								9801H
	C.0.CH	BYTE HIGH CURRENT VALUE								9802H
	C.0.FL	BYTE LOW END VALUE								9803H
	C.0.FH	BYTE HIGH END VALUE								9804H
16 BITS COUNTER										
	C.127.CB					.CKDW	.CKUP	.OUT	.IN	9A7BH
	C.127.CL	BYTE LOW CURRENT VALUE								9A7CH
	C.127.CH	BYTE HIGH CURRENT VALUE								9A7DH
	C.127.FL	BYTE LOW END VALUE								9A7EH
	C.127.FH	BYTE HIGH END VALUE								9A7FH

Table 4.a. Data RAM map

RAM (8000H - FFF7H)

PULSE GENERATORS	P.0						.OUTD	.OUTU	.IN	9C00H
	P.1						.OUTD	.OUTU	.IN	9C01H
	P.126						.OUTD	.OUTU	.IN	9C7EH
	P.127						.OUTD	.OUTU	.IN	9C7FH
REFERRING TIME	T			.2000	.1000	.500	.200	.100	.50	9F00H
SCAN COUNTER	SXS	BYTE LOW SCAN x SECOND								9F08H
		BYTE HIGH SCAN x SECOND								
SPECIAL FLAG	F	.E	.C	.>	.=	.<	.P	.1	.0	9F10H
WATCH / CALENDAR	W.CB								ADJ	9FF0H
	W.SEC	SECONS VALUE BINARY BYTE								9FF1H
	W.MIN	MINUTES VALUE BINARY BYTE								9FF2H
	W.HOUR	HOURS VALUE BINARY BYTE								9FF3H
	W.DAY	DAY OF WEEK VALUE BINARY BYTE								9FF4H
	W.DATE	DAY OF MONTH VALUE BINARY BYTE								9FF5H
	W.MTH	MONTH VALUE BINARY BYTE								9FF6H
	W.YEAR	YEAR VALUE BINARY BYTE								9FF7H
										9FFFH
EXTENDED RETENTIVE MEMORY	X.0	.7	.6	.5	.4	.3	.2	.1	.0	A000H
	X.1	.7	.6	.5	.4	.3	.2	.1	.0	A001H
	X.24566	.7	.6	.5	.4	.3	.2	.1	.0	FFF6H
	X.24567	.7	.6	.5	.4	.3	.2	.1	.0	FFF7H

CAUTION: EXTENDED RETENTIVE MEMORY IS NOT AVAILABLE ON EVERY PLC

Table 4.b. Data RAM map

PROGRAM LIST STRUCTURE:

```
PROGRAM =    LINE 1
             LINE 2
             LINE 3
             ..........
             ..........
             LINE n


LINE =     [“MEMORY_COMMENT]

LINE =     [OPERAND_LABEL = OPERAND]  [‘COMMENT]

LINE =     [JUMP_LABEL:]  [‘COMMENT]

LINE =     [INSTRUCTION]  [‘COMMENT]


INSTRUCTION = OPERATION  [OPERAND 1]  [OPERAND 2]  [OPERAND 3]
```

OPERATION	OPERAND 1 (GROUPS)	OPERAND 2	OPERAND 3	FUNCTION
LD	BIT (any)			Load the first bit
LDNOT	BIT (any)			Load the first reversed bit
AND	BIT (any)			And with a bit
ANDNOT	BIT (any)			And with a reversed bit
OR	BIT (any)			Or with a bit
ORNOT	BIT (any)			Or with a reversed bit
ANDLD				And between intermediate results
ORLD				Or between intermediate results
OUT	BIT (1,2,3,4)			Out of a bit to a port
OUTNOT	BIT (1,2,3,4)			Out of a reversed bit to a port

Table 5.a. Summary table of the instructions

OPERATION	OPERAND 1 (GROUPS)	OPERAND 2 (GROUPS)	OPERAND 3 (GROUPS)	FUNCTION
SET	BIT (1,2,3,4)			Set of a bit (*)
RES	BIT (1,2,3,4)			Reset of a bit (*)
CPL	BIT (1,2,3,4)			Complement of a bit (*)
JMP				Jump to next JME (*)
JME				Target location of JMP
GOTO	LABEL (max 32 charact.)			Jump to LABEL: (*)
GOSUB	LABEL (max 32 charact.)			Subroutine execution included between LABEL: and END (*)
NOP				No operation
END				End of instruction list of program or subroutine
TIM	C.n.IN (n=0-127)	BYTE (5,6,7,8,11)		Timer handling with period 0.1"
CNT	C.n.IN (n=0-127)	BIT (any)	BYTE (5,6,7,8,11)	Counter UP handling
SFR	BYTE (5,6,7,8,9)			Shift left of a byte;input of F.C in the l.s.b. and out of the F.C in the m.s.b. (*)

*** = instruction execution only if the condition is ON.**

Table 5.b. Summary table of the instructions

OPERATION	OPERAND 1 (GROUPS)	OPERAND 2 (GROUPS)	OPERAND 3 (GROUPS)	FUNCTION
ANDB	BYTE1 (5,6,7,8,9)	BYTE2 (5,6,7,8,9,10)	BYTE3 (5,6,7,8,9,10)	BYTE1 <-- BYTE2 AND BYTE3 And of the single bits of 1 byte (*)
ORB	BYTE1 (5,6,7,8,9)	BYTE2 (5,6,7,8,9,10)	BYTE3 (5,6,7,8,9,10)	BYTE1 <-- BYTE2 OR BYTE3 Or of the single bits of 1 byte (*)
XORB	BYTE1 (5,6,7,8,9)	BYTE2 (5,6,7,8,9,10)	BYTE3 (5,6,7,8,9,10)	BYTE1 <-- BYTE2 XOR BYTE3 Exclusive or of the single bits of 1 byte (*)
CPLB	BYTE (5,6,7,8,9)			BYTE <-- NOT BYTE Complement of the single bits of 1 byte (*)
MOV1	BYTE1 (5,6,7,8,9)	BYTE2 (5,6,7,8,9,10)		BYTE1 <-- BYTE2 Copy of 1 byte variable (*)
MOV2	BYTE1 (5,6,7,8)	BYTE2 (5,6,7,8,11)		BYTE1 <-- BYTE2 Copy of 2 bytes variable (*)
MOV4	BYTE1 (5,6,7)	BYTE2 (5,6,7,12)		BYTE1 <-- BYTE2 Copy of 4 bytes variable (*)
CMP1	BYTE1 (5,6,7,8,9,10)	BYTE2 (5,6,7,8,9,10)		BYTE1 = BYTE2 ? Compare two 1 byte variables; risult in F.< F.= F.> (*)
CMP2	BYTE1 (5,6,7,8,11)	BYTE2 (5,6,7,8,11)		BYTE1 = BYTE2 ? Compare two 2 bytes variables; risult in F.< F.= F.> (*)
CMP4	BYTE1 (5,6,7,12)	BYTE2 (5,6,7,12)		BYTE1 = BYTE2 ? Compare two 4 bytes variables; risult in F.< F.= F.> (*)

*** = instruction execution only if the condition is ON.**

Table 5.c. Summary table of the instructions

OPERATION	OPERAND 1 (GROUPS)	OPERAND 2 (GROUPS)	OPERAND 3 (GROUPS)	FUNCTION
ADD1	BYTE1 (5,6,7,8,9)	BYTE2 (5,6,7,8,9,10)	BYTE3 (5,6,7,8,9,10)	BYTE1 <-- BYTE2 + BYTE3 Add 1 byte variables; overflow in F.C (*)
ADD2	BYTE1 (5,6,7,8)	BYTE2 (5,6,7,8,11)	BYTE3 (5,6,7,8,11)	BYTE1 <-- BYTE2 + BYTE3 Add 2 bytes variables; overflow in F.C (*)
ADD4	BYTE1 (5,6,7)	BYTE2 (5,6,7,12)	BYTE3 (5,6,7,12)	BYTE1 <-- BYTE2 + BYTE3 Add 4 bytes variables; overflow in F.C (*)
SUB1	BYTE1 (5,6,7,8,9)	BYTE2 (5,6,7,8,9,10)	BYTE3 (5,6,7,8,9,10)	BYTE1 <-- BYTE2 - BYTE3 Sub 1 byte variables; borrow in F.C (*)
SUB2	BYTE1 (5,6,7,8)	BYTE2 (5,6,7,8,11)	BYTE3 (5,6,7,8,11)	BYTE1 <-- BYTE2 - BYTE3 Sub 2 bytes variables; borrow in F.C (*)
SUB4	BYTE1 (5,6,7)	BYTE2 (5,6,7,12)	BYTE3 (5,6,7,12)	BYTE1 <-- BYTE2 - BYTE3 Sub 4 bytes variables; borrow in F.C (*)
MUL1	BYTE1 (5,6,7,8,9) (2 bytes)	BYTE2 (5,6,7,8,9,10)	BYTE3 (5,6,7,8,9,10)	BYTE1 <-- BYTE2 * BYTE3 Mul byte variables; overflow in F.E (*)
MUL2	BYTE1 (5,6,7,8) (4 bytes)	BYTE2 (5,6,7,8,11)	BYTE3 (5,6,7,8,11)	BYTE1 <-- BYTE2 * BYTE3 Mul 2 bytes variables; overflow in F.E (*)
MUL4	BYTE1 (5,6,7) (8 bytes)	BYTE2 (5,6,7,12)	BYTE3 (5,6,7,12)	BYTE1 <-- BYTE2 * BYTE3 Mu 4 bytes variables; overflow in F.E (*)

*** = instruction execution only if the condition is ON.**

Table 5.d. Summary table of the instructions

OPERATION	*OPERAND 1 (GROUPS)*	*OPERAND 2 (GROUPS)*	*OPERAND 3 (GROUPS)*	*FUNCTION*
DIV1	BYTE1 (5,6,7,8,9) (2 bytes)	BYTE2 (5,6,7,8,9,10)	BYTE3 (5,6,7,8,9,10)	BYTE1 <-- BYTE2 / BYTE3 Div 1 byte variables; divisor = 0 in F.E (*)
DIV2	BYTE1 (5,6,7,8) (4 bytes)	BYTE2 (5,6,7,8,11)	BYTE3 (5,6,7,8,11)	BYTE1 <-- BYTE2 / BYTE3 Div 2 bytes variables; divisor = 0 in F.E (*)
DIV4	BYTE1 (5,6,7) (8 bytes)	BYTE2 (5,6,7,12)	BYTE3 (5,6,7,12)	BYTE1 <-- BYTE2 / BYTE3 Div 4 bytes variables; divisor = 0 in F.E (*)
INC1	BYTE (5,6,7,8,9)			BYTE <-- BYTE + 1 Inc 1 byte variables; overflow in F.C (*)
INC2	BYTE (5,6,7,8)			BYTE <-- BYTE + 1 Inc 2 bytes variables; overflow in F.C (*)
INC4	BYTE (5,6,7)			BYTE <-- BYTE + 1 Inc 4 bytes variables; overflow in F.C (*)
DEC1	BYTE (5,6,7,8,9)			BYTE <-- BYTE - 1 Dec 1 byte variables; borrow in F.C (*)
DEC2	BYTE (5,6,7,8)			BYTE <-- BYTE - 1 Dec 2 bytes variables; borrow in F.C (*)
DEC4	BYTE (5,6,7)			BYTE <-- BYTE - 1 Dec 4 bytes variables; borrow in F.C (*)

