UNI-PRO

DEVELOPMENT ENVIRONMENT FOR PROGRAMMABLE CONTROLLERS

INTRODUCTION MANUAL TO THE C PROGRAM LANGUAGE

CODE 114UPROCLE22

Important Notice

This Instruction Manual should be read carefully before use, and all warnings should be observed; the Manual should then be kept for future reference.

Summary

1. Introduction to the C program language

The main features of the C program language are its availability of complete data structures, concise instructions and high-level control, thus offering the programmer great programming freedom, enabling, in addition, access to hardware devices.

C is a compiled program language. It has been optimised for writing firmware applications, and indeed both the generated executable code and source code are relatively small in size, which is an indispensable feature in order to remain within the limits of flash memories controlling electronic devices.

It is a high-level language and therefore features a particularly simple syntax, in which ordinary English language words are utilised to describe commands corresponding to tens of assembler instructions or hundreds of instructions in machine language (or code).

Among high-level languages, C is considered to be the lowest-level one. This is due to the fact that it has few instructions and manages memory in an efficient way.

In virtue of the possibilities offered by its language, a program written in C enables an exhaustive memory management, while being especially efficient thanks to its contained size.

UNI-PRO uses a third party C compiler designed for embedded systems running on 16-bit microprocessors. There are some differences with the ANSI-C language, specifically due to the particular architecture and constraints of the embedded systems.

This manual is intended for engineers who use the C programming language to develop application programs for the C-PRO family using the UNI-PRO development environment.

2. Basic Concepts

This sections deals with the basic features of the ANSI-C program language used in UNI-PRO.

2.1 Data Types

The data types expected by ANSI-C can be divided into two categories:

- simple data
- structured data

There follows a description of the basic data expected by the ANSI-C program language and the UNI-PRO development environment.

2.1.1 Simple Data Types Expected by C

The ANSI-C program language expects the following data types:

Data Type	Sign	Repr.	Min.	Max.	Notes
char		8 bit	-128	127	(In UNI-PRO, **char** is considered signed, but this generally depends on the compiler being used.)
unsigned char	X	8 bit	0	255	
short int	X	16 bit	–32768	32767	Or just **short (*)**
unsigned short int		16 bit	0	65535	
long int	X	32 bit	-2147483648	2147483647	Or just **long (*)**
unsigned long int		32 bit	0	4294967295	
float	X	32 bit			Single-precision floating-decimal data.
double	X	64 bit			Double-precision floating-decimal data.

NOTE (*): The generic-type integer data with **int** sign is differently represented according to whether the processor is a 16 or 32-bit one, therefore it is recommended to use explicitly the **short** notation per 16-bit data and the **long** notation for 32-bit data.

The following table shows the floating-point data format and expressible value range:

Floating-point Data Format	**Expressible Value Range**
float type	The exponent part is a value between 2^{-126} and 2^{+127}. The fractional portion of the mantissa (the integer portion is normalized to 1) is binary and has 24-digit accuracy.
double type	The exponent part is a value between 2^{-1022} and 2^{+1023}. The fractional part of the mantissa (the integer part is normalized to 1) is binary and has 53-digit accuracy.
long double type*	The exponent part is value between 2^{-1022} and 2^{+1023}. The fractional part of the mantissa (the integer part is normalized to 1) is binary and has 53-digit.

* long double type and double type declarations are recognized as the same type.

2.1.2 Simple Data Types Expected by UNI-PRO

The simple data types adopted by UNI-PRO can be divided into two logical categories: the first one is made up of all data types corresponding to those of the ANSI-C program language; the second category is made up of the new, non-structured data types inserted by the UNI-PRO environment.

The following summary table lists the data types belonging to the first category, indicating for each type:

- Sign : whether or not negative numbers may be represented;
- Repr. : the number of bits actually used by this type of oject;
- Min. : the minimum value it can take on;
- Max. : the maximum value it can take on;
- Corr. ANSI C : the corresponding ANSI C datum.

Data Type	**Sign**	**Repr.**	**Min.**	**Max.**	**Corr. ANSI C**
CJ_BIT		1 bit	0 (FALSE)	1 (TRUE)	(*)
CJ_S_BYTE	X	8 bit	-128	127	Signed char
CJ_BYTE		8 bit	0	255	Unsigned char
CJ_SHORT	X	16 bit	–32768	32767	Signed short
CJ_WORD		16 bit	0	65535	Unsigned short
CJ_LONG	X	32 bit	-{}— 2147483648	2147483647	Signed long
CJ_DWORD		32 bit	0	4294967295	Unsigned long

(*) ANSI-C does not declare a **bool** data type, as is the case with other program language types (e.g. C++), but rather uses **char** with the indications “different from 0” for TRUE and “equal to 0” for FALSE.

On the above data, it is possible to carry out all operations allowed by ANSI C.

The data types not defined by the ANSI-C standard (CJ_VOID, CJ_LED, CJ_BUZZ, CJ_DATE, CJ_TIME, CJ_DATETIME) introduced by the UNI-PRO environment require individual treatment.

CJ_VOID

The CJ_VOID data type is an innovative concept, introduced by the UNI-PRO development environment. It allows a considerable reduction of project development time and a high degree of flexibility.

With this new concept, it is possible to define the data types of generic objects (e.g. a Var or algorithm inputs), by simply linking them to objects whose type has previously been defined.

For example, if a variable is added to a project, this variable is given a default setting CJ_VOID. By linking the variable to a digital input (defined by default as CJ_BIT), the variable type is automatically switched to CJ_BIT.

CJ_LED and CJ_BUZZ

The CJ_LED and CJ_BUZZ data types are very similar. They represent the possibile values that can be taken on respectively by a LED or Buzzer object. These data types can take on values comprised between 0 and 3, with the following corresponding statuses:

0 : Off

1 : On continuously

2 : On with low frequency

3 : On with high frequency

CJ_DATE

The CJ_DATE data type has been implemented for the purpose of carrying out processing involving dates. It represents the number of seconds elapsed since midnight on 1st January 2000 ad is capable of representing dates up to 2068. The use of this data type can prove helpful when, for example, certain operations are to be controlled on the basis of predetermined dates.

When using this data type within algorithms, it may prove easier to use the structure CJ_DATE_STRUCT.

CJ_TIME

The CJ_TIME data type has been implemented for the prupose of carrying out processing involving hours. It can be very useful for managing application time bands of a regulator, or in many other cases. It represents the number of seconds elapsed since the beginning of the day (00:00) and can easily be converted into the CJ_TIME_STRUCT structure via the special conversion function.

CJ_DATETIME

The CJ_DATETIME data type has been implemented for all situations where it is necessary to process together both dates and times. It represents the seconds elapsed since midnight on 1st January 2000 and is capable is representing 2^31 seconds, which is equivalent to about 68 years.

This data type can be used directly within algorithms, or, in order to work more easily, it can be converted into the CJ_DATETIME_STRUCT structure, via library functions (see section on CJ_DATETIME_STRUCT).

Note: CJ_DATE, CJ_TIME and CJ_DATETIME data types are 32-bit data plus sign and are compatible in calculations with the CJ_LONG data type.

2.1.3 Structured Data Types

In addition to the simple data types already described, the C program language enables the definition of more complex types, by combining several basic data types into one structure.

The UNI-PRO development environment features the implementation of structured data types carrying multiple information. They are nothing more than C structures made up of a certain number of elements, called *fields*, which are accessible via the following C-predefined syntax:

structure.fieldname

Example:

```
CJ_ANALOG probe;

CJ_SHORT set;

if (probe.Error != 0)

     if (probe.Value > set)

          ...
```

Structured data types and their meanings are analyzed below.

CJ_ANALOG

The CJ_ANALOG data type represents the status of an analog input. The structure is made up of two fields:

- Short type **Value**, which represents the value read by the probe;
- Byte type **Error**, which represents an error code. If this field is equal to zero there are no probe errors; otherwise it takes on the following values:

    1: the probe is short-circuited; the Value field is forced to module 32766.

    2: the probe is interrupted or missing; the Value field is forced to module 32765.

    3: the reference of the probe is broken; the Value field is forced to module 32762.

CJ_CMD

The CJ_CMD data type is a structure associated with the arrival of a command. It is made up of the following fields:

- Boolean-type **Valid**, which represents the completion of the command notification. If this property takes on the TRUE value, this means that the command has been intercepted and therefore it is possible to proceed with the chosen action; otherwise, no command has been received.

- Byte type **Node**, which indicates the logical node of the controller sending the command.

- Short type **Param**, which represents the command parameter.

CJ_BTN

The CJ_BTN data type is a structure associated with an action on a keyboard key, whether this is pressed, pressed and held or released.

Its is made up of the following fields:

- Boolean-type **Valid**, which represents the completed action (pressing, release or pressing/holding) of the keyboard key. If it takes on the TRUE value, this means that the action indicated in the Btn object has been notified; otherwise the action has not taken place.

- Byte type **Node**, which indicates the logical node where the key action has been verified.

- Short type **Param**, which indicates the number of seconds of persistence of the corresponding key.

CJ_DATE_STRUCT

The CJ_DATE_STRUCT type can be very useful when carrying out operations involving dates. Starting from the CJ_DATE non-structured data type, it is possible to fill the CJ_DATE_STRUCT structure, utilizing the appropriate conversion function.

It is made up of the following fields:

- Byte type **Day**, which indicates the days [1 to 31].

- Byte type **Month**, which indicates the month [1 = January, 2 = February, … 12 = December].

- Byte type **Year**, which indicates the year's last two digits, starting from the year 2000. For example, if this field has a value of 12, this indicates the year 2012.

CJ_TIME_STRUCT

The CJ_TIME_STRUCT data type can prove very useful when carrying out operations with hours, for example to manage time bands. Starting from the CJ_TIME non-structured data type, it is possible to fill the CJ_TIME_STRUCT structure, utilizing the appropriate conversion function.

It is made up of the following fields:

- Byte type **Sec**, which indicates the seconds [0 to 59].
- Byte type **Min**, which indicates the minutes [0 to 59].
- Byte type **Hour**, which indicates the hours [0 to 23].

CJ_DATE_TIME_STRUCT

The CJ_DATETIME_STRUCT data type is used in the conversion from CJ_DATETIME (which represents a date/time, codified in seconds) into an easier format.

