SoHVAC

C Programming Language

User Guide

06/2014

EIO0000000536.02

www.schneider-electric.com

Table of Contents

Safety Information

Important Information

NOTICE

Read these instructions carefully, and look at the equipment to become familiar with the device before trying to install, operate, or maintain it. The following special messages may appear throughout this documentation or on the equipment to warn of potential hazards or to call attention to information that clarifies or simplifies a procedure.

The addition of this symbol to a Danger or Warning safety label indicates that an electrical hazard exists, which will result in personal injury if the instructions are not followed.

This is the safety alert symbol. It is used to alert you to potential personal injury hazards. Obey all safety messages that follow this symbol to avoid possible injury or death.

DANGER

DANGER indicates an imminently hazardous situation which, if not avoided, **will result in** death or serious injury.

WARNING

WARNING indicates a potentially hazardous situation which, if not avoided, **can result in** death or serious injury.

CAUTION

CAUTION indicates a potentially hazardous situation which, if not avoided, **can result in** minor or moderate injury.

NOTICE

NOTICE is used to address practices not related to physical injury.

PLEASE NOTE

Electrical equipment should be installed, operated, serviced, and maintained only by qualified personnel. No responsibility is assumed by Schneider Electric for any consequences arising out of the use of this material.

A qualified person is one who has skills and knowledge related to the construction and operation of electrical equipment and the installation, and has received safety training to recognize and avoid the hazards involved.

About the Book

At a Glance

Document Scope

This document describes the conformance of **SoHVAC** with the ANSI-C standard. It explains what is fully compatible, allowed or not allowed in the writing of C algorithms using the Code Editor.

Validity Note

This document is valid for **SoHVAC** V3.0.

Related Documents

Title of Documentation	Reference Number
SoHVAC Software User Guide	EIO0000000537

You can download these technical publications and other technical information from our website at www.schneider-electric.com

Product Related Information

The application of this product requires expertise in the design and programming of control systems.

⚠ WARNING

UNINTENDED EQUIPMENT OPERATION

- Only qualified persons with the skills and knowledge of electrical control systems, and having the related safety training, should be allowed to program, install, alter and otherwise apply this product.
- Understand and follow all local, regional and national safety codes and standards.

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

⚠ WARNING

LOSS OF CONTROL

- The designer of any control scheme must consider the potential failure modes of control paths and, for certain critical control functions, provide a means to achieve a safe state during and after a path failure. Examples of critical control functions are emergency stop and overtravel stop, power outage and restart.
- Separate or redundant control paths must be provided for critical control functions.
- System control paths may include communication links. Consideration must be given to the implications of unanticipated transmission delays or failures of the link.
- Observe all accident prevention regulations and local safety guidelines.[1]
- Each implementation of this equipment must be individually and thoroughly tested for proper operation before being placed into service.

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

[1] For additional information, refer to NEMA ICS 1.1 (latest edition), "Safety Guidelines for the Application, Installation, and Maintenance of Solid State Control" and to NEMA ICS 7.1 (latest edition), "Safety Standards for Construction and Guide for Selection, Installation and Operation of Adjustable-Speed Drive Systems" or their equivalent governing your particular location.

⚠ WARNING

UNINTENDED EQUIPMENT OPERATION

- Only use software approved by Schneider Electric for use with this equipment.
- Update your application program every time you change the physical hardware configuration.

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

1 Introduction to the C Programming Language

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

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

SoHVAC 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 M168 family using the **SoHVAC** development environment.

2 ANSI-C Language Features in SoHVAC

2.1 Data Types

2.1.1 Simple Data Types Defined by C

The following table shows the scalar data types defined by the ANSI-C standard:

Data Type	Sign	Repr.	Minimum	Maximum	Notes
char	Yes	8 bit	-128	127	
unsigned char	No	8 bit	0	255	
short int	Yes	16 bit	–32768	32767	
unsigned short int	No	16 bit	0	65535	
long int	Yes	32 bit	-2147483648	2147483647	
unsigned long int	No	32 bit	0	4294967295	
float	Yes	32 bit			Single-precision floating-decimal data.
double	Yes	64 bit			Double-precision floating-decimal data.
int	Yes	16 bit	-32768	32767	

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[1]	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.

