Energy XT Pro
Chiller Applications

SUMMARY

1 USE OF MANUAL

To facilitate use of the manual, customers may find the following useful:

Call-outs

Callout column:
Callouts on the topics described are placed to the left of the text to allow the user to find the desired information quickly.

Cross references

***Cross references*:**
All the words in *italics* are listed in the index with a reference to the page where they are described in more detail;
the text below serves as an example:
"activation of the alarm stops the compressors"
The italics indicate that under Compressors in the index there is a reference to the page where compressors are described in more detail.
If the online Help on the PC is used, the words in italics become proper hyperlinks (automatic links activated with a click of the mouse) that connect the different sections in the manual and allow you to navigate through the document.

Highlighted icons

Some parts of the text are highlighted in the callout column using icons that have the following meanings:

Note: draws attention to a specific topic that users should take into account.

Tip: highlights a suggestion that helps users to understand and use the information on the topic described.

Warning! : **highlights information that may damage the system or place persons, equipment, data, etc at risk if not known. These sections must always be read prior to use.**

2 INTRODUCTION TO APPMAKER

2.1 Potential and limitations of AppMaker

The aim of this chapter is to identify the *potential and limitations of AppMaker* in order to put it to optimum use in the development of air conditioning applications.

AppMaker is a SoftPLC environment complying with the IEC 61131-3 standard with the following characteristics:

- It has been on the market since the beginning of the 1990s.
- It is widely referenced.
- It is the only application with a professional version for distributed control for more than five years.
- It has development workbench that is among the most sophisticated currently available.
- It can easily be used for porting and customization (the target sources can be purchased from ICS Triplex).

2.1.1 Application development philosophy limits

The limits are mainly linked to the fact that in the PLC (and SoftPLC) world, codes are infrequently reused since applications are normally not parametric (an exception is the increasingly widespread approach of constructing, in vertical application sectors, super-configurators that can automatically generate applications that can then be reworked with their standard workbenches: for example, IEC 61131 automation application and relative SCADA HMI). As a result, the programming model adopted favours above all quick and easy development and not the reusability or parametrizability of what has been generated.
However, in the air conditioning sector, there is a need to produce units that combine a high level of programmability and a high level of parametrizability.

2.2 Use of IEC-61131-3 languages

AppMaker implements the entire range of languages supported by the IEC 61131-3 standard and also supports a flow-chart programming model. The different parts of an AppMaker application can be written in different languages and the language that is most suitable for the software module to be generated selected each time.

The languages available for development of the applications are the following:

Text languages
IL : infrequently used because it is very cryptic, resembles an "old" assembly language for 8 bit microprocessors.
ST: since it is very similar to languages like "C" or "Pascal" it can easily be used by anyone who has programming experience of these languages.
Graphical languages
LD : In addition to being the most widely used language among PLC programmers, it has the advantage by using the "Quick LD editor" of developing parts of codes that can be dynamically allocated. Its use therefore has a number of positive factors as well as a high computational speed.
FBD : this is used whenever graphical/visual understanding is important.
SFC : this is very powerful and allows traceability (it inserts break-points and allows you to see the exact state of the automa); it is therefore the best language to use especially when there is a relationship between the algorithm and one or more finished state automa;
FC : Since this is an extension that is only available on AppMaker and is not part of the standard IEC 1131, we have decided not to use this language.

2.3 Libraries and functional blocks

The many possibilities offered by AppMaker includes certain functional algorithms that are placed directly in the core of the controller but can however by used from the workbench. Normally the "functional blocks" that are included in the controller BIOS are those that require real time execution or those that are more suitably developed in "C" language. If in the first case, it is not generally speaking possible for the Energy XT-PRO developer to obtain the same behaviour of a functional block included in the BIOS with an IEC 1131 "program unit", in the second case, nothing prevents the IEC 1131software engineer from developing other "program units" in addition to that supplied as the basic library for the chiller application. However, it is important in order to maintain architectural correctness that the I/O interfaces of the new program unit developed (that in "C" language is normally called the "functional prototype") is identical to that in the library functional block.

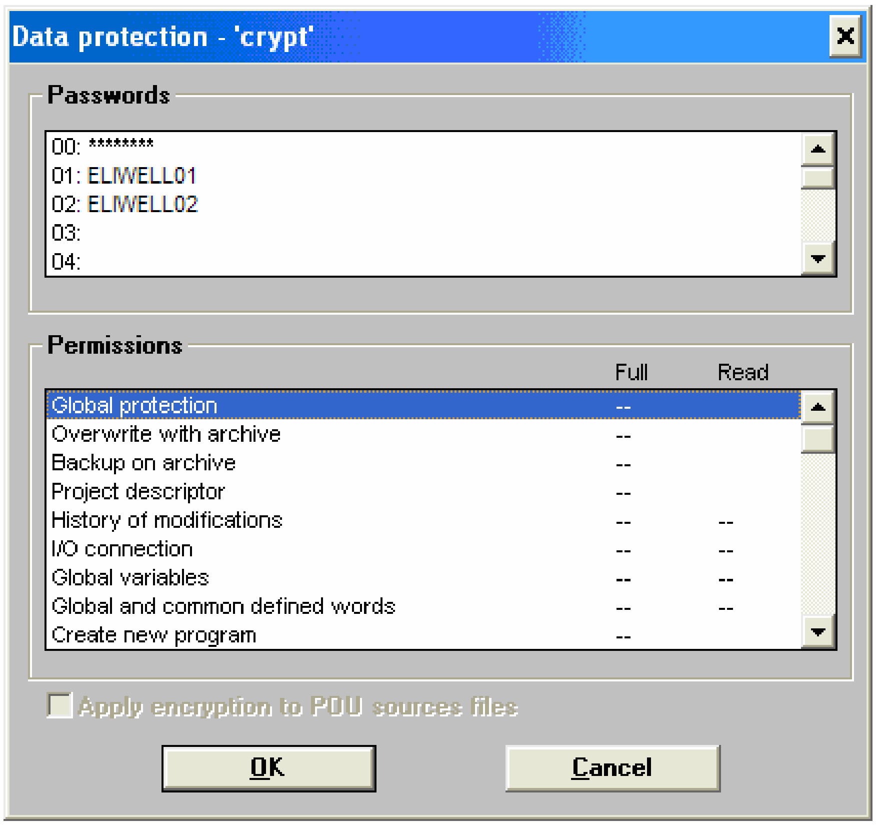
2.4 Possibility of parametrization of application

Although a "programmable application" approach has been used in the development of the chiller "baseline" application, our objective was however to create an application that could also be "parametrizable" to a certain extent.

The non-"structural" machine parameters can be modified or set from a user interface in order to obtain the desired application from the domain of the possible applications. However, since this domain, once it has been compiled and downloaded in the XT-PRO application that has been developed, is finished and limited, it is obvious that parameters cannot be set that exceed its physical limit. For this reason, the number of compressors cannot be set to a value that is higher than the number of compressors that have been compiled and wired in the application.
Another example that sets a limit to the parametrization of the application is linked to the "wiring" of the physical I/Os. With this type of baseline application, the physical wiring diagram cannot be changed. For this reason, the AppMaker I/O frame will use fixed wiring that can only be modified using the AppMaker workbench.

2.5 Applicability of AppMaker applications

AppMaker provides a series of access levels with "password" protection to enable controlled release of the application sources.
There are 16 (00..15) levels available. For each of the levels, a dedicated password can be defined. Access is hierarchical, i.e. the priority decreases going from level "n" to level "n+1" (i.e. the highest level is the level 00). For example, if I want to modify the password in level 03 I can gain access using the passwords for the levels 00..03.
The following figure illustrates the protection window once access has been gained to level 01.


Data protection - 'crypt'

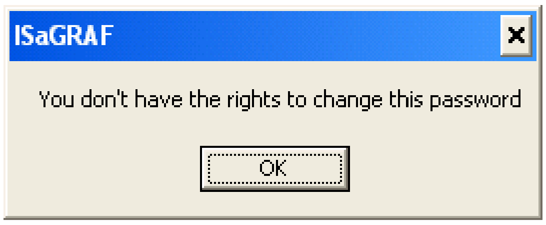
Passwords

00: ********
01: ELIWELL01
02: ELIWELL02
03:
04:

Permissions

	Full	Read
Global protection	--	
Overwrite with archive	--	
Backup on archive	--	
Project descriptor	--	
History of modifications	--	--
I/O connection	--	--
Global variables	--	--
Global and common defined words	--	--
Create new program	--	

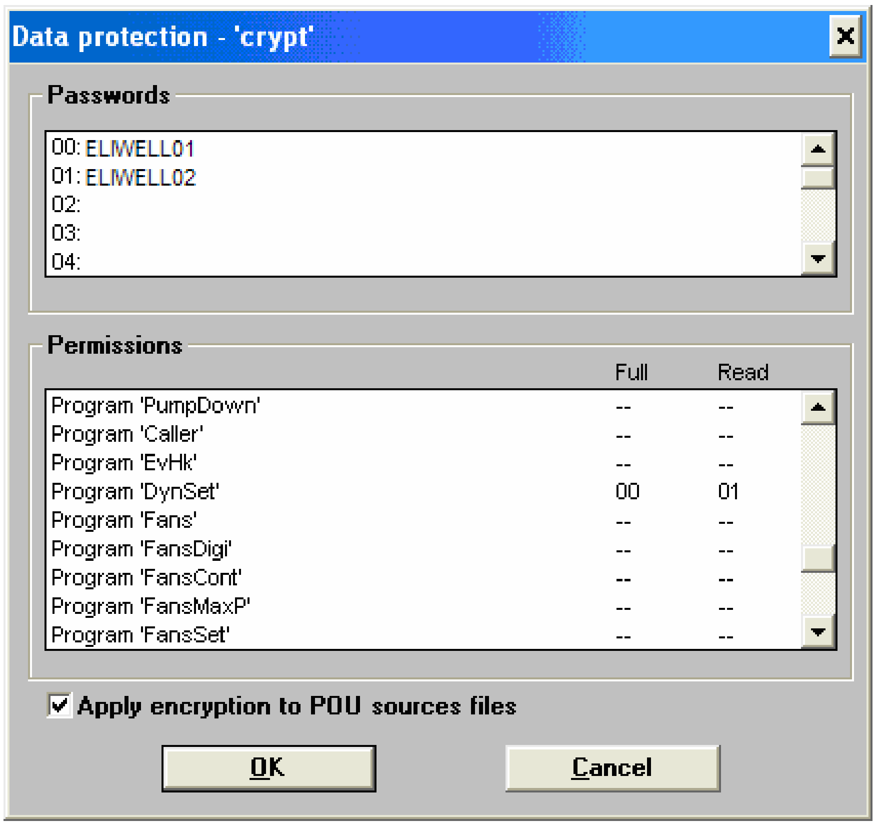
From the figure you can see that the highest level (00) is masked and if I attempt to modify it APPMAKER signals that we are not authorized.


ISaGRAF

You don't have the rights to change this password

The passwords for all the lower levels of the input level can be displayed and modified. The protection levels can then be associated with single files (programs) that constitute the application and can be associated with more general objects such as I/O connections, global variables, etc.
A "Full", i.e. read/write, and a "Read", i.e. read-only level can be associated with most of the objects[1], included in the developed programs.
In the following example, the *DynSet* program can be accessed in read only mode by supplying the level 01 password and in read-write mode, by supplying the level 00 password.

[1] For some types of object, for example, the possibility of creating (adding) programs to an application, only one type of access is possible; these objects are the ones for which a read-only type access is not necessary.



00: ELIWELL01
01: ELIWELL02
02:
03:
04:

	Full	Read
Program 'PumpDown'	--	--
Program 'Caller'	--	--
Program 'EvHk'	--	--
Program 'DynSet'	00	01
Program 'Fans'	--	--
Program 'FansDigi'	--	--
Program 'FansCont'	--	--
Program 'FansMaxP'	--	--
Program 'FansSet'	--	--

[✓] Apply encryption to POU sources files

Note that the AppMaker data protection window contains a flag for encryption.
If this flag is not selected, the files are protected from an access point of view by Workbench (or if the files have passwords, the password is necessary when you try to open the file) but are saved in their original format and can therefore still be read by any "external" editor (e.g. a text editor). As a result, the source would therefore be available and could be copied by a customer.
Beyond this context, some Program Units may become simple Function and Function Block call containers written in C and implemented in the BIOS. At present, the most important parts of the application are considered those that implement temperature control and the resource allocation logics (saturation, balancing, advanced policies).
Even without studying how to use the level hierarchy to provide different levels of visibility with only one source pack (by providing each customer with the correct level password), "cloned" applications can be generated very easily and quickly. In these applications, the level of visibility can be customized protecting the parts that for a specific customer must remain hidden (protected) and unprotecting those that must remain visible.

2.6 Use of Arrays

Use of the arrays provides functions that operate on data sets declared as one-dimensional vectors and identified by a different index for each instance or copy of Program Unit that uses the same function.
Let's consider, for example, a generic Program Unit that controls a compressor. To understand the great potential offered by the use of one-dimensional arrays, just imagine that if this data structure could not be used, you would need N versions of the same program with different names (therefore individually modifiable) and different global variables would be necessary for the different compressors. Obviously, this problem does not exist for the local variables in the Program Unit since they have the scope of the Program Unit in which they are declared.
When defining global variable vectors, on the other hand, each copy of the Program Unit will have one "index" variable with a different value assigned to it. In this way, all the code remains identical in the different copies of the Program Unit using global_variable[index] type notation.
Since neither **MenuMaker PRO** nor **TabMaker** can handle arrays, it is important to bear in mind that these data structures can only used for internal computation variables and cannot be used directly as I/O variables.
There are other situations in which care must be taken with the use of vectors because they cannot be used or improper use would jeopardize the correct working of the application.

A solution that resolves the limits connected with the use of vectors is that of resorting to I/O static variables for both inputs and outputs. This solution, (whose only disadvantage is a waste of memory caused by the fact that a quantity of I/O variables equal to the maximum size that the two I/O variables can assume must be allocated), enables MenuMaker PRO to access the I/Os and allows the MODBUS protocol to read specific I/Os with a fixed MODBUS address.

2.7 System optimization and project criteria

This paragraph is particularly important since it identifies the guide lines followed in the project and those to be followed in the development for optimization of the system.

The first decision concerns the languages to be used when they are not set by AppMaker as with SFC for some sequential parts.
The idea has been to favour the ST language since it allows greater code readability and enables the “C” programmer to understand the code and customize it quickly. In other point, we preferred the use of languages such as FBD and QLD, that compared with text languages, are easier to read for customers who are used to using graphical languages. What is more, the use of QLD with the “in-line” option is necessary when the functions must be nested with status variables although this language, if its use is generalized, leads to a significant increase in the sizes of the code.
The alternative is to use a text language (ST) by transforming the status variables into global variable vectors. In this way, the code size is reduced (to the detriment of execution speed) but the price to be paid is the loss of “interface pins” between the functions.
If absolutely necessary, I could have all the global variables in a single common area and no explicit interface between the modules.

In any case, there are certain restrictions that must be observed.

- the application must be “containable” in the available memory
- the execution time must be compatible with the dynamics of the system to be controlled
- the system must do what has been specified

When developing a new application, we recommend alternative solutions including, during the development stage, ones that save on the resource that from time to time is more critical, space or time.

The right approach to development is therefore as follows:

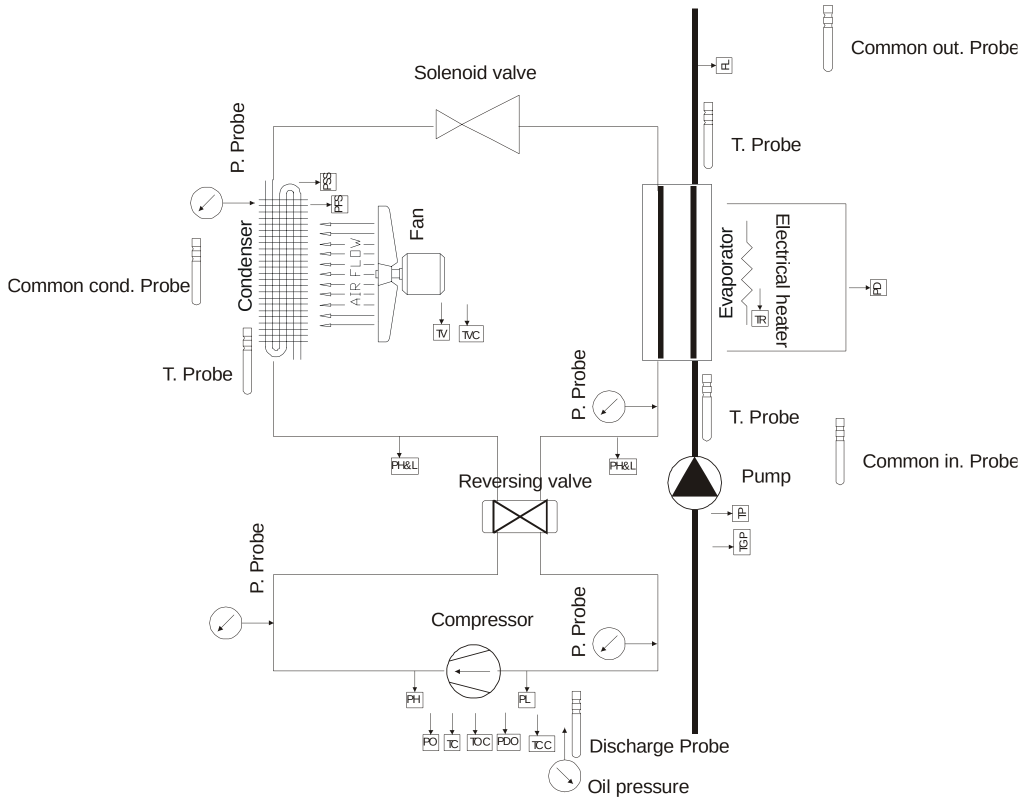
1. development with graphical languages when compatible with restrictions
2. development in text language as an alternative
3. Performance profiling with AppMaker simulation tools
4. optimization, if necessary, of space by “compacting” from FBD to ST
5. optimization, if necessary, of time by increasing the use of memory
6. interaction of steps 3, 4, 5.

3 GENERAL ARCHITECTURE OF CHILLER APPLICATION

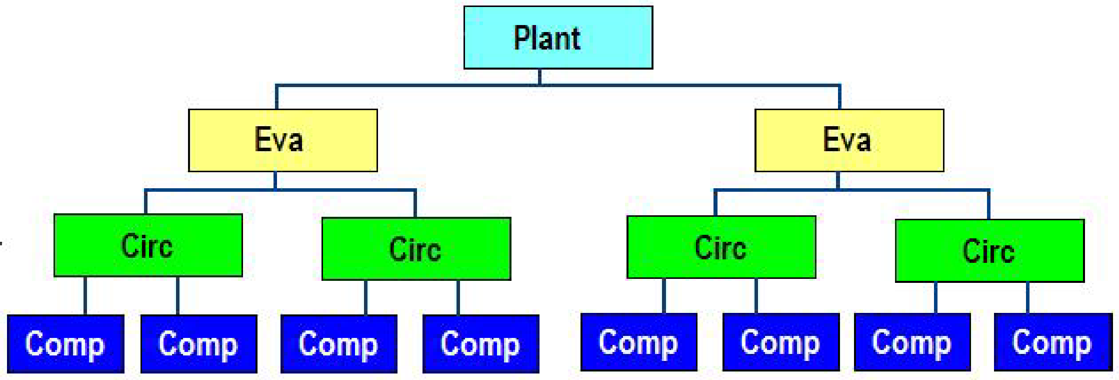
3.1 Introduction

The refrigerating machine is a system comprising a circuit in which refrigerant fluid circulates via a compressor; the compressed (and heated) fluid is sent to a condenser where it loses heat through an expansion device and then passes through an evaporator where heat is extracted.
In an air/water chiller, the evaporator cools down water whereas in an air/air chiller it cools down air.

Diagram of air/water chiller

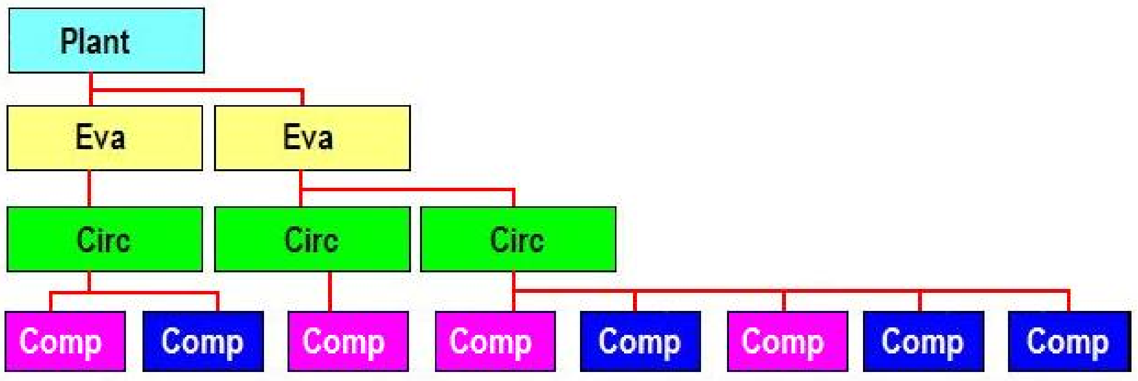


The Energy XT-PRO application is based on a hierarchical, modular structure that can be symmetrical, asymmetrical, uniform or non-uniform.



For example, with this structure each node on the same level must have the same number of child elements. Each evaporator must therefore control the same number of circuits and each circuit must control the same number of compressors.

The Energy XT-PRO overcomes these limitations by asking the developer to define the number of elements of each type. The links between elements can be subsequently handled according to a data driver schema.
This is an example of a structure obtained in this way:



Modularity is obtained by defining the number of child elements that belong to each parent element using parameters. evaporators per machine, circuits per evaporator and compressors per circuit. In addition, the same type of compressor is not necessary. In the diagram, different types of compressor are indicated with different colours.

This example can be extended to all elements that are part of the machine and as a result, evaporators can have a different number of circuits or fan groups can have a different number of fans.

The main advantages in using this type of structure are:

- the possibility of generating unbalanced machines: one circuit with two compressors and one with three;
- the possibility of generating non-uniform machines: one circuit with several different compressors;
- ease in handling asymmetrical elements in standard way: one fan stop for two condensers on two different circuits.

3.2 General architectural principles

For the sake of clarity, we will mention some general characteristics of an AppMaker application:

- Each application consists of the following *Sections*
  Begin
  Sequential
  End
  Function and Function Block local to application;
  note that the use of the sections is optional and the entire application can be inserted in the *Begin section*, if necessary.
- Each section consists of a number of *Program Units* of the desired size (compatible with the physical memory available).
- There are no restrictions or limitations concerning the choice of language used for development of the Program Unit except for limitations imposed by the AppMaker system.
- The *Function Blocks* are independent application units that can be invoked in different program units: they therefore encapsulate recurrent usage logics.
- The data is contained in a *Data Dictionary* that is separate from the program logic. The data has scope as in modern programming languages i.e. the contents of this data during debugging can be displayed and modified during execution of the program.

We will now describe the principles behind the software architecture described in this manual:

1. *Use of Sections*
   The sections are used following the following guidelines:

Section	Program Units contained	Languages used
Begin	This contains program units dedicated to the initialization of the application, pre-processing of the input signals and conversion from physical to logic signals, handling of logic alarms and calculation of temperature control algorithms.	Quick LD/FBD, ST
Sequential	This contains the program units that encapsulate the state logics of the compressors.	SFC for automata and ST, Quick LD/FBD for auxiliary logics
End	This contains additional program units for the control of condensation and fans and conversion of logic signals to physical ones.	Quick LD/FBD, ST
Function	This does not contain library functions but all the functions used for computation and basic data structures.	Quick LD/FBD, ST or C.
Function Block	This does not contain library FBs but generally all the functions used for computation and basic data structures.	Quick LD/FBD, ST or C.

2. *Modularity*
   Modularity regards several aspects:
   a. Program Unit: the organization of the code in program units should make it easier to locate a function in the application. As far as possible, a function will be placed in a specific program unit. Bear in mind

that "function" does not refer to an entire control function but a series of operations. For example, the core of the control logic of compressor 1 will be in a program unit in the *Sequential section* but the pre-processing of its input signals and processing of the alarm conditions will be in other program units in the *Begin section* in the same way as the post-processing of the logic to physical conversion will be in another program unit in the *End section*. This does not infringe the function-program unit mapping principle because there may be cases in which the application maintains the same control logic but modifies the implementation logic, etc.

   b. Function Blocks: these are typically basic functional units and are therefore very small. Normally, they are catalogued in libraries but there can be some application-specific FBs that will be allocated to the last section of the application.
   c. Complex data structures: This is represented by the control of "one-dimensional arrays". With special measures that are also used in the baseline application, two-dimensional structures can be controlled by means of simple one-dimensional arrays.

3. *Names of variables and program units*
   As far as possible, they must be highly 'significant'. If there are tool limits (for example, length of names of program units), these must be compensated for by using suitable comments.

4. *Data driven code*
   As far as possible, data driven codes will be used: i.e. there will be data structures that depending on their parameterization (default value for fixed parameters or value forced from keyboard/serial connection for dynamic parameters) can be used to modify important attributes in the application: for example, presence or not of compressor n, its association with circuit p, etc. Obviously, there will be objective limits in the AppMaker characteristics. Refer to Chapter 1.

5. *Code readability*
   This aspect also has several implications.
   a. Use of descriptive strings in the configuration of parameters: it is obviously useful that a parameter that indicates the configuration of a compressor is not 0 or 1 but is excluded or enabled in the same way as the variable that indicates its alarm status should be "normal" or "alarm" rather than 0 or 1.
   b. Extensive use of indentation: for the ST code.
   c. Extensive use of comments, notes and tags.
   d. Use of a clear graphic layout: for the FDB code.
   e. Use of states that immediately clarify the current situation of logics, including auxiliary states such as those in which a non-configured or dynamically excluded object is trapped: for the STC code.

The general principles described here have not only been complied with when developing the baseline chiller application but are also followed when making future modifications. This is not only to simplify implementation but also to prevent architectural mechanisms becoming disjointed by modifications that are not coherent with the basic philosophy.