This structure is usually filled by the DateTimeToStruct() conversion function, whose C program language syntax is as follows:

CJ_DATETIME_STRUCT DateTimeToStruct(CJ_DATETIME Value);

Its field descriptions are the following:

- Byte type **Sec**, which indicates the seconds [0 to 59].
- Byte type **Min**, which indicates the minutes [0 to 59].
- Byte type **Hour**, which indicates the hours [0 to 23].
- Byte type **Day**, which indicates the days [1 to 31].
- Byte type **WeekDay**, which indicates the day of the week [0 = Sunday, 1 = Monday, … 6 = Saturday].
- Byte type **Month**, which indicates the month [1 = January, 2 = February, … 12 = December].
- Byte type **Year**, which indicates the year's last two digits, starting from the year 2000. For example, if this field has a value of 12, this indicates the year 2012.

To reconvert the structure into the CJ_DATETIME type, use the StructToDateTime function, whose C program language syntax is as follows:

CJ_DATETIME StructToDateTime(CJ_DATETIME_STRUCT rtc);

2.2 Variable Declarations

A variable is declared as follows:

var_type Name_of_variables_separated_by_commas;

Example:

```
short number,sum;

long bignumber,bigsum;
```

A variable can be pre-initialized using the assignation operator = .

Example:

```
short i, j, k=1;

float x=2.6, y;
```

Here we can see two examples of initialization of equivalent variables, not forgetting, however, that the method used in the first example is the more efficient one.

Example 1:

```
float sum=0.0;

long bigsum=0;
```

Example 2:

```
float sum;

long bigsum;

...

sum=0.0;

bigsum=0;

...
```

It is possible to perform multiple assignments, provided that the variables are of the same type.

Example:

```
short a,b,c=3;

a=b=c;
```

Where the instruction a=b=c (with c=3) corresponds to and is more efficient than a=3, b=3 and c=3.

2.3 End-of-Instruction Operator

As can be seen from the above examples, every instruction in C must end with a semi-colon (;). Example:

```
…

short a = 0;

float b = 0;

…
```

2.4 Instruction Blocks

One or more instructions grouped together so as to form a set of instructions, which is treated as a single unit by the compiler, constitutes an instruction block. The block starts with an open curly bracket "**{**" and ends with a closed curly bracket "**}**". The following is an example of an instruction block:

```
if (c < 3)

{

      a = b+c;

      d = c+32;

}
```

In this example, the instructions enclosed within curly brackets, which constitute the block, are controlled by the ***if*** keyword, and are executed only if the condition c < 3 is verified.

2.5 The *Return* Keyword

The **return** keyword is used to define the point and value at exit from a function (or algorithm). The returned type must be consistent with the type defined in the function prototype.

The `return` statement without value is transformed into an equivalent `goto` statement.

The target is the end of the algorithm. The `return` statement with value is transformed into an assignment of the value returned and the `goto` statement to the end of the algorithm.

In UNI-PRO, each algorithm has one output. This means that the value of this output is passed by means of the `return` statement.

In UNI-PRO, if the output is an array, the `return` statement without value must be used.

Example 1:

```
Return (a+b);
```

Example 2:

```
c = a+b;

return;
```

If no `return` statement is used, a `return` statement without value is automatically executed after the last line of the algorithm.

2.6 Function Calls

To call a defined function, one must enter the function name (remembering that C is a case-sensitive program language, thus the name must be written precisely, including upper and lower-case letters), and one must then indicate the various arguments, enclosed within round brackets and separated by commas.

If for example one wants to call a function defined as follows:

```
short max(short a, short b);
```

one must write:

```
short A = 2;

short B = 5;

short maxValue = max( A , B );
```

2.7 Static Variables

The use of `static` variables in the algorithms is allowed, but not recommended, because they can cause undesired results. If a static variable is called in more than one algorithm, all instances of that static variable will use the same memory register.

Example:

If in an algorithm there is this code:

```
static int s=0;

s++;

return s;
```

and this algorithm is used three times in the project, at the beginning the first will calculate 1, the second 2 and the third 3, in the second cycle the first algorithm will calculate 4, and so on.

3. Operators

Like all program languages, C has a number of operators, i.e. symbols representing certain operations on the value of data; the latter is commonly termed operand.

Some C operators are exact equivalents of their counterparts in other program languages, while others are peculiar to C. Before looking at their main features, however, it may be useful to clarify the meaning of two concepts: **precedence** and **associativity**.

When an operator acts on several operands or several operations are defined within an expression, these concepts become significantly important, as they enable a correct interpretation the expression itself, determining which operations must be carried out before others. Let us take as an example the following sum:

```
a = b + c;
```

This expression contains two operators, i.e.: the equal sign (assignment operator) and the plus sign (sum operator). It is easy to understand how this expression is only meaningful if one first calculates the sum of the values contained in **b** and **c**, and only subsequently is the result assigned to **a**. One can say the precedence of the assignment operator is lower than that of the sum operator.

Let us now examine a series of assignments:

```
a = b = c = d;
```

The C compiler executes this by assigning the value of d to c; then the value of c to b; and finally the value of b to a. The result is that the value of d is assigned in cascade to the other variables. In practice, the expression has been evaluated from right to left, that is to say, the assignment operator has a right-to-left associativity.

In other words, **precedence** (or *priority*) is referred to the order in which the compiler evaluates operators, whereas **associativity** concerns the order in which operators with the same priority are evaluated (the order may not necessarily always be from right to left).

Round brackets can always be used to define parts of expressions which are to be evaluated before the operators found outside the brackets. Furthermore, in the presence of nested round brackets, the applicable rule is that the first encountered closed bracket is coupled to the last open bracket, and that the first operations to be evaluated will always be the more internal ones. For example, the following expression:

```
a = 5 * (a + b / (c - 2));
```

is evaluated as follows: first the difference between **c** and **2** is calculated, then **b** is divided by that difference. The result is summed to **a** and the value thus obtained is multiplied by 5. The product is finally assigned to **a**. In the absence of brackets, the compiler would have acted differently, namely:

```
a = 5 * a + b / c - 2;
```

is evaluated by summing the product of **a** and **5** to the quotient of **b** divided by **c**; 2 is then subtracted from the result, and the value thus obtained is assigned to **a**. It is worth presenting the entire set of C operators, by summarising in a table their rules of precedence and associativity; the operators are listed in decreasing order of precedence.

OPERATOR	**DESCRIPTION**	**ASSOCIATIVITY**
`!`	Logical NOT	Right-to-left
`~`	one's complement	
`-`	unary minus (negation)	
`++`	autoincrement	
`--`	autodecrement	
`*`	multiplication	Left-to-right
`/`	Division	
`%`	Remainder of integer division	
`+`	Left-to-right addition	
`-`	Subtraction	
`<<`	Left shift of bit	Left-to-right
`>>`	Right shift of bit	
`<`	Less than	Left-to-right
`<=`	Less than or equal to	
`>`	Greater than	
`>=`	Greater than or equal to	
`==`	Equal to	Left-to-right
`!=`	Different from (Not Equal to)	
`&`	bit-AND	Left-to-right
`^`	bit-XOR	Left-to-right
`|`	bit-OR	Left-to-right
`&&`	Logical AND	Left-to-right
`||`	Logical OR	Left-to-right
`? :`	Conditional expression	Right-to-left
`=, etc.`	Assignment operators (simple and compound)	Right-to-left
`,`	Comma (expression separator)	Left-to-right

Note: In case of sum or multiplication, the UNI-PRO compiler does not check for arithmetic overflow.

Example:

```
CJ_SHORT a = 1000, b = 1000;

CJ_LONG  c = a * b;
```

The multiplication is carried out using int arithmetic, and the result may overflow or be truncated before being promoted and assigned to the long left-hand side.

As is the case for any type of computer programming, it is the responsibility of the programmer to write code that prevents unwanted or unintended results from mathematical operations and memory access operations.

Use an explicit cast on at least one of the operands to force long arithmetic:

```
          long int c = (long int)a * b;
```

or

```
          long int c = (long int)a * (long int)b;
```

(both forms are equivalent).

Note: UNI-PRO compiler does not check for division by zero in case of division and remainder operator.

Example:

```
CJ_SHORT a = 1000, b = 0;

CJ_LONG  d = a / b;
```

The division is carried out giving an indeterminate result value.
It is the responsibility of the programmer to write code that prevents unwanted or unintended results from mathematical operations and memory access operations.

Use an IF...ELSE test to determine that the divisor operand is non-zero:

```
If (b<>0)

CJ_LONG d = a / b;

Else

CJ_Error_divide();
```

3.1 Logical NOT

Logical NOT is represented by the exclamation mark (!). It enables the logical negation of a comparison result, i.e. to turn it "upside down". Therefore, if for example:

```
(a > b)
```

is true, then

```
!(a > b)
```

will prove false.

3.2 One's Complement

The one's complement operator is represented by the tilde (~). The complement to one of a number is obtained by inverting all bits making up the number, for example, with reference to data expressed by a single byte, the one's complement of 0 is 255, while that of 2 is 253. In fact, representing the byte as an 8-bit string, in the first case we pass from 00000000 to 11111111, while in the second from 00000010 we obtain 11111101.

The one's complement operator (or *binary negation*) must not be confused either with the logical negation operator, as just described, or with the algebraic negation one, or unary minus ("–", see below), which are described above. In any case, the difference between the three is evident. The first one turns "upside down" the single bits of a value; the second renders null a non-null value and vice versa; and the third inverts the sign of a value, i.e. renders negative a positive value and vice versa.

3.3 Algebraic Negation

The minus sign " - " can be used as an algebraic negation, i.e. to express negative numbers, or, to be more precise, to invert the sign of a value; in this case, it has a higher priority over all arithmetic operators. Thus

```
a = -b * c;
```

is evaluated by multiplying **c** by the value of **b** with a changed sign. It should be noted that the algebraic negation of a given value does not modify the value itself, but returns it with the opposite sign and identical module; in the above example, the value of **b** is not modified.

3.4 Autoincrement and Autodecrement

The autoincrement and autodecrement operators sum and subtract, respectively, a unit to and from the variable to which they are applied. The expression

```
++a;
```

increments by 1 the value of **a**, while

```
--a;
```

decrements it by 1. It is very important to remember that they can be used as prefixes or suffixes, i.e. they can either precede or follow the variable to which they are being applied. Their significance remains unchanged (adding or subtracting 1), but their priority level differs. In the expression

```
a = ++b;
```

a is assigned the value of **b**, incremented by 1, because variable **b** is first incremented and then its new value is assigned to **a**; unlike in

```
a = b++;
```

where **a** is assigned the value of **b**, and only subsequently is the latter incremented. Similar considerations also apply to the decrement operator.