[1] Long double type and double type declarations are recognized as the same type.

2.1.2 Simple Data Types Expected by SoHVAC

The simple data types adopted by **SoHVAC** are not found in the ANSI-C standard but do have equivalent data types, as defined in the following table:

Data Type	**Sign**	**Repr.**	**Minimum**	**Maximum**	**Corr. ANSI-C**
CJ_BIT	No	1 bit	0 (FALSE)	1 (TRUE)	(*)
CJ_CHAR	No	8 bit	0	255	Unsigned char
CJ_S_BYTE	Yes	8 bit	-128	127	Signed char
CJ_BYTE	No	8 bit	0	255	Unsigned char
CJ_SHORT	Yes	16 bit	–32768	32767	Signed short
CJ_WORD	No	16 bit	0	65535	Unsigned short
CJ_IP_ADRESS	No	32 bit	0.0.0.0	255.255.255.255	BCD Value (4bytes)
CJ_LONG	Yes	32 bit	-2147483648	2147483647	Signed long
CJ_DWORD	No	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 types, 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 **SoHVAC** environment are defined below.

CJ_VOID

The **CJ_VOID** data type is an innovative concept, introduced by the **SoHVAC** development environment, which enables a considerable reduction of project development time, while allowing a high degree of flexibility.

With this new concept, it is possible to define the data types of generic objects (for example, 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 possible values that can be taken on respectively by a LED or a 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 and is capable of representing dates up to 2068.

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 purpose of carrying out processing involving hours. It can be useful for managing application time bands of a regulator as well as in many other needs. 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 can represent 2^31 seconds, which is equivalent to 68 years.

This data type can be used directly within algorithms, or, to make it easier to work with, it can be converted into the **CJ_DATETIME_STRUCT** structure via library functions (refer to 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 language enables the definition of more complex types by combining several basic data types into one structure.

The **SoHVAC** 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 of two fields:

Short type Value: represents the value read by the probe

Byte type detected error: represents an error code. This code is only valid for temperature or 4-20 mA inputs. The error codes are as follows:

- 0: no error detected
- 1: the probe has a short-circuit
- 2: the probe is interrupted or missing

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**: 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**: indicates the logical node of the controller sending the command.
- **Short type Param**: 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 hold or released.

It is made of the following fields:

- **Boolean-type Valid**: 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**: indicates the logical node where the key action has been verified.
- **Short type Param**: indicates the number of seconds of persistence of the corresponding key.

CJ_DATE_STRUCT

The **CJ_DATE_STRUCT** type is used 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 of the following fields:

- **Byte type Day**: indicates the days [1 to 31]
- **Byte type Month**: indicates the month [1 = January, 2 = February, … 12 = December]
- **Byte type Year**: indicates the last two digits of the year 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 is used 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 of the following fields:

- **Byte type Sec**: indicates the seconds [0 to 59]
- **Byte type Min**: indicates the minutes [0 to 59]
- **Byte type Hour**: 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, given in seconds) into an easier format.

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

```
CJ_DATETIME_STRUCT DateTimeToStruct(CJ_DATETIME Value);
```

Its field descriptions are the following:

- **Byte type Sec**: indicates the seconds [0 to 59]
- **Byte type Min**: indicates the minutes [0 to 59]
- **Byte type Hour**: indicates the hours [0 to 23]
- **Byte type Day**: indicates the days [1 to 31]
- **Byte type WeekDay**: indicates the day of the week [0 = Sunday, 1 = Monday, … 6 = Saturday]
- **Byte type Month**: indicates the month [1 = January, 2 = February, … 12 = December]
- **Byte type Year**: indicates the last two digits of the year, 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 syntax is as follows:

```
CJ_DATETIME StructToDateTime(CJ_DATETIME_STRUCT rtc);
```

2.2 Introduction to C Language

2.2.1 Variable Declaration

A variable is declared as follows:

```
var_type Name_of_variable_separated_by_comma;
```

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;
...
```

2.2.2 Comments

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

Example:

```
short somma;
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.

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;
```

2.2.3 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;
…
```

2.2.4 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 comprise the block, are controlled by the `if` keyword, and are executed only if the condition c < 3 is verified.

2.2.5 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.