3.3 General structure

The *general structure* of the application is described below. Note that the purpose of this chapter is only to provide architectural guidelines. You must therefore refer to Chapter 5 for specific information on the various program units shown in the following figures and on the structure used for the program units.
The criterion adopted is that of implementing the algorithm needed for control of the chiller in the various program units. Let's examine the functions in the various sections in greater detail:

- **begin** : Pre-processing of data and calculation of main algorithms

- the pre-processing of auxiliary logics to give the core logics an updated picture;
- control of transfer of input values from physical to logical;
- Handling of logic alarms;
- Calculation of availability;
- Calculation of temperature control;
- Calculation of control;

- **sequential** : compressor linked logics

- **end** : Special algorithms and conversions of values from logical to physical.

- Control of condensation;
- Control of *liquid injection*;
- Pump-Down;
- Logical-physical conversions of results of temperature control.

3.3.1 Begin Section

This section mainly uses purely combinatorial logic. It often involves program units without memory in which the data produced depends exclusively on the input data.
There are exceptions to this rule such as the alarm generation logics that will enable the function blocks containing the alarm development automa.

The principal program units in this section are:

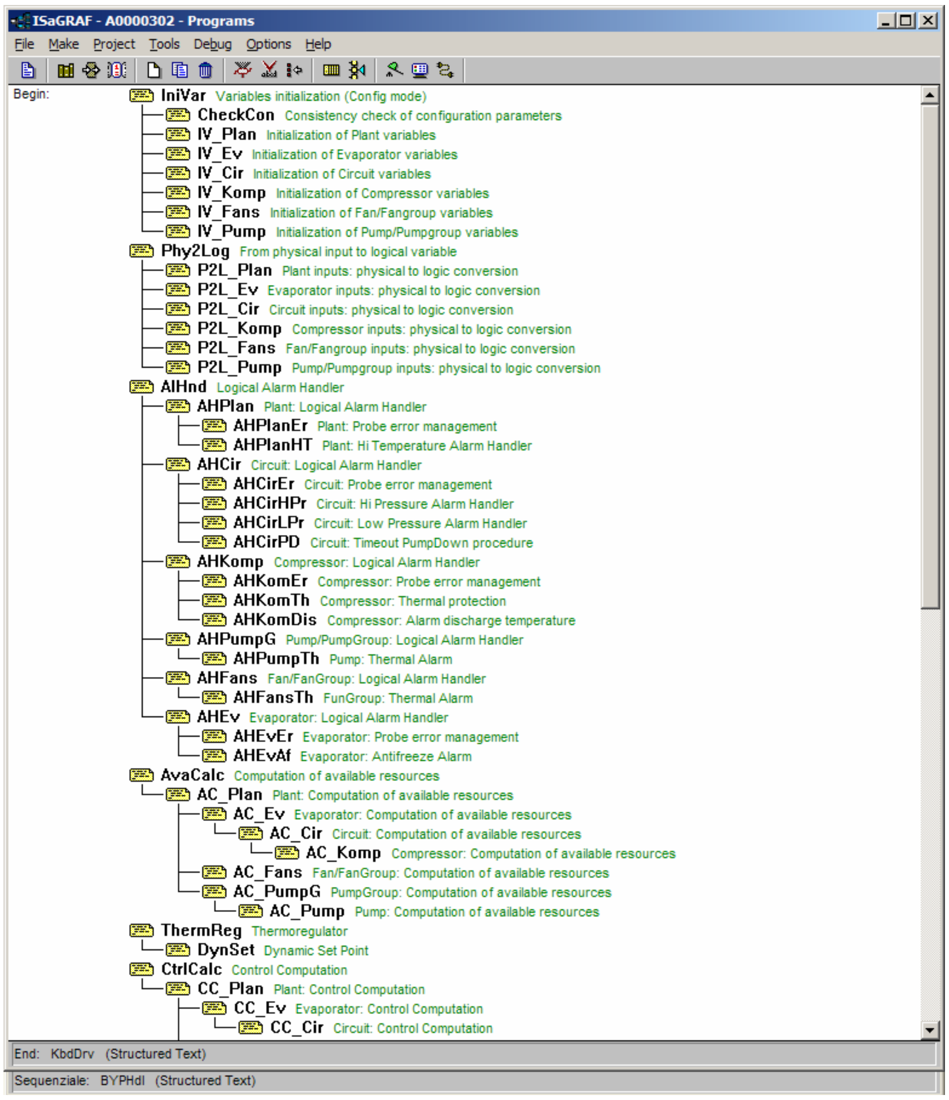
1. Validation of configuration
2. Initialization of variables
3. Converter of physical input into logical variables
4. Generator of logic alarms

5. Calculation of availability
6. Calculation of requested power (temperature control)
7. Calculation of control

By proceeding in top-down mode, each program unit can be broken down into sub-program units to obtain better modularization and encapsulation of the different functions.

The programming language will be FBD/QLD where maximum readability is necessary to make modification easy for a user who is used to these programming languages and ST in the parts where the compactness of the code needs to be optimized or where cycles that are difficult to control with graphical languages need to be implemented.

The diagram below illustrates how the section is structured[2].



ISaGRAF - A0000302 - Programs

```
Begin:  IniVar   Variables initialization (Config mode)
          ├─CheckCon  Consistency check of configuration parameters
          ├─IV_Plan   Initialization of Plant variables
          ├─IV_Ev     Initialization of Evaporator variables
          ├─IV_Cir    Initialization of Circuit variables
          ├─IV_Komp   Initialization of Compressor variables
          ├─IV_Fans   Initialization of Fan/Fangroup variables
          └─IV_Pump   Initialization of Pump/Pumpgroup variables
        Phy2Log  From physical input to logical variable
          ├─P2L_Plan  Plant inputs: physical to logic conversion
          ├─P2L_Ev    Evaporator inputs: physical to logic conversion
          ├─P2L_Cir   Circuit inputs: physical to logic conversion
          ├─P2L_Komp  Compressor inputs: physical to logic conversion
          ├─P2L_Fans  Fan/Fangroup inputs: physical to logic conversion
          └─P2L_Pump  Pump/Pumpgroup inputs: physical to logic conversion
        AlHnd  Logical Alarm Handler
          ├─AHPlan  Plant: Logical Alarm Handler
          │   ├─AHPlanEr  Plant: Probe error management
          │   └─AHPlanHT  Plant: Hi Temperature Alarm Handler
          ├─AHCir  Circuit: Logical Alarm Handler
          │   ├─AHCirEr   Circuit: Probe error management
          │   ├─AHCirHPr  Circuit: Hi Pressure Alarm Handler
          │   ├─AHCirLPr  Circuit: Low Pressure Alarm Handler
          │   └─AHCirPD   Circuit: Timeout PumpDown procedure
          ├─AHKomp  Compressor: Logical Alarm Handler
          │   ├─AHKomEr   Compressor: Probe error management
          │   ├─AHKomTh   Compressor: Thermal protection
          │   └─AHKomDis  Compressor: Alarm discharge temperature
          ├─AHPumpG  Pump/PumpGroup: Logical Alarm Handler
          │   └─AHPumpTh  Pump: Thermal Alarm
          ├─AHFans  Fan/FanGroup: Logical Alarm Handler
          │   └─AHFansTh  FunGroup: Thermal Alarm
          └─AHEv  Evaporator: Logical Alarm Handler
              ├─AHEvEr  Evaporator: Probe error management
              └─AHEvAf  Evaporator: Antifreeze Alarm
        AvaCalc  Computation of available resources
          └─AC_Plan  Plant: Computation of available resources
              ├─AC_Ev  Evaporator: Computation of available resources
              │   └─AC_Cir  Circuit: Computation of available resources
              │       └─AC_Komp  Compressor: Computation of available resources
              ├─AC_Fans  Fan/FanGroup: Computation of available resources
              └─AC_PumpG  PumpGroup: Computation of available resources
                  └─AC_Pump  Pump: Computation of available resources
        ThermReg  Thermoregulator
          └─DynSet  Dynamic Set Point
        CtrlCalc  Control Computation
          └─CC_Plan  Plant: Control Computation
              ├─CC_Ev  Evaporator: Control Computation
              │   └─CC_Cir  Circuit: Control Computation
End:  KbdDrv  (Structured Text)
Sequenziale:  BYPHdl  (Structured Text)
```

3.3.2 Sequential Section

The automa that describe the evolution of the compressor system from a dynamic point of view are implemented in this section.
The languages used must be SFC to reduce the text parts to a minimum and refer, where possible, the executive parts to *functions and function blocks* developed using graphical languages and ST, if necessary.

The program units implemented will be basically the compressor control automa (several copies, if necessary).

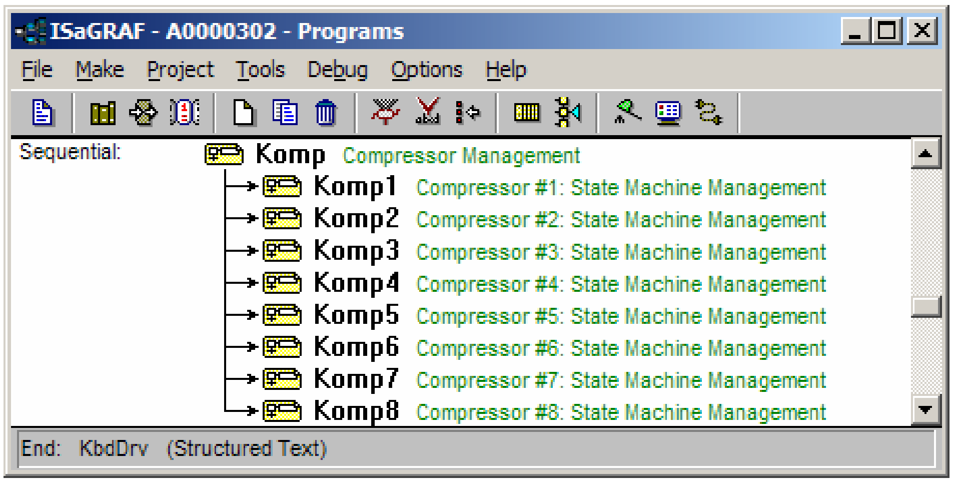
Please note that there is no explicit hierarchical relation (see paragraph 4 for more information on this subject) and therefore the implementation scheme will be:

The diagram uses the nomenclature defined in the next chapter.

1. Compressor 1 control automa
2. Compressor 2 control automa
3. …
4. Compressor n control automa

Generally speaking, the program units will contain compressor control automata: these are N copies of the same program that operate on a series of local variables and the i-nth set of global variables allocated in N size vectors. A compressor actuation logic is assigned to these automa that observes the shutdown time for the different operations and maintains the current availability information of the compressor.

The AppMaker project for the *Sequential section* would therefore be:


ISaGRAF - A0000302 - Programs

Sequential:
Komp Compressor Management
Komp1 Compressor #1: State Machine Management
Komp2 Compressor #2: State Machine Management
Komp3 Compressor #3: State Machine Management
Komp4 Compressor #4: State Machine Management
Komp5 Compressor #5: State Machine Management
Komp6 Compressor #6: State Machine Management
Komp7 Compressor #7: State Machine Management
Komp8 Compressor #8: State Machine Management

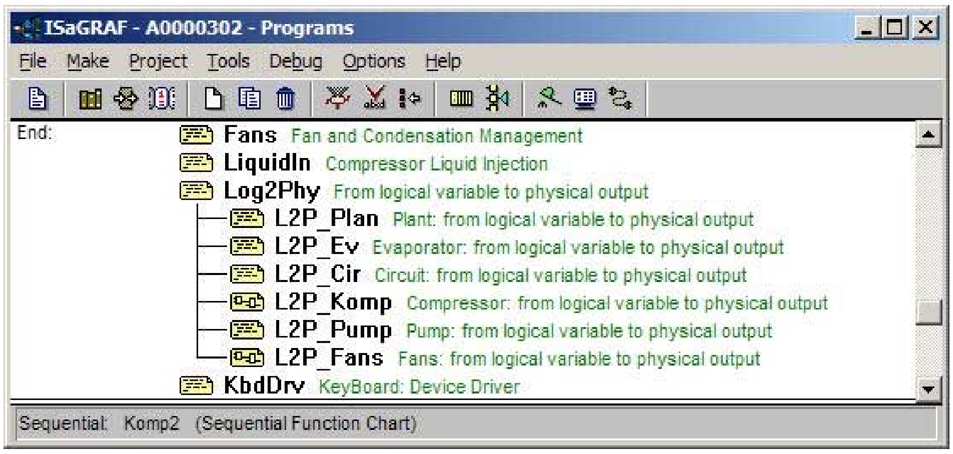
End: KbdDrv (Structured Text)

3.3.3 End Section

The principal program units in this section are:

- Control of condensation and fans;
- Control of *liquid injection* algorithm in compressors;
- Conversion of logic variables to physical output variables;
- Control of data exchange with keyboard.

By proceeding in top-down mode, each program unit can be broken down into program units of a lower level to obtain better modularization and encapsulation of the different functions.
The diagram below illustrates how the section is structured.

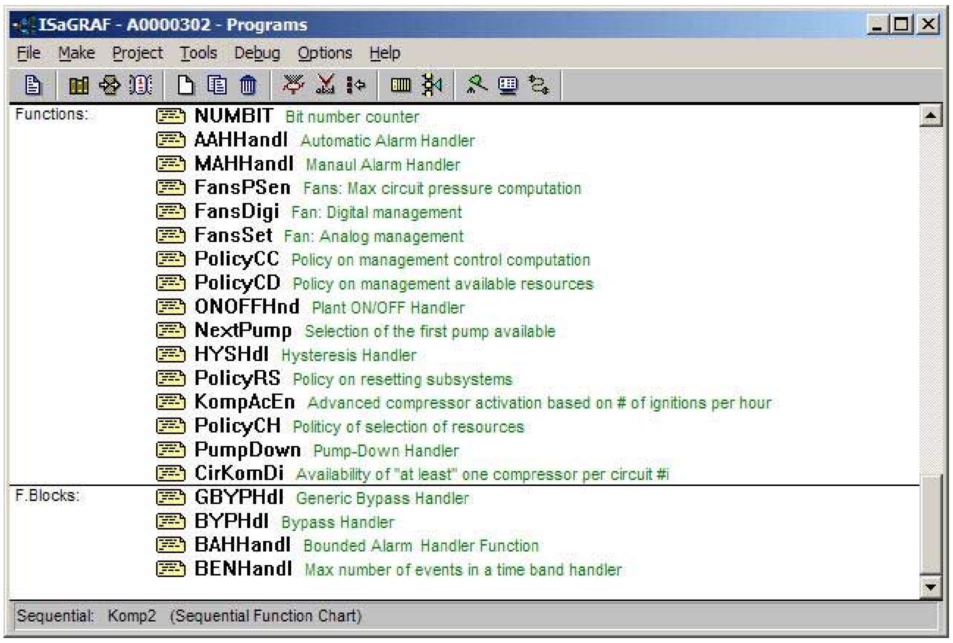

ISaGRAF - A0000302 - Programs

End:
Fans Fan and Condensation Management
LiquidIn Compressor Liquid Injection
Log2Phy From logical variable to physical output
L2P_Plan Plant: from logical variable to physical output
L2P_Ev Evaporator: from logical variable to physical output
L2P_Cir Circuit: from logical variable to physical output
L2P_Komp Compressor: from logical variable to physical output
L2P_Pump Pump: from logical variable to physical output
L2P_Fans Fans: from logical variable to physical output
KbdDrv KeyBoard: Device Driver

Sequential: Komp2 (Sequential Function Chart)

3.3.4 Functions and function blocks

Functions and function blocks for the calculation of temperature control algorithms are implemented in this section. They are part of the final section along with the hysteresis and bypass algorithms and all algorithms in common use that require customization (the bit counter, for example) as well as the “strategic” refrigerating resource selection algorithms that cannot be modified or displayed (the Program Unit PolicyCC, for example). If you require customized refrigerating resource selection algorithms, you must cancel the program unit in the baseline application and then generate a new program.


ISaGRAF - A0000302 - Programs
File Make Project Tools Debug Options Help

Functions:	NUMBIT	Bit number counter
	AAHHandl	Automatic Alarm Handler
	MAHHandl	Manaul Alarm Handler
	FansPSen	Fans: Max circuit pressure computation
	FansDigi	Fan: Digital management
	FansSet	Fan: Analog management
	PolicyCC	Policy on management control computation
	PolicyCD	Policy on management available resources
	ONOFFHnd	Plant ON/OFF Handler
	NextPump	Selection of the first pump available
	HYSHdl	Hysteresis Handler
	PolicyRS	Policy on resetting subsystems
	KompAcEn	Advanced compressor activation based on # of ignitions per hour
	PolicyCH	Politicy of selection of resources
	PumpDown	Pump-Down Handler
	CirKomDi	Availability of "at least" one compressor per circuit #i
F.Blocks:	GBYPHdl	Generic Bypass Handler
	BYPHdl	Bypass Handler
	BAHHandl	Bounded Alarm Handler Function
	BENHandl	Max number of events in a time band handler

Sequential: Komp2 (Sequential Function Chart)

3.3.5 Procedure for modification of baseline procedure

A general outline of the procedure that a developer can follow to obtain a new application based on the baseline application described in this document is given below. Although you can easily modify the application using the AppMaker workbench, we strongly recommend observing the following simple rules:

If the modification involves a change in the wiring diagram, in order to add an output signal, for example, the “I/O frame” must be modified. This modification will have little effect of the program logic since the I/O variables have logical names and therefore when they are moved to the frame, this does not involve changes in the logics. All you need to do is locate the program units and FBs affected by modifications once you have assessed any changes in the parametrization and status global data structures.
If there are changes in the global data structures, changes must be made to the application “dictionary” in compliance with the philosophy of the data structure that supports the baseline application.
If an FB has to be changed, it must be rewritten and its instances in the application must be replaced by those in the newly conceived FB.
If necessary, the program unit code involved in the modification must be changed or extended. In some cases, making changes may lead to side effects even in program units that are not directly involved in the modification: however, these cases are fairly infrequent because of the high modularization of the application and strong mapping between functions and software modules (program units).
A new TIC must now be generated (compilation)
The new TIC must be subjected to validation through initial simulation (validation performed on PC) and then debugging (validation performed on Target) to check that the changes made produce the desired results and no unwanted effects (regression testing). The regression testing will be significantly limited by the strong mapping between features and software modules (program units).
An updated version of the project documentation must be produced using the special feature in the AppMaker workbench.

Note that the conceived architecture may allow significant changes to the structure and application by simply reusing the program units (code) and modifying/extending the code and parametrization and status global data structure (dictionary). For a more in-depth understanding of this mechanism, read the next two chapters.

Note that many changes, even significant ones, to the structure and application (moving a compressor from one circuit to another, for example) may not require any real change to the software since they are obtained by simple re-parametrization of the baseline application.

4 CHILLER APPLICATION DATA DICTIONARY

This chapter describes the global data areas that can therefore be accessed by each program whereas the local data areas of each program will be described in chapter 4.
The global data plays a major role; it constitutes the software interface between the different programs and represents the communication and synchronization mechanisms between the different programs.

The data that the application uses is classified in several areas that can be divided into sub-areas.
It is hierarchically divided as below:

1. Parameters: this data defines how the application works at various levels. The application uses this data by accessing it in read only. By resorting to a classification already used, the parameters are divided into three categories:

   a. FIXED parameters (they will be referred to in the rest of the manual as Defined Words): read only values (non-user modifiable) that identify the maximum machine domain belong to this category such as:

      - maximum number of evaporators;
      - maximum number of circuits;
      - maximum number of compressors;

   b. COLD parameters: these are parameters that can be modified but generally imply the need to stop the plant and restart it. They belong to this category:

      - number of enabled compressors;
      - control algorithm (proportional, *PI*)
      - resource selection algorithm (saturation/balancing)

   c. HOT parameters: these are operating parameters that can be modified during operating conditions with no special measures, for example:

      - minimum time compressor is OFF
      - minimum time between switch-ons of two compressors
      - temperature set point

2. Variables: this is data that defines the status of the machine at various levels. The application produces and uses this data by accessing it in read and write mode. The data is classified by functional area or physical component and scope, i.e. differentiating between global variables of the application and local variables of the different programs and different copies of the same program. The different areas for the global variables will be indicated below whereas for the local variables, refer to the paragraphs on the different functional areas in chap. 4.

   2.1 plant variables
   2.2 condenser variables
   2.3 circuit variables
   2.4 compressor variables
   2.5 fan group variables
   2.6 pump group variables

4.1 Nomenclature

In order to interpret the identifiers for the different objects in the dictionary more easily, the following conventions are used:

1. names of objects: these are characterized by an acronym that identifies the physical area the object belongs to, the function that the object refers to and the type of object:

   1.1 KOMP: compressor area
   1.2 CIR: circuit area
   1.3 EV: evaporator area
   1.4 COND: condenser area
   1.5 FANS: fan/fan group area
   1.6 PUMP: pump/pump group area

   1.7 CH: Chiller
   1.8 HP: heat pump
   1.9 A: alarms
   1.10 DF: de-frost
   1.11 AF: anti-freeze
   1.12 PD: pump down
   1.13 DTSET: dynamic set point
   1.14 SOL: solenoid
   1.15 PRES: Pressure
   1.16 TEMP: temperature

   1.17 DI : digital input (a pressure switch, for example)
   1.18 SENS: analogue sensor (analogue input)

when applicable, the physical area the object belongs to must be used as a prefix before any other acronym (e.g. for the circuit pressure sensor, the full acronym must be CIR+PRES+SENS and not PRES+SENS+CIR or SENS+PRES-CIR)

2. syntax: identifies the type or scope of the object:

2.1	XXX_YYY:	I/O parameter or variable or defined word (constant)
2.2	XxxYyyy:	local or global internal variable; name of function of function block
2.3	xxx_yyy:	internal support variables, temporary indexes, etc.

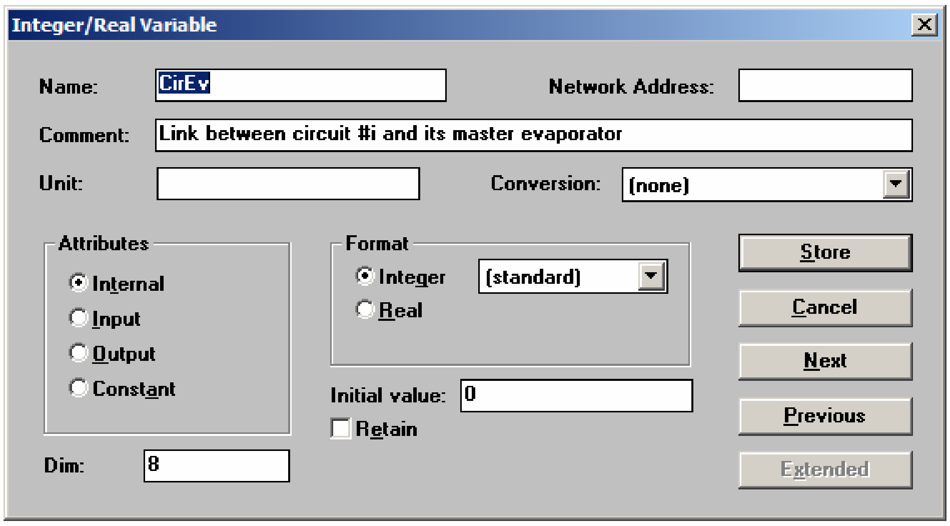
Note that what has previously been defined as “parameter” uses the syntax “XXX_YYY”, i.e. the identifier consists of only upper case characters and the names/acronyms are separated by the character “_” (underscore), whereas what has previously been defined as “variable” uses the syntax “XxxYyy, i.e. the identifier consists of upper case (first letter of the name/acronym) and lower case (other letters of name/acronym) and the names/acronyms are separated by a change from lower case to upper case.
Note that the I/O traceable variables and the constants also use the same syntax as the “parameters”, the former because they are to a certain extent considered external to the AppMaker application (input is read-only and output is write-only) and the latter because it is common practice to use upper case characters to define constants.

4.2 Definition of control structures

The AppMaker application must be developed from the very start with the maximum limits of machine expandability in mind. This means that control structures (control arrays) must be sized considering the “maximum of the maximum numbers”
Specifically, in our example the structural limits imposed are:

Maximum number of circuits on machine	8
Maximum number of evaporators on machine	4
Maximum number of compressors on machine	8
Maximum number of pumps on machine	2
Maximum number of fan groups on machine	8
Maximum number of fans on machine	16
Maximum number of compressors per circuit	8
Maximum number of circuits per evaporator	4
Maximum number of fans per fan group	8
Maximum number of circuits per fan group	8

The vectors must be sized not only on the basis of these elements but also the specific use of the vector; therefore, not all vectors that belong to the circuits, for example, have size 8.
A clear example of this is the fact that CirEv has size 8 since, for each circuit, it indicates the evaporator it belongs to.

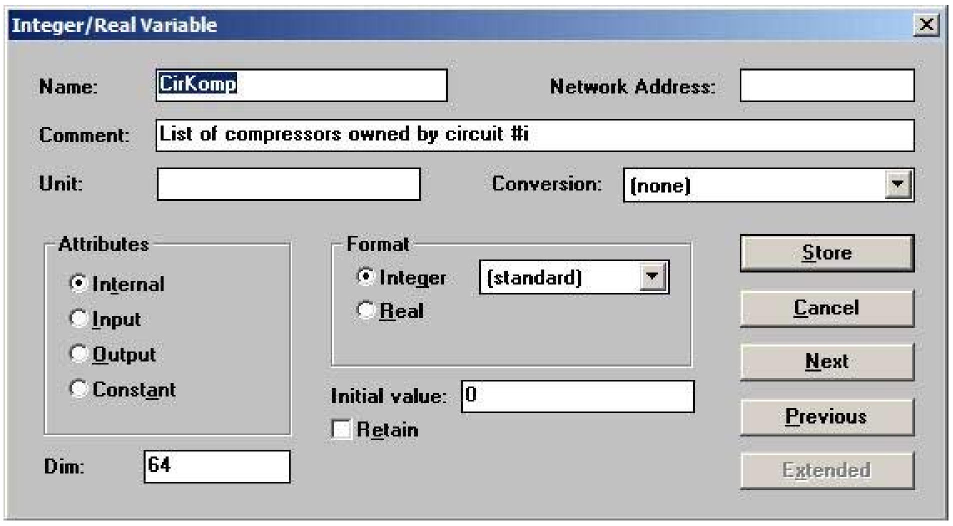

Integer/Real Variable

Name: CirEv
Network Address:
Comment: Link between circuit #i and its master evaporator
Unit:
Conversion: (none)

Attributes: [✓ Internal] Input Output Constant
Format: [✓ Integer] (standard) Real
Initial value: 0
Retain: ☐
Dim: 8

The *CirKomp vector*, on the other hand, has size 64.
This indicates for each compressor which circuit it belongs to. all the 8 compressors may belong to the first circuit or there may be only one compressor for each of the 8 circuits.
Therefore CirKomp must have a size that is equivalent to:

Maximum number of machine circuits* Maximum number of machine compressors.