Again, in

```
if(a > ++b) ....
```

the condition is evaluated after having incremented **b**, whereas in

```
if(a > b++) ....
```

the condition is first evaluated, and *then* **b** is incremented.

The difference between prefixed and suffixed operator disappears when the autoincrement of the variable is a parameter of a function call; with reference to a line such as this:

```
functionName(par1,++a);
```

it is often not possible to know *a priori* whether **a** is incremented before passing its value to the function or whether, on the contrary, the increment is subsequently effected. One could expect the writing **++a** to determine the increment before the call, while **a++** would determine it after; nevertheless, the C program language does not establish a univocal rule. This means that the individual compiler can proceed as deemed best. This, in turn, means that there may be compilers which fix *a priori* a univocal way to proceed, while others decide instead case by case at the compilation stage, for example on the basis of code optimisation options, in relation to speed, dimensions and so forth. It is therefore imperative to study very carefully the compiler's documentation, or, better still, to avoid the risk of any ambiguity, by dedicating to the variable's increment a separate instruction for that of the function call, not least in view of possible subsequent porting of the program to other compilers.

The operators "++" and "--" always modify the value of the variable to which they are applied.

3.5 Arithmetic Operators

The arithmetic operators of the C program language are the symbols of addition " + ", subtraction " – ", division " / " and multiplication " * ". The use of these operators can also appear rather obvious; however, it is useful to emphasize the fact that the normal rules of algebraic priority apply to them, therefore, in the absence of brackets, multiplication and division operations are carried out before those of addition and subtraction. For example, the following expression:

```
a = b + c * 4 - d / 2;
```

is calculated as:

```
a = b + (c * 4) - (d / 2);
```

where the multiplication and the division have priority over the rest.

3.6 Remainder of Integer Division

When carrying out a division between two integer numbers, C returns only the integer part of the result. If there is a remainder, this is lost. For example, the following expression:

```
a = 14 / 3;
```

assigns a = 4;

If one wants to know the remainder of the division, one has to use the " % " operator:

```
a = 14 % 3;
```

assigns a = 2, i.e. the remainder of the operation. In practice, the " % " operator is complementary to the " / " operator, but can only be applied to values belonging to the integer category.

3.7 Bit Shift

Even though it is normally classified among high-level program languages, C often shows its nature of system-oriented program language; indeed, the bit operators at its disposal are one of the features which contribute to make it particular close to the machine. These operators enable intervention on integer data, by considering as simple bit sequences.

Two operators are particularly interesting, as they enable the relocation – or shifting – by a certain number of positions to the right or left, of the bits of a given value: these are the so-called *bit shift operators*. In particular, the left shift is represented by the " << " symbol, the right shift by the " >> " symbol. Example:

```
a = 1;

a <<= 2;

b = a >> 1;
```

The reported code fragment assigns to **a** the value 4 and to **b** the value 2 (without modifying the value of **a**); indeed, number 1 in binary form is 00000001.

By relocating the bits to the left by two positions, one obtains 00000100, which is, namely, 4.

The assignment operator can be made up of bit operators; there follows that the second line of code modifies the value of **a**, assigning to it its own value relocated to the left by two positions, while the third line gives the result of the operation in **b**, without modifying the value contained in **a**.

It must be noted that the shift operation renders meaningless the first or last bits of the value (depending on whether the relocation is to the left or right, respectively); those spaces are filled with bits with an appropriate value. A shift to the left never causes any problems, since the bits which are left free are filled with a zero bit; in the case of a shift to the right, things become complicated.

If the data type on which the shift is performed is without sign, or is positively signed, in this case too null bits are used as fillers. On the contrary, if the data type is negatively signed, then it must be born in mind that its most significant bit – i.e. the one at the extreme left – is used precisely to express the sign. Some processors extend the sign, i.e. they fill in the bits left free by the shift with a bit one, while others insert null bits anyhow. Therefore, depending on the calculator on which it is performed, a left-shift operation as the following one:

```
short sc;

sc = -1; // In bits it is 11111111

sc >>= 4; // remains 11111111 with E.S.; becomes 00001111 without E.S.
```

can result into a final **sc** value still equal to **a** -1, if the processor performs a sign extension (E. S.); or equal to 15, if there is no sign extension. Thus, one needs to proceed with caution, and to consult the machine documentation, before risking any kind of hypothesis.

3.8 Logical Operators

Logical test operators can be divided into two categories: those normally used in comparisons between values; and those used to link the results of two comparisons. The following is a brief series of examples referring to the first group:

```
(a == b) // TRUE if a is EQUAL to b

(a != b) // TRUE if a is different from b

(a < b) // TRUE if a is strictly less than b

(a > b) // TRUE if a is strictly greater than b

(a <= b) // TRUE if a is less than or equal to b

(a >= b) // TRUE if a is greater than or equal to b
```

The writing of the said operators and their meaning appear obvious, perhaps with the exception of the equal operator " == "; in fact, having established that in the codification of programs comparisons of equality are generally about half the assignments, the designers of the C program language, have decided to distinguish the two operators, by doubling the writing of the second to express the first one.

It follows, therefore, that:

```
a = b;
```

assigns to **a** the value of **b**, whereas:

```
(a == b)
```

expresses a condition which is true if the two variables are the same. The different writing of the two operators enables the writing of conditions such as:

```
if(a = b) ....
```

From has just been said, it appears obvious that such writing cannot mean "if **a** is equal to **b**"; what we have here is, in fact, simply a very succinct way of saying:

```
a = b;

if(a) ....
```

which, in turn, is the equivalent of:

```
a = b;

if(a != 0) ....
```

That is to say, "assign **b** to **a**, and if the result (i.e. the new value of **a**) is different from 0…", given that every time a condition is expressed without a second comparison term, the C program language assumes that one wants to verify it non-nullity.

Let us now examine the second category of operators. The logical operators normally used to link the results of two or more comparisons are two: they are the logical product (" && ", or AND) and the logical sum (" || ", or OR).

```
(a < b && c == d) // AND: true if both are TRUE

(a < b || c == d) // OR: true if AT LEAST ONE is TRUE
```

It is possible to write rather complex conditions, but one must bear in mind the rules of priority and associativity. For example, given that all operators of the first category have greater priority over those of the second category, the following:

```
(a < b && c == d)
```

is the equivalent of:

```
((a < b) && (c == d)
```

In expressions where both " && " and " || " appear, one must remember that the first has priority over the second; therefore:

```
(a < b || c == d && d > e)
```

is the equivalent of:

```
((a < b) || ((c == d) && (d < e)))
```

We can deduce from all this – if nothing else – that in many cases the use brackets, even where they are not indispensable, is certainly useful, since it improves considerably the legibility of code, avoiding the risk of making insidious logical mistakes.

3.9 Logical Bit Operators

Logical bit operators enable the relating of two values via a bit-by-bit comparison. Let us examine the logical product operator, or bit-AND. When two bits are put in AND, the result is a null bit, unless both bits have a value of 1. The table illustrates all possible cases with a logical product of two bits, depending on the values which each of them can take on.

The operation which consists in putting two values in AND is often referred to as “masking”, since it has the effect of hiding selectively some of the bits; in particular, it is the second value that is conventionally termed “mask”. If the mask contains a zero, in the result there will always be a zero in that same position, whereas a 1 in the mask leaves the value of the original bit unchanged. Let us suppose, for example, that we want to consider only the 8 least significant bits of a 16-bit value:

```
unsigned short word;

unsigned char byte;

word = 2350;

byte = word & 0xFF;
```

The value 2350, as expressed in 16 bits, results in 0000100100101110, while its hexadecimal 0xFF value is 0000000011111111. The logical product operation can be represented thus:

```
0000100100101110 &

0000000011111111 =

0000000000101110
```

and the result is 46. From this example, we can further deduce that the AND on bit operator is the “ & ” character.

The difference from the logical AND operator appears to be subtler, even though this has the different writing “&&”. The AND on bit acts precisely on the single bits of the two expressions, while the logical AND links the logical values of the same (true or false). For example, the following expression:

```
((a > b) && c)
```

returns a value which is different from 0 if **a** is greater than **b** and if, at the same time, **c** is different from 0; whereas the expression:

```
((a > b) & c)
```

returns a value different from 0 if **a** is greater than **b** and, at the same time, **c** is an odd value. In fact, a true expression returns 1, and all odd values have their least significant bit at 1, therefore the logical product has a bit at 1 (the least significant one, thus different from 0) only if both conditions are true.

The OR on bit operator is used instead to calculate what is commonly known as the logical sum of two values. When two bits are put in OR, the result is always 1, except in the case where both bits are at 0. The behaviour of the logical sum operator is summarised in the table. It should be noted that the mask concept can also be validly applied to OR operations between two values, especially when one wants to assign the value 1 to one or more bits of a variable. In fact, the presence of a 1 in the mask brings to 1 the corresponding result bit, whereas a 0 in the mask leaves the bit of the original value unchanged (this behaviour is exactly the opposite of that of the " & " operator).

The OR operation on the bits of values 2350 and 255 (FFh) can be represented as follows:

```
0000100100101110 |

0000000011111111 =

0000100111111111
```

and it returns 2599. The symbol of the bit-OR operator is " | ", and it should not be confused with the logical OR operator " || "; apart from this, there exist between these two operators differences in meaning which are very similar to those mentioned earlier with reference to the AND on bit and logical AND operators.

There is a third logical bit operator, the bit-XOR operator, also known as "exclusive OR". Its symbol is the circumflex accent " ^ ". An XOR operation between two bits returns a 0 result when the two bits have equal value (i.e. they are both 1 or both 0), whereas it returns a 1 when the two bits have opposite values (the first 1 and the second 0, or vice versa); the table highlights what has been stated. From this we can deduce, therefore, that the presence of a 1 within a mask used in XOR inverts the corresponding bit of the original value.

Brushing up once more the example of the value 2350 masked with a 255, we have:

```
0000100100101110 ^

0000000011111111 =

0000100111010001
```

The result is 2513.

3.10 Conditional Operator

The conditional operator, sometimes also known as ternary operator, since it works on three operands, is represented by the "?:" symbol, and can be compared to a shortened form of the control structure IF….ELSE.

Its general expression is:

```
condition ? expression1 : expression2
```

This means: "If condition is true (i.e. its value is different from 0) return expression1, otherwise return expression2".