*** = instruction execution only if the condition is ON.**

Table 5.e. Summary table of the instructions

OPERATION	OPERAND 1 (GROUPS)	OPERAND 2 (GROUPS)	OPERAND 3 (GROUPS)	FUNCTION
ABS1	BYTE1 (5,6,7,8,9)	BYTE2 (5,6,7,8,9,10)		BYTE1 <-- |BYTE2| Absolute value of 1 byte variable; if negative complements F.C (*)
ABS2	BYTE1 (5,6,7,8)	BYTE2 (5,6,7,8,11)		BYTE1 <-- |BYTE2| Absolute value of 2 bytes variable; if negative complements F.C (*)
ABS4	BYTE1 (5,6,7)	BYTE2 (5,6,7,12)		BYTE1 <-- |BYTE2| Absolute value of 4 bytes variable; if negative complements F.C (*)
NEG1	BYTE (5,6,7,8,9)			BYTE <-- -BYTE Neg (2 complement) of 1 byte variable (*)
NEG2	BYTE (5,6,7,8)			BYTE <-- -BYTE Neg (2 complement) of 2 bytes variable (*)
NEG4	BYTE (5,6,7)			BYTE <-- -BYTE Neg (2 complement) of 4 bytes variable (*)
BINBCD1	BYTE1 (5,6,7,8,9)	BYTE2 (5,6,7,8,9,10)		BYTE1 (BCD) <-- BYTE2 (BIN) Conv BIN-BCD of 1 byte (*)
BINBCD2	BYTE1 (5,6,7,8)	BYTE2 (5,6,7,8,11)		BYTE1 (BCD) <-- BYTE2 (BIN) Conv BIN-BCD of 2 bytes (*)
BINBCD4	BYTE1 (5,6,7)	BYTE2 (5,6,7,12)		BYTE1 (BCD) <-- BYTE2 (BIN) Conv BIN-BCD of 4 bytes (*)

*** = instruction execution only if the condition is ON.**

Table 5.f. Summary table of the instructions

OPERATION	OPERAND 1 (GROUPS)	OPERAND 2 (GROUPS)	OPERAND 3 (GROUPS)	FUNCTION
BCDBIN1	BYTE1 (5,6,7,8,9)	BYTE2 (5,6,7,8,9,10)		BYTE1 (BIN) <-- BYTE2 (BCD) Conv BCD-BIN of 1 byte (*)
BCDBIN2	BYTE1 (5,6,7,8)	BYTE2 (5,6,7,8,11)		BYTE1 (BIN) <-- BYTE2 (BCD) Conv BCD-BIN of 2 bytes (*)
BCDBIN4	BYTE1 (5,6,7)	BYTE2 (5,6,7,12)		BYTE1 (BIN) <-- BYTE2 (BCD) Conv BCD-BIN of 4 bytes (*)
SWAP	BYTE (5,6,7,8,9)			BYTE.7-4 <---> BYTE.3-0 Exchange of the nibbles of a byte (*)
RCL1	BYTE (5,6,7,8,9,10)			Stack(0) <-- BYTE Load in the stack of a signed 8 bits variable (*)
RCL2	BYTE (5,6,7,8,11)			Stack(0) <-- BYTE Load in the stack of a signed 16 bits variable (*)
RCL4	BYTE (5,6,7,12)			Stack(0) <-- BYTE Load in the stack of a signed 32 bits variable (*)
STO1	BYTE (5,6,7,8,9)			BYTE <-- Stack(0) Copy the stack(0) to a signed 8 bits variable; error in F.E (*)
STO2	BYTE (5,6,7,8)			BYTE <-- Stack(0) Copy the stack(0) to a signed 16 bits variable; error in F.E(*)
STO4	BYTE (5,6,7)			BYTE <-- Stack(0) Copy the stack(0) to a signed 32 bits variable; error in F.E (*)

*** = instruction execution only if the condition is ON.**

Table 5.g. Summary table of the instructions

OPERATION	*OPERAND 1 (GROUPS)*	*OPERAND 2 (GROUPS)*	*OPERAND 3 (GROUPS)*	*FUNCTION*
ADD				Stack(0) <--Stack(1) + Stack(0) Signed sum in the 32 bit stack; overflow in F.E (*)
SUB				Stack(0) <--Stack(1) - Stack(0) Signed sub in the 32 bit stack; borrow in F.E (*)
MUL				Stack(0) <--Stack(1) * Stack(0) Signed mul in the 32 bit stack; overflow in F.E (*)
DIV				Stack(0) <--Stack(1) / Stack(0) Signed div in the 32 bit stack; divisor = 0 in F.E (*)
CMP				Stack(1) = Stack(0) ? Signed comp in the 32 bits stack; result in F.< F.= F.> (*)
MOVADD	BYTE1 (5,6,7,8) (2 bytes)	BYTE2 (5,6,7,8,9)		BYTE1 <-- ADDRESS(BYTE2) Load the absolute address to a 2 bytes variable (*)
MOVASC	BYTE (5,6,7,8,9)	|STRING| (max 100 char.)		BYTE1 <-- ASC(STRING) Fill a memory block with ASCII string (*)
MOVBLK	BYTE1 (5,6,7,8,9)	BYTE2 (5,6,7,8,9)	BYTE3 (5,6,7,8,9,10) (max 255)	BYTE1..... <-- BYTE2..... Copy a block of BYTE3 bytes to another (*)
RESMEM	BYTE1 (5,6,7,8,9)	BYTE2 (5,6,7,8,9,10) (max 255)		BYTE1..... <-- K.0 Reset a block of BYTE2 bytes (*)

*** = instruction execution only if the condition is ON.**

Table 5.h. Summary table of the instructions

OPERATION	OPERAND 1 (GROUPS)	OPERAND 2 (GROUPS)	OPERAND 3 (GROUPS)	FUNCTION
IOREFR				Forces update of the external resources (I/O MASTER) (*)
RESWD				Forced reset of the Watch-Dog timer (*)
INCLUDE	STRING (max 8 charact)			Include of program lines from external file
PASSW	STRING (max 8 charact)			Password declaration of a program

*** = instruction execution only if the condition is ON.**

Table 5.i. Summary table of the instructions

GROUP	TYPE	LIST
1	BIT	0.0.0 - 31.127.7
2	BIT	M.0.0 - M.1023.7
3	BIT	H.0.0 - H.1023.7 X.0.0 - X.24567.7
4	BIT	F.C F.E F.< F.= F.> C.*.IN C.*.CKUP C.*.CKDW P.*.IN
5	BYTE	0.0 - 31.127
6	BYTE	M.0 - M.1023
7	BYTE	H.0 - H.1023 X.0 - X.24567
8	BYTE	C.*.FL C.*.CL SXS
9	BYTE	C.*.FH C.*.CH W
10	BYTE	CONSTANT MAX 1 BYTE
11	BYTE	CONSTANT MAX 2 BYTES
12	BYTE	CONSTANT MAX 4 BYTES

Table 6. Validity groups of the operands

ISTRUCTION	BYTES	NOTES	TIME (µS)	NOTES
LD	6/2/8/4	A	5/1/7/3	A
LDNOT	7/3/9/5	A	6/2/8/4	A
AND	6/2	B	6/2	B
ANDNOT	6/2	B	6/2	B
OR	6/2	B	6/2	B
ORNOT	6/2	B	6/2	B
ANDLD	2		2	
ORLD	2		2	
OUT	7/3	B	8/4	B
OUTNOT	9/5	B	9/5	B
SET	9/5	B	2-9/2-5	B,C
RES	9/5	B	2-9/2-5	B,C
CPL	9/5	B	2-9/2-5	B,C
JMP	5		2-4	C
JME	0		0	
GOTO	5		2-4	C
GOSUB	5		2-4	C
NOP	1		1	
END	1		2	

Table 7.a. Instructions performances

ISTRUCTION	BYTES	NOTES	TIME (µS)	NOTES
TIM	45		37-127-110	D
CNT	45		37-127-110	D
SFR	8		2-38	C
ANDB	17		2-19	C
ORB	17		2-19	C
XORB	17		2-19	C
CPLB	17		2-19	C
MOV1	11		2-13	C
MOV2	14		2-20	C
MOV4	20		2-34	C
CMP1	14		2-29	C
CMP2	21		2-36	C
CMP4	35		2-59	C
ADD1	17		2-29	C
ADD2	24		2-42	C
ADD4	38		2-68	C
SUB1	17		2-30	C
SUB2	24		2-43	C
SUB4	38		2-69	C
MUL1	17		2-42	C
MUL2	24		2-110	C
MUL4	38		2-342	C
DIV1	17		2-45	C
DIV2	24		2-240	C
DIV4	38		2-773	C

Table 7.b. Instructions performances

ISTRUCTION	BYTES	NOTES	TIME (µS)	NOTES
INC1	8		2-24	C
INC2	8		2-30	C
INC4	8		2-45	C
DEC1	8		2-24	C
DEC2	8		2-32	C
DEC4	8		2-47	C
ABS1	11		2-17-26	E
ABS2	14		2-28-43	E
ABS4	20		2-49-67	E
NEG1	8		2-14	C
NEG2	8		2-24	C
NEG4	8		2-40	C
BINBCD1	11		2-89	C
BINBCD2	14		2-248	C
BINBCD4	20		2-743	C
BCDBIN1	11		2-27	C
BCDBIN2	14		2-60	C
BCDBIN4	20		2-182	C
SWAP	8		2-9	C
RCL1	11		2-45	C
RCL2	15		2-49	C
RCL4	23		2-57	C
STO1	8		2-24	C
STO2	8		2-27	C
STO4	8		2-25	C
ADD	5		2-38	C
SUB	5		2-39	C
MUL	5		2-185	C
DIV	5		2-895	C
CMP	5		2-22-34	F

Table 7.c. Instructions performances

ISTRUCTION	BYTES	NOTES	TIME (µS)	NOTES
MOVADD	12		2-11	C
MOVASC	18+Ncar		2 - [8+10Nchar]	C
MOVBLK	17		2 - [20+27Nbyte]	C
CMPBLK	17		2 - [28+30NByte]	C
RESMEM	11		2 - [14+8Nbyte]	C
IOREFR	5		2 - see manual	C
RESWD	5		2-12	C
INCLUDE	0		0	
PASSW	0		0	

Table 7.d. Instructions performances

NOTES:

A:	FIRST_INSTR and NEW_BYTE / FIRST_INSTR and SAME_BYTE / FOLLOW_INSTR and NEW_BYTE / FOLLOW_INSTR and SAME_BYTE
B:	NEW_BYTE / SAME_BYTE
C:	NOT_EXECUTED - EXECUTED
D:	IN=0 - IN=1_COUNT - IN=1_END_COUNT
E:	NOT_EXECUTED - POS - NEG
F:	NOT_EXECUTED - DIFFERENT_SIGN - SAME_SIGN

External instructions

Add personalized instructions

The ICL51 language allows the Programmer the possibility to make new personalized instructions, to add to the basic ones already available, as if a version of the software perfectly satisfy to his specific needs.

To add new instructions you need some experience in the programming in ASSEMBLER LANGUAGE of the microprocessors of the INTEL® 80C51 family. In addition you need a common commercial ASSEMBLER program for this language, able to generate executable BINARY FORMAT files.

The used technic to make the ICL51 compiler recognises valid also the new instruction, is to create, for each new instruction, a object file whose name (8 characters maximum) corresponds to the instruction name and the extension is necessarily .IOF (Instruction Object File).

Then the created files must be placed in the current work directory; when the compiler in the program list finds a not basic instruction, before reporting an error, looks in the current directory for a file with the same name of the instruction and extension .IOF. If this file is found, its stored code is used as compilation code of the instruction.

As the basic instructions, the new instructions can have some proper operands, transferred with the same syntax. This allows to handle the personalized instructions as basic: nobody should be able to suspect the new instructions were intentionally realized and added externally to the compiler.

Transferring operands to the external instruction

The compiler handles essentially five modes of transferring parameters to the external assembler routine: these modes follows what happens to most of the basic instructions.

Particularly, indicating generically with NAME the chosen name for the new instruction, they are:

```
NAME1     byte1     byte2     byte3

NAME2     byte1     byte2     byte3

NAME4     byte1     byte2     byte3

NAME      byte

NAME
```

The first three allow to transfer to the external instruction two operands (byte2 e byte3) which can be respectively a 1, 2, 4 bytes variable or constant.