2.2.6 Function Calls

To call a defined function, enter the function name (note that C is a case-sensitive program language), and then indicate the various arguments enclosed within parenthesis and separated by commas.

For example, to call a function is defined as follows:

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

You must write:

```
short A = 2;
short B = 5;
short maxValue = max(A,B);
```

3 Operators

SoHVAC supports the following arithmetic operators on scalar variables a and b:

Operator	Description
-a	Algebric negation
a + b	Sum
a - b	Subtraction
a * b	Multiplication
a / b	Division
a % b	Remainder
++	Auto increment
--	Auto decrement
a << b	Bit-wise left shift [1]
a >> b	Bit-wise right shift [1]
a & b	Bit-wise **And** [1]
a | b	Bit-wise **Or** [1]
a ^ b	Bit-wise **Xor** [1]
~	Ones complement [1]
a < b	Comparison
a <= b	Comparison
a > b	Comparison
a >= b	Comparison
a == b	Comparison
a != b	Comparison

[1] not applicable on float format

NOTE: In case of sum or multiplication, the **SoHVAC** compiler cannot test 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.

⚠ WARNING

UNINTENDED EQUIPMENT OPERATION

- Write programming instructions to test the validity of operands intended to be used in mathematical operations.
- Avoid using operands of different data types in mathematical operations.

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

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: **SoHVAC** compiler cannot test 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.

⚠ WARNING

UNINTENDED EQUIPMENT OPERATION

- Write programming instructions to test the validity of operands intended to be used in mathematical operations.
- Avoid using operands of different data types in mathematical operations.

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

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 Floating Point Arithmetic Operators

SoHVAC 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

SoHVAC compiler supports type conversion to and from integer types.

SoHVAC compiler cannot test 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.

⚠ WARNING

UNINTENDED EQUIPMENT OPERATION

- Write programming instructions to test the validity of operands intended to be used in mathematical operations.
- Avoid using operands of different data types in mathematical operations.

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

4 Basic Functions

4.1 Conditional Statement

SoHVAC allows the use of the conditional statement as described in the ANSI-C standard.

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 Return

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 **SoHVAC**, each algorithm has one output. This means that the value of this output is passed by means of the `return` statement.

In **SoHVAC**, 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.

4.3 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);
else
function (b);
// function (a) is executed IF value a equal value b
// function (b) is executed IF value a is different from value b
if (a == c)
{
function (a);
a = c;
}
If value a equal value c, function (a) is executed then a takes c
value;
```

4.4 Switch

The `switch` statement is a multi-way decision that tests whether an expression matches one of a number of constant integer values.

If a case matches the expression value, execution starts at that case. The `break` statement causes an immediate exit from the switch.

The case labeled "default" is executed if none of the other cases correspond.

SoHVAC provides full support for `switch` statements, including ”fall-through”.

Example:

```
switch(i) {
  case 1: j++;
  case 2: j++; break;
default:j=0;
}
```

if i = 1, case 1 also the case 2 is executed.

if i = 2, only the case 2 is executed.

4.5 While Instruction

Using the `while` instruction, it is possible to define an iteration cycle until a given condition is verified as **TRUE**.

The `break` statement causes an immediate exit from the while.

Example:

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

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.

If the value `a` equal the value `c`, the `while` loop is exited.

4.6 Do … While Instruction

The `do...while` cycle is similar to `while`-type cycles. The expression is tested at the end. The code is executed at least once.

Example:

```
do
{
 function (a);
 a++;
}
while (a < b);
```

The two lines enclosed within curly brackets will be executed a first time, until the `a` variable becomes equal or greater to `b`; at this point, the execution will exit `while` function.

4.7 For Instruction

`for` – the logic "starting point; limit; increment step" is particularly suited to those cases where the number of cycles to be iterated can be determined at the outset.

Example:

```
for (i=1;i<k;i++)
{ ....}
```

Before performing the first iteration, the `i` variable is initialized to 1.

If it proves to be less than the `k` variable, the cycle is performed.

At the end of each iteration, `i` is incremented.

The function is performed until `i` is equal or greater than `k`.

4.8 Considerations When Using Loop-Type Instructions

The use of cycles within the **SoHVAC** development environment entails a variation of program execution times.

Repetitive instruction cycles slow down program execution times. If the exit condition from a cycle used within an algorithm never occurs, it creates an infinite-cycle situation, resulting in the repeated processing of the same operation (irresolvable). In this situation, the application would be stopped after a few hundreds of ms, an auto-reset event follows, and the controller restarts.

4.9 Goto, Break and Continue

The statements `goto`, `break` and `continue` are supported by **SoHVAC**.

5 Advanced Functions

5.1 The Comma Operator

SoHVAC supports the `comma` operator `a, b`. A pair of expressions separated by a comma is evaluated left to right, and the type and value of the result are the type and value of the right operand.

Example:

```
CJ_SHORT x;
CJ_SHORT y = 7;
x = (y = y-5, 24/y);
```

The result will be x=12.

5.2 Type Casts

SoHVAC has full support for `arithmetic type cast`.

Example:

```
CJ_SHORT x;
(CJ_BYTE)x
```

The calculated value is between 0 and 255.

5.3 Side Effects

5.3.1 Assignment Operator Side Effect

SoHVAC allows all `side effect` operators with their respective semantics.

This includes the assignment (=, +=, etc.), pre and post increment and decrement operators.

Example:

```
CJ_WORD x = 0, y = 5;
x = y++;
```

The result will be x=5 and y=6.

After the execution, the variable `x` will contain the initial value of `y`, and `y` will contain the initial value of `y` plus one.

5.3.2 Type Cast Side Effect

SoHVAC performs the `implicit type cast` as required by the ANSI-C standard.

Example:

```
CJ_BYTE c;
CJ_SHORT i = 257;
CJ_LONG l;
l = c = i;
Results: i = 257; c= 2; l = 2;
```

The value of i is converted to the type of the assignment expression c=i, that is, CJ_BYTE (or unsigned char) type. The value of this expression is then converted to the type of the outer assignment expression, that is, CJ_LONG (or long int) type.

5.4 Order of Evaluation

SoHVAC is compliant with the ANSI-C standard.

5.5 Arrays

SoHVAC 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 */
```

SoHVAC 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.

NOTE: Assignment must be made to the variable addressed by the index of the array. Assignment to the array name without an index can cause undesired overwriting of memory.

Example:

```
CJ_WORD a[10];
a = 0;
```

This will cause memory overwrites. The variable is a pointer. Changing the pointer address may lead to unwanted or unintended results.

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.

⚠ WARNING

UNINTENDED EQUIPMENT OPERATION

- Ensure 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.
- Do not attempt to assign a value to an array name without using an appropriate index into the array.

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

SoHVAC supports the definition of single dimensional arrays for some **SoHVAC** entities: variables, parameters and constants. Single dimensional arrays can be used outside of C algorithms by **SoHVAC** entities.

NOTE: 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.

5.6 Structures

SoHVAC 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 practical 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.

5.7 Unions

SoHVAC allows the use of unions to use the same storage for multiple data types.

5.8 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 a pointer to the array.