For example, the following instruction:

```
(x > 0) ? x : 0 ;
```

returns the value of x if x > 0; in all other cases, it returns a value 0.

The conditional operator enables the writing of more compact and efficient code, compared with what can be achieved with the IF….ELSE structure, albeit at the expense of the legibility of the code.

3.11 Assignment Operator

The assignment operator is represented by the equal sign " = ", and it assigns to the variable on its left the result of the expression on its right. Given the intuitive nature of its meaning and usage, it is not worth dwelling on it any longer. Rather, it is worth examining its usage combined with arithmetic operators.

In all cases where there is an expression of the following type:

```
a = a + b;
```

that is, where the variable to the left of the equal sign also appears in the expression on its right, it is possible to use an abbreviated form which is expressed by "compounding" the assignment operator with the equal and the expression operator. We then talk about composite assignment operators, as opposed to a simple assignment operator, i.e. the equal sign. As usual, an example is clearer than any explanation; thus the above expression becomes:

```
a += b;
```

When formalising the whole, an assignment of the following type:

```
variable = variable operator expression
```

can be written (but not necessarily) as follows:

```
operator variable = expression
```

This is the complete list of all composite assignment operators:

```
+=; -=; *=; /=; %=; >>=; <<=; &=; ^=; |=;
```

They enable the creation of expressions which may be a little cryptic, but are undoubtedly very concise.

3.12 Floating Point Arithmetic Operators

UNI-PRO supports the following operators on variables of the types float, double and long double:

Operator	**Description**
-a	Negation
a + b	Sum
a - b	Subtraction
a * b	Multiplication
a / b	Division
a < b	Relation
a <= b	Relation
a > b	Relation
a >= b	Relation

UNI-PRO compiler supports type conversion to and from integer types.

UNI-PRO compiler does not check for arithmetic overflow in case of sum or multiplication.

Note: The IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754 -1985) defines floating-point computation.

This standard is allowed but not required by the ANSI-C standard.
In the controller, the floating point operations are done by the runtime library functions. Although those functions operate according to ANSI/IEEE Std754 -1985, they do not completely conform to the standard. In particular, no interrupts are generated and no status flag is set.

It is the responsibility of the programmer to write code that prevents unwanted or unintended results from mathematical operations and memory access operations.

4. Instructions

Any program can be codified into a program language by using only three control modes of the processing flow: sequential execution, conditional execution and cycles.

Sequential execution is the simplest of the three, and is often not regarded as a proper control mode at all; in fact, it is logical to expect that, in the absence of any other specification, the next instruction to be performed is the one that follows the current one in the codification.

The other two control structures require a closer examination.

4.1 Conditional Control Instructions

The C program language has at its disposal two different instruments for conditioning the execution of programs. It is worth having a careful look at them.

4.1.1 IF…ELSE Instruction

Conditional execution in its simplest form is specified using the **IF** keyword, which indicates to the compiler that the next instruction must be executed if the condition, always specified within brackets, is true. If the condition is not verified then the instruction is not executed and the processing flow jumps to the next instruction. The instruction to be executed upon verification of the condition can be a single line of code, terminated by a semi-colon, or a block of lines of code, each terminated by a semi-colon and all enclosed within curly brackets. Example:

```
if(a == b)
  function(a);
if(a == c)
{
  function(a);
  a = c;
}
```

In the example code, if the value contained in **a** is equal to the value contained in **b**, the function is recalled; otherwise, the *function call (a)* will not be executed, and processing will proceed with the next instruction, which is still an IF. This time, if *a is equal to c*, the block of instructions enclosed within curly brackets will be executed, otherwise it is skipped, and the program proceeds with the first instruction which follows the closed curly bracket.

As a rule of thumb, a condition is expressed via one the logical operators of C, and is always enclosed within round brackets.

IF is completed by the keyword **ELSE**, which is used when one needs to define two possible alternatives; furthermore, several IF…ELSE structures can be nested, when it is necessary to perform tests on multiple cascaded "levels":

```
if(a == b)
  // a is greater than b
else
{
  // a is less or equal to b
  if(a < b)
    // a is strictly less than b
  else
    // a is strictly equal to b
}
```

In the presence of ELSE, if the condition is true, only what is contained between IF and ELSE is executed; otherwise, only the code that follows ELSE itself is executed. In other words, the execution of the two blocks of code is alternative.

It is extremely important to remember that every ELSE if referenced by the compiler to the last IF encountered; therefore, when IF…ELSE constructions are nested, one must pay attention to the logical construction of alternatives.

Let us try to clarify this concept with an example. Let us suppose that we want to codify in C the following algorithm: *if a is equal to b then check if a is greater than c. If this condition is true too, a is incremented. If instead the first of the two conditions is false, i.e. a is not equal to b, then c is assigned the value of b*. Let us now examine a coding hypothesis:

```
if(a == b)
  if(a > c)
    a++;
else
  c = b;
```

The left-margin indents of the various lines highlight good intentions: it is visually immediate to link ELSE to the first IF. Pity that the compiler should be totally uninterested in indentations! Indeed, it links ELSE to the second IF, namely, to the last encountered IF.

Thus, it is necessary to find a remedy:

```
if(a == b)
  if(a > c)
    a++;
  else;
else
  c = b;
```

The one we have just examined in one possibility. By introducing an "empty" ELSE, one can achieve the aim, as this is linked to the last encountered IF, i.e. the second one. When the compiler encounters the second ELSE, the last and not yet "completed" IF, moving backwards within the code, is the first of the two. It all tallies… Yet there is a more elegant way.

```
if(a == b)
{
  if(a > c)
    a++;
}
else
  c = b;
```

In this case, curly brackets clearly indicate to the compiler which code portion is directly dependent on the first IF, and there is no risk that ELSE may be linked to the second one, given that this is entirely enclosed within the block in curly brackets and therefore certainly "complete". As can be seen, with the exception of some peculiarities, nothing separates the logic of IF in the C program language from that of IFs (or equivalent keywords) available in other program languages.

4.1.2 Switch Instruction

IF manages excellently those situations where, following the evaluation of a condition, there appear to be only two possible alternatives. However, when there are more than two alternatives, one is forced to use several nested IF instructions, and this can complicate considerably the logical structure of the code, impairing its legibility.

When it is possible to express the condition to be evaluated with an expression returning an integer number or a character, the C program language offers the **switch** instruction, which enables the evaluation of any number of alternatives for the result of said expression.

The value returned by the expression must be of integer type, and type conversion is normally performed automatically.

The general form of this instruction is the following:

```
switch (expression)
{
     constant case 1:
          //… instructions
          break;
     constant case 2:
          //… instructions
          break;
     //… other cases
     default :
          //..instructions
          break;
}
```

The keywords **case** and **default** are labels which are arrived at on the basis of evaluation of the expression.

There may be an arbitrary number of *cases*, whereas *default* must be unique. The flow starts at the case whose constant value is equal to the returned value of the switch expression, and proceeds until it encounters an explicit interrupt instruction, determined by the keyword **break**. When no case value is equal to the expression value, the default label is reached.

Let us immediately look at a practical example:

```
int rc; // Let us assume a return value

switch(input)

{

     case 1:

          rc = 33;

          break;

     case 2:

          rc = 66;

          break;

     case 3:

          rc = 100;

          break;

     default:

          rc = 0;

          break;

}
```

The reported code fragment is referred to a very simple algorithm which, depending on the *input* entry, sets an *rc* value. It is best to examine this in greater detail. It should first of all be noted that the expression to be evaluated must be enclosed within round brackets. Furthermore, the *switch* body, i.e. the set of alternatives, is enclosed within curly brackets. Every single alternative is defined by the **case** keyword, followed by an integer constant (variables or non- constant expressions are not admissible) or char, which is in turn followed by a colon " : ". Everything that follows the colon represents the code that will be executed if the evaluated expression takes on precisely the value of the constant found between the **case** and the colon, up to the first encountered **break** instruction; the latter determines the exit from the **switch**, i.e. a jump to the first instruction that follows the closed curly bracket. The **default** keyword, followed by the colon, introduces the section of code to be executed, if the expression does not take on any of the values specified by the different *cases*.

Within the curly brackets, at least one condition must be specified: this means that the **switch** could also be followed by a single **case** or by the **default**, and therefore there can be switches without default or case. The default, however, if present, is unique.

It is perhaps superfluous to specify that there can – if necessary – be more than one **break**, and that they can depend on other conditions evaluated within a case, e.g. via an IF. Furthermore, a case can contain an entire switch, within which a third one can be nested, and so forth. The important thing is not to lose the logical thread of controls. Example:

```
switch(a)
{
  case 0:
    switch(b)
    {
      case 25:
        ....
      break;
      case 30:
      case 31:
      ....
      case 40:
      ....
      break;
      default:
      ....
      }
    ....
  break;
  case 1:
    ....
  break;
  case 2:
    ....
}
```

In the above examples, case-dependent instruction blocks are never enclosed in curly brackets. In fact, the brackets are not necessary (whereas they are – let us repeat this – to open and close a switch); their presence, however, does not hurt. In one word, these brackets are optional.

4.2 Cycles

The C program language also has at its disposal instructions for cycle control; these instructions make it possible to force iterations on code blocks of varying dimensions.

4.2.1 While Cycle

Using the **while** instruction, it is possible to define an iteration cycle until a given condition is verified as true. Let us immediately look at an example:

```
while(a < b)

{

  functionName(a);

  ++a;

}
```

The two lines enclosed within curly brackets will be executed until the *a* variable – increment by increment – becomes equal to *b*; at this point, the execution will proceed with the first instruction that follows the closed curly bracket.

It is worth delving deeper into the algorithm, examining in greater detail what actually happens. The first operation is to evaluate whether *a* is less than *b* (this condition must be expressed within round brackets). If this is true, the function and the autoincrement of *a* are executed, to return then to the comparison between *a* and *b*; if the condition is true, the cycle is repeated, otherwise the operation proceeds – as already said above – with what comes after the closed curly bracket.

From this, one can first of all deduce that if at the first test the condition is not true, the cycle is not executed – not even once. Furthermore, it is indispensable that within the curly brackets something happens to determine the necessary conditions for exiting the cycle; in this case, the subsequent increments of *a* sooner or later render false the condition from which the entire **while** cycle depends.