The effective value of these two variables, which generally are the source operands of an instruction, are loaded in some internal registers of the microprocessor, before calling the external code; the compiler requires obligatory the name of the instruction ends with the character 1 or 2 or 4, to distinguish if you have to transfer 1, 2, 4 bytes operands. The registers of passing these operands are the following:

```
01CH  <—  byte2(0)
01DH  <—  byte2(1)          (only  NAME2 and NAME4)
01EH  <—  byte2(2)          (only  NAME4)
01FH  <—  byte2(3)          (only  NAME4)

018H  <—  byte3(0)
019H  <—  byte3(1)          (only  NAME2 and NAME4)
01AH  <—  byte3(2)          (only  NAME4)
01BH  <—  byte3(3)          (only  NAME4)
```

Regarding the byte1 operand, generally destination operand of the instruction, it is transferred exclusively the absolute address of its data RAM location (or the lower byte for NAME2 e NAME4); This address is loaded in the 16 bits pointer register called DPTR (refer to the specialized documentation of the family 80C51 of microprocessors).

The fourth mode of transferring parameters allows to pass the absolute address of the byte operand in the pointer register DPTR, in the same way of the preceding modes; in addition, if the operand is a bit type, besides the loading of the absolute address of the byte in the DPTR, in the register B the mask (only one bit is 1) identifying the position of the bit in the byte.

At the end the fifth mode does not allow to transfer parameters, but only to execute the present code in the external file.

To finish the description of transferring parameters we can say, for example, the first three modes are equivalent to what happens for an instruction respectively of type ADD1, ADD2, ADD4, the fourth mode to what happens for the instruction SFR and at the end the fifth one for the instruction ANDLD.

At last we show the absolute address of the internal memory of the microprocessor, used for the operating stack and handled by the evaluation instructions of the equations in according with the Polish reverse notation:

004H	<— byte(0)	STACK(3)
005H	<— byte(1)	
006H	<— byte(2)	
007H	<— byte(3)	
008H	<— byte(0)	STACK(2)
009H	<— byte(1)	
00AH	<— byte(2)	
00BH	<— byte(3)	
00CH	<— byte(0)	STACK(1)
00DH	<— byte(1)	
00EH	<— byte(2)	
00FH	<— byte(3)	
010H	<— byte(0)	STACK(0)
011H	<— byte(1)	
012H	<— byte(2)	
013H	<— byte(3)	

How to create an external instruction

To crate an external instruction you need, as already said in the preceding sections, to edit a file in assembler 80C51 language. In succession you need to assemble this source file to generate a binary file containing the machine language of the microprocessor and change the extension of that file in .IOF.

When the compiler ICL51 will find an externally defined instruction, it will provide automatically to generate the code corresponding to the loading of the instruction operands and then it will queue to that code the contents of the file .IOF.

The external instructions requiring the transferring of parameters will have to process the passed values in the registers and return the result in the data RAM using for example the present address in the DPTR.

The written code in assembler is fully copied in the user program code (file .OBJ), so it is essential it ends always its execution without infinite loop, otherwise the watchdog circuit of the Logic Controller will occur to force a reset.

The assembler program structure must be therefore “open”, i.e. it must allow to continue the execution of all the other instructions of the user program. In addition the generated code must not contain particular absolute addresses proper of the program: the possible jumps of program must be relative because it is not predictable what absolute address the compiler will place the code. So we suggest to start the assembler program with the instruction:

```
ORG   0000H
```

Now we make some considerations on another key-point of the assembler programming. The Boolean instructions are processed by means the help of a stack of bits of LIFO type (Last In First Out): in this stack are stored temporarily all the intermediate bits of the boolean calculations.

In detail the stack is made by 8 bits; the level 0 bit, corresponding to the input and output position of the stack, coincides with the carry bit C of the microprocessor.

The handling of this stack is made automatically by the compiler; particularly a position pointer, initialized on the bit 71H, allows to store temporarily the calculated value in the carry C every time a new instruction LD or LDNOT is met.

If you have only one instruction LD or LDNOT only the carry C is enough for the calculation, otherwise the intermediate results are stored in succession from the bit 71H to the bit 77H of the internal memory of the microprocessor. The instructions ANDLD and ORLD decrement by a level the stack performing the operation between the carry bit C and the last intermediate result stored in the bit 71H.

If the execution of the external instructions needs the carry bit C for its necessities, it can be temporarily stored in the bit 70H and resumed at the end of the instruction code; the stack indeed must be unchanged after the instruction execution.

The principal interest is for the level 0 bit i.e. the carry C of the microprocessor, because it contains the value ON/OFF of the current point of the program; generally many instructions are executed only if the logical value of this bit is “1”, while they are skipped if it is “0”. When this characteristic is required you will have to close the assembler list between the following two lines of program:

```
 JNC          Istruction_End

 -------------------------------

 (instruction list)

--------------------------------

Istruction_End:
```

In this way if the contained value in the carry C is the logic “0”, the assembler instructions block are skipped and the external instruction executes nothing.

In addition because the logic preceding result, can enable more output instructions, you must not affect the carry C value, so after the external instruction execution, other instructions can use its contents. We suggest at this purpose to save the carry C contents in the bit of internal memory 70H:

```
 MOV          70H,C

--------------------------------------------------------

 (instruction list with the bit C available)

--------------------------------------------------------

 MOV          C,70H
```

The assembler program can refer to any internal and external (data RAM) RAM block of the microprocessor, provided that you don’t execute writing instructions that affect, in a fatal way frequently, the execution of the operating system. To make easier the things we suggest to use, for the proper assembler program, only the following internal register of the microprocessor:

ACC
PSW
B
BANK 0 REGISTERS
BANK 1 REGISTERS
BANK 2 REGISTERS
BANK 3 REGISTERS
DPTR
STACK (max deep available: 32 bytes)

The external instruction MUX

With the following example an external instruction is created, allowing to realize a MULTIPLEXER device with 8 output. The instruction MUX needs only one operand that is the identifier of the byte you want to convert; after the conversion the result is returned in the same byte. The conversion is made only if the condition of the instruction MUX is ON; in this way the conversion is the following:

BYTE (before the MUX)	BYTE (after the MUX)	
0	00000001B	
1	00000010B	
2	00000100B	
3	00001000B	
4	00010000B	
5	00100000B	
6	01000000B	
7	10000000B	
8÷255	00000000B	(F.E = 1)

The instruction works only with 1 byte variables which validity field of the operand before the conversion is 0-7; in all the other cases you have an error condition reported forcing ON the flag bit F.E.

This example, properly modified, is the base for developing any other conversion operation of the value of a byte in according with the stored values in a table.

We remember the object code corresponding to an external instruction (file with extension .IOF) is wholly inserted in the object code (file .OBJ) of the user program every time you call the instruction.

So pay attention not to create too long files .IOF and, in the case this is not possible, not to call too frequently the instruction in the program file. A way to remove this difficulty is to define a subroutine containing the call to the external instruction; in this way the object code of the instruction will be incorporated only once in the file .OBJ of the user program.

At the end pay attention to remove the possible ending part of the created binary file by the used assembler program; often indeed the generated binary code have dimensions multiple of some quantities (for example 128 byte) and they fill the ending part with operating codes corresponding to the NOP instructions. The ending part if not removed, will fill unnecessarily the program memory and will slow the instruction execution.

We show later the assembler list of the external instruction MUX:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                           ;;
;; instruction:     MUX                                                      ;;
;;                                                                           ;;
;; operand:         byte           (dimension 1 byte)                        ;;
;;                                                                           ;;
;; syntax:          MUX   byte   ‘comment                                    ;;
;;                                                                           ;;
;; function:        when the condition is ON it converts the value of the    ;;
;;                  byte in the position of the bit; if the byte to convert  ;;
;;                  is not 0,1,2,3,4,5,6,7, after the instruction, the byte  ;;
;;                  is 0 and the error flag F.E is ON                        ;;
;;                                                                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


EXT_FLAG          =     09F10H                  ;byte pointer of the FLAGS

                  ORG   0000H


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


                  JNC   MUX_END                 ;jump if the condition is OFF
                  MOV   70H,C                   ;save temporarily the carry
                  MOVX  A,@DPTR                 ;load in ACC the byte
                  CJNE  A,#8,TRY_LESS           ;verify if byte < 8
                  SJMP  MUX_ERROR
TRY_LESS:         JNC   MUX_ERROR
                  ADD   A,#2
                  MOVC  A,@A+PC                 ;load in ACC the table byte
                  SJMP  MUX_OK
                  DB    00000001B               ;value for byte = 0
                  DB    00000010B               ;value for byte = 1
                  DB    00000100B               ;value for byte = 2
                  DB    00001000B               ;value for byte = 3
                  DB    00010000B               ;value for byte = 4
                  DB    00100000B               ;value for byte = 5
                  DB    01000000B               ;value for byte = 6
                  DB    10000000B               ;value for byte = 7

MUX_ERROR:        MOV   A,#0
                  MOVX  @DPTR,A
                  MOV   DPTR,#EXT_FLAG          ;byte pointer of the FLAGS
                  MOVX  A,@DPTR
                  SETB  ACC.7                   ;set the bit F.E
                  MOVX  @DPTR,A
                  MOV   C,70H                   ;restore the carry
                  SJMP  MUX_END

MUX_OK:           MOVX  @DPTR,A
                  MOV   C,70H                   ;restore the carry
MUX_END:


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


                  END
```

External instruction SQR

The following example shows how to create new arithmetic instructions to use on the operating stack. Particularly a SQUARE ROOT instruction is developed, operating on the present value in the level 0 of the stack and returning the result in the same position.

The used algorithm for the evaluation of the square root is very easy and it is scheduled in the flow diagram of the Figure 1.

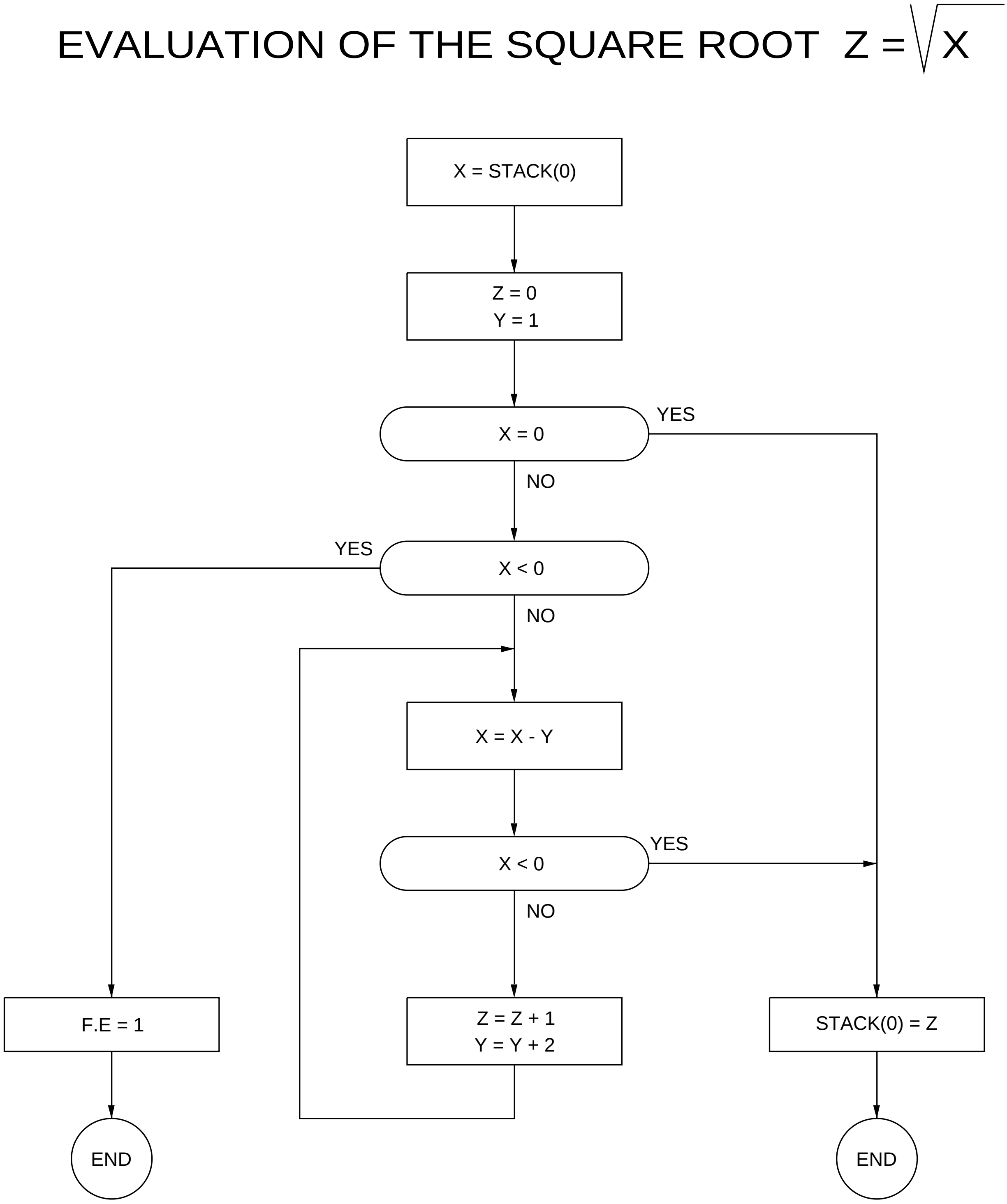


Figure 1. Flow diagram for the square root evaluation

To make easier the correspondence between the showed flow diagram and the assembler list implementing the instruction SQR, we make reference to symbolic names (X, Y, Z) for the used variables in the algorithm. The variable X is the source value of which you want to know the square root and it is contained in the level 0 of the operating stack; the variable Y is a counter necessary inside the algorithm and at last Z is the destination value i.e. the wanted value of the square root. At the end if the square root calculus does not fail, the Z value is returned in the level 0 of the stack.

The assembler list corresponding to the SQR instruction is the following:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                           ;;
;; instruction: SQR                                                          ;;
;;                                                                           ;;
;; syntax:      SQR      ‘comment                                            ;;
;;                                                                           ;;
;; function:    integer square root calculus of the value present in         ;;
;;              the stack. The result replaces the value without             ;;
;;              affecting the stack position.                                ;;
;;                                                                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


BIT_STACK       =     02EH                    ;ram byte for bit stack
EXT_FLAG        =     09F10H                  ;user flag of status
RESET_WD        =     0FC0H                   ;reset watchdog routine


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


                  ORG   0000H

                  MOV   BIT_STACK.0,C           ;save the bit stack

                  MOV   014H,#0                 ;reset the result Z = 0
                  MOV   015H,#0
                  MOV   016H,#0
                  MOV   017H,#0
                  MOV   018H,#1                 ;odd counter Y = 1
                  MOV   019H,#0
                  MOV   01AH,#0
                  MOV   01BH,#0

                  MOV   A,010H                  ;test if X = 0
                  ORL   A,011H
                  ORL   A,012H
                  ORL   A,013H
                  JZ    SQR_END

                  MOV   A,013H                  ;test if X < 0
                  JB   ACC.7,SQR_NEG

SQR_LOOP:         CLR   C                       ;calculate X = X - Y
```

```
                  MOV   A,010H
                  SUBB  A,018H
                  MOV   010H,A
                  MOV   A,011H
                  SUBB  A,019H
                  MOV   011H,A
                  MOV   A,012H
                  SUBB  A,01AH
                  MOV   012H,A
                  MOV   A,013H
                  SUBB  A,01BH
                  MOV   013H,A

                  JB    ACC.7,SQR_END           ;test if X < 0

                  MOV   A,014H                  ;increment Z = Z + 1
                  ADD   A,#1
                  MOV   014H,A
                  MOV   A,015H
                  ADDC  A,#0
                  MOV   015H,A
                  MOV   A,016H
                  ADDC  A,#0
                  MOV   016H,A
                  MOV   A,017H
                  ADDC  A,#0
                  MOV   017H,A

                  MOV   A,018H                  ;increment Y = Y + 2
                  ADD   A,#2
                  MOV   018H,A
                  MOV   A,019H
                  ADDC  A,#0
                  MOV   019H,A
                  MOV   A,01AH
                  ADDC  A,#0
                  MOV   01AH,A
                  MOV   A,01BH
                  ADDC  A,#0
                  MOV   01BH,A

                  LCALL RESET_WD                ;reset of watchdog
                  SJMP  SQR_LOOP

SQR_NEG:          MOV   DPTR,#EXT_FLAG          ;square root of the negative value
                  MOVX  A,@DPTR
                  SETB  ACC.7                   ;set the error bit
                  MOVX  @DPTR,A
                  SJMP  SQR_EXIT

SQR_END:          MOV   010H,014H               ;transferring the result Z
                  MOV   011H,015H
                  MOV   012H,016H
                  MOV   013H,017H

SQR_EXIT:         MOV   C,BIT_STACK.0


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


                  END
```

Development environment

Programming with PC

The ICL51 software runs on IBM® (or compatible) Personal Computer with MS-DOS operating system; it allows all the develop operations of the user program with ICL51 language and it allows also an immediate utilization of the Logics on the machine or installation for automation.

To programme you need therefore a normal Personal Computer with the following characteristics:

- 640 Kbytes of RAM memory
- a RS232 serial interface
- a floppy-disk driver for 1.4 MBytes diskette
- Hard-disk with at least 5 MBytes free

You see immediately that the required qualifications for the Computer to programme are extremely small and widely satisfied by any electronic machine in sale.

You can write the programme of the automatic machine using any text Editor, on condition that this is able to elaborate text files "pure" that is without special and control characters typical of the specific Editor. The choice of this Editor is left to the Programmer, to allow him to use that he knows better; in any case, with the ICL51 software release 4.0, the EDIT51.EXE editor is given, especially qualified for this task. In every way the external Editor can be called from within the develop environment ICL51.

Through the Editor you will have to prepare a text file containing all the lines of the program, in ICL51 language, according to the syntax described in the special sections of this manual.

Successively a compiler programme will allow you to translate all the informations contained in the source list, before edited, in informations which the microprocessor of the logic understands. This happens with creation, by the compiler, of a object file that forms a equivalent version of the edited programme but in assembler language, directly employed by the microprocessor.

The transfer functions of the program and of its data from and to the Logic are possible connecting the MASTER board to the serial interface of the Personal Computer.

The ICL51 programme allows you also to debug the machine program by means his execution in real time: during the execution is possible monitoring and forcing all the internal variables of the Logic.

An easy flow diagram of the operations, presented by the principal menu of the ICL51 software, will guide also the less skilled Programmer during all the phases of the automation program developing.The great facility and immediateness of the system will allow the Programmer to concentrate exclusively on the program to develop and not on the "jungle" of menu and submenu of the develop software.

ICL51 software installation

The ICL51 software packet is normally provided with floppy-disks containing all the necessary files for its working and a wide set of applicative and demonstrative programs . Future releases are possible by means of floppy disks containing new files to overwrite on the old ones or to add up these. Diffusion of ICL51 software is not subject to any licence for using; moreover you can take the ICL51 software packet, its releases, auxiliary programs and all the documentation in .PDF extension every time you want in the INTERNET network.

Beginning to work you have first to install the ICL51 program on the Personal Computer you want to use for programming the Logics. Full freedom in installation and in laying of the directory is left to the user of the software; afterwards we give only some indications and informations useful in this phase.

Create a directory, for example with name ICL51_40, and copy all the files from the disk to such directory; supposing to work on hard disk C: and to have inserted the floppy-disk of the ICL51 program in drive A:, the operations are the following:

```
C:\

MD ICL51_40

CD ICL51_40

COPY A:*.* C:
```

Warning: normally, because of a large amount of memory required by all the files, the provided disks with the software packet or its releases contains compressed versions of data. Decompress such files to get them in a primary and useful form.

We suggest also to crate a subdirectory of ICL51_40 to put the user made programs, in order to keep them apart from the executable programs of the develop environment.

To recall the ICL51 program from a possible other directory, add in the file AUTOEXEC.BAT the operand C:\ICL51_40 to the command PATH.

Finally you have to make a copy of all the configuration files *.CFG in the current directory of work. These files allow you to store all the chosen configurations to operate with the system; the configurations can be also different for every particular user program, arranging before some copies of the configuration files in every subdirectory of work and changing appropriately the options in function of the particular program to develop.

The release 4.0 of the ICL51 software is been completely reorganized to make easy and automatic its updating in according to the following develop of new Logic Controller. Actually some files are general purpose and good for all the Logic Controllers, whereas others are specific for each Logic Controller. In this way to update the software for a new product you must only add in the ICL51_40 directory new files concerning it and automatically it will be recognized by the programming environment.

To understand better how is structured ICL51 software release 4.0 we show a summary list of all the files which compose it. With the generic word LOGIC we want to mean one or more Logic Controllers one has to use and for which is required the presence of the corresponding files; with the word FILENAME we mean the name of one or more user programs of example given like equipment of the software:

ICL51.EXE	main executable program (main menu)
CFG51.EXE	general configuration program (recalled from the main menu)
ICL51.CFG	general configuration file of the environment
VIEW51.EXE	program for files visualization (recalled from the main menu)
EDIT51.EXE	program for editing of files (recalled from the main menu)
ICL51.HLP	file for help recalled inside the environment

C_LOGIC.EXE	compilation program (recalled from the main menu)
T_LOGIC.EXE	transfer program (recalled from the main menu)
M_LOGIC.EXE	monitor program (recalled from the main menu)
M_LOGIC.CFG	configuration file for the monitor

Warning: you need these last 4 files to use a particular Logic Controller. Normally the ICL51 software set is given with the files of all the commercial Logic Controllers.

FILENAME.PRG	source file of the user program for the specific application
FILENAME.OBJ	object file of the user program (to transfer to the logic)
FILENAME.ERR	report file for possible errors in the compilation of the user program
FILENAME.TXT	text file stored in the user program after its back up
FILENAME.HLD	H type retentive memory data file
FILENAME.XMF	X type retentive memory data file (available only for some Logics)

We remember at last that the files with extension .IOF correspond to external instructions; particularly with the ICL51 software release 4.0 are given two examples (MUX and SQR) explained in this manual.

ICL51 develop environment

Whole develop environment of the user program is handled by the executable ICL51.EXE file. To start working with ICL51 program you have to call the principal menu typing from the DOS command line:

```
ICL51
```

The principal screen of ICL51 environment is very easy and clear. A graphic diagram in five phases can condense the principal phases to follow during the develop of the own application; you can recall these phases whether by means the showed functional key, or typing the ENTER key after being placed with the directional keys (Arrows) on the desired square. The pointer itself is placed automatically on the proper square in according with the previous executed phase and on the basis of its results.

Other auxiliary functions, recalled by the function keys, are available in this principal menu and a status window shows the conditions of general configuration of the environment.

The functions available in the principal menu are the following:

Edit (F1)

Recall the selected text Editor for writing the instruction list of the source program .PRG. This is the first step in the development of an applicative program; the selected text editor is executed transferring as a parameter the source program name.

Compile (F2)

It compiles the source program .PRG generating the object file .OBJ and a error file .ERR. This step resorts to the specific of the selected Logic Controller and changes the list program, written in ICL51 language, in a equivalent program but understandable by the microprocessor of the Logic.

View Error (F3)

Recall the selected text visualization program transferring as a parameter the error file name .ERR of the current program. It is necessary to see that file when the compilation process, at its end, reports the presence of errors.

Transfer (F4)

Recall the specific program of transferring of the selected Logic. This program shows a new menu allowing several choices of transferring from and to the Logic Controller. A thorough knowledge of this subject will be performed in a paragraph provided for the purpose.