Integer/Real Variable

Name: CirKomp

Network Address:

Comment: List of compressors owned by circuit #i

Unit:

Conversion: (none)

Attributes: [✓ Internal] Input Output Constant

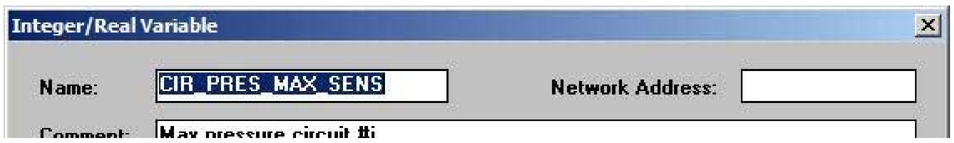
Format: [✓ Integer] (standard) Real

Initial value: 0

Retain: ☐

Dim: 64

Since in AppMaker the vectors cannot be sized using the constant identifier ("Defined words" in the AppMaker dictionary), as with a high level programming language, the vectors must be sized using a numeric value in the "Dim" field in the definition window of the parameter in the dictionary as in the following example:

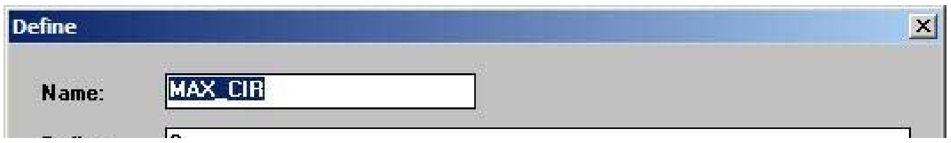

Integer/Real Variable

Name: CIR_PRES_MAX_SENS

Network Address:

Comment: Max pressure circuit #i

It is clear that even if we have a *nomenclature* that could be defined as "self-identifying" because of the acronyms used, the use of constants in the definition mask of the dictionary variables could cause ambiguity or difficulty in identifying the vectors since most of the maximum limits are usually 8 (e.g. circuits, compressors) or 4 (e.g. evaporators).
In this case, the only possibility is to clearly state in the comments field of the vector type variable ("Comment" field in AppMaker) what is being referred and how many elements (e.g. "i-nth circuit" in the previous example) must be specified for the "Dim" field, within the limit of 60 characters imposed by AppMaker for comments for variables.
As an aid when writing the ST code and in order to execute so-called "parametric" loops (loops limited by constant identifiers and not "magic numbers" inserted in the ST code), we intend to insert the definition of constants as "Defined words" in the AppMaker dictionary as in the following example:


Define

Name: MAX_CIR

All the "Defined words" used to define these maximum limits must have an identifier ("Name" field in AppMaker) a "MAX_XXX" string where "XXX" is the object whose maximum is about to be defined and a suitable comment.
Any loops in the AppMaker programs written in ST can be written as:

```
for ind_cir := 0 to (MAX_CIR – 1) do
(* control to be performed on each circuit *)
if <circuit vector>[ind_cir] = ... then
... (* actions to be performed on each circuit *)
end_if;
end_for;
```

in this way, the architecture is:

a) sized according to the maximum and capable of being used (without no modification) on down-sized machines
b) potentially ready to be expanded beyond the currently set limits. For example, to increase the number of controllable circuits, all you have to do is change the sizing of all the circuit vectors (CIR_XXX variables) and modify the relative "Defined word" value (the "Define" field in MAX_CIR must be set with the new maximum value for the number of circuits).

4.3 Global and support variables

This chapter and its sub-chapters list the *global and support variables* that are of major importance for the machine architecture being proposed.

For AppMaker, the subdivision between parameters and variables does not really exist since the database has variables (analogue and digital), timers, messages (strings), FB instances and constants ("Defined words").
Generally speaking, we can define the presence of two types of variables, global and support.
The former are "operating" variables that we could define as "truly global" in the sense that they are the ones that are used to monitor the behaviour and state of the plant.

They contain both configuration and setup information that make the AppMaker application parametrizable and data driven (e.g. KOMP_CIR_EV), and states, analogue or digital that are used (as input) by the algorithm to react correctly (generation of output) (e.g. CIR_PRES_MAX_SENS[]). In practice, these are the application and virtual and physical I/O parameters. The main parameters, i.e. those that have an impact on the architecture of the AppMaker application, are described in detail below.

The choice to insert all the parameters or not has been made bearing in mind that:

a) insertion of parameters is reasonably quick
d) there are a significant number of parameters (more than three hundred)
e) it is easier to follow how variables evolve in an application if there are less variables
f) AppMaker can be used to generate special “spy lists” that are both general and connected to individual programs

The latter (support variables) that the ones that, although they are also considered “operating” variables, are used to implement, facilitate and speed up the execution of algorithms.
The link tables constructed according to the configuration parameters during initialization and other variables that are not related to parameters belong to this category. From many points of view, both can be considered real “support variables”.
Some of them, like the look-up tables, are global since they are used by different algorithms even for completely unrelated purposes that require an external support (for example, for the pump down function of a circuit I need to know which compressors belong to this circuit and so I need a look-up table).
A detailed description of these variables is given in the following sub-chapters since they are general variables.
It is highly probable that many other variables will also have to be “global” meaning “non local” to a single application.
These variables, however, unlike those presented in these chapters on the “*Chiller application data dictionary*” should be considered as specific applications. Even if these variables will have to be used in several applications (and will therefore have to be “global”), there will always be a well-defined producer-consumer (or producer-consumers) relationship

The “local” variables of the individual applications are not considered relevant to an architecture document and will not be dealt with here.

4.3.1 KOMP_CIR_EV vector

(Part of *IV_Plan*)

KOMP_CIR_EV[8]: integer type 8 element vector; it associates by means of a decimal two figure identifier the compressor with the circuits and the circuits with the evaporators. Each element identifies the evaporator that the circuit belongs to with the first figure and the circuit that the compressor belongs to with the second. Only the first n elements can be initialized with n <= 8. The value 0 indicates that the corresponding compressor is not present. **This vector is manually configured**.
Only one analogue value associates the individual compressor to its own circuit (decimal value module 10) and the evaporator that the circuit belongs to (decimal value divided by 10).

Note that the values to be inserted are numbers 1..n and for the sake of clarity, the value 0 is avoided.
The vector is therefore the entry point that generates a different machine family that is also potentially non-symmetrical and unbalanced.

For example, for the BASELINE APPLICATION machine, the vector must be initialized as follows and care taken that there are no gaps and that numbering is in monotonically ascending order from left to right:

KOMP_CIR_EV for 2-4-4 machine (Baseline Application)

Number in vector	Compr.	Content	(Evaporator)	(Circuit)	Remarks
0	1	11	1	1	First compressor, compressor in first circuit of first evaporator
1	2	12	1	2	Second compressor, compressor in second circuit of first evaporator
2	3	21	2	1	Third compressor, compressor in first circuit of second evaporator
3	4	22	2	2	Fourth compressor, compressor in second circuit of second evaporator
4	5	0	-	-	0 = compressor not present
5	6	0	-	-	0 = compressor not present
6	7	0	-	-	0 = compressor not present
7	8	0	-	-	0 = compressor not present

A machine configured with 2 evaporators, 4 circuits and 8 compressors must have the vector KOMP_CIR_EV initialized in this way:

KOMP_CIR_EV for 2-4-8 machine (balanced symmetry)

KOMP_CIR_EV for 2-4-8 machine (balanced symmetry)					
Number in vector	**Compr.**	**Content**	**(Evaporator)**	**(Circuit)**	**Remarks**
0	1	11	1	1	First compressor, first compressor in first circuit of first evaporator
1	2	11	1	1	Second compressor, second compressor in first circuit of first evaporator
2	3	12	1	2	Third compressor, first compressor in second circuit of first evaporator
3	4	12	1	2	Fourth compressor, second compressor in second circuit of first evaporator
4	5	21	2	1	Fifth compressor, first compressor in first circuit of second evaporator
5	6	21	2	1	Sixth compressor, second compressor in first circuit of second evaporator
6	7	22	2	2	Seventh compressor, first compressor in second circuit of second evaporator
7	8	22	2	2	Eighth compressor, second compressor in second circuit of second evaporator

4.3.2 CirPresence Vector

(Part of *IV_Cir*)

`CirPresence[8]`: Boolean type 8 element vector: defines the presence of the i-nth circuit. Calculated by KOMP_CIR_EV[ ].

The *KOMP_CIR_EV vector* is scanned during initialization and for each element whose value is not zero its sets to true the circuit presence value with index

((<value> MOD 10) – 1) + offset

where the offset takes into account the number of circuits belonging to the previous evaporator.

Note that this vector will be used in mode "0..(n – 1)" in the applications. i.e. with index 0 used (index for the first circuit).

CirPresence				
Number in vector	**Circuit**	**Content (2-4-4) (BASELINE APPLICATION)**	**Content (2-4-8)**	**Content (2-8-8)**
0	1	true (present)	true (present)	true (present)
1	2	true (present)	true (present)	true (present)
2	3	true (present)	true (present)	true (present)
3	4	true (present)	true (present)	true (present)
4	5	false (not present)	false (not present)	true (present)
5	6	false (not present)	false (not present)	true (present)
6	7	false (not present)	false (not present)	true (present)
7	8	false (not present)	false (not present)	true (present)

4.3.3 CirEv vector

(Part of *IV_Cir*)
`CirEv[8]`: integer type 8 element vector, one per circuit; it lists the evaporator it belongs to for the i-nth circuit (with i = 0..7), calculated by KOMP_CIR_EV[ ].
It is to be used when, at circuit level, information on the evaporator that the circuit belongs to is necessary.

The *KOMP_CIR_EV vector* is scanned during initialization and for each element with a "k" index whose value is not zero it sets to "**`(<value> DIV 10) - 1`**" the CirEv element with index

((<value> MOD 10) – 1) + offset

where the offset takes into account the number of circuits belonging to the previous evaporator.

Note that this vector will be used in mode "0..(n – 1)" in the applications. i.e. with index 0 used (index for the evaporator on the first circuit).

CirEv				
Number in vector	**Circuit**	**Evaporator (2-4-4) (BASELINE APPLICATION)**	**Evaporator (2-4-8)**	**Evaporator (2-8-8)**
0	1	0	0	0
1	2	0	0	0

CirEv				
Number in vector	Circuit	Evaporator (2-4-4) (BASELINE APPLICATION)	Evaporator (2-4-8)	Evaporator (2-8-8)
2	3	1	1	0
3	4	1	1	0
4	5	-1 (not used)	-1 (not used)	1
5	6	-1 (not used)	-1 (not used)	1
6	7	-1 (not used)	-1 (not used)	1
7	8	-1 (not used)	-1 (not used)	1

4.3.4 CirKomp vector

(Part of *IV_Cir*)

CirKomp[8*8]: integer type 8*8 element vector for each circuit; it lists the circuits that are part of the i-nth circuit (with i = 0..7). Calculated by KOMP_CIR_EV[].
The AppMaker support for only the one-dimensional arrays implies the use of a single vector instead of a two-size matrix (this would be the most logical solution for this variable).

Since each circuit can have a maximum of 8 compressors and we can have a maximum of 8 circuits, this vector is sized for 64 elements even if in this case the overall limit of 8 compressors considers the fact that the vector is partially filled (many elements, the unused ones, will contain “-1”).

For the “i-nth” circuit (with 0 <= i <= (MAX_CIR – 1)), the relevant starting index will be given by the expression:

i * (MAX_KOMP4CIR) (it represents an offset in the vector)

where “MAX_KOMP4CIR” is the constant that defines the maximum number of compressors per circuit; the valid elements for this circuit will therefore be MAX_KOMP4CIR.
The *KOMP_CIR_EV vector* is scanned during initialization and for each element with a “k” index whose value is not zero it sets to “k” the CirKomp element with index

(((<value> MOD 10) – 1) + offset) * MAX_KOMP4CIR

where the offset takes into account the number of circuits belonging to the previous evaporator.

Note that this vector will be used in mode “0..(n – 1)” in the applications. i.e. with index 0 used (index for the first compressor of the first circuit).

CirKomp					
Number in vector	Circuit	Content (2-4-4) (BASELINE APPLICATION)	Content (2-4-8)	Content (2-8-8)	Content (1-1-4)
0	1	0	0	0	0
1	1	-1 (not used)	1	-1 (not used)	1
2	1	-1 (not used)	-1 (not used)	-1 (not used)	2
3	1	-1 (not used)	-1 (not used)	-1 (not used)	3
4	1	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
5	1	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
6	1	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
7	1	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
8	2	1	2	1	-1 (not used)
9	2	-1 (not used)	3	-1 (not used)	-1 (not used)
10	2	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
11	2	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
12	2	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
13	2	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
14	2	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
15	2	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
16	3	2	4	2	-1 (not used)
17	3	-1 (not used)	5	-1 (not used)	-1 (not used)
18	3	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
19	3	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
20	3	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
21	3	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
22	3	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
23	3	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
24	4	3	6	3	-1 (not used)
25	4	-1 (not used)	7	-1 (not used)	-1 (not used)
26	4	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
27	4	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
28	4	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
29	4	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
30	4	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)

CirKomp					
Number in vector	**Circuit**	**Content (2-4-4) (BASELINE APPLICATION)**	**Content (2-4-8)**	**Content (2-8-8)**	**Content (1-1-4)**
31	4	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
32	5	-1 (not used)	-1 (not used)	4	-1 (not used)
33	5	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
34	5	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
35	5	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
36	5	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
37	5	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
38	5	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
39	5	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
40	6	-1 (not used)	-1 (not used)	5	-1 (not used)
41	6	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
42	6	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
43	6	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
44	6	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
45	6	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
46	6	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
47	6	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
48	7	-1 (not used)	-1 (not used)	6	-1 (not used)
49	7	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
50	7	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
51	7	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
52	7	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
53	7	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
54	7	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
55	7	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
56	8	-1 (not used)	-1 (not used)	7	-1 (not used)
57	8	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
58	8	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
59	8	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
60	8	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
61	8	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
62	8	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
63	8	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)

It should be noted that although this type of vector uses more variables (and therefore more memory) it offers significant advantages in the execution stage.
For example, in this case, even an integer type 8 element vector (one per compressor, each element would be the circuit that the compressor belongs to) would be sufficient but this vector would always have to be scanned from the first to the last element each time anything is done to the compressors that belong to the same circuit.
With this vector, on the other hand, a simple multiplication and cycle that can be interrupted at the first element met equal to "-1" are sufficient to reduce significantly the processing time necessary for run-time as will be shown in the next chapters.

4.3.5 EvPresence vector

(Part of *IV_Ev*ap)

EvPresence[4 (MAX_EV)]: Boolean type 4 element vector; defines the presence of the i-nth evaporator.
Calculated by KOMP_CIR_EV[ ].
The *KOMP_CIR_EV vector* is scanned during initialization and for each element whose value is not zero its sets to true the evaporator presence value with index

```
((<value> DIV 10) – 1)
```

Note that this vector will be used in mode "0..(n – 1)" in the applications. i.e. with the index 0 used (index for the first evaporator).

EvPresence				
Number in vector	**Evaporator**	**Content (2-4-4) (BASELINE APPLICATION)**	**Content (2-4-8)**	**Content (4-8-8)**
0	1	true (present)	true (present)	true (present)
1	2	true (present)	true (present)	true (present)
2	3	false (not present)	false (not present)	true (present)
3	4	false (not present)	false (not present)	true (present)

4.3.6 EvCir vector

EvCir[4*4]: Integer type 4*4 element vector; lists the circuits that are part of the i-nth evaporator (with i = 0..3). Calculated by KOMP_CIR_EV[ ];

Here too, the AppMaker support for only the one-dimensional arrays implies the use of a single vector instead of a two-size matrix (this would be the most logical solution for this variable).
Since each evaporator can have a maximum of 4 circuits and we can have a maximum of 4 evaporators, this vector is sized for 16 elements and also in this case the vector is partially filled (many elements, the unused ones, will contain "-1").
For the "i-nth" evaporator (with 0 <= i <= (MAX_EV – 1)), the relevant starting index will be given by the expression:

```
i * (MAX_CIR4EV)
```

where "MAX_ CIR4EV" is the constant that defines the maximum number of circuits per evaporator; the valid elements for this circuit will therefore be MAX_CIR4EV.
The *KOMP_CIR_EV vector* is scanned during initialization and for each element with a "k" index whose value is not zero it sets to "(<value> MOD 10) – 1 + offset" the EV_CIR element with index

```
(((<value> DIV 10) – 1)*MAX_CIR4EV ) + (<value> MOD 10) – 1)
```

where the offset takes into account the number of circuits belonging to the previous evaporator.

Note that this vector will be used in mode "0..(n – 1)" in the applications. i.e. with the index 0 used (index for the first evaporator on the circuit).

EvCir					
Number in vector	**Evap.**	**Content (2-4-4) (BASELINE APPLICATION)**	**Content (2-4-8)**	**Content (2-8-8)**	**Content (4-4-4)**
0	1	0	0	0	0
1	1	1	1	1	-1 (not used)
2	1	-1 (not used)	-1 (not used)	2	-1 (not used)
3	1	-1 (not used)	-1 (not used)	3	-1 (not used)
4	2	2	2	4	1
5	2	3	3	5	-1 (not used)
6	2	-1 (not used)	-1 (not used)	6	-1 (not used)
7	2	-1 (not used)	-1 (not used)	7	-1 (not used)
8	3	-1 (not used)	-1 (not used)	-1 (not used)	2
9	3	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
10	3	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
11	3	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
12	4	-1 (not used)	-1 (not used)	-1 (not used)	3
13	4	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
14	4	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)
15	4	-1 (not used)	-1 (not used)	-1 (not used)	-1 (not used)

Even in this case, it should be noted that although this type of vector uses more variables (and therefore more memory) it offers significant advantages in the execution stage.
For example, in this case, even an integer type 8 element vector (one per circuit, each element would be the evaporator that the circuit belongs to) would be sufficient but this vector would always have to be scanned from the first to the last element each time anything is done to the circuits that belong to the same evaporator.
With this vector, on the other hand, a simple multiplication and cycle that can be interrupted at the first element met equal to "-1" are sufficient to reduce significantly the processing time necessary.

4.3.7 KOMP_STEP vector

(Part of *IV_Komp*)

KOMP_STEP[8]: number of capacity steps per compressor. Range allowed 1..4.

KOMP_STEP					
Number in vector	**Compr**	**Content (2-4-4) (BASELINE APPLICATION)**	**Remarks (2-4-4)**	**Content (2-4-8)**	**Remarks (2-4-8)**
0	1	3	On/off plus two capacity steps (0-33-66-100%)	1	On/off compressors for Ist circuit (0-100%)
1	2	3	On/off plus two capacity steps (0-33-66-100%)	1	On/off compressors for Ist circuit (0-100%)
2	3	3	On/off plus two capacity steps (0-33-66-100%)	2	On/off compressors plus one capacity step for 2nd circuit (0-50-100%)
3	4	3	On/off plus two capacity steps (0-33-66-100%)	2	On/off compressors plus one capacity step for 2nd circuit (0-50-100%)

KOMP_STEP					
Numbe r in vector	Com pr	Content (2-4-4) (BASELINE APPLICATION)	Remarks (2-4-4)	Content (2-4-8)	Remarks (2-4-8)
4	5	0	(not used	4	On/off compressors plus three capacity steps for 3rd circuit (0-25-50-75-100%)
5	6	0	(not used	4	On/off compressors plus three capacity steps for 3rd circuit (0-25-50-75-100%)
6	7	0	(not used	4	On/off compressors plus three capacity steps for 4th circuit (0-25-50-75-100%)
7	8	0	(not used	4	On/off compressors plus three capacity steps for 4th circuit (0-25-50-75-100%)

4.3.8 KompCir vector

(Part of *IV_Komp*)

KompCir[8]: integer type 8 element vector, one per compressor; it lists the circuit it belongs to for the i-nth compressor (with i = 0..7). Calculated by KOMP_CIR_EV[].
It is to be used when, at compressor level, information on the circuit that the compressor belongs to is necessary.
The *KOMP_CIR_EV vector* is scanned during initialization and for each element with a “k” index whose value is not zero it sets to

```
(<value> MOD 10)
```

where the offset takes into account the number of circuits belonging to the previous evaporator.

the KompCir element with “k” index.

Note that this vector will be used in mode “0..(n – 1)” in the applications. i.e. with index 0 used (index to determine the circuit of the first compressor).

KompCir				
Number in vector	Compr.	Circuit (2-4-4) (BASELINE APPLICATION)	Circuit (2-4-8)	Circuit (2-8-8)
0	1	0	0	0
1	2	1	0	1
2	3	2	1	2
3	4	3	1	3
4	5	-1 (not used)	2	4
5	6	-1 (not used)	2	5
6	7	-1 (not used)	3	6
7	8	-1 (not used)	3	7

4.3.9 KompFans (Part of IV_Komp)

KompFans[8]: integer type 8 element vector, one per compressor; it lists the fan group it belongs to for the i-nth compressor (with i = 0..7). Calculated by CIR_FANS[] and KompCir[].
It is to be used when, at compressor level, information on the fan group that the compressor belongs to is necessary.

4.3.10 EvNo, CirNo, KompNo variables

(Part of *IV_Komp*)

These variables are not considered mandatory since the look-up tables defined in the previous chapters (with the “-1” “plug” values) are (and must be) sufficient to guarantee correct access to all the vectorized objects.
The decision to implement the “XxxNo” parameters/variables depends, above all, on the type of consistency checks to be implemented before being run “one shot” during the initialization stage.
These parameters that define:

a) the total number of evaporators that must be controlled
(EvNo, with 0 < EvNo <= MAX_EV)
b) the total number (overall, not per evaporator) of circuits that must be controlled
(CirNo, with 0 < CirNo <= MAX_CIR)
c) the total number (overall, not per circuit) of compressors that must be controlled
(KompNo, with 0 < KompNo <= MAX_KOMP)

can however also be calculated by KOMP_CIR_EV[].
The *KOMP_CIR_EV vector* is scanned during initialization and for each element with a "k" index whose value is not zero:

a) increases the number of compressors (KompNo)
b) increases the number of evaporators (EvNo) if and only if the evaporator
   "(<value> DIV 10) - 1"
   has not already been considered
c) increases the number of circuits (CirNo) if and only if the circuit
   "(<value> MOD 10) - 1"
   has not already been considered

EvNo CirNo KompNo			
Var	**Value for (2-4-4) (BASELINE APPLICATION)**	**Value for (2-4-8)**	**Value for (2-8-8)**
EvNo	2	2	2
CirNo	4	4	8
KompNo	4	8	8

These variables can become local to the initialization phase (therefore not visible to the rest of the applications) depending on the selections made for the consistency checks.

4.3.11 Consistency checks and cycles on objects

Preliminary note
Normally PLC applications do not contain consistency checks on the content of the parameters that the algorithms operate on. It is to be assumed that it is the MMI part that performs the checks on the configuration input provided by an operator and refuses illegal parameters. Assuming that this is also true for applications developed by the final user, we supply guidelines that are intended to highlight any critical points that could have a negative effect on machine behaviour. The consistency checks described in this chapter (and in others) can therefore be considered as requirements for the control software of the operator interface (MenuMaker PRO).

KOMP_CIR_EV initialization requirements

- The array must be filled without leaving gaps;
- It must always be filled from evaporator 1;
- The evaporator index must be in monotonically ascending order (max. increase of 1);
- For each evaporator the circuit index must be in monotonically ascending order (max. increase of 1);

This is an example of an incorrect configuration (no evaporator 2).

KOMP_CIR_EV for 2-4-4 machine (configuration error)					
Number in vector	**Compr.**	**Content**	**(Evaporator)**	**(Circuit)**	**Remarks**
0	1	11	1	1	First compressor, compressor in first circuit of first evaporator
1	2	12	1	2	Second compressor, compressor in second circuit of first evaporator
2	3	31	3	1	Third compressor, compressor in first circuit of third evaporator
3	4	32	3	2	Fourth compressor, compressor in second circuit of third evaporator
4	5	0	-	-	0 = compressor not present
5	6	0	-	-	0 = compressor not present
6	7	0	-	-	0 = compressor not present
7	8	0	-	-	0 = compressor not present

Note that the fact that the previous configuration is considered incorrect is
debatable. The decision whether to make this type of configuration (defined as "spread") possible must be taken at the start of development since it has an impact on:

a) the presence of the EvNo, CirNo and KompNo variables
b) their use in the scanning cycles of the objects
b) the overall performance of the machine

CIR_FANS initialization requirements

- The array must be filled without leaving gaps;
- The fan group index must be in monotonically ascending order (max. increase of 1)

If "spread" configurations are considered possible, the "XxxNo" type variables MUST NOT be present since they could cause incorrect behaviour in these configurations. In these configurations, the control cycles (see below) must be effected by analysing all the possible vector elements up to their maximum value (vector limit).

Vice versa, if the “spread” configurations are not considered possible, the “XxxNo” type variables can be used by the applications subject to a consistency check during initialization and the control cycles can be strongly optimized.