SoHVAC 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.
```

5.9 Static Variables

The use of `static` variables in the algorithms is allowed, but discouraged, 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.

6 Define

6.1 The DEFINE Instruction in SoHVAC

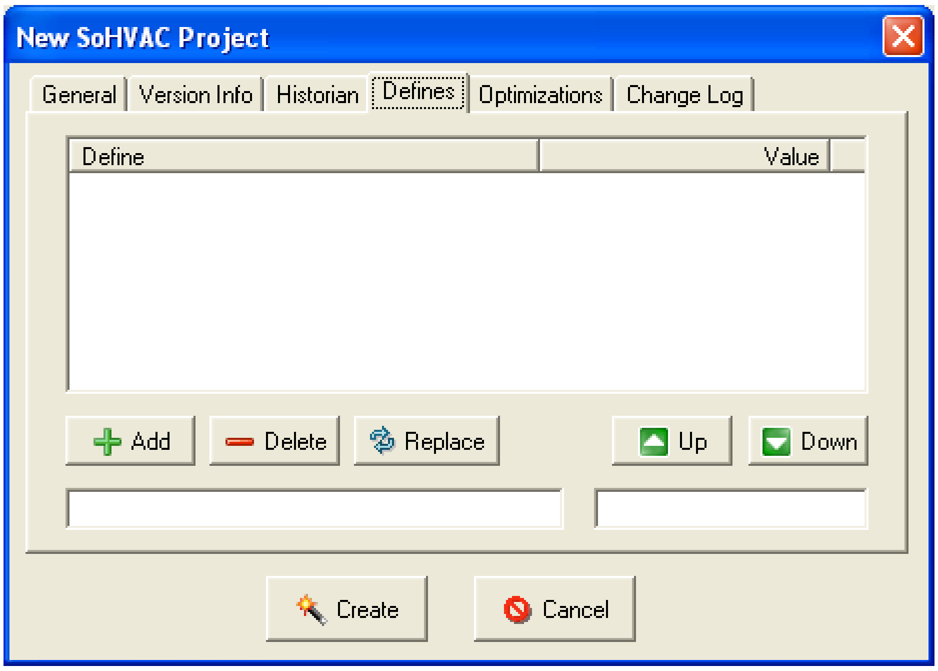
Within **SoHVAC**, the **DEFINE** instruction can be used in two modalities:

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

6.1.1 Project DEFINE

DEFINE allows you to define global constants which can be used by all C algorithms inside the application. You can create DEFINES within the **New SoHVAC Project** window under the **Defines** Tab.

In order to make constants global to all algorithms, the following window is used:

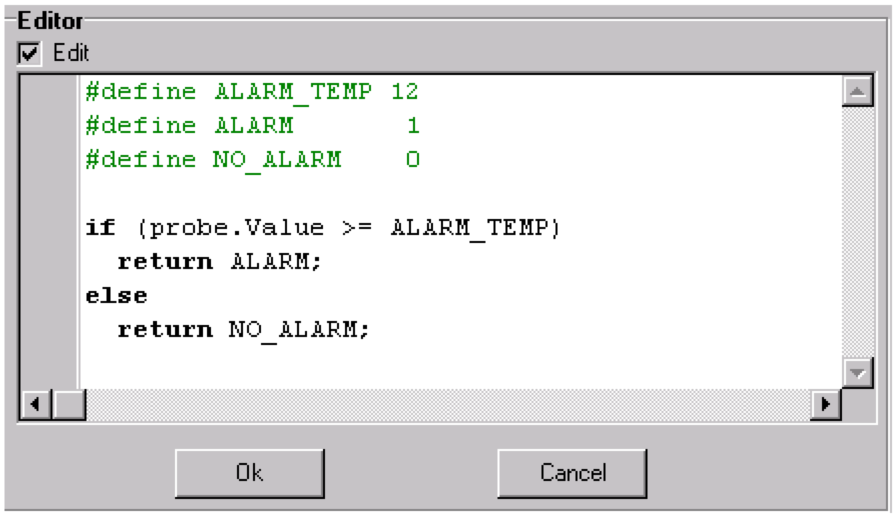


This way, the constants as defined above will be usable in every part of the program without any need for redefinition.

6.1.2 Algorithm DEFINE

If a constant is necessary only within a given algorithm, you can 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.



```
#define ALARM_TEMP 12
#define ALARM       1
#define NO_ALARM    0

if (probe.Value >= ALARM_TEMP)
  return ALARM;
else
  return NO_ALARM;
```

The above example demonstrates an implementation for sensor alarm handling. The same code can be used in multiple algorithms.

If you need to modify the DEFINE constant value, you need to modify the value in each of the algorithms in which you have defined it.

7 Limitation

7.1 Function Calls

SoHVAC only supports function calls of pre-defined functions (**CJ_GetWeekDay**(), **CJ_GetTime**(), **CJ_ReadVarExpo**(), etc.).

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

NOTE: It is possible to create global functions in the **SoHVAC** application program. These global functions can be used as function calls of pre-defined functions in the algorithm editor.

7.2 Assertions

SoHVAC does not support *assertions*. The `assert` statement will generate a compiler error message.

7.3 Pointers

The use of pointers is allowed in **SoHVAC**. 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

- Ensure 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.

7.4 Dynamic Memory

The use of dynamic memory allocation is allowed in **SoHVAC**. 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 code that prevents unwanted or unintended results from memory access operations.

⚠ WARNING

UNINTENDED EQUIPMENT OPERATION

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

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

8 Standard Library

The C standard library provides macros, type definitions, and functions for tasks like string handling, mathematical computations, input/output processing, memory allocation and several other operating system services.

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>
```