Monitor (F5)

Recall the specific program of monitor of the selected Logic Controller. On the screen of the monitor is showed the list of the current program and in the half below is possible to open the window of the monitor. This window allows to see and force the status of all the internal variables in the RAM memory of the Logic Controller (resources) to test the program execution. The program of the monitor will be more examined later.

Dos Shell (F6)

Allows you to go momentarily out of ICL51 environment and to execute any DOS command or program. To return back, ending the shell, you have to type the EXIT command through the keyboard.

Configure (F7)

Recall the configuration program CFG51.EXE if you need to change one or more among the general setting of the ICL51 environment. The information of general setting are stored in the file ICL51.CFG.

Help (F8)

Shows, by means the selected program of visualization, the file ICL51.HLP containing the text of help for using the environment.

Esc

The Escape key allows you to quit the principal menu of the programming environment ending the ICL51 program.

Software configuration

Before starting the first time to work to a new project, you need to configure properly the ICL51 develop environment.

From the principal menu type the function key F7 to have access to the configuration program; temporarily CFG51.EXE program is executed which allows you to see and change the existing setting up in the general configuration file of the environment (ICL51.CFG).

We present eight different options of configuration. To change the configuration, place yourself through the direction keys (Arrows) on the chosen option and type the ENTER key. Some options can have only few alternate values; in this case typing repeatedly the ENTER key will allow to change the value of the option. In the other cases, on the contrary, it is necessary to give a name typing it in full or selecting it in a showed list; with the Esc key you can cancel the selection. Typing the Esc key from the configuration menu you end this program and you return to the principal menu of the environment.

We report later a specific and particularized description of the possible options of configuration and how you can change them:

Logic type

Selects the type of MASTER Logic Controller chosen for the application. Typing the ENTER key is showed a list of all the available Logic Controllers; for the use of the Logic Controller is tested the presence at least, in the current directory of work, of the executable file of compilation (C_LOGIC.EXE). The name of the Logic Controller is extracted out of the name itself of the compiler program removing the prefix “C_” and next it will be used to call also other specific program of the Logic (transfer “T_” and monitor “M_”).

To select the Logic place yourself on the chosen one and confirm with ENTER; the Esc key close the window of the Logic list and cancel the selection.

File

Means the file name of the project to use currently in all the operation performed by the environment. The specified name (max 8 characters) will be automatically provided with the correct extension required in the different steps of program development.

To select the user program name type ENTER on the option itself. At this stage it is possible to type the full name of the file or recall one or more programs, already present in the current directory of work, typing a name with the jolly character ? (question mark) and * (asterisk). The character ? replaces only one generic character of the name to search, while the character * is equivalent to a part of the name made up of generic characters. All the names conforming to the characteristics of search will be listed in a window; at this way, placing with the direction keys and confirming with ENTER, you will select the needed program name.

Logic at

Select the number of the serial output RS232 to use for all the communication operations between the Personal Computer and the chosen Logic Controller.
Typing in sequence the ENTER key you can change the value of the current selection; the serial outputs you can chose, are COM1, COM2, COM3 and COM4.

Printer at

Select the number of parallel output CENTRONICS to use for all the printing operation executed inside the programming environment.
Typing several times the ENTER key you can select one of the possible parallel output (LPT1, LPT2 and LPT3).

Automatic

Allows to enable or not the function of automatic recall of compilation, transfer (download) and monitor at the exit from the Editor of the source file.
With the ENTER key you can change from "Yes" to "No" this function. Selecting "N" at the end of the Editing program, you return to the principal menu of the environment and all the following stages must be recalled manually. With the option "Y" instead, at the exit of the program of Editor, automatically the compiler is started and, if no errors are checked in this phase, the corresponding object file will be transferred to the Logic; finally the menu of monitor will be recalled automatically. This automation of the operations is very useful to accelerate the phase of regulation of the program when you need to do continuous changes and tests.

Editor

Allows to select the Editor program name to use for writing the source list. Typing ENTER it is possible to insert the name (without extension) of the executable program to use; we underline that this text Editor must be available in the current directory of work or at least viewable through the command PATH in the file autoexec.bat.
The Editor program is started by the principal menu by means a command line DOS formed by the name of the Editor succeeded by the applicative program name with extension .PRG. If the Editor you want to use, does not provide the transfer in the command line of the file name to edit, you will need to make a batch file which works in a intermediate way between the programming environment call and the external program of editing.
With the software packet ICL51 release 4.0 is issued the program EDIT51, particularly studied for writing the source list. Such program is very easy and practical; through the function key F1 it's possible to see from inside a list of command and available functions.

Viewer

Allows to select the program name of visualization of text files to use in several places of the environment where you need that function.
Typing the ENTER key you can insert an external file name of visualization; the transfer of the file name to show happens in the same way adopted for the Editor program.
We advise you to use the VIEW51 program given with the ICL51 release 4.0 software packet because it holds, in a easy and direct way, all the possible functions this program needs.

Password

Setting the Password of the programming environment is necessary only in the recovery phases of the informations from the Logic to the Personal Computer.
The Password is made up of a string (maximum 8 characters) and is compared with the stored one in the program memory of the Logic (by means the instruction PASSW of the source program). When the two Password are not the same, some operations like UpLoad and Compare are avoided.
With the ENTER key is possible to type the environment Password. We remember, if you don’t establish in the source list the Password of the Logic, by means the instruction PASSW, the compiler of default will insert in the object file the Password “PASSWORD”; in this case you will need to verify that also the environment Password is the string “PASSWORD”.

Writing the program

Writing the text file containing the instructions list in ICL51 language is the first required step in the develop of own applicative program.

For the writing of this text file, named source program and with extension .PRG, it’s necessary to use a common text Editor. Many commercial programs exist for the writing of this files; some of them are been specially performed to write programs, independently of the language. The freedom to use a text Editor which one already knows, allows the Programmer to begin to work immediately and with the maximum efficacy.

However, to complete the ICL51 software packet release 4.0, it is given a program of text Editor called EDIT51: this Editor is been produced to allow the programming of the Logic Controllers all those who don’t have a proper preferential Editor.

The text file must contain all the instruction needed by the program in execution in according with the syntactic rules described in the parts of this manual provided for this purpose. Not recognized instructions or applied in a wrong way will be identified and listed by the compiler.

The compilation and the error file

The compilation is the following phase to the writing of the source text file of the program. This phase is necessary when you want to test the program on the Logic Controller after you have made a modify in the source file. Indeed you need to update the object file .OBJ when you have new informations in the source file .PRG.

The compiler tests row by row the instructions of the source file and generates the corresponding assembler code understandable to the microprocessor of the Logic Controller. During the verify of the lines of program, all the errors of using the instructions are listed in the error file .ERR for a next reference.

The error file .ERR is always generated by the compiler because contains also other kinds of informations like per cent of memory occupation of the current program; this data is very important because it allows you to control the availability of the program memory to accept the add of new instructions.

Transferring menu

Starting the transferring menu are showed some available command options. In this menu are putted together all the transferring functions from and to the Logic both of the machine program (stored in the FLASH-EPROM memory) and of the data and work parameters (stored in the RAM memory).

All the transferring operations happen through the serial Input/Output RS232 between the Personal Computer and the Logic Controller. Before proceeding verify the connection is been correctly realized and the configuration options (serial number COM used) are opportunely selected.

The function keys of the keyboard of the Personal Computer are redefined for the functions of the transferring menu in accordance with the following things:

DownLoad (F1)

Starts the transfer of the program from the Personal Computer to the selected Logic Controller.

The object file .OBJ is programmed in a permanent way aboard the FLASH-EPROM memory of the Logic after the prior erasure of any existing program. Possible problems occurred during the programming of the FLASH-EPROM memory will be opportunely indicated by the software.

At this purpose we analyse a particular situation which can happens during the programming.

The Logic Controller, on power on, tests the presence of a program loaded in the FLASH-EPROM memory and in this case it enters automatically in the execution state of the (RUN).

In the case instead of a empty FLASH-EPROM memory, is automatically initialized the stop state (STOP) waiting for the DownLoad of a program.

A third possible case exists, which is that one of a wrong program, for any reason, or with wrong jump or loop without end produced by a wrong functional writing of the user program. In these cases there will be a continuous occurrence of the watchdog circuit of the board with resultant impossibility to update the program. To leave this situation you need to force a STOP on power on for the Logic Controller, by means the jumper or dip-switch provided for this purpose or by means the function key F9 (for the Logics that contemplate it) of this menu. Following to the enter of the Logic Controller in the STOP state it is possible to execute the DownLoad of a correct program.

UpLoad (F2)

Starts the transfer of the program from the selected Logic Controller to the Personal Computer.

With this command it is possible to read the program inside the FLASH-EPROM memory of the Logic Controller storing it in the file .OBJ. Any area of comment texts (comments of the original source program preceded by the character " double quote) will be stored in the file .TXT for a further reference.

This transferring operation is possible only if the Password stored on the Logic Controller corresponds with the one setted in the environment.

We want to underline that the recovered program file from the Logic Controller is only the object file .OBJ; the source file .PRG cannot be recovered because its size in bytes is often very big. You have to think indeed to all the alphanumeric Label of 32 characters or the comments to the instructions that can be present in a source file; we could limit the size of the source file and make it less generous in its style to store it in the memory of the Logic Controller.

View Text (F3)

Recalls the visualizer of selected text file passing the file .TXT.
The text file .TXT is automatically created by the function of UpLoad if, in the file .OBJ recovered from the memory of the Logic, is been found at least one comment of recordable type.
In the source list .PRG it is possible indeed to type full lines of comment text starting them with the character “ (double quote); these text lines are included, with a secret code, in the file .OBJ to transfer to the memory of the Logic Controller.
The purpose of these comments is to have a kind of "block-notes" aboard any single Logic Controller. Up to 8176 text characters are available to can describe possible characteristics or changes of the program of the machine or to store a long sequence of informations on it like if it holds a “ Logbook” readable only by whom knows the Password.

Compare (F4)

Starts the compare between the file .OBJ present on the Personal Computer and the content of the FLASH-EPROM memory of the Logic Controller.
During the programming function, by means the command DownLoad, is made automatically a control of the occurred writing of the FLASH-EPROM memory, for this reason the command of Compare is not tight necessary.
The command Compare can be useful to verify if the program stored on the Logic Controller is in accordance with a particular program you have the file .OBJ.

Backup H/X bytes (F5)

Allows to read the current value of the retentive memories of H type from the Logic Controller and to store them in the file with extension .HLD.
For the Logic Controllers that have retentive memory of X type, it is possible to execute an equivalent operation with storing on the file .XMF. Only in this case the option /X is present in the entry of the menu and the first displayed dialogue window will request to specify if you want to read the memories of H type or X type.
The backup command needs the insertion of the number of the starting and ending byte of the memory area you want to read. In the case of H type memories the field of validity of the extremes is 0÷1023 whereas for X type memories is 0÷24567; furthermore the ending number of byte musts not be less than initial.

The purpose of the command of backup is principally that to recover the values of work parameters of the machine and then load them to other Logic Controllers; actually, when a Logic Controller is new, the retentive RAM memory cannot obviously hold the work parameters yet. Storing in a file the information contained in a Logic Controller of reference already configured and then loading this file with the Restore command in a second Logic Controller, it is possible to preset this one to work in according with the correct parameters.
The format of the files .HLD and .XMF is a text type and it can be visualized, printed and also processed by specific programs outside the ICL51 environment.

Restore H/X bytes (F6)

Allows to write data before stored in the file .HLD in the retentive memory of H type of the Logic.
For the Logic Controllers that have retentive memory of X type, it is possible to execute an equivalent operation taking data from the file .XMF. Only in this case the option /X is present in the entry of the menu and the first displayed dialogue window will request to specify if you want to write the memories of H type or X type.
The restore command needs the insertion of the number of the starting and ending byte of the memory area you want to write. In the case of H type memories the field of validity of the extremes is 0÷1023 whereas for X type memories is 0÷24567; furthermore the final number of byte musts not be less than initial.
In the case is specified an area not corresponding to the content of the file .HLD (or of the file .XMF for the X memories), the bytes to fill not found in the backup file will be zeroed.
This thing suggests an artifice to fill with extreme easiness all or a section of the retentive RAM of the Logic Controller with the value zero; it is enough indeed to create a file .HLD (or .XMF for the X memories) containing the reading of only one byte corresponding with the initial element and with decimal value zero. Later you can make the restore specifying this initial element and a desired final one: the first byte will be really read from the file, while all the following, not found in the same file, will be automatically zeroed.