The complexity of a general conditioning machine often imposes cross-references between its various objects. By the term “cross-reference”, we mean that, in addition to the obvious evaporator-circuit-compressor (top-down) hierarchy, it is sometimes necessary to go up the hierarchical structure to the higher objects to obtain information on the higher objects themselves. Normally, this concept can also be applied to “transverse” objects such as fans and pump-down type functions, for example.
For example, with single condensation, each circuit has its own pressure/temperature sensor/digital input even if several circuits are coupled on a single fan group.
The fan group control must therefore scan (this is why a cycle is necessary) all the circuits belonging to the specific fan group to measure the pressure/temperature value according to which the fans must be adjusted.
The fact that the number of these cross-references is far from limited and the fact that I might need to go up and down the hierarchy in an indented way (from a cycle on the circuits I must control something related to the circuit compressors but the control on the compressors dictates the use of something related to the circuit) means that the problem must not be underestimated.
For avoid wasting calculation time, the variables can be used in the cycles instead of the relative constants “MAX_XXX”.
A first cycle of the type:

```
(* scan all the circuits *)
for cir_idx := 0 to (MAX_CIR - 1) do
        if CirPresence[cir_idx] = present then
                result := PumpDown(cir_idx);
        end_if;
end_for;
```

is definitely less efficient than a second cycle of the type (it is also not allowed in spread configurations):

```
(* scan all the circuits (until last one) *)
cir_idx := 0;
while CirPresence[cir_idx] = present do
        result := PumpDown(cir_idx);
        cir_idx := cir_idx + 1;
end_while;
```

that is interrupted as soon as the first circuit set as “not present” is encountered”[3].
Note that in the first cycle we have used “for” as the construct to perform the cycle and “(MAX_CIR - 1)” to identify the final limit of the scanning, in this case on the circuits, whereas the second cycle uses the construct “while”[4].
A variant of the first loop that is definitely simpler and therefore clearer to non-programmers would be:

```
(* scan all the circuits *)
for cir_idx := 0 to (NumCir - 1) do
        result := PumpDown(cir_idx);
end_for;
```

that still uses the construct “for” but uses (NumCir - 1)” to identify the final limit of scanning, could be used if the “spread” configurations are not considered possible.
The optimization deriving from the second cycle is significant and must not be underestimated: for a BASELINE APPLICATION machine (four circuits) the time needed for scanning the second cycle is approximately half that necessary for the first cycle (that scans all eight circuits to see if they are present).
There is however a problem in understanding if it is necessary to allow complete disabling of a circuit or an evaporator and keep the enabling for the others, for example, “I have three circuits and I want to disable the second one”).
In this case, the code of the second cycle would incorrectly be interrupted at the second circuit (that has been disabled) and would not continue to the third circuit that is still active.
For these cases, the proposed solution is to insert a new variable, or better still, a new vector, to indicate the fact that the object can be active or not leaving the presence flag on “present”.

Execution of the algorithms would become conditioned, if necessary, by a dual condition:

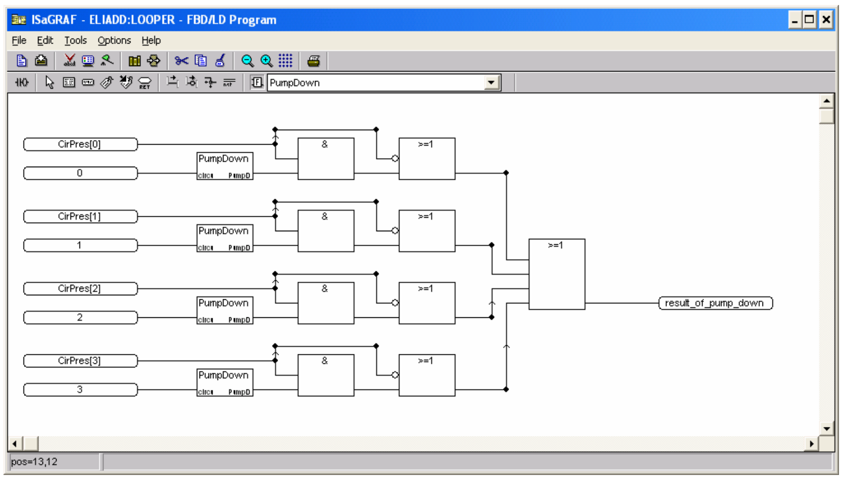
```
if (CirPresence[cir_idx] = present) and
   (CirActive[cir_idx]   = active ) then

...
end_if;
```

where the CirActive[] variable indicates the “possibility” that the control uses the i-nth circuit and can therefore be modified by the MMI something that is not possible for the CirPresence[] variable that defines a structural characteristic of the plant.

Another consideration to be made is the control of the cycles on the objects (compressors, circuits, evaporators) in addition to the ST code.
If the choice of cycle to be effected is not critical for a program written in “Structured Text” (it can be modified very quickly), for other types of program (e.g. FBD/SFC) the control of a while cycle can be complicated albeit impossible or introduce additional programming levels and make the structure rigid.

[3] This construct can also only be used if the “spread” configuration is not possible.
[4] AppMaker provides the construct “while … do … end_while” to effect conditioned loops but generates a warning during compilation to signal the use of a potentially dangerous statement in so much as an incorrect exit from the loop condition could affect the execution type of the PLC cycle.

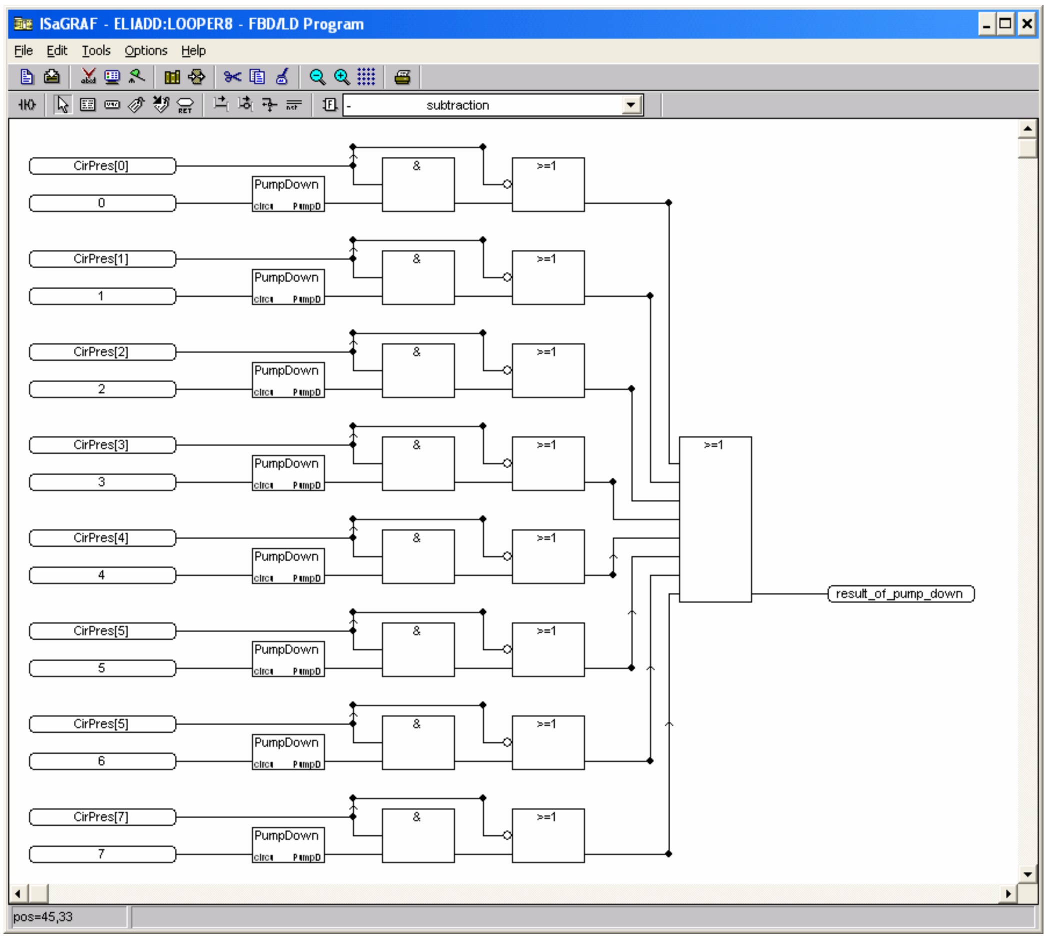
The simple ST cycle mentioned above cannot be implemented as FBD unless feedback is introduced that since it is controlled in various PLC[5] cycles would make the application very slow and unpredictable. In these cases, the FBD should be line exploded as in the following example involving 4 circuits.



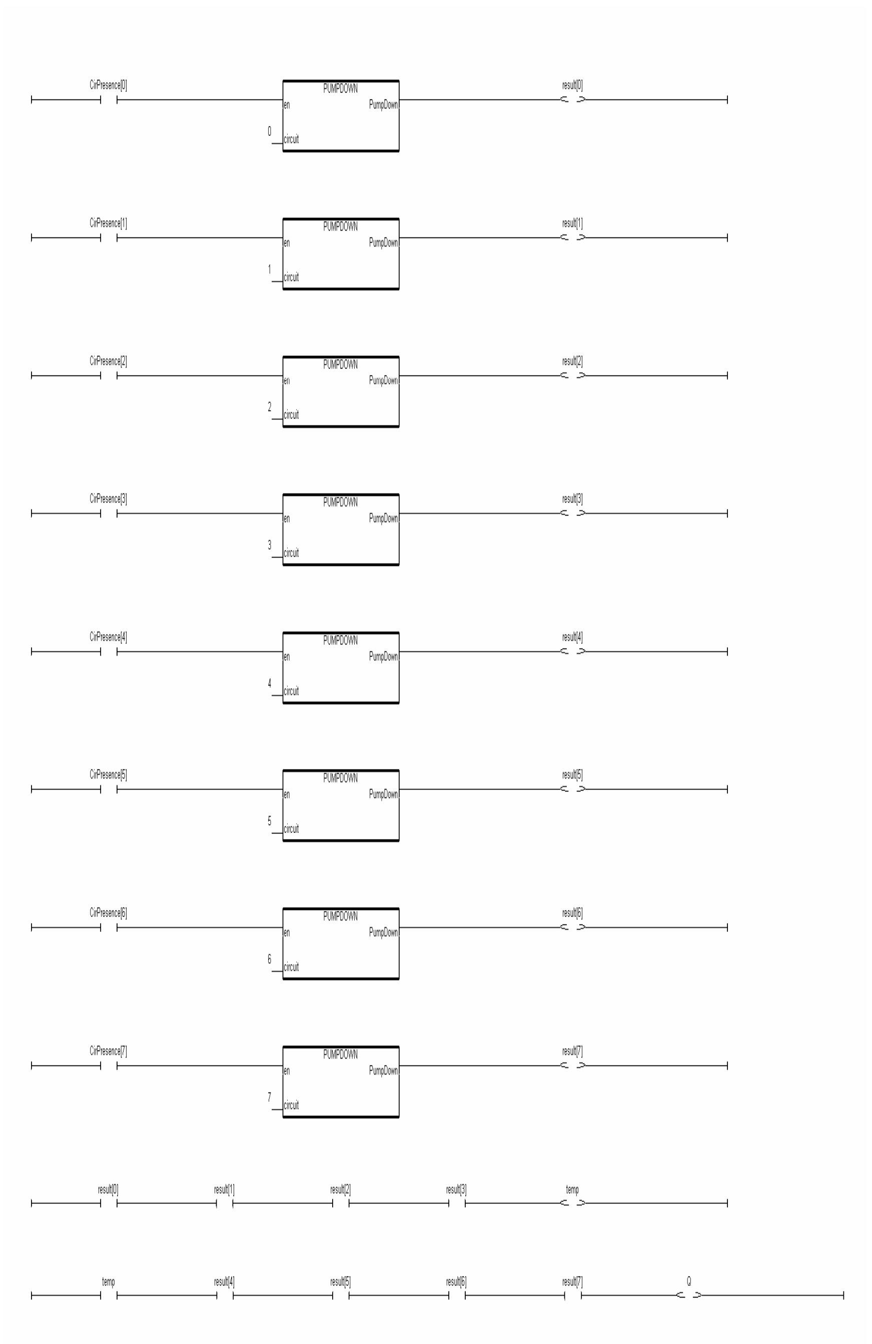
Note that by line exploding the circuits, the use of “XXX_NO” variables and “MAX_XXX” variables is not necessary.

Implementation of the same function for 8 circuits instead of 4, would impose the implementation of a similar schema that is however different (the final OR, for example, would be with 8 input). If, on the one hand, this work is simple and can be performed by personnel without specific high level programming background, the numerous copy-paste-modify operations that are subject to potential errors certainly make performing modifications (the following FBD contains an error, for example) more critical.

[5] At each PLC cycle only one turn would be effected and the output used as input (feedback) would only be considered on the next PLC cycle.



Remember that the code in the FBD would be executed irrespective of whether the objects it operates on are present or not (for example, the “PumpDown” function would always be called for all 8 circuits).
One could argue that it is possible to avoid execution within the function by checking in the function that the object it operates on is actually present but in this case I would have a useless overhead for function calling (note that this overhead is higher the smaller the machine to be produced).
Alternatively, the use of Ladder Diagram could prevent execution of the code using the presence flag of the object in the contact to the left of the function.
This possibility, although valid, leads to the use of local variables and additional rungs to recover the results of function execution as in the following example.



In the example, the function that implements the circuit pump down will only be executed for the circuits whose presence flag (CirPresence) is true. Note that to obtain the overall function result, we had to introduce the local "result" vector, a "temp" variable and two rungs to line up (and) the results of the individual calls in a "readable" and printable way.
Although it is more efficient than the FBD, even the LD must be modified if the maximum number of objects to be dealt with changes.
For these reasons, processing of loops is only recommended for the ST language.
The proposed architecture and variables we refer to in this chapter recommend implementation at two levels of the control logics. The lower level must contain the "pure" control algorithm to be effected on the single object (for example, the pump-down logic of the individual circuit) whereas the higher level, written in ST, must contain the loop to be executed on all the objects present.
Both algorithms would belong to one library. In this way, the user would then be allowed to use the low level, the high level, configure the low level according to the customer and use it for all the circuits, configure the low level according to the customer and use it only for certain circuits with the maximum freedom of configuration.

4.3.12 FANS_NO parameter (Part of IV_Fans)

FANS_NO[8]: integer 8 element vector; it contains the number of fans per fan group. It is only initialized once during start-up by the FANS_NO_1…8 parameters. Range allowed 1…4.
This vector is manually configured.
For the BASELINE APPLICATION machine, for example, the vector must be initialized as follows and care taken that there are no gaps:

FANS_NO			
Number in vector	**Fan Group**	**Content (BASELINE APPLICATION)**	**Remarks (2-4-4)**
0	1	3	Three fans in first fan group
1	2	3	Three fans in second fan group
2	3	1	(not used
3	4	1	(not used
4	5	1	(not used
5	6	1	(not used
6	7	1	(not used
7	8	1	(not used

4.3.13 CIR_FANS ((Part of IV_Plant)

CIR_FANS[8]: integer 8 element vector; it contains the association between circuit and fan group. It is only initialized once during start-up by the CIR_FANS_1…8 parameters. Range allowed 0…2, the value 0 indicates that no fan group is associated to that circuit.
This vector is manually configured.
For example, for the BASELINE APPLICATION machine, the vector must be initialized as follows and care taken that there are no gaps and that numbering is in monotonically ascending order (maximum increase of 1):

CIR_FANS			
Number in vector	**Circuit**	**Content (BASELINE APPLICATION)**	**Remarks**
0	1	1	The first circuit belongs to the first fan group
1	2	1	The second circuit belongs to the first fan group
2	3	2	The third circuit belongs to the second fan group
3	4	2	The fourth circuit belongs to the second fan group
4	5	0	(not used
5	6	0	(not used
6	7	0	(not used
7	8	0	(not used

4.3.14 FansCir (Part of IV_Fans)

FansCir[8*8]: MAX_FANGROUPS * MAX_CIR4FANGROUP integer element vector; lists the circuits that are part of the i-nth fan group (with i = 0.0.7). Calculated by CIR_FANS [].
The AppMaker support for only the one-dimensional arrays implies the use of a single vector instead of a two-size matrix (this would be the most logical solution for this variable).

Since each fan group can have a maximum of 8 circuits and we can have a maximum of 8 fan groups, this vector is sized for 64 elements even if in this case the overall limit of 8 circuits considers the fact that the vector is partially filled (many elements, the unused ones, will contain "-1").

For the "i-nth" fan group (with 0 <= i <= (MAX_FANGROUPS – 1)), the relevant starting index will be given by the expression:

i * (MAX_CIR4FANGROUP**)** (it represents an offset in the vector)

where "MAX_CIR4FANGROUP" is the constant that defines the maximum number of circuits per fan group; the valid elements for this fan group will therefore be MAX_CIR4FANGROUP.
The CIR_FANS vector is scanned during initialization and for each element with a "k" index whose value is not zero it sets to "k" the FansCir element with index

```
(value – 1) * MAX_CIR4FANGROUP + offset
```

Note that this vector will be used in mode “0..(n – 1)” in the applications. i.e. with index 0 used (index for the first circuit of the first fan group).

FansCir			
Number in vector	Fan Group	Content (BASELINE APPLICATION)	Remarks
0	1	0	The first circuit belongs to the first fan group
1	1	1	The second circuit belongs to the first fan group
2	1	-1 (not used)	(not used
3	1	-1 (not used)	(not used
4	1	-1 (not used)	(not used
5	1	-1 (not used)	(not used
6	1	-1 (not used)	(not used
7	1	-1 (not used)	(not used
8	2	2	The third circuit belongs to the second fan group
9	2	3	The fourth circuit belongs to the second fan group
10	2	-1 (not used)	(not used
11	2	-1 (not used)	(not used
12	2	-1 (not used)	(not used
13	2	-1 (not used)	(not used
14	2	-1 (not used)	(not used
15	2	-1 (not used)	(not used
16	3	-1 (not used)	(not used
17	3	-1 (not used)	(not used
18	3	-1 (not used)	(not used
19	3	-1 (not used)	(not used
20	3	-1 (not used)	(not used
21	3	-1 (not used)	(not used
22	3	-1 (not used)	(not used
23	3	-1 (not used)	(not used
24	4	-1 (not used)	(not used
25	4	-1 (not used)	(not used
26	4	-1 (not used)	(not used
27	4	-1 (not used)	(not used
28	4	-1 (not used)	(not used
29	4	-1 (not used)	(not used
30	4	-1 (not used)	(not used
31	4	-1 (not used)	(not used
32	5	-1 (not used)	(not used
33	5	-1 (not used)	(not used
34	5	-1 (not used)	(not used
35	5	-1 (not used)	(not used
36	5	-1 (not used)	(not used
37	5	-1 (not used)	(not used
38	5	-1 (not used)	(not used
39	5	-1 (not used)	(not used
40	6	-1 (not used)	(not used
41	6	-1 (not used)	(not used
42	6	-1 (not used)	(not used
43	6	-1 (not used)	(not used
44	6	-1 (not used)	(not used
45	6	-1 (not used)	(not used
46	6	-1 (not used)	(not used
47	6	-1 (not used)	(not used
48	7	-1 (not used)	(not used
49	7	-1 (not used)	(not used
50	7	-1 (not used)	(not used
51	7	-1 (not used)	(not used
52	7	-1 (not used)	(not used
53	7	-1 (not used)	(not used
54	7	-1 (not used)	(not used
55	7	-1 (not used)	(not used
56	8	-1 (not used)	(not used
57	8	-1 (not used)	(not used
58	8	-1 (not used)	(not used
59	8	-1 (not used)	(not used
60	8	-1 (not used)	(not used
61	8	-1 (not used)	(not used
62	8	-1 (not used)	(not used
63	8	-1 (not used)	(not used

4.3.15 FansNo (Part of IV_Fans)

These variables are not considered mandatory since the look-up tables defined in the previous chapters (with the “-1” “plug” values) are (and must be) sufficient to guarantee correct access to all the vectorized objects.
The decision to implement the “XxxNo” parameters/variables depends, above all, on the type of consistency checks to be implemented before being run “one shot” during the initialization stage.
This parameter defines the total number of fan groups that must be controlled
(FansNo, with 0 < FansNo <= MAX_FANGROUPS) and can however be calculated by CIR_FANS[].
The CIR_FANS vector is scanned during initialization and for each element with a “k” index whose value is not zero it increases the number of fan groups (FansNo) if and only if the fan group
“<value>” has not already been taken into consideration.

In the Baseline application FansNo=2.

4.3.16 PUMP_NO parameter

This parameter is used to set the number of pumps that the plant must handle.
The range for this parameter will have to be 0..MAX_PUMP, where MAX_PUMP is the constant that defines the maximum number of pumps that can be controlled by the application.
For BASELINE APPLICAITON, the maximum number of pumps, equal to the number of pumps to be controlled, is two,
The control code of all that is related to the pumps is conditioned by the fact that the number of pumps is “greater than zero” as in the following example in “ST” code:

```
if PUMP_NO > 0 then
        (* control of pumps *)
end_if;
```

Obviously, cycles operating on PumpsXxx[MAX_PUMP] type vectors are also possible. These cycles will have as a termination logic condition “index < (PUMP_NO – 1) :

```
for pump_idx := 0 to (PUMP_NO – 1) do
(* control of i-nth pump (with index “pump_idx) *)
...
end_for;
```

4.4 Constants (Defined words)

Although it may seem excessive to reach this definition level in an architecture document, a preliminary list of the principal constants is provided below.
Obviously the list is not complete but identifies the areas in which the constants are necessary.

MAX_CIR	8	(*Maximum number of circuits on machine*)
MAX_EV	4	(*Maximum number of evaporators on machine (2 CVM)*)
MAX_KOMP	8	(*Maximum number of compressors on machine*)
MAX_PUMP	2	(*Maximum number of pumps on machine*)
MAX_FANGROUPS	8	(*Maximum number of fan groups on machine (2 CVM)*)
MAX_FANS	16	(*Maximum number of fans on machine (8 CVM)*)
MAX_KOMP4CIR	8	(*Maximum number of compressors per circuit (1 CVM)*)
MAX_CIR4EV	4	(*Maximum number of circuits per evaporator*)
MAX_FANS4FANGROUP	8	(*Maximum number of fans per fan group (4 CVM)*)
MAX_CIR4FANGROUP	8	(*Maximum number of circuits per fan group (4 CVM)*)

KIM_STANDARD	0	(Standard compressor power on*)
KIM_PARTWINDING	1	(*Part winding compressor start-up*)
KIM_STARDELTA	2	(Star/delta compressor start-up*)

SENS_ERROR	-32768	(*Probe error code*)

P_KOMP	0	(*Compressor subsystem*)
P_CIR	1	(*Circuit subsystem*)
P_EV	2	(*Evaporator subsystem*)
P_PLAN	3	(*Plant subsystem*)

SATURATION	False	(*definition for saturation policy*)
BALANCING	True	(*definition for balancing policy*)

KOMP_OFF	0	(*OFF state of i-nth compressor*)
KOMP_ON_NR	1	(**ON Not Ready* state of i-nth compressor*)
KOMP_ON	2	(*ON state of i-nth compressor*)
KOMP_OFF_NR	3	(**OFF Not Ready* state of i-nth compressor*)

ENTRY_SENS	0	(*definition for inlet water sensor*)
EXIT_SENS	1	(*definition for outlet water sensor*)

TREG_*PI*	2	(*P.I. temperature control*)
TREG_TIMEPROP	1	(*Time-proportional temperature control*)
TREG_PROPORTIONAL	0	(*Proportional temperature control*)

SEMI-HERMETIC	0	(*Semi-hermetic type compressor*)
SCREW	1	(*Screw type compressor*)

PLAN_OFF	0	(*Plant OFF state*)
PLAN_SHUTDOWN	1	(*Plant SHUTDOWN state*)
PLAN_CHILLING	2	(*Plant ON state in Chiller mode*)

PUMPGROUP_OFF	0	(*OFF state of pump group*)
PUMPGROUP_GOING_UP	1	(*ON state of pump group from plant ON*)
PUMPGROUP_ON	2	(*ON state of pump group*)
PUMPGROUP_GOING_DOWN	3	(*ON state of pump group from plant OFF*)
PUMPGROUP_ON4HEATERS	4	

AAH_NOMINAL	0	(*AAH nominal status*)
AAH_ALARM	1	(*AAH alarm status*)

MAH_NOMINAL	0	(*MAH nominal status*)
MAH_ALARM	1	(*MAH alarm status*)
MAH_WAIT_RESET	2	(*MAH resettable status*)

GB_OFF	0	(*Generic bypass status off*)
GB_ON	2	(*Generic Bypass status on*)
GB_BYPASS_OFF_ON	1	(*Generic Bypass status off-on*)
GB_BYPASS_ON_OFF	3	(*Generic Bypass status on-off*)

B_OFF	0	(*Bypass status off*)
B_BYPASS	1	(*Bypass status bypass*)
B_ON	2	(*Bypass status on*)

DTSET_NONE	0	(*Dynamic set point not enabled*)
DTSET_TEMP	1	(*Dynamic set point enabled in temperature mode*)
DTSET_CURR	2	(*Dynamic set point enabled in current mode*)

HY_OFF	0	(*Low value status of hysteresis*)
HY_ON	1	(*High value status of hysteresis*)

BAH_NOMINAL	0	(*BAH alarm nominal status*)
BAH_AUTO_RES	1	(*BAH automatically resettable status*)
BAH_MAN_RES	3	(*BAH active auto -> manual status*)
BAH_WAIT_RES	2	(*BAH manually resettable status*)

ASYMMETRICAL	True	(*Asymmetrical fans*)
SYMMETRICAL	False	(*Fans with same power*)

PDA_START	0	(*start pump-down during start-up*)
PDA_HANDLE	1	(*handle pump-down during start-up*)
PDS_START	2	(*start pump-down during shutdown*)
PDS_HANDLE	3	(*handle pump-down during shutdown*)
PD_RESET	4	(*reset pump-down handling*)

solenoid_open	False	(*solenoid valve open*)
solenoid_close	True	(*solenoid valve closed*)

PD_NONE	0	(*pump-down not supported*)
PD_ONSTART	1	(*pump-down during start-up*)
PD_FULL	2	(*pump-down during start-up and shutdown*)

pd_max_press_reached	false	(*pump-down pressure switch indicates max pressure has been reached*)
pd_min_press_reached	true	(*pump-down pressure switch indicates min pressure has been reached*)

NOT_IN_PD	0	(*not in pump-down*)
PDA1	1	(*pump-down during start-up phase 1*)
PDA2	2	(*pump-down during start-up phase 2*)
PDS	3	(*pump-down during shutdown*)
FINISH_PDA	4	(*pump-down during start-up finished*)
FINISH_PDS	5	(*pump-down during shutdown finished*)

Note that the constants have been divided into groups
The first group consists of the "MAX_XXX" constants, i.e. the constants that restrict the physical dimensions of the plant and assume values on the basis of which the arrays have been sized.
The other groups include the definition of the Boolean logic values that are traceable to only the "true" and "false"[6] values or the enumerative values of states in the machine.
The use of Defined Words makes the application more readable and less ambiguous For example, the fact that for an open solenoid valve the status of the output line that controls it must be 1/on/active, i.e. "true" could be misleading whereas the use of a construct similar to the one below makes the behaviour required by the algorithm clear.

```
if SOL_VALVE_FLAG[circuit] = TRUE then
(* if solenoid valve present ... *)
        SOL_VALVE[circuit] := solenoid_close;
        (* ... set solenoid close command *)
end_if;
```

[6] These aliases are currently indicated in the document using lower case characters but as constants can follow the nomenclature for constants (all upper case characters)

5 DESCRIPTION OF BASE-LINE APPLICATION

Details of the main data structures and the algorithmic control part for each of the program units referred to in the previous chapter are given below.