SoHVAC supports by default only the following headers.

```
<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>;
```

8.1 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);
```

8.2 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*);
```

8.3 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);
```

8.4 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);
```

APPENDICES

APPENDIX 1: Reserved Words of the C Program Language

The ANSI C standard recognizes 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
```

The following words are not part of ANSI-C standard, but are reserved by the **SoHVAC** compiler:

```
__far
__near
__interrupt
__EI
__DI
```

APPENDIX 2: 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_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_BIT CJ_DivByZero_Error_Read (void)

Returns 1 if there was a division by zero.

void CJ_DivByZero_Error_Reset (void)

Resets the division by zero indication.

void CJ_DivByZero_Error_Write (void)

Sets the division by zero indication.

CJ_BIT CJ_IsFirstMain (void)

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

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_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 **SoHVAC**, 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. The semaphore can be set only when it is not locked by another process. The owner of the semaphore can reset the semaphore.

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 the tenth semaphore as free; therefore, the resources are free to be used by other processes.

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_WORD CJ_SemaphoreRead(CJ_WORD i)

Returns the value of semaphore i.

CJ_WORD i: the semaphore number.

CJ_BYTE CJ_SemaphoreWrite(CJ_WORD i, CJ_WORD value)

Sets the value of semaphore i.

CJ_WORD i: the semaphore number. The semaphore i must be between 0 and 9.

CJ_WORD value: value to be stored in the semaphore.

Returns the status of the semaphore write operation:

4	Semaphore write unsuccessful.
5	Semaphore write successful.
6	Semaphore locked by another algorithm.

CJ_BYTE CJ_SemaphoreForce(CJ_WORD i, CJ_WORD value)

Forces the value of semaphore i. This function must be used only to recover from errors detected in your error handling routines.

The function returns the status of the semaphore force operation:

7	Semaphore force unsuccessful.
8	Semaphore force successful.

NOTE: The semaphore 0 and 1 are reserved for Modbus communication. These semaphores must not be used in the application.

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_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_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_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_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_GetSeconds(CJ_DATETIME dt)

This function returns the number of the seconds [0...59] 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_BYTE CJ_GetWeekDay(CJ_DATETIME dt)

This function returns the weekday contained in the parameter *dt* of **CJ_DATETIME** type.

9	Sunday
10	Monday
11	Tuesday
12	Wednesday
13	Thursday
14	Friday
15	Saturday

CJ_BYTE CJ_GetYear(CJ_DATETIME dt)

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

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 error, use the following **READ** functions.

void CJ_Math_Error_Reset (void)

Reset the global mathematical error flag.

CJ_BYTE CJ_LanguageRead (void)

Returns the selected language for the **EIML** pages:

16	English
17	Italian
18	French
19	Spanish
20	German
21	Russian
22	Portuguese
7…255	For future use

CJ_BYTE CJ_LanguageWrite (void)

Sets the selected language for the **EIML** pages:

23	English
24	Italian
25	French
26	Spanish
27	German
28	Russian
29	Portuguese
7…255	For future use

CJ_BYTE CJ_LCDBackLightRead (void)

Returns the status LCD backlight of the build in display:

30	Off
31	On
32	Time based

void CJ_LCDBackLightWrite (CJ_BYTE value)

Sets the LCD backlight function of the build in display:

33	Off
34	On
35	Time based

CJ_BYTE CJ_MaxInterruptTime (void)

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

CJ_WORD CJ_MaxMainTime (void)

Returns the maximum cycle time of the main application, 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_WORD CJ_MinMainTime (void)

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

CJ_BIT CJ_NaN_Error_Read (void)

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

void CJ_NaN_Error_Reset (void)

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

void CJ_NaN_Error_Write (void)

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

CJ_BIT CJ_Overflow_Error_Read (void)

Returns 1 if there was an overflow.

void CJ_Overflow_Error_Reset (void)

Resets the overflow indication.

CJ_SHORT CJ_ParKey_Error_Read (void)

Returns the parameter key status:

36	**OK**
37	**READ_ERROR**. It is not able to read the key.
38	**WRITE_ERROR**. An error is happened during the write.
39	**BAD DATA**. The data between key and controller are not compatible.

CJ_BIT CJ_PowerSupply_Error_Read (void)

Returns the power supply status:

40 OK

41 The power supply voltage is below 18 V or above 30 V.

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_SHORT CJ_RTC_Error_Read (void)

Returns the Real Time Clock status:

42 **CJ_RTC_OK**: Operating correctly.

43 **CJ_RTC_READ_ERROR**: A RTC access error has been detected.

44 **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.

CJ_WORD CJ_RunMainTime (void)

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

CJ_BIT CJ_Stack_Error_Read (void)

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

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.

void CJ_SetCondVisBit (CJ_WORD idx, CJ_BIT value)

This function controls the visibility of a variable on a display page.

The variable must be exported in the export table using the **Tools/Export Entities** menu of the programming environment.

All entities which have set the conditional visibility to hidden will be substituted with points…and are not editable.

Input parameters

Idx: (**CJ_WORD**) ID of the entity exported on Export table (**Tool_Export Entities**).

Value (**CJ_BIT**) Visibility:

0 Entity is shown.

1 Entity is hidden.

CJ_DATE StructToDate (CJ_DATE_STRUCT date)

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

CJ_DATETIME StructToDateTime (CJ_DATETIME_STRUCT rtc)

This function converts a data structure **CJ_DATETIME_STRUCT** to a **CJ_DATETIME** 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_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_BIT CJ_Underflow_Error_Read (void)

Returns 1 if there was an underflow.

void CJ_Underflow_Error_Reset (void)

Resets the underflow indication.

void CJ_Underflow_Error_Write (void)

Sets the underflow indication.

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_BIT CJ_5V_Ratio_Error_Read (void)

Returns the 5 V sensor power supply status:

45 OK

46 The power supply voltage is below 4 V or above 5.5 V.

CJ_BIT CJ_24V_Bus_Error_Read (void)

Returns the 24 V expansion bus power supply status:

47 OK

48 The power supply voltage is below 18 V or above 30 V.

CJ_BIT CJ_24V_Probe_Error_Read (void)

Returns the 24 V sensor power supply status:

49 OK

50 The power supply voltage is below 18 V or above 30 V.

The following functions are used for managing high speed counter digital inputs.

CJ_DWORD CJ_Read_DI_PulseCnt (CJ_BYTE in);

Returns the value of the digital input counter.
CJ_BYTE in: index of the digital inputs. The input in must be between 1 and 2.
NOTE: This function is only supported by the TM168●21●●, TM168●32●● and TM168●40●● controllers.
Refer to the Modicon M168 Controller Hardware Guide.

void CJ_Clear_DI_PulseCnt(CJ_BYTE in);

This function resets the value of the digital input counter.
CJ_BYTE in: index of the digital inputs. The input in must be between 1 and 2.
NOTE: This function is only supported by the TM168●21●●, TM168●32●● and TM168●40●● controllers.
Refer to the Modicon M168 Controller Hardware Guide.

APPENDIX 3: Common Considerations and Style Rules in C

- **Assignment (=) instead of comparison (==)**: The comparison **a == b** returns a **TRUE** only when the two variables have the same value, which is different from the assignment **a = b**. If you use an assignment in place of a comparison, the result will always be **TRUE** except in the case where **b = 0**.
- **Missing ( ) for a function**: When making function calls, use parentheses even if the function has no parameters.
- **Array Indices**: When arrays are initialized or used, you must be careful with the used indices (thus also with the number of elements), because, if an array is initialized with N elements, its index must have a range between 0 (the first element) and N-1 (the Nth 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, you need to be careful, especially when variables are being used.
- **The semicolon ";" closes every instruction**: Every instruction must end with a semi-colon.
- **Usage, function and position of comments**: Comments should 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, it is advisable to enclose within parenthesis both the expression and the values.

APPENDIX 4: Glossary of Terms

ANSI-C: Standard version of the C program language, 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 **SoHVAC** 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 **SoHVAC**, it represents the operating system 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.

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

APPENDIX 5: 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

Index