There is, however, another method for abandoning a cycle in the presence of a given condition: this is the **break** instruction. Example:

```
while(a < b)

{

  function(a);

  if(++a == 100)

   break;

  --c;

}
```

In this example, *a* is incremented and then compared with the value 100; if equal to it, the cycle is interrupted, otherwise it proceeds with the decrement of *c*. It is also possible to exclude from execution part of the cycle, forcing a return to the test:

```
while(a < b)
{
  if(a++ < c)
    continue;
  function(a);
  if(++a == 100)
    break;
  --c;
}
```

In the last example above, *a* is compared with *c* and is incremented. If, before the increment, *a* is less than *c*, the processing flow returns to the test of the **while** instruction; the responsibility of the forced jump rests with the **continue** instruction, which enables a new iteration to be started from the beginning. Otherwise, function(a) is called, and a new test is subsequently performed, with possible exit from the cycle.

While cycles can be nested:

```
while(a < b)
{
  if(a++ < c)
    continue;
  function(a);
  while(c < x)
    ++c;
  if(++a == 100)
    break;
  --c;
}
```

Inside the cycle for (a > b) there is a second cycle for (c < x). Already in the first iteration of the "external" cycle, if the (c < x) condition is true, we enter into the "internal" cycle, which is entirely processed (i.e. *c* is incremented until it takes on a value equal to *x*), before executing the next

instruction of the external cycle. In practice, at every iteration of the external cycle there is a complete series of iterations in the internal cycle.

It should be emphasised that any **break** or **continue** instructions present within the internal cycle relate exclusively to this cycle; **break** would cause exiting from the internal cycle, while **continue** would produce a return to test, still within the internal cycle. Finally, we can also note that the cycle for (c < x) is made up of a single instruction – it is for this very reason that it was possible to omit the curly brackets.

4.2.2 Do…While Cycle

The **do...while** cycle is very similar to *while*-type cycles. Let us take a look at one:

```
do
{
  if(a++ < c)
    continue;
  function(a);
  while(c < x)
    ++c;
  if(++a == 100)
    break;
  --c;
}
while(a < b);
```

It is not by chance that this example is the same as was used earlier with reference to the **while** instruction; in fact, the two cycles are identical in every way, with the exception of a single detail.

In **do…while** type cycles, the test on condition is performed at the end of the iteration, not at the beginning, and this has two important consequences.

First of all, a **do…while** cycle is always executed at least once; in fact, the processing flow has to go through the cycle's entire code block, before getting to evaluate the condition for the first time. If the condition is false, the cycle is not repeated, and processing proceeds with the first instruction following **while**, but it remains obvious anyway that the cycle has already been completed once.

Secondly, the **continue** instruction does not determine a jump backwards, but forwards. In fact, it forces in any type of cycle a new condition check; in **while** cycles, the condition is at the beginning of the code block, therefore to reach it from an intermediate point within the latter, it is necessary to jump backwards, whereas in **do...while** cycles the test is found at the end of code and is obviously reached with a jump forward.

For all other aspects of the behaviour of **do...while** cycles, especially the **break** instruction, the same considerations apply as already made about *while*-type cycles.

The compiler generates an advisory if the assignment operator is used as condition of a control flow statement such as `if` or `while`.

Example:

```
If (a == b)
     s++;
;


// The s++ is executed only if a and b are equal.


If (a = b)
     s++;
;
// The value b is copied into a and s++ is always executed.
```

4.2.3 For Cycle

Among C instructions for cycle control, **for** is undoubtedly the most versatile and efficient. **For** is found in all – or nearly all – languages, but nowhere else does it have the same power which it has within C. Indeed, generally speaking, *while*-type cycles and their derivations are used in situations where it is impossible to know *a priori* the exact number of iterations, whereas **for** – thanks to its "starting point; limit; increment step" logic – is particularly suited to those cases where the number of cycles to be iterated can be determined at the outset.

In the C **for**, three-coordinate logic still applies, but, unlike in nearly all other program languages, they are reciprocally unbound and not necessary. This means that while in Basic **for** acts on a single variable, which is initialised and incremented (or decremented) until a predetermined limit is reached, in C **for** can manipulate, for example, three different variables (or, rather, three expressions of different types).

What is more, none of the three expressions need necessarily be specified; a **for** without iteration conditions is perfectly admissible.

At this point, we may as well dispense with stating the obvious, to concentrate instead on the possible definition modalities of the three conditions which drive the cycle.

Let it immediately be said, therefore, that also **for** demands that conditions be specified within round brackets, and that if the cycle's code block includes more than one instruction, one needs to use the usual, open and closed curly brackets.

We can use the **break** and **continue** instructions in **for** cycles too: the first to exit the cycle, the second to "instantly" return to test evaluation.

Also **for** cycles can be nested, and one should bear in mind that the most internal cycle performs an entire series of iterations at every iteration of its most immediate containing cycle.

Let us examine a few examples of **for** cycle, which, in its most banal form, can look as follows:

```
for(i = 1; i < k; i++)

{  ....  }
```

There is nothing exceptional here. Before performing the first iteration, the *i* variable is initialised to 1. If it proves to be less than the *k* variable, the cycle is performed a first time. At the end of each iteration, *i* is incremented and subsequently compared with *k*; if it proves less than the latter, the cycle is repeated.

It is worth emphasising that all three logical coordinates are contained within round brackets and are separated from one another by a semi-colon (" ; "); only the (; ;) sequence is obligatory within a **for** cycle.

Indeed, we can have **for** with the following syntax:

```
for( ; ; )

{  ....  }
```

What is the meaning of this? Nothing is initialised. No test is performed. Nothing is modified. The secret lies in the fact that the absence of test is equivalent of an always verified condition: thus, in this example, for defines an infinite iteration. The program remains "trapped" in the cycle, until a condition arises that enables it to abandon the cycle in some other way, e.g. with the help of a **break**.

Let us look at another example:

```
for( ; a++; )

{  ....  }
```

There is nothing remarkable here, after all: it is verified that *a* does not have a null value, and *a* is then incremented. If the verification returns a positive result, the cycle's code is executed. Verification is then repeated, closely followed by the increment, and so forth… By paying a little closer attention to all this, one can notice that this **for** cycle is a perfect equivalent of a **while** cycle. In fact:

```
while(z++)

{  ....  }
```

Indeed, we could go so far as to say that in C the **while** instruction is perfectly useless, given that it can always be replaced by **for**, which, on the contrary, generally makes it possible to achieve a more compact and efficient coding of the algorithm. The greater compactness derives from the possibility of using, if need be, contextually to the condition, an initialisation instruction and a variation, too. The greater efficiency, on the other hand, depends on the technical behaviour of the compiler, which, where possible, automatically manages **for** cycle counters, as **register** variables.

It is perhaps worth emphasising once more that the contents of the round brackets depend heavily on the cycle one wants to execute and on the processing set-up one wants it to have, whereas the use of the double semi-colon is mandatory. The first and last parameter do not necessarily have to initialise and increment (or decrement) the counter, just as the intermediate parameter does not necessarily have to be a condition to be evaluated. Each of these parameters can be any C instruction, or can be omitted. The compiler, however, always interprets the central parameter as a condition to be verified, independently of what it actually is. This parameter is therefore always evaluated as true or false, and from it depend both entry into the cycle and its subsequent iterations.

4.2.4 Usage of Cycles within UNI-PRO

It is worth underlining the fact that the use of cycles within the UNI-PRO development environment entails a variation of the execution times of programs. In fact, a very heavy cycle might slow down considerably the execution times of the program.

Let us suppose that, for some reason (e.g. a programmer error or a badly-set variable), the exit condition from a cycle used within an algorithm never occurs, thus creating an infinite-cycle situation; this would entail an infinite main cycle, resulting in the repeated processing of the same operation (irresolvable). In this situation, the application would be stopped after a few hundreds of milliseconds and an auto-reset event will follow. The controller restarts from the reset interrupt.

It is very important, therefore, to pay great attention to the use of the cyclical instructions described in the preceding paragraphs.

5. Array and Structures

5.1 Array

An array can be described as "an organised collection of objects". First of all, the mere concept of **"collection"** implies that such objects are of the same type, thus, taking inspiration from the real world, we could define an array of apples, which could not include any "pear objects". Thus in C an array is a collection of variables of the same kind.

"Organised" entails that it should be possible to identify univocally all objects within the array, in a systematic way; in C, this is done through the use of numerical indexes, which in an N-dimensioned array range from 0 to N-1.

Let us see in detail how it is possible to declare an array:

```
short myarray[10];
```

As can be seen above, an array is declared by writing the variable name (here *myarray*) and, within square brackets, a figure which identifies the number of elements of the same type (here *short*), i.e. the size of the array.

Going back to the C program language, let us see how it is possible to declare **float** or **char** arrays:

```
float float_array[12];

char char_array[7];
```

Once an array has been declared, it is possible to assign a value to the corresponding position, by recalling it via the index; for example, if one wanted to insert the value 87.43 into the float array in fifth position, it would be enough to write:

```
float_array[4] = 87.43;
```

If on the other hand one wanted to utilise the value contained in the third position of the array, and store it into another variable, one should proceed as follows:

```
float myvar;

myvar = float_array[2];
```

There is a strong relationship between the usage of arrays and **for** cycles; this is due to the fact that a cycle enables counting a certain number of times. Thus, by using a variable that increments (or decrements) its own value at every cycle, its is possible to scroll the array's positions in a simple way, with considerable economy of written code.

As an example, let us assume that we have an **int** array of 100 elements, and that we want to calculate the sum of the contents of all the array's positions. Given the array's peculiar nature, we are not going to start counting from 1, but from 0, and up to ninety-nine (thus having one-hundred actual elements). We are going to perform the sum of every element of the array using the index, incremented every time, taken from the **for** cycle.

In code:

```
short my_array[100];

unsigned short i;

long sum = 0;

for (i=0; i<100; i++)

{

  sum += my_array[i];

}
```

In this case, the control variable “i” of the **for** cycle is used as an index for the array; it should be noted that visually it is immediately obvious that we are dealing with a cycle of 100 elements, since the given condition is to have “i<100”, starting the count from zero; thus the cycle is executed up to the ninety-ninth element, which is less than one-hundred.

At the next iteration, before executing the cycle’s body, “i” is incremented by 1 and therefore is worth 100, a value which no longer verifies the condition (since 100 is not *less* than 100, and may only be equal to it), and which causes the interruption of the **for** cycle; the counted interval (0 to 99) is useful to represent all one-hundred elements of the array, given that, as already explained earlier, the elements in an array are counted starting from zero.