5.1 IniVar

In this first section of the initial shutdown, all the data structures that describe the machine structure as in Chapter 3 are initialized and all the features that are normally present in each AppMaker project such as going into configuration mode are dealt with. Initialization involves "expanding" any "compressed" parameters set by the configurator/MMI in the vectors used by the PLC applications. Here we are talking about single parameters that can be set by the configurator/MMI that are valid for all the elements they refer to (a single threshold for all the circuit alarms, a single flag to enable a specific type of sensor for all the compressors, etc.). If the configurator/MMI initially allows the setting of a single parameter that is valid for all the system elements (compressor, circuit, evaporator) whereas the PLC applications are developed to allow non-uniform handling (element by element) of this parameter, this program unit will see to its relative expansion with a code:

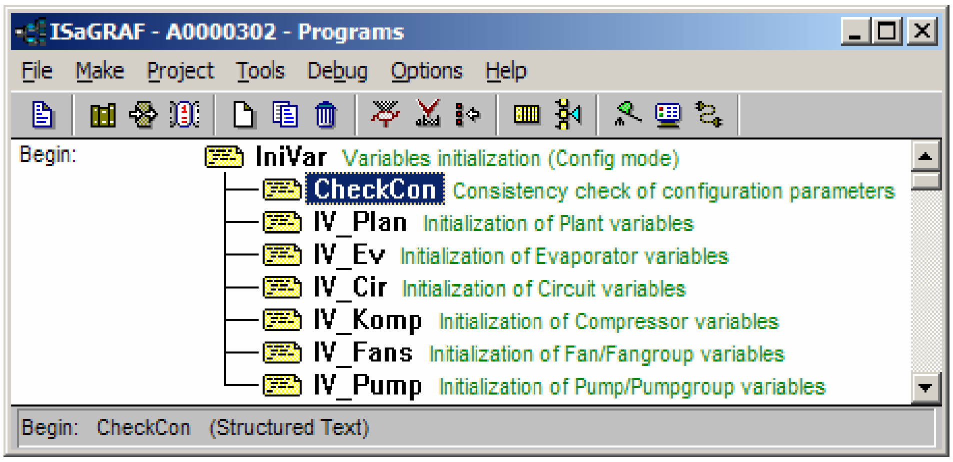
```
for cir_idx := 0 to (MAX_KOMP – 1) do
  CirHPresADelta[cir_idx] := A_MAX_DELTA_PRES;
end_for;
```

Once the configurator/MMI has been updated, this expansion will no longer be necessary and can be removed. In this way, you can decide which parameters will be different for the various elements of the same type or if there will be one single parameter. The choice will be based on functional requirements (compressors with different capacity steps, for example) within the limits set by the platform with regard to size of memory and execution time.

It is important to note the structure of the chiller is non-symmetrical and potentially unbalanced, the data vectors that describe the machine structure can be used to "reconstruct" and therefore pass through the whole hierarchical tree from top to bottom (from the roots to the leaves) and from bottom to top (from the leaves to the roots). This is the reason for the presence of data structures that indicate the "children" of each element and structures that indicate the "parent" (or parents) of each node.

5.1.1 CheckCon

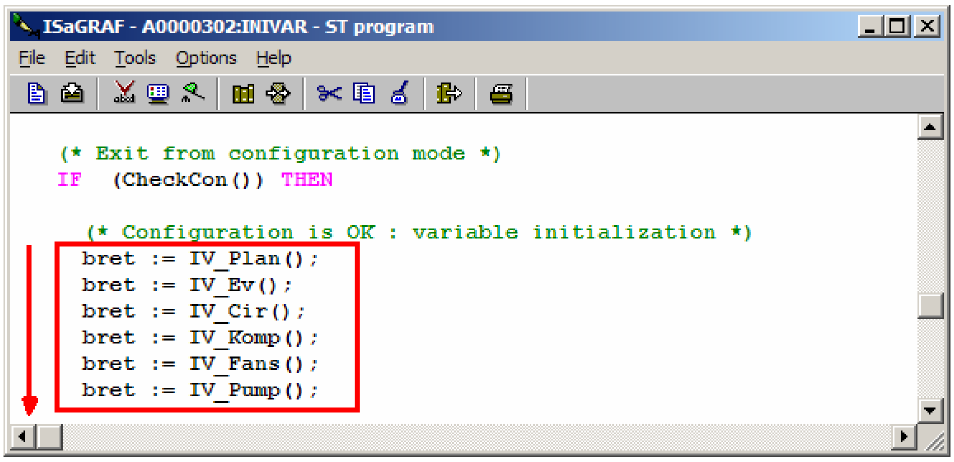
For each program unit, there are no special support structures except for the variable/parameter that allows one-shot execution and the result of the consistency checks that do not allow any PLC operation.


ISaGRAF - A0000302 - Programs
File Make Project Tools Debug Options Help

```
Begin:   IniVar  Variables initialization (Config mode)
           ├─ CheckCon  Consistency check of configuration parameters
           ├─ IV_Plan   Initialization of Plant variables
           ├─ IV_Ev     Initialization of Evaporator variables
           ├─ IV_Cir    Initialization of Circuit variables
           ├─ IV_Komp   Initialization of Compressor variables
           ├─ IV_Fans   Initialization of Fan/Fangroup variables
           └─ IV_Pump   Initialization of Pump/Pumpgroup variables
Begin:  CheckCon  (Structured Text)
```

In the BASELINE APPLICATION this is a dummy function that always return a "true" value (*CheckCon* := true;). For this reason, we decided to leave application developers free to customize this feature according to the "sensitive" points of their applications.

It is very important to note that the order of execution of the single "child" Program Units of another Program Unit DO NOT follow the order in which they are inserted into the AppMaker project but the order in which they are invoked in the parent Program Unit.


ISaGRAF - A0000302:INIVAR - ST program
File Edit Tools Options Help

```
  (* Exit from configuration mode *)
  IF  (CheckCon()) THEN

    (* Configuration is OK : variable initialization *)
    bret := IV_Plan();
    bret := IV_Ev();
    bret := IV_Cir();
    bret := IV_Komp();
    bret := IV_Fans();
    bret := IV_Pump();
```

In the figure the two orders coincide but if *IV_Plan*() and *IV_Ev*() were to change place in the ST program in InitVar, the Evaporator variables would be initialized first and then the Plant variables.

5.1.2 IV_Plan

The main function of the "InitVar" of the "Plant" is that of copying the I/O support variables onto the relative one-dimensional arrays that will then be used in the calculation of the architecture. This first involves "filling" the KOMP_CIR_EV[] vector that, as seen in chapter 3, represents the "relationship" between the compressors, circuits and evaporators. Since it is a high level configuration, the vector that describes the relationship between the fan groups and the circuits, CIR_FANS[], is also acquired. All the alarms and Plant timers are then reset.

5.1.3 IV_Ev

Initialization of the Evaporator variables involves filling the EvPresence[] and EvCir[] structures, i.e. the structures that define the presence or non-presence of the evaporator (enabling) and the relationship with the circuits. Through these data structures, it will be possible to determine which circuit hierarchically belongs to which evaporator. All the Evaporator alarms and timers are then reset.

5.1.4 IV_Cir

The compressors present (enabled) are described and the hierarchical links between the circuits and higher structures (evaporators) and lower structures (compressors) are calculated. As already seen in chapter 3, the CirPresence[], CirEv[] , CirKomp[] vectors represent these links.
Here too, all the circuit alarms and timers are then reset.

5.1.5 IV_Komp

Since the architecture supports different types of compressor, the KOMP_STEP[MAX_KOMP] structure must be filled in by describing the characteristics (in terms of capacity steps) of each compressor. The KompCir[] structure must also be filled in by describing the hierarchical relationship between each compressor and its circuit. All the compressor alarms and timers must then be reset.

5.1.6 IV_Fans

In this *program unit* the FANS_NO[MAX_FANGROUPS] structure that, as seen before, describes the link between fans and circuits is initialized. All the data structures related to the fans, i.e. FANS_CSTART_SET_PRES[] and FANS_CSTOP_DELTA_PRES[] are also initialized. FansCir[] is calculated and all the fan group timers and alarms are reset.

5.1.7 IV_Pump

The two pumps do not require special initialization but the timers must be realigned (in PumpHours[]) and the relevant alarm variables reset.

5.2 Phy2Log

5.2.1 P2L_xxx

This *program unit* group's task is to copy all the physical system values onto the "logic" variables. As defined in the chapter on the *nomenclature* of variables and parameters, all the variables that refer to a physical value have names that end in _PHY. In the same way, the "HOT" parameters have names that end in "HOT".

The following program units will be provided, one for each structural element of the chiller:

• *Phy2Log*	(Main program unit)
• P2L_Plan	(Conversion from physical to logic IN at Plant level)
• P2L_Ev	(Conversion from physical to logic IN at Evaporator level)
• P2L_Cir	(Conversion from physical to logic IN at Circuit level)
• P2L_Komp	(Conversion from physical to logic IN at Compressor level)
• P2L_Fans	(Conversion from physical to logic IN at Fan Group level)
• P2L_Pump	(Conversion from physical to logic IN at Pump Group level)

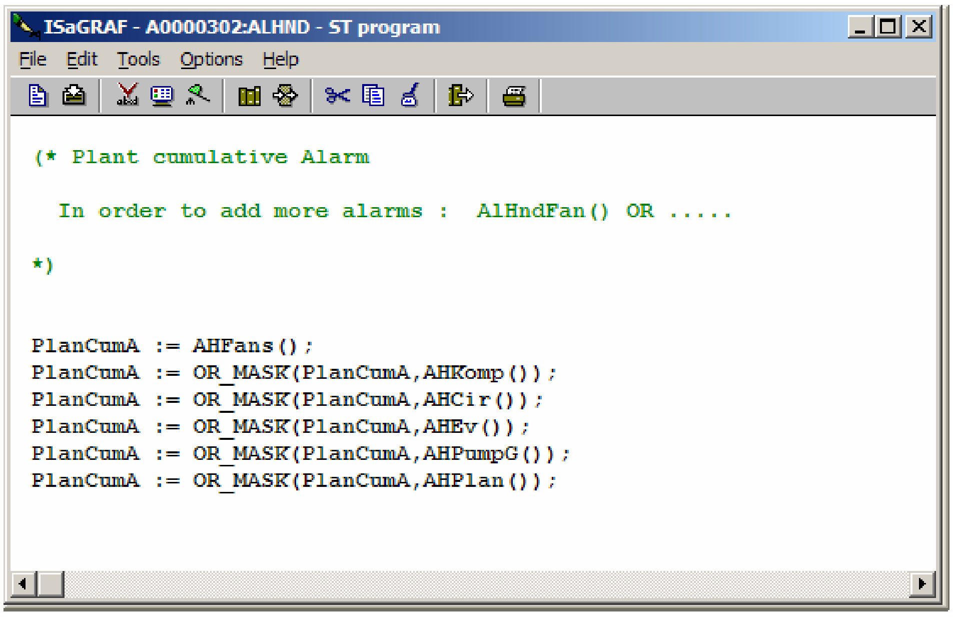
This type of operation (copying from physical variables to logical variables) is necessary because only I/O defined variables that CANNOT be vectors can be assigned to the controller terminals. If I/O variables defined on arrays need to be used, you must pass through support variables that are then copied into vectors used for the algorithmic calculation.

To maintain code “cleanliness” and “rationality”, copying from physical to logical is effected on all the *chiller* subsystems even in situations in which this is not necessary since in AppMaker what is not “updated” is not “refreshed”.

5.3 AlHnd

For the system elements, plant, evaporator, circuit, etc. but also for some functions (free cooling, recovery, anti freeze etc) nominal behaviour is used but there are also one or more abnormal conditions (alarms, for example) that affect behaviour that must be implemented by the logics.

Having said this, this program unit has the task of checking the alarm conditions whatever type they are and generating any shutdown conditions that will have direct consequences on the calculation of control and therefore output.



```
(* Plant cumulative Alarm

  In order to add more alarms :  AlHndFan() OR .....

*)


PlanCumA  := AHFans();
PlanCumA  := OR_MASK(PlanCumA,AHKomp());
PlanCumA  := OR_MASK(PlanCumA,AHCir());
PlanCumA  := OR_MASK(PlanCumA,AHEv());
PlanCumA  := OR_MASK(PlanCumA,AHPumpG());
PlanCumA  := OR_MASK(PlanCumA,AHPlan());
```

As previously explained, the order with which the different functions of the “parent” Program Unit are invoked must be followed to proceed. In this case, the first to be calculated will be the handling of the fan alarms (AHFans()).

It is very important to note that the alarm of a plant component can have a domino effect in the alarm state of other components. It is for this reason that it is essential to pinpoint an alarm “hierarchy” and assess “cumulatively” the alarm state of the machine from the least important element to the most important one in the hierarchy of the alarms.

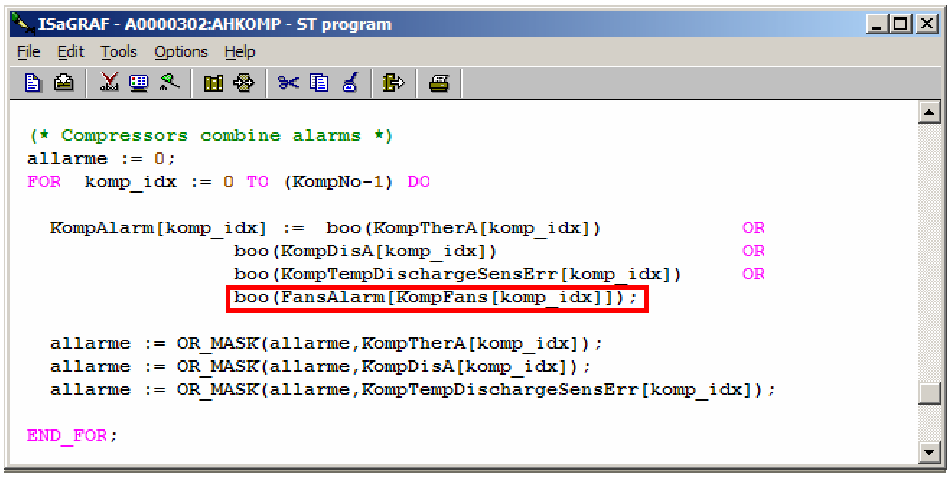
Below we will see that the order is which the various Alarm Handling program units are considered follows this principle: the alarm state of the *AH_component* function considered will depend on the alarm state of the component and the alarm state of what has been examined up to that moment.

5.3.1 AH_Fans

After having assessed the presence of a fan group thermal switch alarm, it will be sufficient to analyse the state of each single fan and a logical “or” will be made of any alarms.
Note that *AH_Fans* is executed first since any alarm state of the fan group depends on the fans only and therefore the calculation of alarms does not need to assess if other parts of the plant are in alarm mode or not.

5.3.2 AH_Komp

Assessment of the alarm state of the compressors depends on the state of the compressors themselves and any alarm state of the fans.



ISaGRAF - A0000302:AHKOMP - ST program

```
(* Compressors combine alarms *)
allarme := 0;
FOR  komp_idx := 0 TO (KompNo-1) DO

  KompAlarm[komp_idx]  :=  boo(KompTherA[komp_idx])            OR
                           boo(KompDisA[komp_idx])             OR
                           boo(KompTempDischargeSensErr[komp_idx])  OR
                           boo(FansAlarm[KompFans[komp_idx]]);

  allarme := OR_MASK(allarme,KompTherA[komp_idx]);
  allarme := OR_MASK(allarme,KompDisA[komp_idx]);
  allarme := OR_MASK(allarme,KompTempDischargeSensErr[komp_idx]);

END_FOR;
```

As we can see from the part selected in the red frame, the alarm state of each compressor is the total sum of all the potential alarm conditions as well as the alarm state of the fans that must have already been previously calculated and therefore available.

5.3.2.1 AH_KompEr

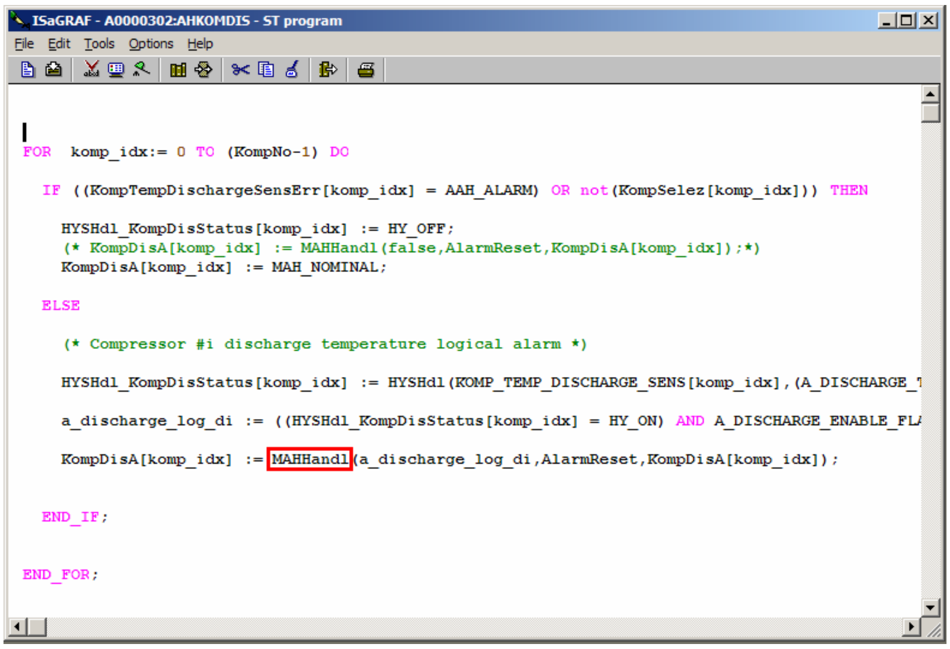
In this program unit the presence of the discharge temperature probe errors is examined. Note the presence of the AAHHandl() function, i.e. the handling function of the “fault discharge temperature probe” automatically reset alarm.

5.3.2.2 AH_KompTh

In this program nit the presence of a compressor thermal switch alarm is examined for the compressors present.

5.3.2.3 AH_KompDis

In this program unit the presence of a compressor discharge temperature alarm is examined.



ISaGRAF - A0000302:AHKOMDIS - ST program

```
FOR  komp_idx:= 0 TO (KompNo-1) DO

  IF ((KompTempDischargeSensErr[komp_idx] = AAH_ALARM) OR not(KompSelez[komp_idx])) THEN

    HYSHdl_KompDisStatus[komp_idx] := HY_OFF;
    (* KompDisA[komp_idx] := MAHHandl(false,AlarmReset,KompDisA[komp_idx]);*)
    KompDisA[komp_idx] := MAH_NOMINAL;

  ELSE

    (* Compressor #i discharge temperature logical alarm *)

    HYSHdl_KompDisStatus[komp_idx] := HYSHdl(KOMP_TEMP_DISCHARGE_SENS[komp_idx],(A_DISCHARGE_

    a_discharge_log_di := ((HYSHdl_KompDisStatus[komp_idx] = HY_ON) AND A_DISCHARGE_ENABLE_FL

    KompDisA[komp_idx] := MAHHandl(a_discharge_log_di,AlarmReset,KompDisA[komp_idx]);


  END_IF;


END_FOR;
```

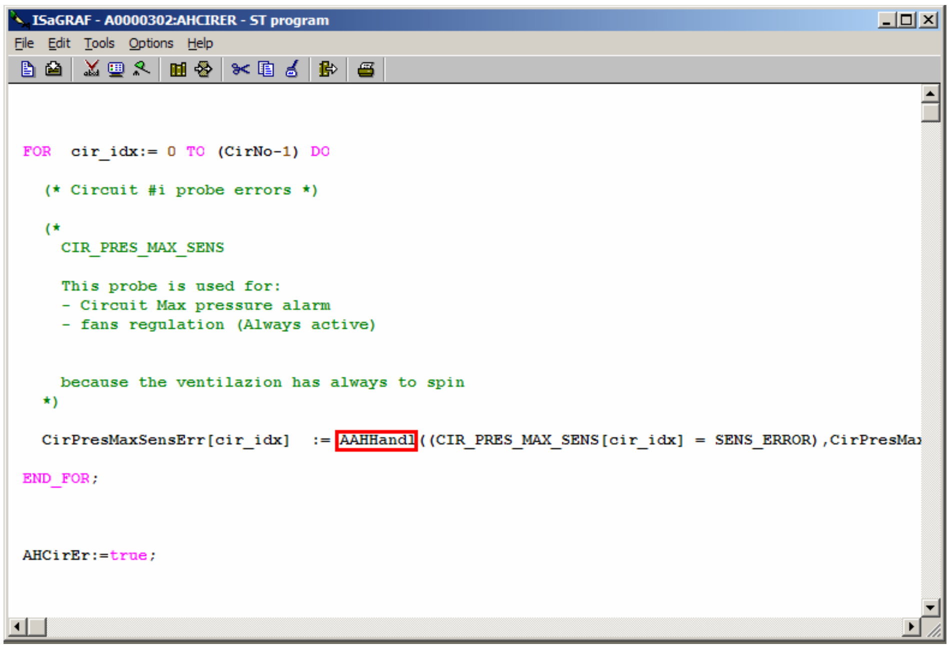
Note in the calculation of the discharge alarm of the i-nth compressor, the presence of the MAHHandl() function that is used for the handling of the **manually reset** alarms.

5.3.3 AH_Cir

The handling of circuit alarms comes in hierarchical order immediately after compressor handling. In its logics, it is the same as what has already been seen for the compressors.

5.3.3.4 AH_CirEr

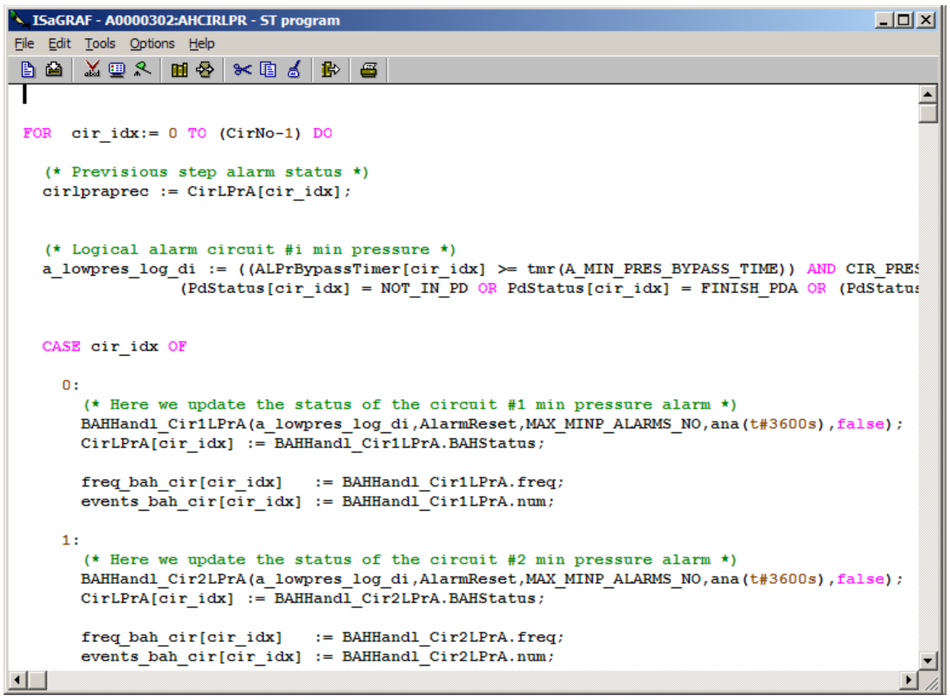
The purpose of this program unit is to calculate any probe errors in the i-nth circuit. The logic is the same as what has already been seen for the compressors but it is important to note the presence of the AAHHandl() function, i.e. the handling function of the “faulty maximum pressure probe” automatically reset alarm.