View Holding file (F7)

Recall the visualizer of selected file passing as parameter the file .HLD.
With this command it is possible therefore to view, analyse and print the backup file.

Update Watch (F8)

The command of Update Watch execute a forcing of the time and of the date of the Computer in the watch/calendar of the Logic Controller (when this optional is installed).
This operation can be performed both with the Logic Controller in RUN and in STOP; the only difference is that in case of Logic Controller in RUN, that command will not perform its functions if in the user program are running the instructions of writing the watch/calendar. This happens because you could have a overlapping between the operations made by the running program on the Logic Controller and the transferring operations of the Computer.

Before performing the update of the watch/calendar the time and the date of the Computer are viewed; if they were not correct, before transferring to the Logic, update them by means of the TIME and DATE Dos commands (to do this it is enough to go temporarily out of the develop environment through the Dos Shell command).
The confirmation of the update of the watch/calendar begins the transmission operations of the values in the bytes of W type of the Logic Controller.
This operation can be executed with the running machine to preset correctly the watch/calendar. Following corrections of the time and the date can be performed again on the Logic with that command or, if you don’t have a Computer, by means a terminal panel programmed for this purpose.

Stop Logic (F9)

This command allows you to force the Logic Controller in the STOP state on power on. In some Logic Controllers this function is got by means of closing of a specified dip-switch or jumper and then the present entry in the menu is not available. In the Logic Controllers of last generation this operation is got without any hardware configuration on the board.

The steps to follow to force the Logic Controller in the STOP state on power on are the following.

Turn off the power to the Logic Controller and then set the command with F9. Turn on the power and then wait the Logic Controller ends the initialization operations (normally 2 seconds are enough). At this point shut down the command typing again the function key F9.

The Logic Controller is now in STOP and it is possible to program it (DownLoad); remember the STOP function on power on has principally the purpose to exit from stalemate situations in which the stored program, for any reason, is not correct but the Logic Controller on power on equally goes in RUN, producing a cyclic occurrence of the watchdog circuit.

The monitor of the program

The monitor program allows to view and modify the content of RAM memory of the Logic Controller during the execution in real time of the user program.

Starting such a function, on the screen of the Personal Computer the list of the currently selected program is recalled. This list will always do as a background for all the monitor operations; if you start the monitor window the screen area used for the list will be limited to the upper half of the screen, whereas in the lower half will be viewed the monitor window.

To scan the whole list of the program on the screen use the following keys:

• arrow up	move the list a line up
• arrow down	move the list a line down
• Page up	move the list a page up
• Page down	move the list a page down
• Home	move to the beginning of the list
• End	move to the end of the list

To start monitor window type the function F1=Watch. If the Automatic configuration option is been selected, the monitor window is automatically recalled and the Logic is immediately forced in the RUN state (Start of the program).

The monitor window allows to view in the same time 8 variables in so many rows of the screen. Every row gives you several informations of the variable:

- The T field shows the variable type with and without sign (U=unsigned, S=signed) necessary to the correct understanding by the monitor of the used convention for the variable (integer without sign or integer with sign through a 2-complement notation).

- The N field specify the bytes number (1, 2 or 4) of the variable; also this choice must be transmitted to the monitor program to a full view of the value in 8, 16, 32 bits.

- The Byte field specifies the variable name through the mnemonic of the operand in according with the things established by the ICL51 language.

- The Description field allows to associate to the variable a descriptive text string (maximum 16 characters). If in the configuration M_LOGIC.CFG file, any description (null string) does not be found, the system automatically will verify in the source list of the user program the presence of a alphanumeric Label assigned to the operand and it will use that Label as description of the monitor.

- The Decimal field views the decimal value currently monitored of the pointed variable. That value means the value with or without sign of the variable ad 1, 2 or 4 bytes.

- The visualization field of the single bits of the variable allows to analyse the state of all the relays or flags associated to that memory area.

The system provides the arrangement in the configuration file M_LOGIC.CFG of a list of 512 variables that can be recalled at once in the monitor window. This window of 8 variables wideness can be shifted up and down on the continuous list of all the variables stored in the configuration file.

One particularly of the 8 variables viewed in the window is highligthed on the screen: this is the variable currently pointed and on it will be performed all the operations of modification offered by the monitor. On the pointed variable is also highligthed one of its bits to allow the modification operations on the single bit.

To move the visualization of the monitor window in the list of all the 512 variables use the following keys :

• Ctrl + Arrow up	point to the preceding variable
• Ctrl + Arrow down	point to the following variable
• Ctrl + Page up	move the window on the 8 preceding variables
• Ctrl + Page down	move the window on the 8 following variables
• Ctrl + Home	move the window to the first 8 variables
• Ctrl + End	move the window to the last 8 variables
• Ctrl + Left arrow	move the bit pointer a location on the left
• Ctrl + Right arrow	move the bit pointer a location on the right

The monitor menu allows to recall several functions of which some work on the specific pointed variable or on its pointed bit. For this reason the actions performed by the above keys are very important; before working on a fixed variable or one bit of its you must verify the correct position of the pointer.

The available functions in the monitor program are the following:

Start (F1)

Force the Logic Controller in the RUN state starting the scan of the user program. The Logic Controller executes the program till the following Stop command.

Stop (F2)

Force the Logic Controller in the STOP state stopping the scan of the user programs.

In this state the monitor window will view the starting values present in the data RAM before going in RUN state (like on power on). The Logic remains in stop program till the following Start command.

Change (F3)

Allows to select a different variable in the position currently pointed.

Starting this command a dialogue window will require the insertion of the T, N, Byte fields and if necessary of the Description field.

With the ENTER key it is possible to confirm the values of the single fields while with the Esc key you cancel the command.

The new selected variable will be updated in the configuration file of the monitor

M_LOGIC.CFG at the pointed position.

Force (F4)

Allows to force the value of the currently pointed variable. The dialogue window will require the insertion of a decimal, binary (ending with B character) or hexadecimal (ending with H character) value.
The value to force must be in the field of validity of the pointed variable; in case of error however the software will provide to indicate it.
You can think that the forcing operation of a value in the data RAM is possible only if are not present in the program instructions doing the same things on the same variables: in this case the action executed by the machine program is prevailing. The same thing happens for all the resources already forced by the operating system of the Logic Controller as the input bytes.

Set (F5)

Force at the logic state “1” the bit currently pointed of the selected variable.
The forcing of a bit of the data memory is possible only if instructions forcing the same bit, do not exist in the program of the Logic Controller or if the bit is not defined in its value by the operating system (ex: input bit).

Res (F6)

Force at the logic state “0” the bit currently pointed of the selected variable.
The forcing of a bit of the data memory is possible only if instructions forcing the same bit, do not exist in the program of the Logic Controller or if the bit is not defined in its value by the operating system (ex: input bit).

Help (F7)

View a summary list of the functions performed by positioning keys on the list and on the monitor window.

Search (F8)

Allows the research forward of a text string inside of the viewed program list.

Esc

The Escape key allows to exit from the monitor menu and return to the principal menu of the ICL51 environment.

Communication protocol

General informations

The RS232 interface of the MASTER board allows all the communication operations with a Personal Computer or with any device able to handle any similar interface.

In both cases it is necessary to follow some rules to control the flow of exchanged informations between the Logic Controller and external devices: these rules establish the communication protocol of the MASTER board.

Later we will show these rules referring for simplicity to a connection between MASTER Logic Controller and Personal Computer, reporting illustrative parts of program written in BASIC language.

The purpose of this section is that to enable the Programmer to make the MASTER boards speaking with a Computer, accessing in reading and in writing both the functioning program of the Logic Controller and all the stored work data in the memory RAM.

With the informations we will report, it is possible to write programs with high level language, operating on the Computer, monitoring the running of the program on the MASTER, allowing to get data in real time during the running of the machine; also it is possible to operate the machine forcing informations from the Computer to the Logic Controller.

These handling operations by means Computer can be temporary, i.e. connecting provisionally to the MASTER a Computer only when it is necessary, or permanent, leaving the Computer always connected to the Logic Controller. In this last case we refer to the use of a Industrial Computer as an integral part of the machine and which functions can be both of monitoring and control of the automatic process. Limit case of this application is the control of the process completely made by the Computer, using the logic controller exclusively as INPUT/OUTPUT interface towards the machine.

Opening of the communication channel

Before any operation of communication with the Logic Controller it is necessary to open, in the program on the Computer, its serial communication channel RS232.

Supposing to connect the Logic Controller to the serial port COM1 of the Computer, the instruction BASIC allowing to do this is the following:

```
OPEN “COM1:9600,N,8,1” FOR RANDOM AS #1
```

This instruction opens the communication file number 1 on the communication serial port COM1. The choice of the file number is arbitrary and it has no reference with the serial port number; all the access operations to the COM1 will have later be performed obviously on that file number.

From the open instruction we guess the serial communication parameters are:

Baud rate:	9600
Parity:	NONE
Bits number:	8
Stop bits:	1

These parameters are defined inside the operating system of the Logic Controller and they must be closely respected.

Ended the communication you must execute on the Computer the close instruction of the communication channel:

```
CLOSE #1
```

The use of the open and close instructions of the RS232 communication channel is completely leaved at discretion of the Programmer; nothing forbids to open and close continuously the channel in according with the needs to communicate with the Logic Controller.

Handling of the communication errors

We suggest to insert in the program running on the Computer a routine of communication errors handling; the lack of this routine does not compromise the communication with the Logic Controller neither its functioning, but it can produce a program stop on the Computer or the exit to the operating system.

In BASIC the errors handling can be made putting at the top of the program the following instruction:

```
ON ERROR GOTO Error Handling
```

At the bottom of the principal program insert instead the handling routine of the possible error; following we show an example of this one:

```
Error Handling:

        SELECT CASE ERR
        CASE 24
                Error$ = “Timeout error”
        CASE 52, 57
                Error$ = “Communication error”
        CASE 68
                Error$ = “Not available port”
        CASE 69
                Error$ = “Overflow in the buffer”
        -------------------------------------------------
        -------------------------------------------------
        END SELECT

        BEEP
        PRINT Error$
        SLEEP
        RESUME NEXT
```

These lines of program are showed exclusively for example; it will be up the Programmer to deepen the argument referring to the proper sections of the manual of the used programming language.

The errors handling type depends strongly on the whole developed program structure; actually the errors detections depends on the type and the way of using of the employed instructions in the communication. The return after one error (ex: RESUME NEXT) depends on the program structure and on its specifications.

In some cases it is suitable to handle the TIMEOUT communication error also by means the Watch dog TIMER of the BASIC:

```
ON TIMER (3) GOSUB TimeOut

---------------------------------------------
---------------------------------------------

Error = FALSE
TIMER ON

DO UNTIL LOC (1) = 4
        IF Error THEN
                EXIT LOOP
        END IF
LOOP

TIMER OFF

IF NOT Error THEN
        ReceivedString$ = INPUT$(4, #1)
END IF

----------------------------------------------
----------------------------------------------



TimeOut:

        Error = TRUE
        BEEP
        PRINT "Timeout error"
        SLEEP

RETURN
```

Generally in the communication program the errors can occur during a opening of the communication channel; or you can find a waiting situations (by means “polling”) of receiving data from the Logic Controller never ending because of an accidental event on the communication line. In this case the use of the TIMER, instruction as in the preceding example, can end the receiving message continuing the program execution.