In this way, it could also be fairly simple to initialise all the values in the array. Let us imagine that we wished to give each element in the array the value which equals its position; with the following solution, we would save a lot of code:

```
short my_array[100];

unsigned short i;

for (i=0; i<100; i++)

{

  my_array[i] = i;

}
```

Of course, an array can be initialised during declaration, also passing values directly, for example, as in the following case:

```
short numbers[] = { 7, 23, 4, 94, 120 };
```

Here an array of 5 elements is created, and this is why the square brackets do not contain a size, as this can be derived from the number of elements enclosed within the curly brackets.

Let us take, as a further example, character arrays, which have the peculiar capacity of being initialised without being divided by commas; we are now going to propose two equivalent forms of initialisation, the first of which is undoubtedly the more practical one:

```
char my_string[] = "Hello World!";

char my_string[] = { 'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!' };
```

A peculiarity of the C is that there is no proper “string type”, i.e. what is known in other program languages as “string” (think of C++ or Java). In C, strings are represented by character arrays, therefore one must be careful with the operation to be performed and one must remember that anyway they do offer the advantage of enjoying all the qualities of an array, among them that of being able to scroll at leisure in the position of the string itself. Every character array ends with the escape sequence, i.e. the value 0.

Obviously, the power of arrays also lies in the fact that it is possible to use multi-size arrays. In practice, every element contained in an array can in turn be itself an array; in this way, it is easily possible to represent tables and matrixes, or anything else that requires an even higher level of representation.

The example given below uses a bi-dimensional array to define a matrix of N lines and M columns:

```
short matrix[n][m];
```

whose elements can, for example, be scanned, using only two **for** cycles, as illustrated below:

```
short n = 10;

short m = 12;

short matrix[n][m];

short i, j;

for (i=0; i<n; i++)

   for (j=0; j<m; j++)

      if (matrix[i][j] == 0)

          { ... }
```

5.1.1 Usage of Arrays within UNI-PRO

UNI-PRO supports multi-dimensional `arrays` in C algorithms as defined by the ANSI-C standard.

Multi-dimensional arrays are for local variables and constants, and thus cannot be accessed outside of the C algorithm in which they are defined.

Example:

```
CJ_WORD a[10][5];/* the order is [row] [col] : a is a matrix of 10 rows
and 5 columns */
```

UNI-PRO does not support dynamic arrays (arrays with non-constant size).

Example:

```
CJ_BYTE size=10;

CJ_SHORT a[size];
```

This declaration will produce a compiler message: “invalid array subscript: integral constant expression is expected”.

The declaration

```
CJ_WORD a[10];
```

defines an array of size 10, that is, a block of 10 consecutive objects named a[0], a[1], …,a[9], because C arrays are zero-indexed.

The notation a[i] refers to the i-th element of the array.

Note: As the C compiler does not recognize if the program uses an array index out of limits, be aware about the dimension of the arrays. If the indexed element is outside of limits, it can cause undesired overwrites of memory.

Example:

```
CJ_WORD a[10];

a[10] = 0;
```

This will cause memory overwrites.

As is the case for any type of computer programming, it is the responsibility of the programmer to write code that prevents unwanted or unintended results from mathematical operations and memory access operations. In addition, UNI-PRO enables the definition of some array-type entities, with the following limitations:

- These entities are only variables, parameters or constants, and thus cannot, for example, be analogue inputs, digital outputs or timers.
- Arrays are only mono-dimensional.
- The maximum size for an entity array is 100 elements.
- Only the following data types can be used in these arrays: CJ_VOID, CJ_BIT, CJ_BYTE, CJ_S_BYTE, CJ_SHORT, CJ_WORD, CJ_DWORD, CJ_LONG.

Note: The maximum size for an entity array is 100 elements.

5.2 Structures

UNI-PRO allows arbitrary structure types. The structures may be nested, and may contain arrays.

The sizeof operator applied to a structure type yields the sum of the sizes of the components. However, the ANSI-C standard allows arbitrary padding between components and also requires that the fields of a `struct` be allocated in the order they are declared.

The controller has a 16 bit architecture with 2 byte boundary alignment.

Example:

```
struct foo

{

  CJ_SHORT a; // 2 byte

  CJ_CHAR  b; // 1 byte

};

struct foo my_var;

return sizeof(my_var);
```

It will return the value 4.

Recursive structures are allowed, but their use is not useful because it is not possible to handle objects that are dynamically allocated like `lists`, `queues` or `trees`.

Moreover structures with dynamic arrays are not permitted.

5.3 Unions

UNI-PRO allows the use of unions to use the same storage for multiple data types.

5.4 Strings

ANSI-C implements strings of characters as an array. The internal representation of a string has a null character '\0' at the end, so the physical storage required is one more than the number of characters written between the quotes. `Strings` are then often represented by means of a pointer pointing to the array.

UNI-PRO provides full support for string constants, usable either in initializers or as

a constant.

Example:

```
char s[]="abc"; // A string array s is initialized using a string
constant.

i=1;

s[i]='y'; // The second character of s is modified.
```

6. Comments

It is possible to insert into the code text comments, which are totally ignored by the compiler and in which the programmer can comment on the code or on the functionality of a given block of code.

There are two types of comments:

- single-line comments
- multi-line comments

To insert a single-line comment, it is necessary to enter the sequence "//", whereas a multi-line comment starts with the sequence "/*" and ends with the sequence "*/".

The following is an example of single-line comment:

```
// Declaration of variables

short a,b;

a = 0; // Variable a initialised at 0

b = 1; // Variable b initialised at 1
```

Or, in the case of a multi-line comment:

```
/* Declaration of variables */

short a, b;

/*

Initialisation of variables

to values 0 and 1

*/

a = 0;

b = 1;
```

7. Define

With the help of the **define** (or **#define**) command, it is possible to define constants, with the aim of making the source code more easily legible.

Example: Let us suppose that we wanted to compare a number with a fixed value:

```
if (a > 32655)

     a = 32655;
```

This portion of code if much less clear than this one:

```
//Definition of a constant

#define MAX_VALUE 32655

if (a > MAX_VALUE)

     a = MAX_VALUE;
```

As one can see, the second solution is much better optimised, with the result that the written code is more readily legible and understandable.

Let us now suppose that a given value is used several times within a program, and that there is a need to modify it; in that case, one would have to intervene on every single instruction containing that value, to modify it. This operation could cause some bugs, indeed one could forget to update a portion of code, or update it incorrectly. This type of problems can be solved by using the define command; in fact, if one defines a constant:

```
#define MY_VALUE 12345
```

it will be possible to use MY_VALUE in the code lines, thus avoiding the direct use of the numeric value. Now, if it should be necessary to modify the value, it would be sufficient to modify it only on that instruction, in order to implicitly modify it also on all other instructions that require its usage.

```
#define MY_VALUE 12333

if (a = MY_VALUE){…}

for (i=0; i<MY_VALUE;i++) {…}

return MY_VALUE;
```

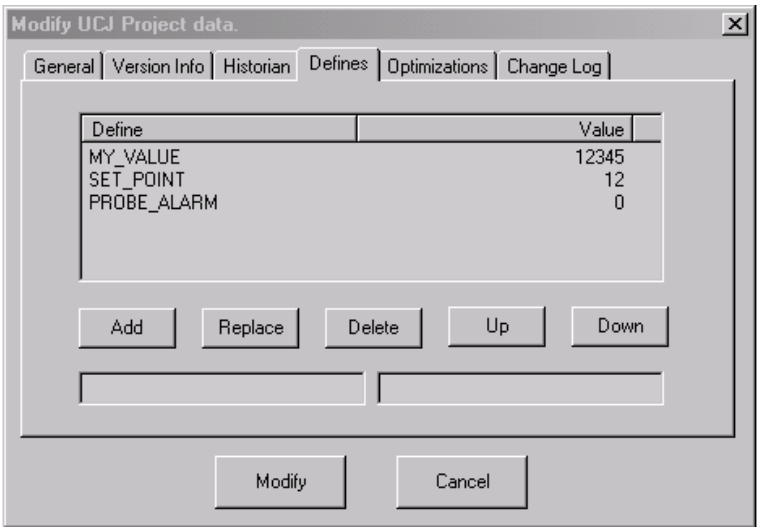
7.1 The DEFINE Instruction in UNI-PRO

Within UNI-PRO, the DEFINE instruction can be used in two modalities:

1. Project DEFINE – as property which is visible to all the project code.
2. Algorithm DEFINE – as property which is limited to certain parts of the code.

7.1.1 Project DEFINE

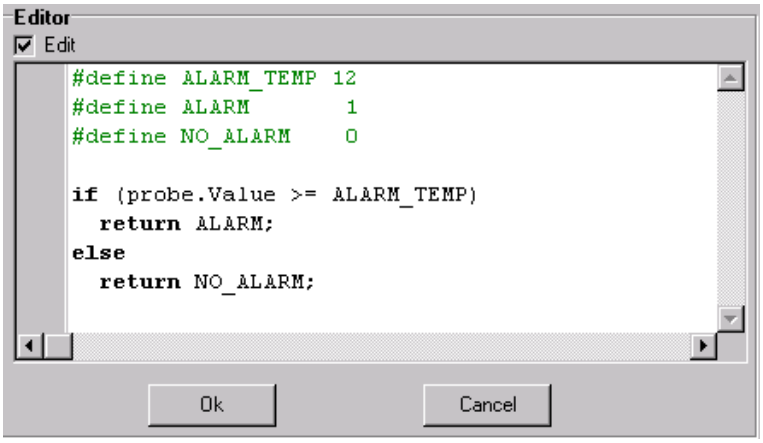
DEFINE, as used in this way, has the property of being usable in any algorithm contained in the project in question, without the need to redefine it in every single block. In order to exploit this functionality, it is sufficient to define the project constants, via the function Property - Defines (available in UNI-PRO) of every created project. For the creation of the necessary DEFINE, the following window undet the **Defines** Tab is used:



In this way, the constants as defined above will be usable in every part of the program, without any need for redefinition.

7.1.2 Algorithm DEFINE

If a constant is necessary only within a given algorithm, it is more convenient to define it exclusively for that portion of code. Using the normal C syntax, the chosen constant is defined, and this will only be valid in the algorithm in which it has been defined.



This is an example of code which could be written using the *Algorithm Editor* available in UNI-PRO.