ISaGRAF - A0000302:AHCIRER - ST program
File Edit Tools Options Help

```
FOR  cir_idx:= 0 TO (CirNo-1) DO

  (* Circuit #i probe errors *)

  (*
    CIR_PRES_MAX_SENS

    This probe is used for:
    - Circuit Max pressure alarm
    - fans regulation (Always active)


    because the ventilazion has always to spin
  *)

  CirPresMaxSensErr[cir_idx]   := AAHHandl((CIR_PRES_MAX_SENS[cir_idx] = SENS_ERROR),CirPresMa

END_FOR;


AHCirEr:=true;
```

5.3.3.5 AH_CirHPr and AH_CirLPr

The purpose of these two functions is to calculate the presence of high pressure (*AH_Cir*HPr) or low pressure (*AH_Cir*LPr) alarms. Calculation is more difficult than before and the need to use bounded alarms which means that vectors cannot be used compels us to use special methods when writing the code and repeating the code for the maximum number of circuits. Note that if you want to extend the DOMAIN of the BASELINE APPLICATION by increasing the number of possible circuits, this Program Unit must also be extended by modifying it so that a larger domain can be used. One immediate example is that, since the circuit index variable (cir_idx) can have values greater than 8 (if the number of maximum circuits is increased), the number of considered “cases” must be increased by adding code in the same way as shown here:


ISaGRAF - A0000302:AHCIRLPR - ST program
File Edit Tools Options Help

```
FOR  cir_idx:= 0 TO (CirNo-1) DO

  (* Previsious step alarm status *)
  cirlpraprec := CirLPrA[cir_idx];


  (* Logical alarm circuit #i min pressure *)
  a_lowpres_log_di := ((ALPrBypassTimer[cir_idx] >= tmr(A_MIN_PRES_BYPASS_TIME)) AND CIR_PRES
                (PdStatus[cir_idx] = NOT_IN_PD OR PdStatus[cir_idx] = FINISH_PDA OR (PdStatu


  CASE cir_idx OF

    0:
      (* Here we update the status of the circuit #1 min pressure alarm *)
      BAHHandl_Cir1LPrA(a_lowpres_log_di,AlarmReset,MAX_MINP_ALARMS_NO,ana(t#3600s),false);
      CirLPrA[cir_idx] := BAHHandl_Cir1LPrA.BAHStatus;

      freq_bah_cir[cir_idx]   := BAHHandl_Cir1LPrA.freq;
      events_bah_cir[cir_idx] := BAHHandl_Cir1LPrA.num;

    1:
      (* Here we update the status of the circuit #2 min pressure alarm *)
      BAHHandl_Cir2LPrA(a_lowpres_log_di,AlarmReset,MAX_MINP_ALARMS_NO,ana(t#3600s),false);
      CirLPrA[cir_idx] := BAHHandl_Cir2LPrA.BAHStatus;

      freq_bah_cir[cir_idx]   := BAHHandl_Cir2LPrA.freq;
      events_bah_cir[cir_idx] := BAHHandl_Cir2LPrA.num;
```

5.3.4 AH_Ev

The handling of evaporator logic alarms is the same as that previously described. It involves handling the potential errors caused by the breaking of one or more probes (computed in *AH_EvEr*), and the evaporator antifreeze alarm (computed in *AH_EvAf*).

5.3.4.6 AH_EvEr

The function handles the probe error in the evaporator module.
Note the presence of the AAHHandl() function, i.e. the handling function of the "faulty outlet water temperature probe error" automatically reset alarm.

5.3.4.7 AH_EvAf

The evaporator antifreeze alarm is handled in this Program Unit.

5.3.5 AH_PumpG

Handling of the pump/pump group logic alarms by integrating pump swapping with the possibility of a flow switch alarm.

5.3.5.8 AH_PumpTh

The purpose of this program unit is to check for the presence of a thermal switch alarm on the pumps.

5.3.6 AH_Fans

The handling of the fan/fan group logic alarms passes through the handling of any fan thermal switch errors computed in the child Program Unit.

5.3.6.9 Ah_FansTh

The purpose of this program unit is to check for the presence of a thermal switch alarm on the fans (fan group).

5.3.7 AH_Plan

Handling of the plant alarms must combine both the plant alarms and the lower level alarms so that they reach the "brain" of the machine and can be used in subsequent control phases.

5.3.7.10 AH_PlanEr

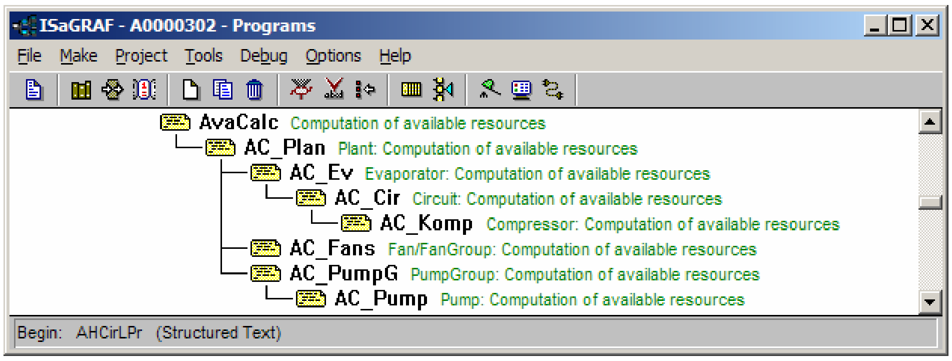
The local plant errors are errors linked to faulty probes and in particular to the water inlet, water outlet and set point control probes.

5.3.7.11 AH_PlantHT

In this case, the purpose of this Program Unit is to assess computation of the plant high temperature error.

5.4 AvaCalc

After performing the alarm management part, an important part in the temperature control process is that of calculating



```
AvaCalc  Computation of available resources
└─AC_Plan  Plant: Computation of available resources
    ├─AC_Ev  Evaporator: Computation of available resources
    │  └─AC_Cir  Circuit: Computation of available resources
    │      └─AC_Komp  Compressor: Computation of available resources
    ├─AC_Fans  Fan/FanGroup: Computation of available resources
    └─AC_PumpG  PumpGroup: Computation of available resources
        └─AC_Pump  Pump: Computation of available resources
Begin:  AHCirLPr  (Structured Text)
```

the availability of refrigerating (machine) resources.

Conceptually speaking, the calculation of availability is a process that starts with less important elements of the machine which each inform their "parent element" of which and how many resources (refrigerating and physical) they are able to supply if necessary.
Therefore, each compressor will have to inform the circuit it belongs to of how many power "steps" it can supply on request. The circuit, one it has gathered information on the availability of all its compressors, must then inform the evaporator it belongs to of how many resources it can supply on request. Obviously, this power will be the sum total of the available resources of each compressor belonging to it. In the same way, the power that each evaporator will supply as information available to the plant will be the sum total of the powers available in each of its circuits.

It should be noted that the calculation of available resources is not limited to a simple mathematical sum total of the availability of the compressors but also considers any situations/states in which parts of the system may find themselves in which mean that although the compressors may be "available" to supply refrigerating power, one or more components of a higher hierarchical level are not.

5.4.1 Status variables:

The data structures that this program unit operates on are mainly used to maintain the static and dynamic availability data, the level requested and the actual level of the evaporator, circuit and compressor. For a more detailed description, refer to the paragraph on compressor logic.

KompMinLevDin[8]	Compressor minimum dynamic availability
KompMaxLevDin[8]	Compressor maximum dynamic availability
CirMinLevDin[8]	Circuit minimum dynamic availability
CirMaxLevDin[8]	Circuit maximum dynamic availability
EvMinLevDin[4]	Evaporator minimum dynamic availability
EvMaxLevDin[4]	Evaporator maximum dynamic availability
PlanMinLevDin	Plant minimum dynamic availability
PlanMaxLevDin	Plant maximum dynamic availability
FansActivable[8]	Fan group availability
PumpGMinLevDin	Pump group minimum dynamic availability
PumpGMaxLevDin	Pump group maximum dynamic availability
PlanPumpGMinLevDin	Pump group minimum dynamic availability at plant level
PlanPumpGMaxLevDin	Pump group maximum dynamic availability at plant level

All the vectors are allocated for the maximum number of elements of each type. Where this is not a problem because of memory restrictions, this means that the code can be developed invariantly with regard to the number of evaporators, circuits and compressors.
In this case, if the number of elements changes, the code must be modified. For a code developed in ST language, the modification is limited to changing the numeric value of a parameter whereas for codes developed with graphical languages (FBD / QLD), blocks must be added or removed from the relative program unit.

5.4.2 AC_Plan

The purpose of this program unit is to calculate the availability of the refrigerator resources for the entire plant by acquiring the availability of each one from all the system elements.

5.4.3 AC_Ev

To calculate evaporator availability, just check for the presence of alarms once the availability of all the circuits has been obtained. The calculation is performed from the PolicyCD() function located in the functions area of the AppMaker project.

5.4.4 AC_Cir

Circuit availability is also calculated only after calculating compressor availability and assessing any shutdown or alarm situations.

5.4.5 AC_Komp

The calculation of compressor availability is characterized by more complex algorithms that take into account how long a compressor has been in operation and the number of times that it has been switched on and off. By using the BASELINE APPLICATION, the resource calculation control algorithms can be customized especially the calculation of compressor availability. If you wish to use compressors with different powers, for example, modifications can be made in this part of the program so that the plant is aware (via circuits and evaporators) of the sum total of resources available.

5.4.6 AC_Fans

Although the fans are a plant element, as far as availability is concerned, they are completely disconnected from the other plant elements. This explains why the *AC_Fans* program unit, although it hierarchically depends on the *AC_Plan*t, is not related to the other program units.

5.4.7 AC_PumpG

The calculation of the pump group availability is based on the sum total of the availability of the individual pumps.

5.4.7.1 AC_Pump

Availability of the individual pumps is checked by examining the presence of alarms on the pump and giving non-availability if necessary.

5.5 ThermReg

This program unit’s task is to calculate the request for refrigerating power based on the offset between a measured temperature (feedback variable) and a set temperature (set point). The measured temperature can be that of the evaporator inlet water or outlet water.
For plants with several evaporators, the outlet temperature can be taken as the mean value of the outlet temperature of the various evaporators and taken from a single common sensor.

5.5.1 ThermReg

In this program unit, once the presence of the dynamic set point has been checked (see next program unit), the value of the temperature control variable is calculated using a *PI* (or only P) algorithm.

5.5.1.1 DynSet

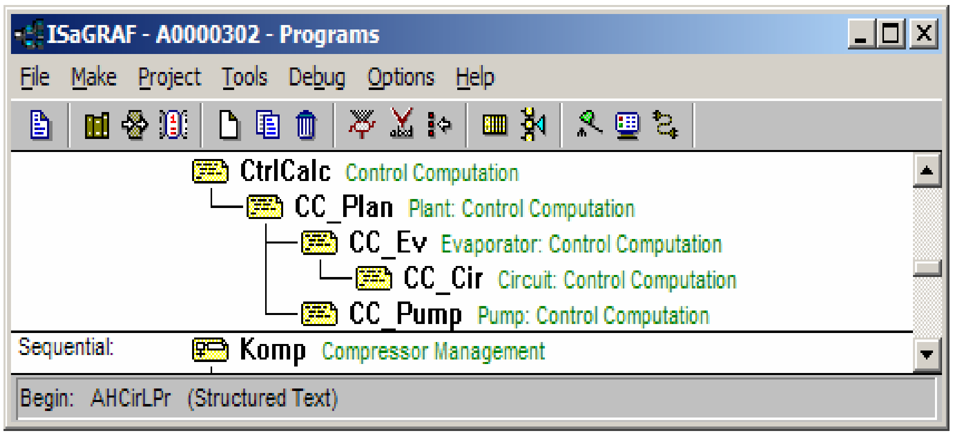
This function modifies the operator set point according to an analogue input signal.
The function calculates a delta that must be added to the control set point and as such must be effected by the temperature controller in order to modify the effective set point on the basis of which a decision is made to switch the evaporators/circuits/compressors on or off.
The function is linearly modulated with selection of the type of dynamic set point to be applied according to the type of sensor that affects the calculation that must be performed for the delta (both temperature and pressure).

5.5.2 Status variables:

TregReqLev	Power required by temperature controller

5.6 CtrlCalc

After calculating availability and temperature controller requirements, an important part of the temperature control process is that of calculating the control of refrigerating resources.


ISaGRAF - A0000302 - Programs
File Make Project Tools Debug Options Help
CtrlCalc Control Computation
CC_Plan Plant: Control Computation
CC_Ev Evaporator: Control Computation
CC_Cir Circuit: Control Computation
CC_Pump Pump: Control Computation
Sequential:
Komp Compressor Management
Begin: AHCirLPr (Structured Text)

Conceptually speaking, the calculation of control is a process that starts with the most important elements of the machine which each inform their “child element” of which resources and how many of them (refrigerating and physical) require implementation.
The plant must therefore inform all the evaporators on the basis of the refrigerating resource selection policy how much power is required.
In turn each evaporator, once the assigned resources have been defined, must inform all the circuits on the basis of the refrigerating resource selection policy how much power it requires.
In the same way, each circuit, once the assigned resources have been defined, must inform all the compressors on the basis of the refrigerating resource selection policy how much power it requires.

5.6.1 Status variables:

The data structures that this program unit operates on are mainly used to maintain the data for the requested power and actual power supplied for the evaporator, circuit and compressor.

PlanReqLev	Power required by plant
PlanOutLev	Power supplied by plant
EvReqLev[4]	Power required by each evaporator
EvOutLev[4]	Power supplied by each evaporator
CirReqLev[8]	Power required by each circuit
CirOutLev[8]	Power supplied by each circuit
KompReqLev[8]	Power required by each compressor
KompOutLev[8]	Power supplied by each compressor

All the vectors are allocated for the maximum number of elements of each type. Where this is not a problem because of memory restrictions, this means that the code can be developed invariantly with regard to the number of evaporators, circuits and compressors.
In this case, if the number of elements changes, the code must be modified. For a code developed in ST language, the modification is limited to changing the numeric value of a parameter whereas for codes developed with graphical languages (FBD / QLD), blocks must be added or removed from the relative program unit.

5.6.2 CC_Plan

The purpose of this program unit is to calculate the power required by the plant according to the temperature control request and plant availability.
This information is then used to implement the policy of assignment of resources to the evaporators.

The calculation is performed from the PolicyCC() function located in the functions area of the AppMaker project. The power actually supplied (PlanOutLev) coincides with PlanReqLev.
It also transfers the plant alarm to its sub-systems.

5.6.3 CC_Ev

The purpose of this program unit is to implement the policy of allocation of resources to the circuits that belong to each evaporator. The power actually supplied (EvOutLev) coincides with EvReqLev.
It also transfers each evaporator's alarm to its sub-systems.

5.6.4 CC_Cir

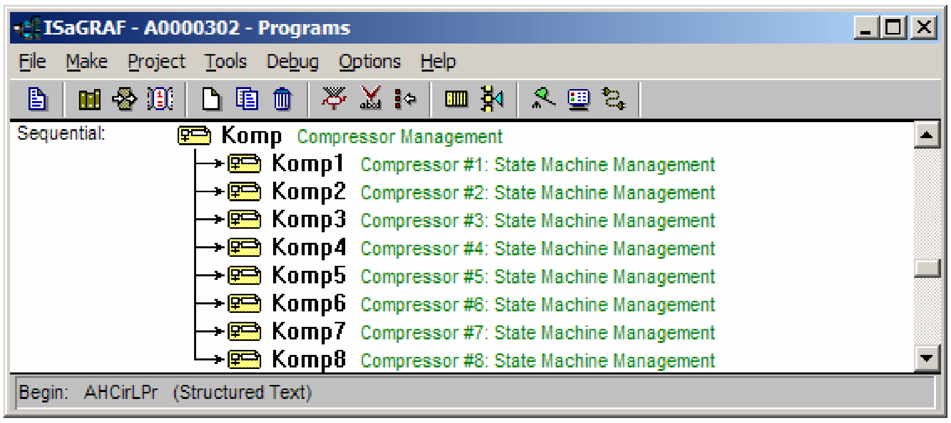
The purpose of this program unit is to implement the policy of allocation of resources to the compressors that belong to each circuit according to the activation state of the pumpdown function. The power actually supplied (CirOutLev) coincides with CirReqLev.
It also transfers each circuit's alarm to its sub-systems.

5.6.5 CC_Pump

The purpose of this program unit is to control the development of the pump group state machine and operate the currently selected pump.

5.6.6 Komp: control

The control of the compressor can by summed up in the same way as a state machine. The control program of the single compressors operates on a set of local variables and a set of global variables through which it interacts with the other

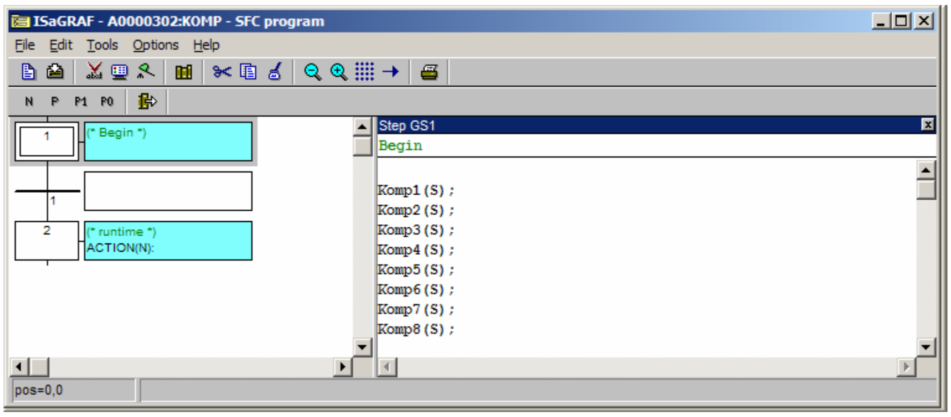


Sequential: Komp Compressor Management
Komp1 Compressor #1: State Machine Management
Komp2 Compressor #2: State Machine Management
Komp3 Compressor #3: State Machine Management
Komp4 Compressor #4: State Machine Management
Komp5 Compressor #5: State Machine Management
Komp6 Compressor #6: State Machine Management
Komp7 Compressor #7: State Machine Management
Komp8 Compressor #8: State Machine Management
Begin: AHCirLPr (Structured Text)

parts of the control.

The goal that has been set is to have a single code that can be cloned for the control of several compressors. To this end, the global variables are allocated on vectors of the same size to number NC of compressors in the plant. The different copies of the compressor control program operate on the set of n index global variables with n ranging from 0 to NC-1; there will therefore be a local variable of the compressor control program that identifies the set of global variables on which it effects the n-th copy of the program.

The automa for the individual compressors are "independent" units. The purpose of the Komp program unit will be to invoke all the automa of the NC compressors.

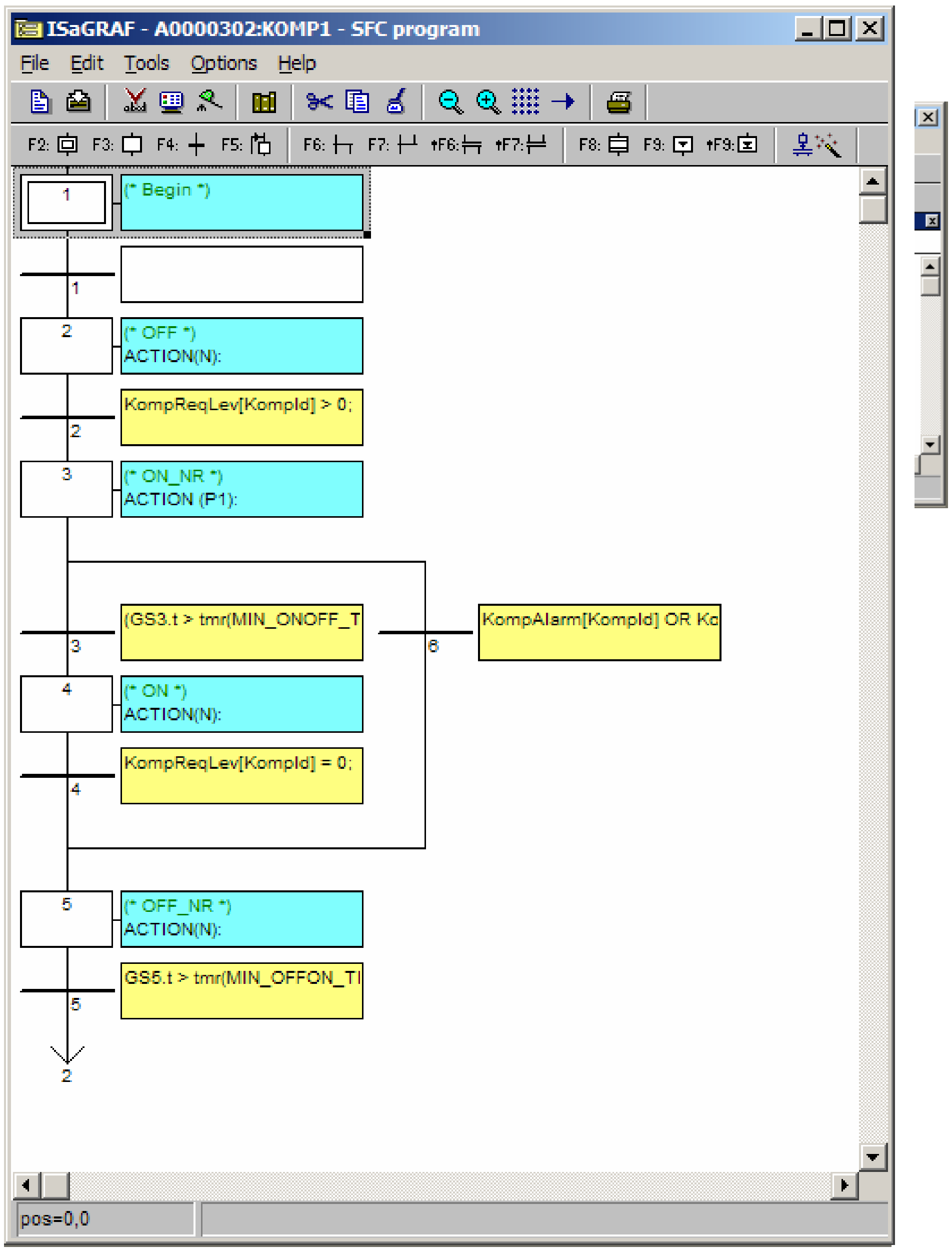


```
1 (* Begin *)
1
2 (* runtime *)
ACTION(N):
```

```
Step GS1
Begin

Komp1 (S) ;
Komp2 (S) ;
Komp3 (S) ;
Komp4 (S) ;
Komp5 (S) ;
Komp6 (S) ;
Komp7 (S) ;
Komp8 (S) ;
```

pos=0,0

Note that the order with which the calls to the KompX(S) automa are inserted in the Komp program unit IS NOT important since all the automa are independent and are executed SIMULTANEOUSLY. Therefore a situation of this kind:

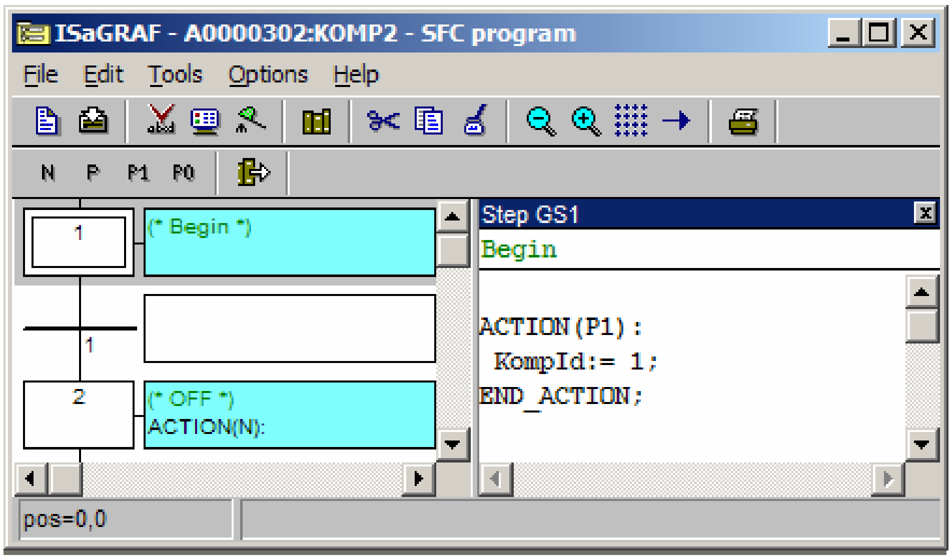


would be THE SAME as the previous one.

Let's take a detailed look at SFC implementation of the compressor state machine:

5.6.6.1 Initial Status

In this status the variables for each compressor are initialized, in particular the index of the data set of global variables and the global variables. The basic program initializes all the local variables at default value.

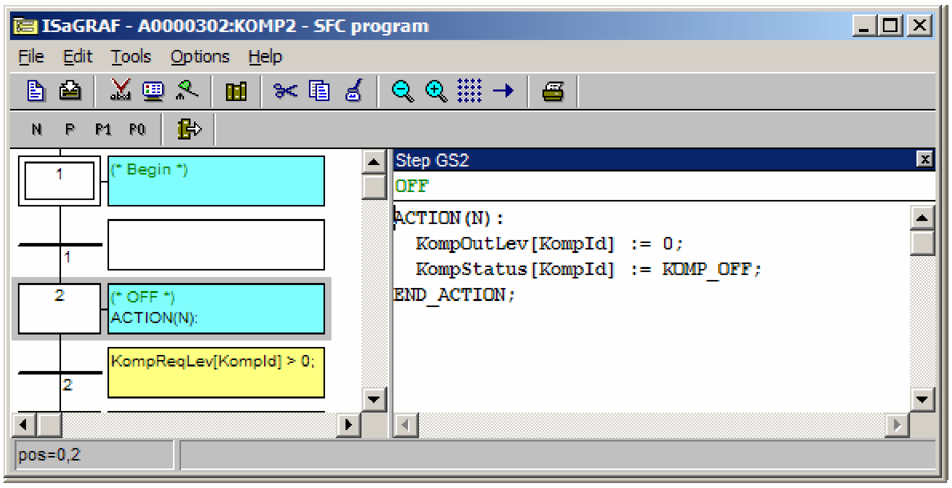

ISaGRAF - A0000302:KOMP2 - SFC program
File Edit Tools Options Help
N P P1 P0
1
(* Begin *)
1
2
(* OFF *)
ACTION(N):
Step GS1
pos=0,0

```
Begin

ACTION(P1):
 KompId:= 1;
END_ACTION;
```

If the structure of the Komp1 block is analysed, you can seen that its “KompId” (dataset) is initialized at “1”.