In alternative to the “polling” handling of the receiving messages from the Logic Controller, you can use in BASIC the events handler ON COM(1) to execute the instructions of answer to the serial when the data come to the Computer.

Communication commands

All the communication operations between the Computer and the MASTER Logic Controller happen by means the sending from the Computer of a COMMAND BYTE to the Logic Controller. Then the Computer has the control of the communication state; the Logic Controller executes only the sent commands by the Computer.

Obviously the Logic Controller must continue its job to process the loaded machine program in the memory; to understand the RS232 communication mechanism we say every time the Computer sends to the Logic Controller a packet of bytes made in the top by the command byte and then by the parameters of the command, the Logic Controller executes a INTERRUPT routine saving in a internal buffer the bytes of the packet.

When the Logic Controller ends the scan of the machine program, it executes the Computer command; then it starts again the program scanning or because of certain commands it stands in a STOP condition.

In this way all the communication operations with the Computer are synchronized with the machine program scanning of the Logic Controller and more than one communication operation per scan cycle cannot be executed. Additionally you must consider the processing speed of the program in the Computer; if the control speed of the Computer is less than that of a scan cycle execution of the Logic Controller, it is obvious that a communication operation will happen every two or more scan cycle. However this does not slow in a remarkable way the supervision of the Logic Controller by the Computer, and neither the active control of the machine functions by the program on the Computer.

For all those commands not providing for an answer from the Logic Controller, you must pay attention not to send consecutively two different packets, because the Logic Controllers does not have the time to process them. So for some types of command we suggest to introduce a software delay in the Computer allowing the Logic Controller to process the just sent packet; the need of this delay depends on the type of the required operation as you will see later.

A BASIC routine which delays a multiple number of 55 ms is the following:

```
SUB WaitNx55ms (N)

        FOR I= 0 TO N
                SOUND 32767, 1
        NEXT I

END SUB
```

This routine utilizes the SOUND instruction with frequency over the audible and during 1/18.2=0.05494 seconds and it determine a wait of the program N times 55ms.

Before proceeding to a detailed analysis of the communication commands, we show in the Table 8 the map of the data RAM memory; the use of this table is essential in the communication operations because all the reading and writing data commands refer to the absolute addresses of the variables.

The RAM memory is divided in some blocks each assigned to a particular type of variable or resource of the program. For example the first block of 4096 bytes is used as rest memory for all the I/O both of the MASTER and of the 31 SLAVES; the following two blocks are used for the 1024 bytes of the not retentive memory and for the 1024 bytes of the retentive memory. There are also the used block to implement the software counters, the block for the pulse generators on edge and at the end the filled block by a various ensemble of small dimension resources.

These tables allow to individualize, for each program variable, the corresponding hexadecimal address; this address, made by four hexadecimal digits, will have to be divided in two parts (LOW and HIGH) two digits each. Every part forms the value of one byte to send in a proper way to the Logic Controller, to specify which variable you want read or write.

RAM (8000H - FFF7H)

	0.0	.7	.6	.5	.4	.3	.2	.1	.0	8000H
	0.1	.7	.6	.5	.4	.3	.2	.1	.0	8001H
MASTER (BOARD 0)										
	0.126	.7	.6	.5	.4	.3	.2	.1	.0	807EH
	0.127	.7	.6	.5	.4	.3	.2	.1	.0	807FH
	1.0	.7	.6	.5	.4	.3	.2	.1	.0	8080H
	1.1	.7	.6	.5	.4	.3	.2	.1	.0	8081H
SLAVE 1 (BOARD 1)										
	1.126	.7	.6	.5	.4	.3	.2	.1	.0	80FEH
	1.127	.7	.6	.5	.4	.3	.2	.1	.0	80FFH
	31.0	.7	.6	.5	.4	.3	.2	.1	.0	8F80H
	31.1	.7	.6	.5	.4	.3	.2	.1	.0	8F81H
SLAVE 31 (BOARD 31)										
	31.126	.7	.6	.5	.4	.3	.2	.1	.0	8FFEH
	31.127	.7	.6	.5	.4	.3	.2	.1	.0	8FFFH
	M.0	.7	.6	.5	.4	.3	.2	.1	.0	9000H
	M.1	.7	.6	.5	.4	.3	.2	.1	.0	9001H
NOT RETENTIVE MEMORY										
	M.1022	.7	.6	.5	.4	.3	.2	.1	.0	93FEH
	M.1023	.7	.6	.5	.4	.3	.2	.1	.0	93FFH
	H.0	.7	.6	.5	.4	.3	.2	.1	.0	9400H
	H.1	.7	.6	.5	.4	.3	.2	.1	.0	9401H
RETENTIVE MEMORY										
	H.1022	.7	.6	.5	.4	.3	.2	.1	.0	97FEH
	H.1023	.7	.6	.5	.4	.3	.2	.1	.0	97FFH
	C.0.CB					.CKDW	.CKUP	.OUT	.IN	9800H
	C.0.CL	BYTE LOW CURRENT VALUE								9801H
	C.0.CH	BYTE HIGH CURRENT VALUE								9802H
	C.0.FL	BYTE LOW END VALUE								9803H
	C.0.FH	BYTE HIGH END VALUE								9804H
16 BITS COUNTER										
	C.127.CB					.CKDW	.CKUP	.OUT	.IN	9A7BH
	C.127.CL	BYTE LOW CURRENT VALUE								9A7CH
	C.127.CH	BYTE HIGH CURRENT VALUE								9A7DH
	C.127.FL	BYTE LOW END VALUE								9A7EH
	C.127.FH	BYTE HIGH END VALUE								9A7FH

Table 8.a. Map of the data RAM

RAM (8000H - FFF7H)

PULSE GENERATORS	P.0						.OUTD	.OUTU	.IN	9C00H
	P.1						.OUTD	.OUTU	.IN	9C01H
	P.126						.OUTD	.OUTU	.IN	9C7EH
	P.127						.OUTD	.OUTU	.IN	9C7FH
REFERRING TIME	T			.2000	.1000	.500	.200	.100	.50	9F00H
SCAN COUNTER	SXS	BYTE LOW SCAN x SECOND								9F08H
		BYTE HIGH SCAN x SECOND								
SPECIAL FLAG	F	.E	.C	.>	.=	.<	.P	.1	.0	9F10H
WATCH / CALENDAR	W.CB								ADJ	9FF0H
	W.SEC	SECONS VALUE BINARY BYTE								9FF1H
	W.MIN	MINUTES VALUE BINARY BYTE								9FF2H
	W.HOUR	HOURS VALUE BINARY BYTE								9FF3H
	W.DAY	DAY OF WEEK VALUE BINARY BYTE								9FF4H
	W.DATE	DAY OF MONTH VALUE BINARY BYTE								9FF5H
	W.MTH	MONTH VALUE BINARY BYTE								9FF6H
	W.YEAR	YEAR VALUE BINARY BYTE								9FF7H
										9FFFH
EXTENDED RETENTIVE MEMORY	X.0	.7	.6	.5	.4	.3	.2	.1	.0	A000H
	X.1	.7	.6	.5	.4	.3	.2	.1	.0	A001H
	X.24566	.7	.6	.5	.4	.3	.2	.1	.0	FFF6H
	X.24567	.7	.6	.5	.4	.3	.2	.1	.0	FFF7H

CAUTION: EXTENDED RETENTIVE MEMORY IS NOT AVAILABLE ON EVERY PLC

Table 8.b. Map of the dat RAM

STOP

Stops the user program execution

Command code

(1)

Answer from the Logic Controller

None.

Description

The STOP command allows the PC to stop the user program looping putting the Logic Controller in a idle state. The RAM memory is reset excluding the retentive bytes; continuously the call to all the SLAVES is executed with the related updating of the reserved bytes. The Logic controller stands in this state till the following RUN command.

Example

In the BASIC language the following instruction forces the MASTER Logic Controller in the STOP state:

```
PRINT #1, CHR$(1);
```

See also

RUN

RUN

Starts the user program execution

Command code

(10)

Answer from the Logic Controller

None.

Description

The RUN command allows the PC to restart the user program. On power on, if there is present a user program in the FLASH-EPROM memory, the Logic Controller automatically goes in the RUN state; but it is also possible to force, on power on, the Logic Controller in the STOP state by means the insertion of the proper switch or by means the F9 command of the transferring menu (in according with the model of the Logic Controller).

Attention: normally after a RUN command other command packets follows, because what interests is the communication when the program is running. Before sending other command packets, we suggest to wait some tens of milliseconds to allow the Logic Controller to start the activity.

Example

In BASIC language the following instruction forces the MASTER Logic Controller in the RUN state:

```
PRINT #1, CHR$(10);
```

See also

STOP

STATUS

Reads the command byte currently present in the Logic Controller

Command code

(250)

Answer from the Logic Controller

(Byte)

1 byte corresponding to the current state of the command byte in the Logic Controller.

Description

The STATUS command reads the present value in the first byte of the receiving buffer of the command packet. In this way it is possible to know the top byte, corresponding to the command code, of the last sent packet.
The purpose of this command is to verify if the Logic Controller is in the STOP or RUN state when this condition is not known.

Example

In BASIC language the following part of program request to the MASTER Logic Controller the value of the command byte preceding sent:

```
PRINT #1, CHR$(250);

DO UNTIL LOC(1) = 1: LOOP

State$ = INPUT$(1, #1)
```

MONITOR1

Reads the current value of a 1 byte variable

Command code

(200) + (Address_Low) + (Address_High)

Answer from the Logic Controller

(Byte)

1 byte corresponding to the current value of the 1 byte variable.

Description

The MONITOR1 command allows to read the current value of the 1 byte variable with absolute address of RAM indicated in the word made by two bytes Address_High and Address_Low. The answer for this 3 bytes command packet, the Logic Controller sends 1 byte corresponding to the current value of the required 1 byte variable; following the receiving of the answer it is possible to send another command packet.

Example

In BASIC language the following part of program reads the value of the 1 byte variable H.1022:

```
PRINT #1, CHR$(200) + CHR$(254) + CHR$(151);

DO UNTIL LOC(1) = 1: LOOP

Value = ASC(INPUT$(1, #1))
```

See also

MONITOR2, MONITOR4

MONITOR2

Reads the current value of a 2 bytes variable

Command code

(201) + (Address_Low) + (Address_High)

Answer from the Logic Controller

(Byte_0) + (Byte_1)

2 byte corresponding to the current value of the 2 byte variable.

Description

The MONITOR2 command allows to read the current value of the 2 bytes variable with absolute address of RAM indicated in the word made by two bytes Address_High and Address_Low.

The answer for this 3 bytes command packet, the Logic Controller sends 2 bytes corresponding to the current value of the required 2 bytes variable; following the receiving of the answer it is possible to send another command packet.

Example

In BASIC language the following part of program reads the value of the 2 bytes variable H.0:

```
PRINT #1, CHR$(201) + CHR$(0) + CHR$(148);

DO UNTIL LOC(1) = 2: LOOP

LowValue = ASC(INPUT$(1, #1))
HighValue = ASC(INPUT$(1, #1))

Value = 256 * HighValue + LowValue
```

See also

MONITOR1, MONITOR4

MONITOR4

Reads the current value of a 4 bytes variable

Command code

(202) + (Address_Low) + (Address_High)

Answer from the Logic Controller

(Byte_0) + (Byte_1) + (Byte_2) + (Byte_3)

4 byte corresponding to the current value of the 4 bytes variable.

Description

The MONITOR4 command allows to read the current value of the 4 bytes variable with absolute address of RAM indicated in the word made by two bytes Address_High and Address_Low.

The answer for this 3 bytes command packet, the Logic Controller sends 4 bytes corresponding to the current value of the required 4 bytes variable; following the receiving of the answer it is possible to send another command packet.