It is recommended to pay particular attention to this functionality, in case the same DEFINE should be declared within different algorithms, since any modification to the value of these DEFINE will also have to be reproduced in all other algorithms where it is used; any omission would be very likely to cause malfunction of the entire project.

8. Limitation

8.1 Function Calls

UNI-PRO only supports function calls of pre-defined functions (CJ_GetWeekDay(), CJ_GetTime(), CJ_ReadVarExpo(), etc.). It is not possible to create your own functions.

Note: In the algorithm window, press CTRL + Space to display the list of available pre-defined functions.

8.2 Assertions

UNI-PRO does not support *assertions*. The `assert` statement will generate a compiler detected error message.

8.3 Pointers

The use of pointers is allowed in UNI-PRO. However, they can be a source of many difficulties to find programming errors, and care must be exercised by the programmer.

As is the case for any type of computer programming, it is the responsibility of the programmer to write codes that prevent unwanted or unintended results from memory access operations.

⚠ WARNING

UNINTENDED EQUIPMENT OPERATION

- Be sure that all variables are initialized to an appropriate value before their first use as array indices or pointers.
- Write programming instructions to test the validity of operands intended to be used as array indices and memory pointers.
- Do not attempt to access an array element outside the defined bounds of the array.

Failure to follow these instructions can result in death, serious injury, or equipment damage.

8.4 Dynamic Memory

UNI-PRO does not allow programs that attempt to allocate dynamic memory. All the functions that are used to manage the dynamic memory, such as `malloc()` or `free()`, cannot be compiled.

9. UNI-PRO Compilation

9.1 Unused Code

UNI-PRO compiles all codes, even when the code is not used. Unused codes, however, will not be linked, which will reduce the memory size of the application.

10. Standard Library

The standard library is not part of the C language proper, but an environment that supports standard C will provide the function declarations and type and macro definitions of this library, through header files.

The ANSI standard defines these headers:

```
<assert.h> <float.h>  <math.h>   <stdarg.h> <stdlib.h>
<ctype.h> <limits.h>  <setjmp.h> <stddef.h> <string.h>
<errno.h>  <locale.h> <signal.h> <stdio.h>  <time.h>
```

UNI-PRO supports by default only the following headers. They do not need to be included in each algorithm.

```
<math.h> <stdlib.h>   <ctype.h>  <string.h> <errno.h>          <stddef.h>
```

This chapter provides the prototypes of the functions used in each library. To use those functions in an algorithm, they must be included. Example:

```
#include <math.h>
```

10.1 Mathematical Functions: <math.h>

```
double acos(double);
double asin(double);
double atan(double);
double atan2(double, double);
double cos(double);
double sin(double);
double tan(double);
double cosh(double);
double sinh(double);
double tanh(double);
double exp(double);
double frexp(double, int *);
double ldexp(double, int);
double log(double);
double log10(double);
double modf(double, double *);
double pow(double, double);
double sqrt(double);
double ceil(double);
double fabs(double);
double floor(double);
double fmod(double, double);
```

10.2 Strings Functions: <string.h>

```
Void *memcpy(void*, const void*, size_t);
void *memmove(void*, const void*, size_t);
char *strcpy(char*, const char*);
char *strncpy(char*, const char*, size_t);
char *strcat(char*, const char*);
char *strncat(char*, const char*, size_t);

int  memcmp(const void*, const void*, size_t);
int  strcmp(const char*, const char*);
int  strncmp(const char*, const char*, size_t);
void *memchr(const void*, int, size_t);
char *strchr(const char*, int);
size_t strcspn(const char*, const char*);
char *strpbrk(const char*, const char*);
char *strrchr(const char*, int);
size_t strspn(const char*, const char*);
char *strstr(const char*, const char*);
char *strtok(char *, const char*);
void *memset(void *, int, size_t);
size_t strlen(const char*);
```

10.3 Character Class Test: <ctype.h>

```
int  isalnum( int);
int  isalpha( int);
int  iscntrl( int);
int  isdigit( int);
int  isgraph( int);
int  islower( int);
int  isprint( int);
int  ispunct( int);
int  isspace( int);
int  isupper( int);
int  isxdigit( int);
int  tolower( int);
int  toupper( int);
```

10.4 Utility Functions: <stdlib.h>

```
Double atof(const char*);
Int atoi(const char*);
long int atol(const char*);
double strtod(const char*, char **);
long int strtol(const char*, char **, int);
unsigned long int strtoul(const char *, char **, int);
int rand(void);
void srand(unsigned int);
void abort(void);
void *bsearch(const void*, const void *, size_t, size_t, int(*)(const
void *, const void *));
void qsort(void *, size_t, size_t, int (*)(const void *, const void *));
int abs(int);
div_t div(int, int);
long int labs(long int);
ldiv_t ldiv(long int, long int);
```

APPENDIX 1: Common Mistakes and Style Rules in C

- **Assignment (=) instead of comparison (==)** – One must be careful when, using a conditional structure such as **if-else**, one writes the comparison operation (==), as it can, because of a simple typing error, become an assignment (=). If this should happen, for example, trying to compare two numbers for equality, one could have the unpleasant situation in which, instead of checking whether **a == b** (which returns a TRUE only when the two variables have the same value), we are saying instead that **a = b**, which is practically always TRUE, its value being FALSE only when **b** is worth 0.
- **Missing ( ) for a function –** The inexperienced programmer tends to believe that a function to which no parameters are passed needs no round brackets; this is wrong, as the round brackets must always be used, even in the absence of parameters.
- **Array Indices** – When arrays are initialised or used, one must be careful with the used indices (thus also with the number of elements), because, if an array is initialised with N elements, its index must have a range between 0 (the first element) and N-1 (the N^{th} element).
- **C is Case-Sensitive** – The C program language (just as C++ and Java) differentiates between upper and lower-case letters, thus interpreting them as two different characters; therefore, one needs to be careful, especially when variables are being used.
- **The semicolon ";" closes every instruction** – This is such a common mistake, that it must be mentioned: every instruction must end with a semi-colon; this easy omission, which is reported by the compiler, can waste precious time and renders the programmer's job unduly burdensome.
- **The names of variables, structures, constants and functions must be significant** – Given that the keywords in this language are in English, it is recommended to use, for variables and functions, mother-tongue names, so as to make it possible to understand from the name itself whether what is being used is really a keyword of the language or a construction created inside our program. Furthermore, when the name is made up of several words, it would be a good rule to highlight with initial capitals all words after the first one; this rule does not, however, apply to constants, which must be written entirely in upper-case letters and separated, when made up of more than one word, by the underscore character "_".
- **Usage, function and position of comments** – Comments must accompany nearly all instructions, to explain the meaning of what is being done; in addition, they must be concise and be updated as soon as an instruction is modified. On the other hand, comments must be more exhaustive if they are needed to explain a given algorithm, or when they accompany a function.
- **Ternary Expression** – The ternary expression <cond> **?** <val> **:** <val> is generally a substitute for an **if-else** structure, and it returns the first value if the expression is TRUE, or the second if the expression is FALSE. Whenever possible, a ternary expression should be put all on one line, and, in order to avoid problems, it is advisable to enclosed within round brackets both the expression and the values.

APPENDIX 2: Reserved Words of the C program language

The ANSI C standard recognises the following keywords:

auto
break
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if
int
long
register
return
short
signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile
while

APPENDIX 3: Documentation of Built-in Functions

In the edit box of the algorithm, to see all the available functions press "Ctrl" + "Space" at the same time.

CJ_DATETIME StructToDateTime(CJ_DATETIME_STRUCT rtc)

This function converts a data structure CJ_DATETIME_STRUCT to a CJ_DATETIME data type.

CJ_DATE StructToDate(CJ_DATE_STRUCT date)

This function converts a data structure CJ_DATE_STRUCT to a CJ_DATE data type.

CJ_TIME StructToTime(CJ_TIME_STRUCT time)

This function converts a data structure CJ_TIME_STRUCT time to a CJ_TIME data type.

CJ_DATETIME_STRUCT DateTimeToStruct(CJ_DATETIME Value)

This function returns a data structure CJ_DATETIME_STRUCT created from the value of the *Value* parameters of CJ_DATETIME type used as input.

CJ_DATE_STRUCT DateToStruct (CJ_DATE Value)

This function returns a data structure CJ_DATE_STRUCT created from the value of the *Value* parameters of CJ_DATE type used as input.

CJ_TIME_STRUCT TimeToStruct (CJ_TIME Value)

This function returns a data structure CJ_TIME_STRUCT created from the value of the *Value* parameters of CJ_TIME type used as input.

CJ_BYTE CJ_GetSeconds(CJ_DATETIME dt)

This function returns the number of the seconds [0...59] contained in the parameter *dt* of CJ_DATETIME type.

CJ_BYTE CJ_GetMinutes(CJ_DATETIME dt)

This function returns the number of the minutes [0...59] contained in the parameter *dt* of CJ_DATETIME type.

CJ_BYTE CJ_GetHours(CJ_DATETIME dt)

This function returns the number of the hours [0...23] contained in the parameter *dt* of CJ_DATETIME type.

CJ_BYTE CJ_GetDay(CJ_DATETIME dt)

This function returns the number of the day [1...31] contained in the parameter *dt* of CJ_DATETIME type.

CJ_BYTE CJ_GetWeekDay(CJ_DATETIME dt)

This function returns the weekday [0 = Sunday, 1…6 = Saturday] contained in the parameter *dt* of CJ_DATETIME type.

CJ_BYTE CJ_GetMonth(CJ_DATETIME dt)

This function returns the number of the month [1...12] contained in the parameter *dt* of CJ_DATETIME type.

CJ_BYTE CJ_GetYear(CJ_DATETIME dt)

This function returns the year [00...68] contained in the parameter *dt* of CJ_DATETIME type.

CJ_TIME CJ_GetTime(CJ_DATETIME dt)

This function returns the time CJ_TIME derived to *dt* parameters, that is the number of seconds starting from midnight of the same day.

CJ_DATE CJ_GetDate(CJ_DATETIME dt)

This function returns the date CJ_DATE derived to *dt* parameters, that is the number of seconds starting from midnight of the year 2000 to the midnight of the same day.

CJ_BIT CJ_GetSecondTic(void) and CJ_BIT CJ_GetMinuteTic(void)

These two functions are managed under the system interrupt. These functions have the same rule as of the corresponding TIMER entities, but they do not use the controller memory because they are managed in the firmware and therefore they can be directly used in the algorithm without further definition.