5.6.6.2 OFF Status

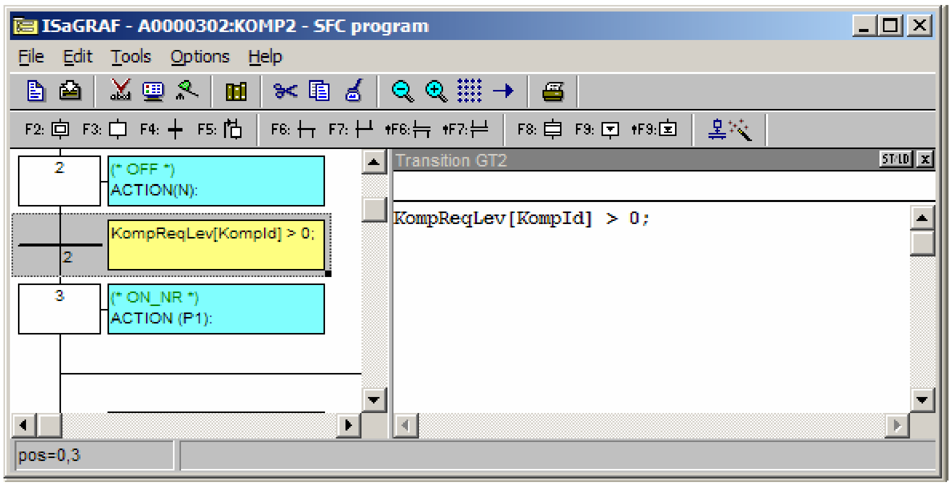
In this status the work variables related to the status and power supplied by each compressor are initialized. To define these variables using indexing the compressor index must already be defined and for cleanliness in writing the code, the index has been defined in the previous block. In this mode, all the OFF blocks will be identical in the different compressors.



```
OFF
ACTION(N):
  KompOutLev[KompId] := 0;
  KompStatus[KompId] := KOMP_OFF;
END_ACTION;
```

5.6.6.3 GT2 transition

The transition from OFF to the subsequent status of ON_NR (*ON NOT Ready*) occurs in the presence of a power request from the compressor

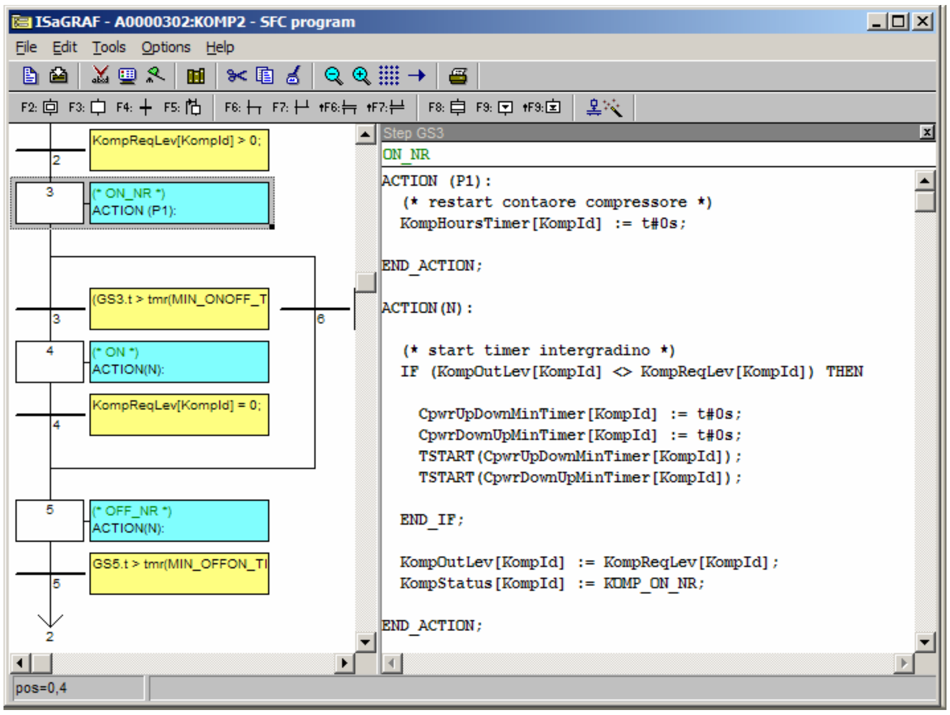

ISaGRAF - A0000302:KOMP2 - SFC program
File Edit Tools Options Help
F2: F3: F4: F5: F6: F7: ↑F6: ↑F7: F8: F9: ↑F9:
2
(* OFF *)
ACTION(N):
2
3
(* ON_NR *)
ACTION (P1):
Transition GT2
ST LD
pos=0,3

KompReqLev[KompId] > 0;

```
KompReqLev[KompId] > 0;
```

5.6.6.4 ON Not Ready

In this phase all the timers related to control of the compressor are initialized and any steps are handled.

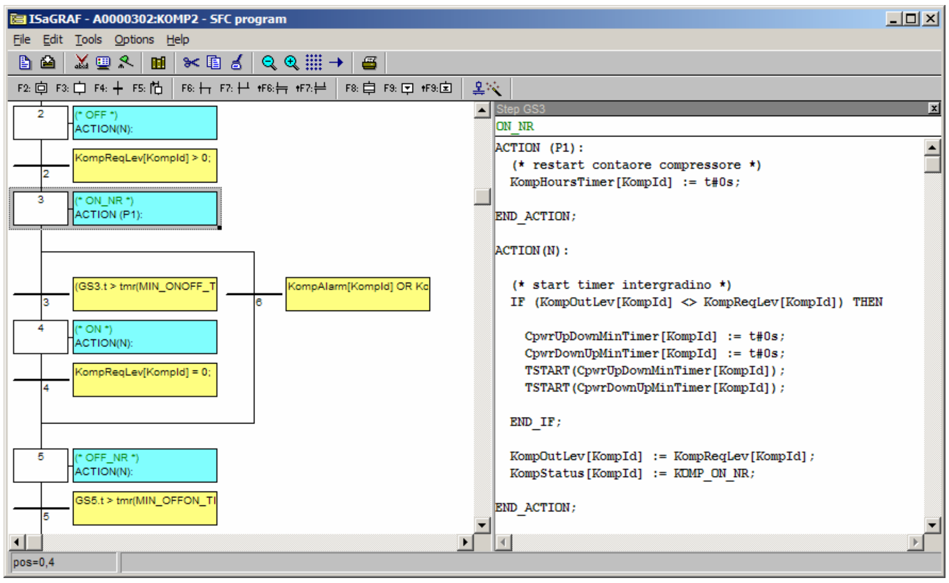


```
ON_NR
ACTION (P1):
  (* restart contaore compressore *)
  KompHoursTimer[KompId] := t#0s;

END_ACTION;

ACTION(N):

  (* start timer intergradino *)
  IF (KompOutLev[KompId] <> KompReqLev[KompId]) THEN

    CpwrUpDownMinTimer[KompId] := t#0s;
    CpwrDownUpMinTimer[KompId] := t#0s;
    TSTART(CpwrUpDownMinTimer[KompId]);
    TSTART(CpwrDownUpMinTimer[KompId]);

  END_IF;

  KompOutLev[KompId] := KompReqLev[KompId];
  KompStatus[KompId] := KOMP_ON_NR;

END_ACTION;
```

If these times are configured (and their control enabled) the compressor switched on at partial power for a period of time equal to the maximum time at reduced power is taken to maximum power observing the minimum step up times beyond control of the temperature controller.
Once maximum power has been reached, it remains in this status beyond control of the temperature controller for a period equal to the minimum time at maximum power.
Once this period has elapsed, the compressor returns under the control of the temperature controller.
Special attention must be paid to the possibility of assigning and therefore controlling the value 0 for some times.
In principle, there is no problem when a compressor is immediately switched on again once it has been switched off (min toff = 0) even when treating the period 0 as the cycle time of the PLC. If this simplification leads to undesired behaviour, double transitions for t <> 0 and t = 0 must be provided.

5.6.6.5 Step GS3

Here, the active compressor timer is reset and restarted and whether a step situation has to be handled or not is assessed.

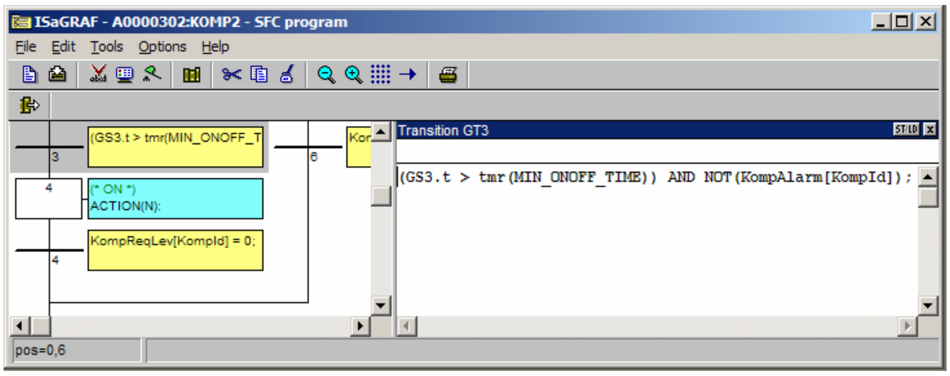


```
ON_NR
ACTION (P1):
  (* restart contaore compressore *)
  KompHoursTimer[KompId] := t#0s;

END_ACTION;

ACTION(N):

  (* start timer intergradino *)
  IF (KompOutLev[KompId] <> KompReqLev[KompId]) THEN

    CpwrUpDownMinTimer[KompId] := t#0s;
    CpwrDownUpMinTimer[KompId] := t#0s;
    TSTART(CpwrUpDownMinTimer[KompId]);
    TSTART(CpwrDownUpMinTimer[KompId]);

  END_IF;

  KompOutLev[KompId] := KompReqLev[KompId];
  KompStatus[KompId] := KOMP_ON_NR;

END_ACTION;
```

Once this point has been reached, two potential transitions must be handled:

5.6.6.6 Transition GT3

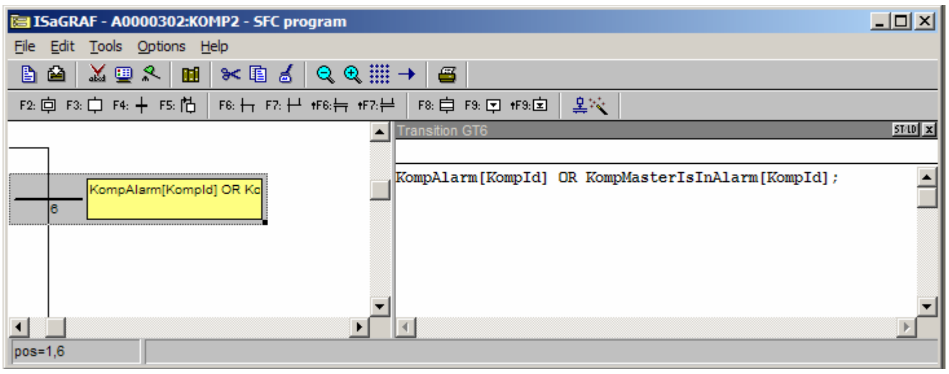
Once the compressor protection time has elapsed and there are no alarm situations, it can go on to the *ON status*.



```
(GS3.t > tmr(MIN_ONOFF_TIME)) AND NOT(KompAlarm[KompId]);
```

5.6.6.7 Transition GT6

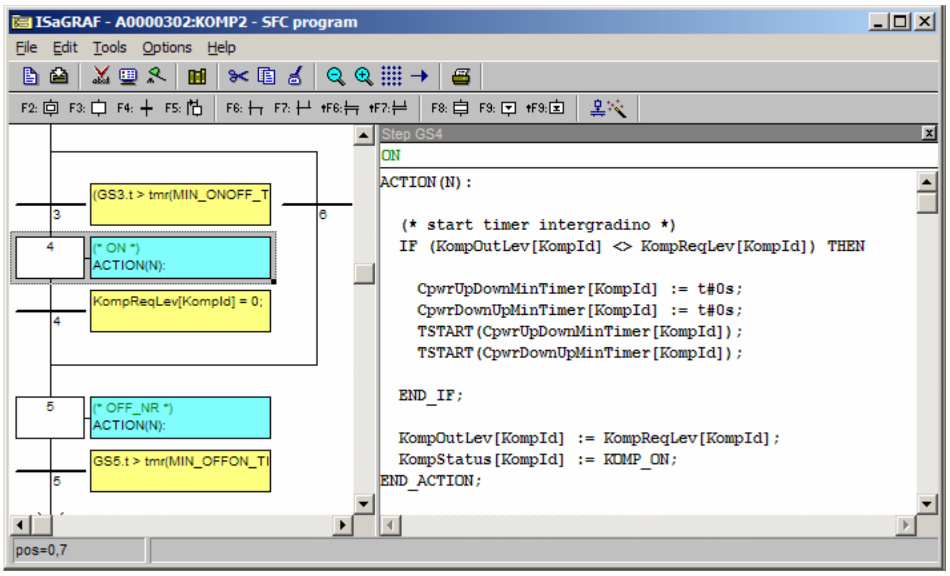
This transition is used to handle any alarm situations of the specific compressor. Whenever there is an alarm situation for the KompId compressor (that handled by the special automa) it goes immediately to OFF_NR.


ISaGRAF - A0000302:KOMP2 - SFC program
File Edit Tools Options Help
KompAlarm[KompId] OR Ko
6
Transition GT6
ST LD
pos=1,6

```
KompAlarm[KompId] OR KompMasterIsInAlarm[KompId];
```

5.6.6.8 ON Status

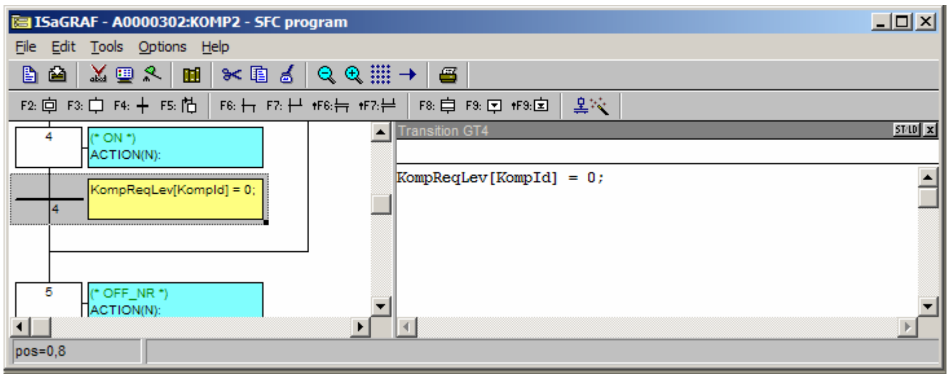
The *ON status* corresponds to the status in which the compressor operates normally. It involves checking if the power supplied corresponds to the power required and if this is not the case, after updating all the timers, updating the supplied power.



```
ACTION(N):

  (* start timer intergradino *)
  IF (KompOutLev[KompId] <> KompReqLev[KompId]) THEN

    CpwrUpDownMinTimer[KompId] := t#0s;
    CpwrDownUpMinTimer[KompId] := t#0s;
    TSTART(CpwrUpDownMinTimer[KompId]);
    TSTART(CpwrDownUpMinTimer[KompId]);

  END_IF;

  KompOutLev[KompId] := KompReqLev[KompId];
  KompStatus[KompId] := KOMP_ON;
END_ACTION;
```

5.6.6.9 Transition GT4

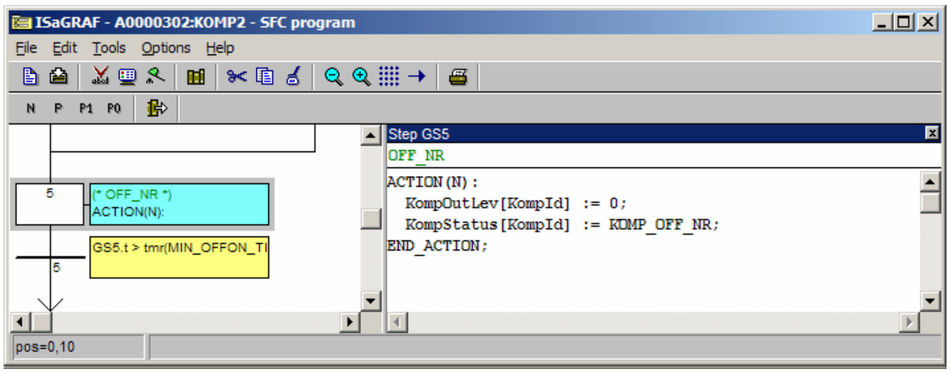
This transition takes place when the control request is equal to zero and the compressor asks to be switched off.


ISaGRAF - A0000302:KOMP2 - SFC program
File Edit Tools Options Help
F2: F3: F4: F5: F6: F7: ↑F6: ↑F7: F8: F9: ↑F9:
4
(* ON *)
ACTION(N):
KompReqLev[KompId] = 0;
4
5
(* OFF_NR *)
ACTION(N):
pos=0,8
Transition GT4
ST LD

```
KompReqLev[KompId] = 0;
```

5.6.6.10 OFF Not Ready

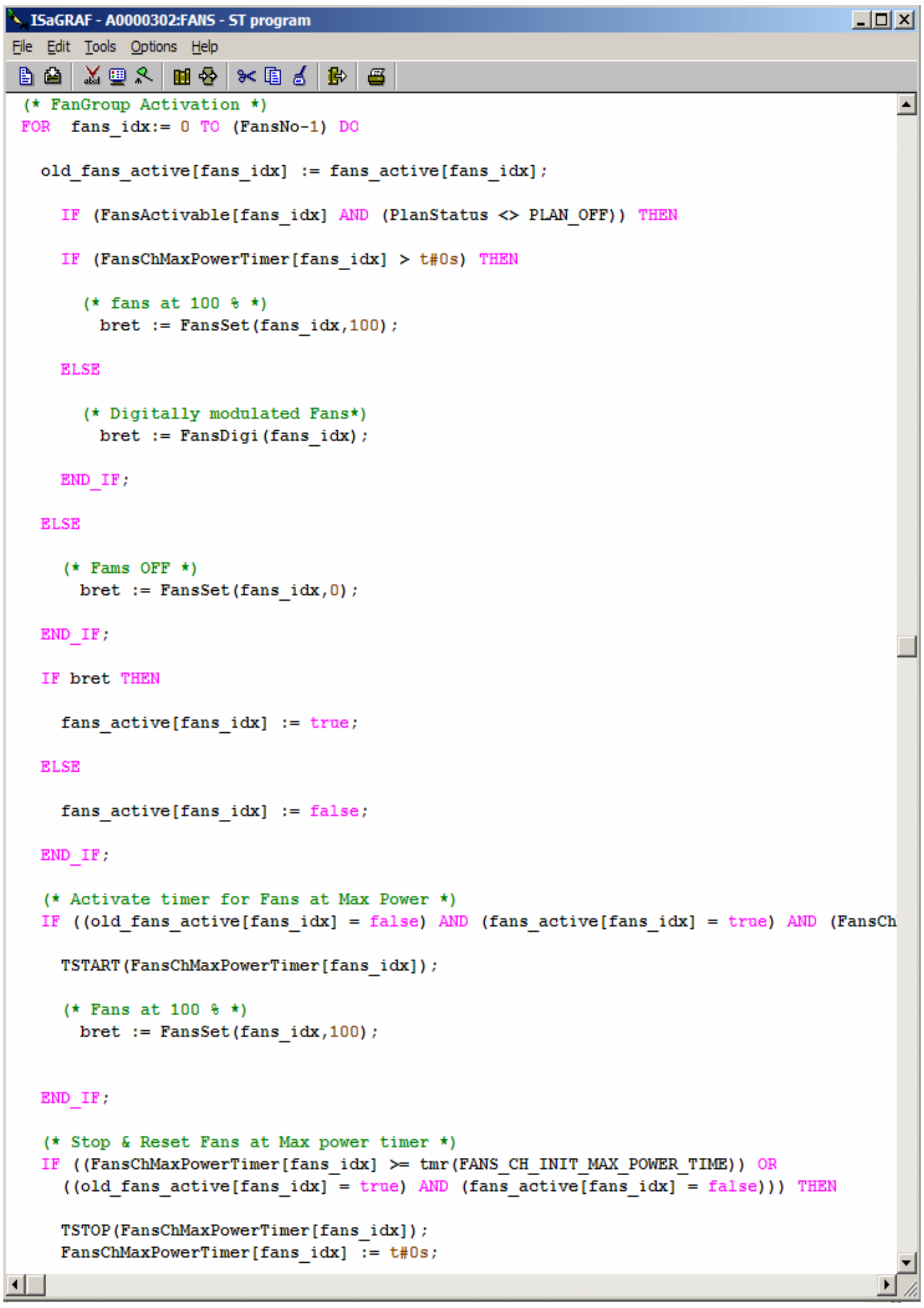
The compressor is switched off, its power taken to zero and its status set to *OFF not ready*. The protection times must be observed before it can be switched on again.



```
ACTION(N):
  KompOutLev[KompId] := 0;
  KompStatus[KompId] := KOMP_OFF_NR;
END_ACTION;
```

5.7 Fans (Single condensation)

This program unit controls the condensation and fans.

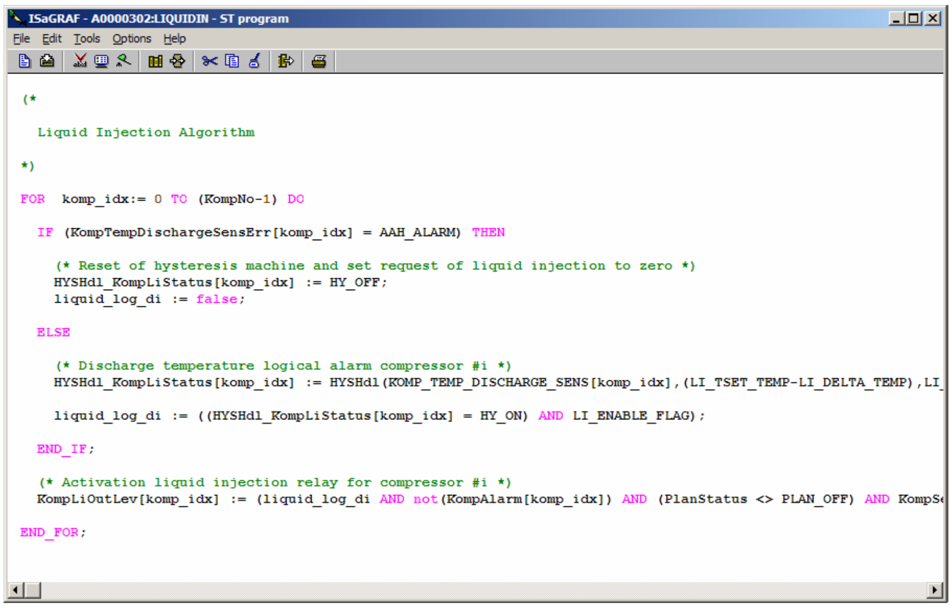

ISaGRAF - A0000302:FANS - ST program
File Edit Tools Options Help

```
(* FanGroup Activation *)
FOR  fans_idx:= 0 TO (FansNo-1) DO

  old_fans_active[fans_idx] := fans_active[fans_idx];

    IF (FansActivable[fans_idx] AND (PlanStatus <> PLAN_OFF)) THEN

    IF (FansChMaxPowerTimer[fans_idx] > t#0s) THEN

      (* fans at 100 % *)
       bret := FansSet(fans_idx,100);

    ELSE

      (* Digitally modulated Fans*)
       bret := FansDigi(fans_idx);

    END_IF;

  ELSE

    (* Fams OFF *)
     bret := FansSet(fans_idx,0);

  END_IF;

  IF bret THEN

    fans_active[fans_idx] := true;

  ELSE

    fans_active[fans_idx] := false;

  END_IF;

  (* Activate timer for Fans at Max Power *)
  IF ((old_fans_active[fans_idx] = false) AND (fans_active[fans_idx] = true) AND (FansCh

    TSTART(FansChMaxPowerTimer[fans_idx]);

    (* Fans at 100 % *)
     bret := FansSet(fans_idx,100);


  END_IF;

  (* Stop & Reset Fans at Max power timer *)
  IF ((FansChMaxPowerTimer[fans_idx] >= tmr(FANS_CH_INIT_MAX_POWER_TIME)) OR
    ((old_fans_active[fans_idx] = true) AND (fans_active[fans_idx] = false))) THEN

    TSTOP(FansChMaxPowerTimer[fans_idx]);
    FansChMaxPowerTimer[fans_idx] := t#0s;
```

Control of single condensation can be seen as a high level function that affects the calculation of the behaviour of the fans.

5.8 Liquid Injection

This feature is used for cooling control on the compressor discharge temperature:


ISaGRAF - A0000302:LIQUIDIN - ST program
File Edit Tools Options Help

```
(*

  Liquid Injection Algorithm

*)

FOR  komp_idx:= 0 TO (KompNo-1) DO

  IF (KompTempDischargeSensErr[komp_idx] = AAH_ALARM) THEN

    (* Reset of hysteresis machine and set request of liquid injection to zero *)
    HYSHdl_KompLiStatus[komp_idx] := HY_OFF;
    liquid_log_di := false;

  ELSE

    (* Discharge temperature logical alarm compressor #i *)
    HYSHdl_KompLiStatus[komp_idx] := HYSHdl(KOMP_TEMP_DISCHARGE_SENS[komp_idx],(LI_TSET_TEMP-LI_DELTA_TEMP),LI_

    liquid_log_di := ((HYSHdl_KompLiStatus[komp_idx] = HY_ON) AND LI_ENABLE_FLAG);

  END_IF;

  (* Activation liquid injection relay for compressor #i *)
  KompLiOutLev[komp_idx] := (liquid_log_di AND not(KompAlarm[komp_idx]) AND (PlanStatus <> PLAN_OFF) AND KompSe

END_FOR;
```

It is to all effects a function that can be called during control and as such can be seen as an independent program unit.