Example

In BASIC language the following part of program reads the value of the 4 bytes variable H.0:

```
PRINT #1, CHR$(202) + CHR$(0) + CHR$(148);

DO UNTIL LOC(1) = 4: LOOP

Value0 = ASC(INPUT$(1, #1))
Value1 = ASC(INPUT$(1, #1))
Value2 = ASC(INPUT$(1, #1))
Value3 = ASC(INPUT$(1, #1))

Value = 16777216 * Value3 + 65536 * Value2 + 256 * Value1 + Value0
```

See also

MONITOR1, MONITOR2

FORCE1

Writes a value on a 1 byte variable

Command code

(210) + (Address_Low) + (Address_High) + (Byte)

Answer from the Logic Controller

None.

Description

The FORCE1 command allow to write a fixed value on the 1 byte variable with absolute address of RAM indicated by the word formed by two bytes Address_High and Address_Low. The Logic Controller does not answer to this 4 bytes command packet; we suggest to insert a fixed delay after sending the command packet (no less than a program cycle) at the purpose to allow the Logic Controller to execute the command.

Example

In BASIC language the following part of program writes the value 78 on the 1 byte variable H.1022:

```
PRINT #1, CHR$(210) + CHR$(254) + CHR$(151) + CHR$(78);
```

See also

FORCE2, FORCE4

FORCE2

Writes a value on a 2 bytes variable

Command code

(211) + (Address_Low) + (Address_High) + (Byte_0) + (Byte_1)

Answer from the Logic Controller

None.

Description

The FORCE2 command allow to write a fixed value on the 2 bytes variable with absolute address of RAM indicated by the word formed by two bytes Address_High and Address_Low. The Logic Controller does not answer to this 5 bytes command packet; we suggest to insert a fixed delay after sending the command packet (no less than a program cycle) at the purpose to allow the Logic Controller to execute the command.

Example

In BASIC language the following part of program writes the value 32578=256*127+66 on the 2 bytes variable H.0:

```
PRINT #1, CHR$(211) + CHR$(0) + CHR$(148) + CHR$(66) + CHR$(127);
```

See also

FORCE1, FORCE4

FORCE4

Writes a value on a 4 bytes variable

Command code

(212) + (Address_Low) + (Address_High) + (Byte_0) + (Byte_1) + (Byte_2) + (Byte_3)

Answer from the Logic Controller

None.

Description

The FORCE4 command allow to write a fixed value on the 4 bytes variable with absolute address of RAM indicated by the word formed by two bytes Address_High and Address_Low. The logic controller does not answer to this 7 bytes command packet; we suggest to insert a fixed delay after sending the command packet (no less than a program cycle) at the purpose to allow the Logic Controller to execute the command.

Example

In BASIC language the following part of program writes the value 2355455890 (decomposed in its 4 bytes corresponds, starting from the most significant byte, 140, 101, 103, 146) on the 4 bytes variable H.0:

```
PRINT #1, CHR$(212) + CHR$(0) + CHR$(148) + CHR$(146) + CHR$(103) + CHR$(101) + CHR$(140);
```

See also

FORCE1, FORCE2

RESBIT

Forces the logic “0” value in one or more bits of a byte

Command code

(220) + (Mask) + (Address_Low) + (Address_High)

Answer from the Logic Controller

None.

Description

The RESBIT command allows to force to “0” one or more bits of the byte with absolute address of RAM indicated by the word made by two bytes Address_High and Address_Low. To specify which bits of the byte have to be reset you must provide in the command packet the value of the mask byte (Mask); this byte must have the logic value “1” in all and only the bits to reset.

The Logic Controller does not answer to this 4 bytes command packet; we suggest to insert a fixed delay after sending the command packet (no less than a program cycle) at the purpose to allow the Logic Controller to execute the command.

Example

In BASIC language the following part of program writes the logic value “0” on the bits 0 and 5 of the byte H.1022 i.e. it resets the bits H.1022.0 and H.1022.5:

```
PRINT #1, CHR$(220) + CHR$(33) + CHR$(254) + CHR$(151);
```

See also

SETBIT

SETBIT

Forces the logic “1” value in one or more bits of a byte

Command code

(221) + (Mask) + (Address_Low) + (Address_High)

Answer from the Logic Controller

None.

Description

The SETBIT command allows to force to “1” one or more bits of the byte with absolute address of RAM indicated by the word made by two bytes Address_High and Address_Low. To specify which bits of the byte have to be reset you must provide in the command packet the value of the mask byte (Mask); this byte must have the logic value “1” in all and only the bits to set.

The Logic Controller does not answer to this 4 bytes command packet; we suggest to insert a fixed delay after sending the command packet (no less than a program cycle) at the purpose to allow the Logic Controller to execute the command.

Example

In BASIC language the following part of program writes the logic value “1” on the bits 1 and 4 of the byte H.1022 i.e. it sets the bits H.1022.1 and H.1022.4:

```
PRINT #1, CHR$(221) + CHR$(18) + CHR$(254) + CHR$(151);
```

See also

RESBIT

BACKUP

Reads the bytes of a RAM memory block

Command code

(120) + (Address_Low) + (Address_High) + (Number_Low) + (Number_High)

Answer from the Logic Controller

(Byte_0) + (Byte_1) + ... + (Byte_N-2) + (Byte_N-1)

N bytes corresponding to the current value of the RAM block beginning from the address “Address”.

Description

The BACKUP command allows to read the current value of all the bytes of a RAM memory block beginning from the absolute address indicated by the word formed by two bytes Address_High and Address_Low, for a number of bytes contained in Number.
The Logic Controller answers to this 5 bytes command packet, sending all the bytes corresponding to the current value of the required memory block; at the end of such a command, the Logic Controller goes automatically in a STOP state.

Example

In BASIC language the following part of program reads the memory block from the byte H.0 to the byte H.399 (totally 400=256*1+144 bytes):

```
PRINT #1, CHR$(120) + CHR$(0) + CHR$(148) + CHR$(144) + CHR$(1);

DO UNTIL LOC(1) = 400: LOOP

FOR I = 1 TO 400
        Value(I) = ASC(INPUT$(1, #1))
NEXT I
```

See also

RESTORE

RESTORE

Writes a sequence of bytes in a RAM memory block

Command code

(130) + (Address_Low) + (Address_High) + (Number_Low) + (Number_High)

Answer from the Logic Controller

(130)

Assent for the availability by the Logic Controller to receive the sequence of bytes.

Sending of the values

(Byte_0) + (Byte_1) + ... + (Byte_N-2) + (Byte_N-1)

N bytes to write in the RAM memory block beginning from the address “Address”.

Description

The RESTORE command allows to write some values in all the bytes of a RAM memory block beginning from the absolute address indicated by the word formed by two bytes Address_High and Address_Low, for a number of bytes contained in Number.
The Logic Controller answers to this 5 bytes command packet, sending a bytes of value 130 (echo of the command) as assent of its availability to receive data. At this point the PC can send consecutively the sequence of values; at the end of such a command, the Logic Controller goes automatically in a STOP state.

Example

In BASIC language the following part of program writes in the memory block from the byte H.0 to the byte H.399 (totally 400=256*1+144 bytes) a sequence of values:

```
PRINT #1, CHR$(130) + CHR$(0) + CHR$(148) + CHR$(144) + CHR$(1);

DO UNTIL INPUT$(LOC(1), #1) = CHR$(130): LOOP

FOR I = 1 TO 400
        PRINT #1, CHR$(Value(I));
NEXT I
```

See also

BACKUP

UPLOAD

Reads the bytes of a FLASH-EPROM program memory block

Command code

(110) + (Start_Page) + (Pages_Number)

Answer from the Logic Controller

(Byte_0) + (Byte_1) + ... + (Byte_N-2) + (Byte_N-1)   N = 256 * Pages_Number

N bytes corresponding to the FLASH-EPROM block beginning from the page Start_Page.

Description

The UPLOAD command allows to read the values of the bytes of a FLASH-EPROM program memory block beginning from the page Start_Page, for a number of pages contained in Pages_Number .

For page we mean a continuous block of 256 bytes and the page address is a number in the field 0÷255; so with the UPLOAD command it is possible to read blocks starting and ending on integer multiples of 256 bytes blocks.

The Logic Controller answers to this 3 bytes command packet, sending all the bytes corresponding to the required program memory block; the number of received bytes is a integer multiple of 256 (value of Pages_Number). At the end of such a command, the Logic Controller goes automatically in the STOP state.

Example

In BASIC language the following part of list reads the program memory block from the page 4 to the page 23 (totally 20 pages = 5120 bytes):

```
PRINT #1, CHR$(110) + CHR$(4) + CHR$(20);

Area$ = STRING$(5120, CHR$(0))

NumberRx = 0

DO WHILE NumberRx < 5120
        DimBuffer = LOC(1)
        MID$(Area$, 1 + NumberRx) = INPUT$(DimBuffer, #1)
        NumberRx = NumberRx + DimBuffer
LOOP
```

DOWNLOAD

Programs the FLASH-EPROM memory

Command code

(100) + (Start_Page) + (Pages_Number)

Answer from the Logic Controller

(100)	if the erasure is correctly done.
(15)	if the erasure is not correctly done.

Sending of the values

(Byte_0) + (Byte_1) + ... + (Byte_N-2) + (Byte_N-1)   N = 256 * Pages_Number

N bytes corresponding to the FLASH-EPROM block beginning from the page Start_Page.

Answer from the Logic Controller

(0)	if the programming is correctly done.
(255)	if the programming is not correctly done.

Description

The DOWNLOAD command allows to program the whole FLASH-EPROM memory with the user program code. The programming of the FLASH-EPROM memory normally happens through the DownLoad function of the develop environment; but this command is been equally included in the protocol command list to explain more how the programming of the FLASH-EPROM occurs.

Before every programming, the memory must be totally erased; this happens electrically on the Logic Controller board without taking the device out of its socket. Following the total erasure of all the memory locations, you must program all the necessary bytes because it is not possible to program a portion time and again.
The programming of each byte of the memory requires less time than that necessary for the communication of a byte through RS232 at 9600 Baud; for this reason the programming of the single bytes happens simultaneously to the serial transferring.

The programming steps are the following. The PC sends a command packet made by three bytes in which are indicated the beginning page and the number of pages to program. At the receiving of the packet by the Logic Controller, this provides to the immediate erasure of the whole FLASH-EPROM memory; ended this operation it sends to the PC a byte to confirm the erasure. This byte is 100 if the erasure is successful, or it is 15 it is not possible to erase the memory for its fault. In this case the RUN led present on the board of the Logic Controller flashes with the error code " 2 flashes/pause" for 10 times; such situation requires a change of the memory device.

If the erasure is done correctly (answer with the byte with value 100), you can proceed to transfer from the PC to the Logic Controller all the bytes of the pages to program. During the programming the RUN led flashes, changing state every programmed page in the memory.
After the programming the Logic Controller sends to the PC a second byte of confirmation which value depends on the result of the operation; particularly the byte is 0 if the programming is done correctly, or it is 255 if a programming error occurred. In this case on the Logic Controller the error is signalled through the error code " 3 flashes/pause" for 10 times.
More exactly, if a programming error occurs, this is immediately signalled by sending of the byte 255, without waiting the end of the programming, while in the case of executed programming, the sending of the byte 0 occurs at the end.

Example

In BASIC language the following part of list executes the programming of the first 64 pages of the FLASH-EPROM memory:

```
PRINT #1, CHR$(100) + CHR$(0) + CHR$(64)

DO
        SELECT CASE INPUT$(LOC(1), #1)
        CASE CHR$(100)
                EXIT DO
        CASE CHR$(15)
                BEEP
                PRINT "Erasure error"
                SLEEP
                RETURN
        END SELECT
LOOP

FOR I = 0 TO 63
        PRINT #1, MID$(Code$, 1 + I * 256, 256);
        IF INPUT$(LOC(1), #1) = CHR$(255) THEN
                BEEP
                PRINT Programming error"
                SLEEP
                RETURN
        END IF
NEXT I
```

```
DO UNTIL LOC(1) = 1: LOOP

IF INPUT$(1, #1) = CHR$(0) THEN
        PRINT “Executed Programming”
END IF
```