The function CJ_BIT CJ_GetSecondTic (void) returns the logic value 1 for each elapsed second.

The function CJ_BIT CJ_GetMinuteTic (void) returns the logic value 1 for each elapsed minute.

It is advisable to use these functions when a great number of TIMER operations are required in the project to maintain an acceptable level of performance.

CJ_BIT CJ_FlagWrite (CJ_WORD i, CJ_BIT val) and CJ_BIT CJ_FlagRead (CJ_WORD i)

These functions help to manage the *Semaphores* in the algorithm.

For example, to manage a shared resource between entities, each entity has to know the status of the other entity to correctly utilize the shared resource. In UNI-PRO, this involves using several links between algorithms, or each algorithm has to have the status of every other algorithm that would use the shared resources as input. This solution has a disadvantage as it consumes a lot of memory in the controller.

To solve this, a typical solution of the concurrent computer programming is proposed: to use some entities called *Semaphores*. *Semaphore* is a structure which has the ability to manage access to some shared resources to control and assign them in the correct mode.

Two functions to realize this data management are:

CJ_BIT CJ_FlagWrite (CJ_WORD i, CJ_BIT val)

The function CJ_BIT CJ_FlagWrite (CJ_WORD i, CJ_BIT val) sets the status of the i-th semaphore.

CJ_WORD i: the semaphore number

CJ_BIT val: available/busy semaphore status

If *val=0,* the semaphore is set to free.

If *val=1,* the semaphore is set to busy.

CJ_BIT CJ_FlagRead(CJ_WORD i)

CJ_WORD i: the semaphore number

If *val=0,* the semaphore is set to free.

If *val=1,* the semaphore is set to busy.

For example,

CJ_FlagWrite (10, 1)

sets the status of the tenth semaphore as busy. Therefore, the resources managed by this semaphore will be accessible only by the entity that set the status.

A busy semaphore can be freed only from the same entity that has set its status.

Function call:

CJ_FlagWrite (10, 0)

sets the status of tenth semaphore as free; therefore, the resources are free to use by other entities.

The status of a semaphore is controllable using the following function:

CJ_BIT CJ_FlagRead (CJ_WORD i)

reads the status of the i-th semaphore. It returns 1 if the semaphore is busy, otherwise, it returns 0. If the semaphore is busy, it is not possible to use the controlled resources until the resources are freed.

CJ_SHORT CJ_WriteVarExpo(word add, long value)

This function allows to write directly from an algorithm the *value* value of an exported variable at the *add* address on Modbus protocol. Thus the variable is modifiable from all project algorithms. This feature is analog to the *global variable* concept used in computer programming.

To correctly use the function, it is required that the value is exported on Modbus protocol using *Export Entities* functionality that can be activated from *Tools/Export Entities* menu of the programming environment.

The CJ_SHORT output can have the following values:

0 = operation completed

-1 = operation correctly activated but not completed (for example, if you write a parameter, the operation is completed only when the value is saved in EEPROM, but it is considered valid when it is saved in RAM)

-11 = item not present

1 = out of range

2 = busy

CJ_LONG CJ_ReadVarExpo(word add)

This function allows to read the value of the exported variable at the *add* address on Modbus protocol. The function is the complement of *CJ_WriteVarExpo(word add, long value).*

To correctly use the function, it is required that the value is exported on Modbus protocol using *Export Entities* functionality that can be activated from *Tools/Export Entities* menu of the programming environment.

CJ_WORD CJ_MaxMainTime(void)

Returns the maximum cycle time of the main application, returned in milliseconds.

CJ_WORD CJ_MinMainTime(void)

Returns the minimum cycle time of the main application, returned in milliseconds.

CJ_WORD CJ_RunMainTime(void)

Returns the current cycle time of the main application, returned in milliseconds.

CJ_BYTE CJ_MaxInterruptTime(void)

Returns the maximum cycle time of applicative interrupt, returned in milliseconds.

CJ_BYTE CJ_MinInterruptTime(void)

Returns the minimum cycle time of applicative interrupt, returned in milliseconds.

CJ_BYTE CJ_RunInterruptTime(void)

Returns the current cycle time of applicative interrupt, returned in milliseconds.

CJ_BYTE CJ_ModbusAskQueue(void)

This function returns the free items number of the Modbus queue.

Note: If the MBS2 serial line is not configured as Modbus Master in the Hardware expert, a compilation error (unresolved external symbol (CJ_ModbusAskQueue) appears).

CJ_BYTE CJ_SendCommand(CJ_BYTE channel, CJ_BYTE node, CJ_BYTE command, short par1)

Allows to send a command from inside of an algorithm;

Returns 0 = command sent, 1= full queue

Channel: 0 = ExpBus

Node: logic node of ExpBus channel

Command: command index

Par1: 16 bit parameter associated to the command.

CJ_BIT CJ_IsFirstMain(void)

Returns 1 for the entire first loop of the main cycle of applicative.

CJ_BIT CJ_Stack_Error_Read(void)

Returns 1 if there was a stack overflow of the program.

CJ_BIT CJ_Math_Error_Read(void)

Returns 1 if there was a mathematical error detected: division by zero, overflow, underflow, Not a Number.

To find the specific detected error, use the following READ functions.

CJ_BIT CJ_DivByZero_Error_Read(void)

Returns 1 if there was a division by zero.

CJ_BIT CJ_Overflow_Error_Read(void)

Returns 1 if there was an overflow.

CJ_BIT CJ_Underflow_Error_Read(void)

Returns 1 if there was an underflow.

CJ_BIT CJ_NaN_Error_Read(void)

Returns 1 if there was a NaN (Not a Number) generic detected error.

void CJ_DivByZero_Error_Write(void)

Sets the division by zero indication.

void CJ_Overflow_Error_Write(void)

Sets the overflow indication.

void CJ_Underflow_Error_Write(void)

Sets the underflow indication.

void CJ_NaN_Error_Write(void)

Sets the arithmetic generic detected error; NaN means Not a Number, for example, the square root of a negative number.

void CJ_Overflow_Error_Reset(void);

Resets the overflow indication.

void CJ_Underflow_Error_Reset(void);

Resets the underflow indication.

void CJ_DivByZero_Error_Reset(void);

Resets the division by zero indication.

void CJ_NaN_Error_Reset(void);

Resets the arithmetic generic detected error; NaN means Not a Number, for example, the square root of a negative number.

void CJ_Math_Error_Reset(void);

Reset the global mathematical detected error flag.

CJ_SHORT CJ_E2_Error_Read (void);

Returns the E2 retained memory status:

0 CJ_E2_OK. Operating correctly

1 CJ_E2_READ_ERROR. An E2 access error has been detected.

2 CJ_E2_WRITE_ERROR. An E2 write error has been detected.

3 CJ_E2_CRC_ERROR. Inconsistent data in the memory.

CJ_SHORT CJ_RTC_Error_Read (void);

Returns the Real Time Clock status:

0 CJ_RTC_OK. Operating correctly

1 CJ_RTC_READ_ERROR. A RTC access error has been detected.

2 CJ_RTC_LOW_VOLTAGE. The RTC chip is below the minimum threshold voltage necessary for maintaining information. The data present may no longer be valid.

APPENDIX 4: Glossary of Terms

ANSI C : Standard version of the C program language C, as defined by the *American National Standards Institute*.

Assembly : Low-level program language, whose instructions can be directly converted into machine code.

Assembler : A program which converts assembly code into machine code.

C++ : An extension of the C program language which enables object-oriented programming.

Compiler : A program which automatically translates code, written in a high-level language, into assembly language to be executed by the machine.

Entity : An element of the UNI-PRO development environment, which can feature at least one input or one output, represented by a data type. By combining several entities, one can create an application program.

Firmware : A type of software which is stored in memory and is usually read-only. Within UNI-PRO, it represents the software of the application executed by the controller.

Linker : A program which links together a series of separately-compiled sub-programs, in order to obtain a complete operating program.

Software : Generic term used to indicate all non-tangible components of an information system, such as the programs and the data being processed.

UNI-PRO : Graphic development environment which enables the creation of controller programs in an assisted and simplified way.

Bibliography:

Brian W.Kernighan and Dennis M.Ritchie,

"**The C Programming Language”** Second Edition,

Prentice Hall, 1988

Herbert Schildt,

“**C: The Complete Reference**” Second Edition,

Osborne, 1990

UNI-PRO – Introduction manual to the C Program Language.

Version 2.2 - January 2011.

Code 114UPROCLE22.

File 114UPROCLE22.pdf.

HEADQUARTERS
Evco
Via Mezzaterra 6, 32036 Sedico Belluno ITALY
Tel. +39 0437-852468
Fax +39 0437-83648
info@evco.it
www.evco.it

OVERSEAS OFFICES
Control France
155 Rue Roger Salengro, 92370 Chaville Paris FRANCE
Tel. 0033-1-41159740
Fax 0033-1-41159739
control.france@wanadoo.fr

Evco Latina
Larrea, 390 San Isidoro, 1609 Buenos Aires ARGENTINA
Tel. 0054-11-47351031
Fax 0054-11-47351031
evcolatina@anykasrl.com.ar

Evco Pacific
59 Premier Drive Campbellfield, 3061, Victoria Melbourne, AUSTRALIA
Tel. 0061-3-9357-0788
Fax 0061-3-9357-7638
everycontrol@pacific.com.au

Evco Russia
111141 Russia Moscow 2-oy Proezd Perova Polya 9
Tel. 007-495-3055884
Fax 007-495-3055884
info@evco.ru

Every Control do Brasil
Rua Marino Félix 256, 02515-030 Casa Verde São Paulo SÃO PAULO BRAZIL
Tel. 0055-11-38588732
Fax 0055-11-39659890
info@everycontrol.com.br

Every Control Norden
Cementvägen 8, 136 50 Haninge SWEDEN
Tel. 0046-8-940470
Fax 0046-8-6053148
mail2@unilec.se

Every Control Shangai
B 302, Yinhai Building, 250 Cao Xi Road, 200235 Shangai CHINA
Tel. 0086-21-64824650
Fax 0086-21-64824649
evcosh@online.sh.cn

Every Control United Kingdom
Unit 19, Monument Business Park, OX44 7RW Chalgrowe, Oxford, UNITED KINGDOM
Tel. 0044-1865-400514
Fax 0044-1865-400419
info@everycontrol.co.uk