5.9 Log2Phy

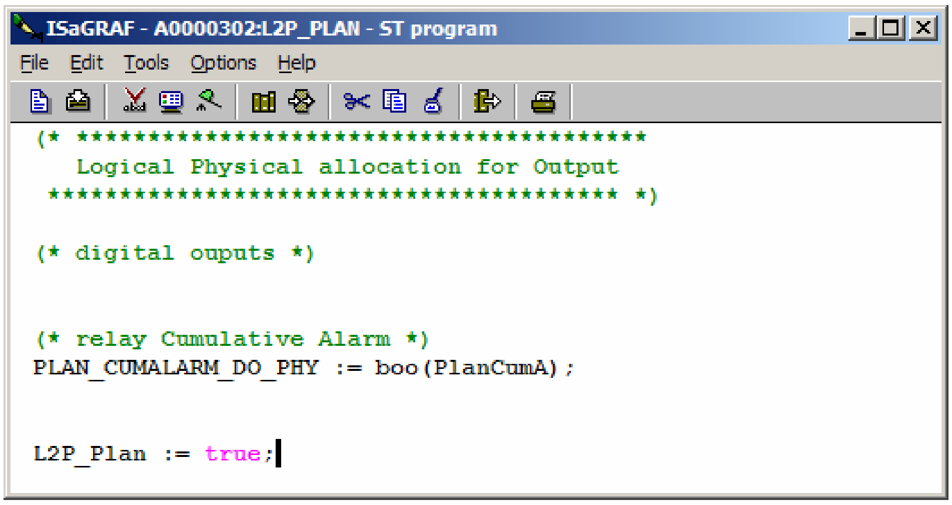
This program unit performs the task of converting the logical variables calculated by the control into physical output variables. Once again, we have decided to divide this program unit into sub-program units each one related to a specific area of the machine or type of component. This decision has been made to meet the need for program unit modularity and readability rather than any practical requirement.
An important point is the level of abstraction and generalization that has to be encapsulated in the program unit that must balance the need for program units and function blocks that are for general use (therefore more complex) and at the same time are simple and efficient to use.
Consider, for example, the program unit for implementation of compressors this can be sized for the maximum number of compressors that can be handled by the maximum application or resized for their effective number (for the BASELINE APPLICATION). The general program unit could have the implementation parts of the 4 compressors not present disabled by the presence flags of the relative compressors but this requires a larger application and longer execution time. In any case, the program unit resized for 4 compressors should be modified to handle a different number of compressors.

The second point concerns the library Function Blocks that make the outputs operative. Here too, I could decide to have the “maximum” FB that can handle all types of compressor (on/off, 1, 2, 3 capacity steps, analogue) and different FBs each one specifically for a different type of compressor. In the same way, start-up problems (simple, star/delta, part winding) can be handled in a general use FB or FBS to be specifically used for different types of compressor). The general use FB has the advantage of not requiring modification of the application to handle different compressors and/or different start-up modes. A disadvantage is that it is much more complex and less efficient in terms of execution. The solution with specialized blocks has both advantages and disadvantages. The FBs are more simple and efficient but require modification of the application to handle different types of compressor.

In the BASELINE APPLICATION, we decided to adopt the solution that implements “maximum” FBs that can handle several types of physical device (compressors, fans, etc.) and leave the alternative solutions for use in future projects.

5.9.1 L2P_Plan

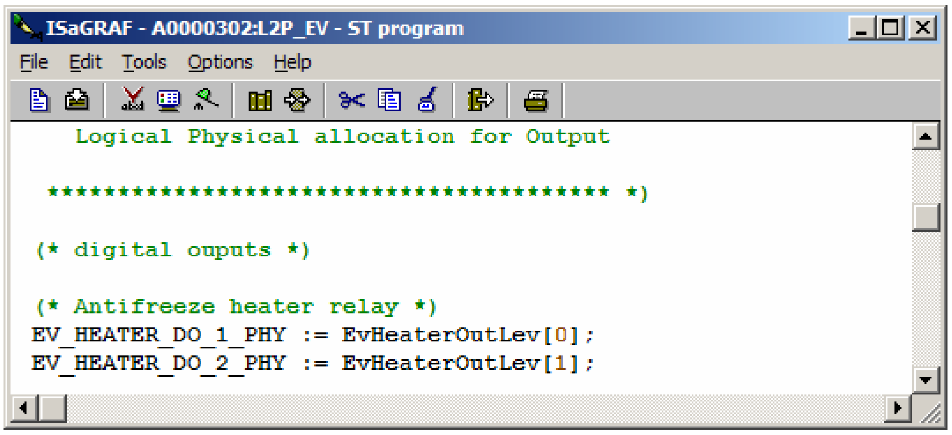
In the chiller BASELINE APPLICATION, the only physical output related to the plant only is the general plant alarm that activates a control relay of a visual or acoustic indicator. In this way, the physical output will coincide with the logical OR of the plant alarms.


ISaGRAF - A0000302:L2P_PLAN - ST program
File Edit Tools Options Help

```
(* ****************************************
   Logical Physical allocation for Output
 **************************************** *)

(* digital ouputs *)


(* relay Cumulative Alarm *)
PLAN_CUMALARM_DO_PHY := boo(PlanCumA);


L2P_Plan := true;
```

5.9.2 L2P_Ev

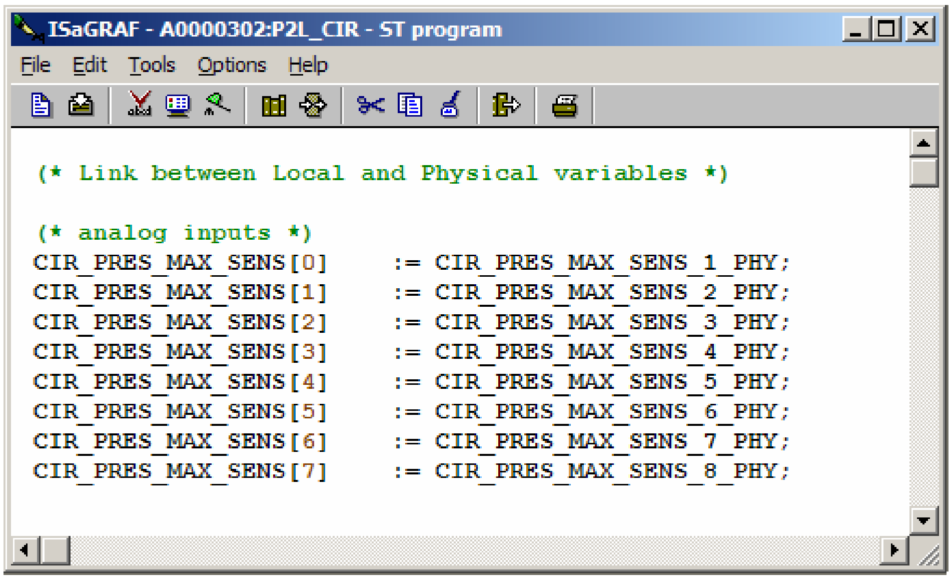
The only two evaporator variables that must be converted at physical level are the two antifreeze heaters that are two elements of a vector in the AppMaker program. Each vector element (that can only be internal variables) must therefore be assigned to two I/O static variables:


ISaGRAF - A0000302:L2P_EV - ST program
File Edit Tools Options Help

```
   Logical Physical allocation for Output

 **************************************** *)

(* digital ouputs *)

(* Antifreeze heater relay *)
EV_HEATER_DO_1_PHY := EvHeaterOutLev[0];
EV_HEATER_DO_2_PHY := EvHeaterOutLev[1];
```

5.9.3 L2P_Cir

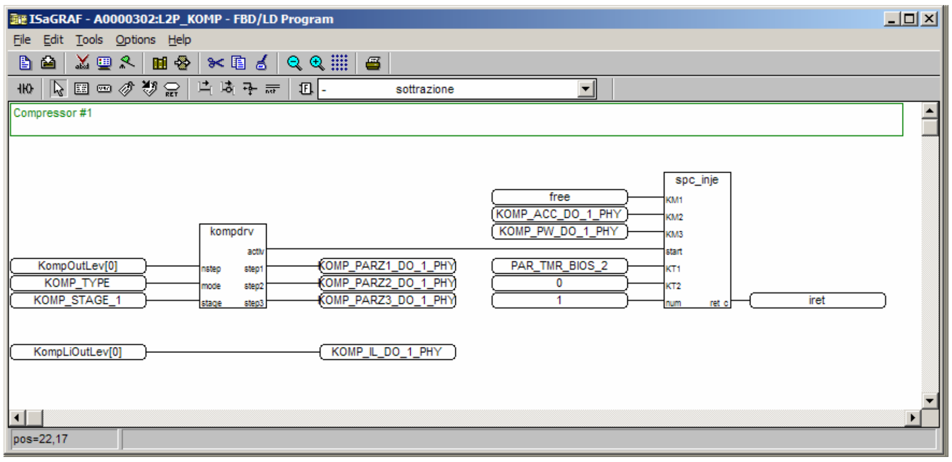
The circuit also has physical elements, solenoid valves, that belong to it. Since these are also handled as internal vector elements, they must be assigned to I/O static variables so that they can be connected to the I/O rack that determines the physical allocation of inputs and outputs.


ISaGRAF - A0000302:P2L_CIR - ST program
File Edit Tools Options Help

```
(* Link between Local and Physical variables *)

(* analog inputs *)
CIR_PRES_MAX_SENS[0]    := CIR_PRES_MAX_SENS_1_PHY;
CIR_PRES_MAX_SENS[1]    := CIR_PRES_MAX_SENS_2_PHY;
CIR_PRES_MAX_SENS[2]    := CIR_PRES_MAX_SENS_3_PHY;
CIR_PRES_MAX_SENS[3]    := CIR_PRES_MAX_SENS_4_PHY;
CIR_PRES_MAX_SENS[4]    := CIR_PRES_MAX_SENS_5_PHY;
CIR_PRES_MAX_SENS[5]    := CIR_PRES_MAX_SENS_6_PHY;
CIR_PRES_MAX_SENS[6]    := CIR_PRES_MAX_SENS_7_PHY;
CIR_PRES_MAX_SENS[7]    := CIR_PRES_MAX_SENS_8_PHY;
```

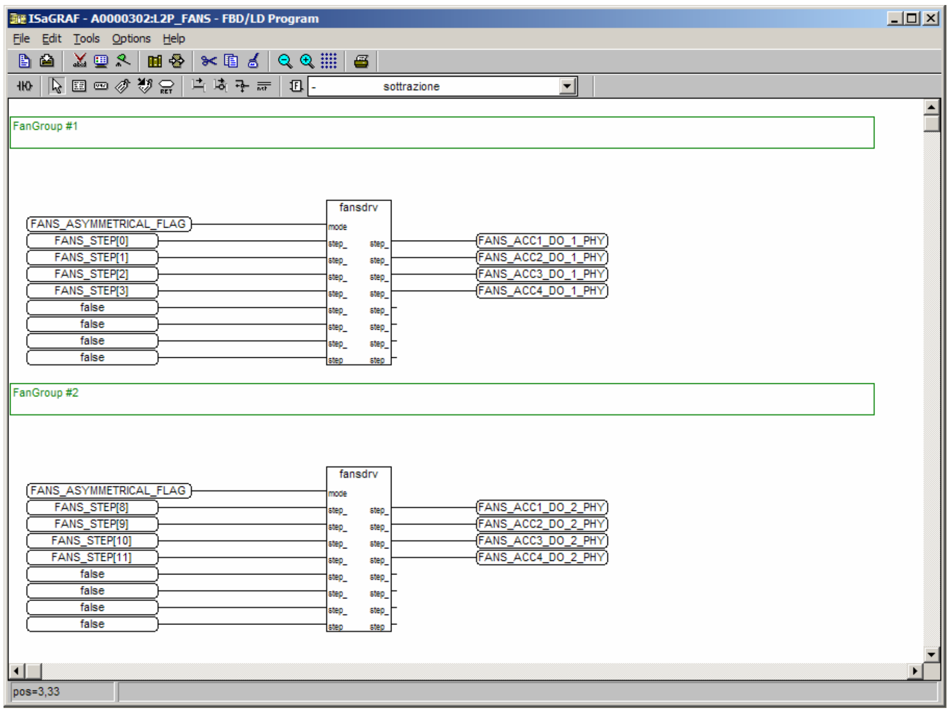
5.9.4 L2P_Komp

Handling of the compressors has been developed using FBD language. This code must be repeated as many times as the maximum number of compressors physically present in the plant occurs.
As you can see in the figure that shows the logical physical conversion of a compressor, the *KompDrv* function is used that, as we will see in the next chapter, represents the driver device of the compressor. As already stated at the beginning of this document, a programmable device offers the possibility of having components such as compressors that are not uniform in the same plant. This means that compressors that not only have different power but also different start-up/control modes can exist in the same plant.
If the "physical nature" of the start-up of a compressor remains unchanged (or n relays are always necessary to start it up) but the sequence to use on the relays changes, intervention must occur on the driver device function as described further on. If, on the other hand, the use of a new compressor requires a different number of I/Os for its control, a new device driver must be implemented in the function area and the program unit *L2P_KOMP* modified since a different number of I/Os will be used.



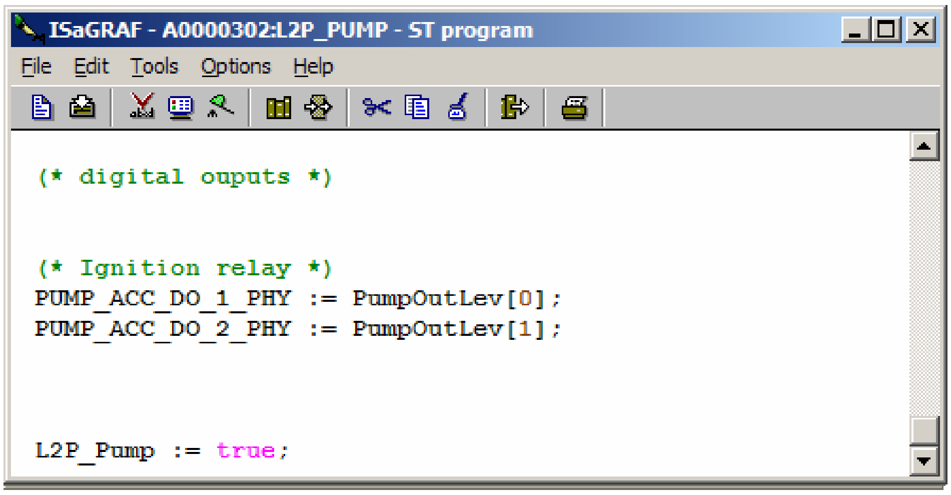
5.9.5 L2P_Fans

As for the compressors, conversion from logic to physical for fans has also been handled using a FBD program unit. Here too, non-uniform fans and different types of driver can be used. The same considerations made for the compressors therefore remain valid.
The figure below shows that the DB code must be repeated as many times as the maximum number of fan groups present in the plant occurs.



5.9.6 L2P_Pump

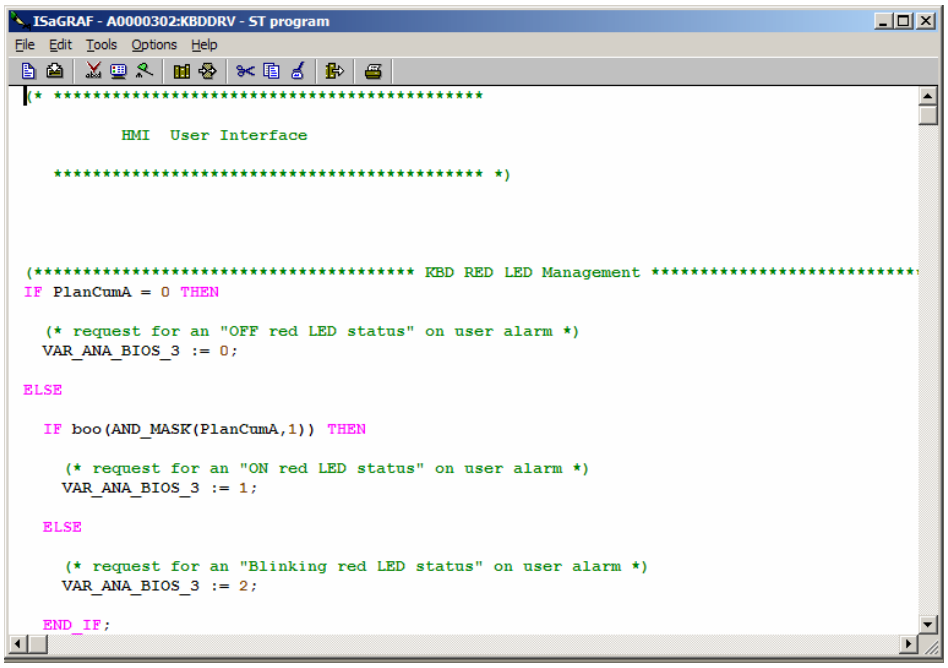
Two pumps have been inserted in the BASELINE APPLICATION whose control signal is contained in the PumpOutLev[] vectors. The values of this vector must therefore be copied into two I/O static variables.


ISaGRAF - A0000302:L2P_PUMP - ST program
File Edit Tools Options Help

```
(* digital ouputs *)


(* Ignition relay *)
PUMP_ACC_DO_1_PHY := PumpOutLev[0];
PUMP_ACC_DO_2_PHY := PumpOutLev[1];



L2P_Pump := true;
```

5.10 KbdDrv

This program unit's task is to produce the information that the XTK Pro keyboard must display on the Baseline Application. Although navigation is programmed and handled using the MenuMaker PRO program, the control of the red LED (normally used to signal the presence of an alarm) and the production of display data occurs in this part of the code.


ISaGRAF - A0000302:KBDDRV - ST program
File Edit Tools Options Help

```
(* ********************************************

         HMI  User Interface

  ******************************************** *)



(*************************************** KBD RED LED Management ***************************
IF PlanCumA = 0 THEN

  (* request for an "OFF red LED status" on user alarm *)
  VAR_ANA_BIOS_3 := 0;

ELSE

  IF boo(AND_MASK(PlanCumA,1)) THEN

    (* request for an "ON red LED status" on user alarm *)
    VAR_ANA_BIOS_3 := 1;

  ELSE

    (* request for an "Blinking red LED status" on user alarm *)
    VAR_ANA_BIOS_3 := 2;

  END_IF;
```

6 FUNCTION BLOCKS SPECIFICATION

The functional block library is in the last part of the A0000301 and A0001301 project. It contains all the library functions. These include the most widely used ones such as the Bypass Handler (general or not) and the Bounded Alarm Handler. If any type of new driver has to be implemented (for compressors or fans) this will be inserted in this part of the project.

6.1 GBYPHdl function

This is a function developed to implement a General Bypass that handles the off <---> bypass_offon ---> on <---> bypass_onoff ---> off transactions regulated by a timer. This function is widely used in alarm handling. In particular, a general bypass is used (parametrized on a time T) so that the alarm becomes enabled once it has been present without interruptions for at least a time T. (off->on transaction). The condition that generated the alarm must no longer be present for at least the time "**T**" in order for the alarm to disappear.

6.2 BYPHdl function

The Bypass function is different from the General Bypass function because the on/off transaction occurs instantly unlike the off/on function. This type of bypass will therefore be used if you want the alarm to appear once the alarm condition has been enabled for at least a time T (parameter-fixed) but as soon as the condition that generates the alarm disappears, the bypass will immediately switch to off.

6.3 BAHHand function

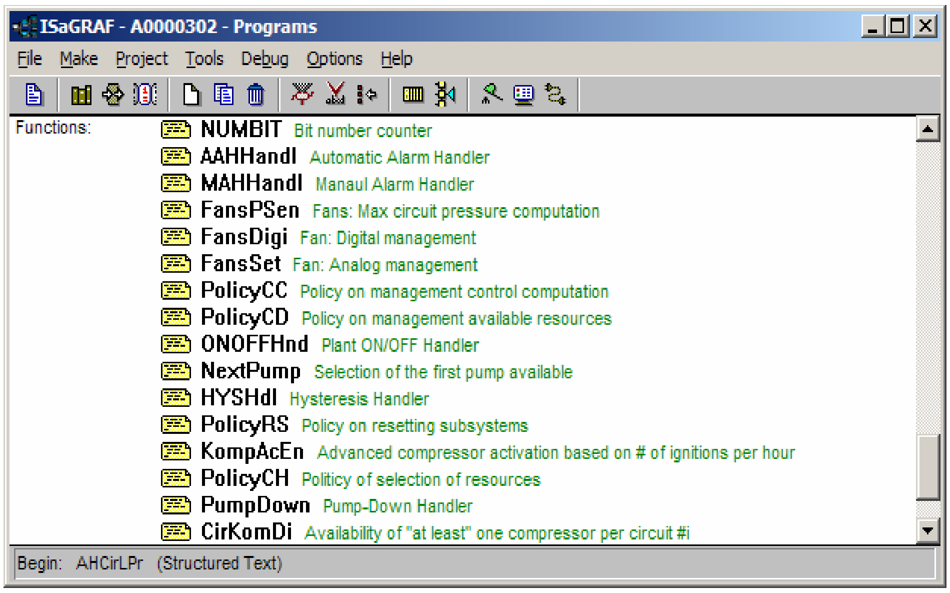
This function, also useful for handling alarms like the others, gives an "active" output once in a parameter-definable time window at least a quantity Q of alarm events is present.
The output only passes to a "non-active" condition via operator intervention (manual reset)

6.4 BENHandl function

This function also controls the alarm system. In this case, output activation occurs when there are at least a certain number of events in a time window.

7 FUNCTION SPECIFICATION

Functions commonly used in the program are fund in the Functions section of the A0000301 and A0001301 project.


ISaGRAF - A0000302 - Programs
File Make Project Tools Debug Options Help

Functions:

NUMBIT	Bit number counter
AAHHandl	Automatic Alarm Handler
MAHHandl	Manaul Alarm Handler
FansPSen	Fans: Max circuit pressure computation
FansDigi	Fan: Digital management
FansSet	Fan: Analog management
PolicyCC	Policy on management control computation
PolicyCD	Policy on management available resources
ONOFFHnd	Plant ON/OFF Handler
NextPump	Selection of the first pump available
HYSHdl	Hysteresis Handler
PolicyRS	Policy on resetting subsystems
KompAcEn	Advanced compressor activation based on # of ignitions per hour
PolicyCH	Politicy of selection of resources
PumpDown	Pump-Down Handler
CirKomDi	Availability of "at least" one compressor per circuit #i

Begin: AHCirLPr (Structured Text)

NUMBIT	It counts the number of one-bits in a 32-bit integer. It is used in BAHHANDL.
AAHHandl	Automatic alarm handling
MAHHandl	Manual alarm handling
FansPSen	It supplies the value of the maximum pressure sensor of the specified fan unit.
FansDigi	Digital control of the fans of the selected fan unit
FansSet	Set of all the fans of the selected fan group
PolicyCC	Calculation policy of system control. This function is password-protected for reading and modification
PolicyCD	Calculation policy of system availability. This function is password-protected for reading and modification
PolicyRS	Reset policy of system control. This function is password-protected for reading and modification
PolicyCH	Hourly system resources selection policy. This function is password-protected for reading and modification
Pumpdown	Pumpdown state machine control of selected circuit. This function is password-protected for reading and modification
ONOFFNHD	Handling of ON/OFF request of plant
NextPump	Function for search for next available pump
HYSHDL	Handling of state machine hysteresis
KompAcEn	Enabling of compressor activation for number of start-ups per hour
CirKomDi	This checks the availability of at least one compressor in the selected circuit.

8 LIBRARY FUNCTION BLOCKS SPECIFICATION

8.1 KompDrv

This is the driver that controls the capacity steps (stages) of a “step controlled” SEMI-HERMETIC or SCREW compressor according to the specifications indicated below:
As described in the program unit technical notes, the parameters it operates on are the type of compressor, the maximum number of capacity steps and the number of steps to be implemented.
The output corresponds to the start-up status and the capacity step “relay”.
Since two types of compressor are implemented in this program unit and there is one variable that identifies the type of compressor among the input variables, machines can have non-uniform types of compressor. The calculation of the refrigerating power goes beyond the type of compressor used (although it affects the calculation of resources). It therefore involves invoking the right driver for each compressor (considering that it is a part-loading compressor).

8.2 FansDrv

This driver controls the fans in SYMMETRICAL or ASYMMETRICAL mode. The fan device driver follows the same logics as those previously mentioned for compressors. It is a function that when it receives the quantity of ventilating power to be supplied and the mode as input, orders the relative outputs to switch on the correct number of fans.
If, for example, you want to switch on the fans using combinatorial logic rather than sequential logic, the code modifications will be localized in this point.

8.3 PI

This implements a *PI* controller in which once the set point, maximum integral time proportional band and sampling time (with other parameters) values are input, they can independently calculate the Ki and Kp quantities and perform a *PI* control producing an output (control) expressed in an integer between 0 and 1000.

Note that this program unit is protected by a special password reserved for each user. This password is essential for the compilation of the program unit and display of the interface/parameters. Remember that this program unit must always be compiled when it is first imported into AppMaker so that it can be recalled in any project.

9 USE OF THE DEVICE

9.1 Permitted Use

This unit is used to control small, medium and large sized chillers with 1 to 8 compressors and circuits.

For safety purposes, the control device must be installed and used in accordance with the instructions supplied. Users must not be able to access parts with dangerous voltage levels under normal operating conditions. The unit must be resistant to water and dust, depending on the specific application, and be accessible only by using special tools. This unit can be fitted on domestic appliances and/or similar units used for air conditioning.
In accordance with the reference standards, this unit is classified:

- as an automatic electronic control device to be installed in a standalone configuration or on other units with regard to manufacturing;
- As a Type 1 control unit in relation to its manufacturing tolerances and derivatives with regard to its automatic operating characteristics;
- As a Class 2 device with regard to protection against electric shocks (referring to the parts that can be accessed during normal use: front keypad);
- As a Class A device with regard to software class and structure

9.2 Unpermitted Use

The use of the unit for applications other than those described is forbidden.
Please note that the relay contacts supplied are functional and may be subject to failure (since the electronics controlling them may short circuit these relays or leave them open). For this reason, any protection devices needed to comply with product requirements or dictated by common sense due to obvious safety reasons should be installed externally.

10 RESPONSIBILITY AND RESIDUAL RISKS

Eliwell & Controlli s.r.l. shall not be liable for any damages deriving from:

- installation/use other than that prescribed which does not comply with the safety standards specified in the regulations and/or herein;
- use on equipment that does not guarantee adequate protection against electric shock, water or dust when assembled.
- use on equipment that allows dangerous parts to be accessed without the use of tools;
- Installation/use on equipment that is not compliant with the standards and regulations in force.

11 DISCLAIMER

12 ANALITIC INDEX

Eliwell & Controlli s.r.l.
Via dell'Industria, 15 Zona Industriale Paludi
32010 Pieve d'Alpago (BL) ITALY
Telephone +39 0437 986111
Facsimile +39 0437 989066
Internet http://www.eliwell.it

Technical Customer Support:
Telephone +39 0437 986300
Email: techsuppeliwell@invensys.com

Invensys Controls Europe
An Invensys Company

CERTIFIED QUALITY SYSTEM UNI EN ISO 9001/2000 SINCERT