Energy XT Pro
ADD Chiller / Heat Pump Applications

CONTENTS

1 Use of the Manual ... 5
2 Introduction to AppMaker ... 6
2.1 AppMaker potential and constraints ... 6
2.1.1 Limit of development philosophy of the Applications ... 6
2.2 Use of languages IEC-61131-3 ... 6
2.3 Libraries and function blocks ... 6
2.4 Possibilities of application parameterisation ... 6
2.5 AppMaker application implementation possibilities ... 7
2.6 Using the Arrays ... 8
2.7 Design criteria and system optimisation ... 8
3 General architecture of the reversible heat pump application ... 10
3.1 Introduction ... 10
3.2 Main principles of architecture ... 12
3.3 General structure ... 13
3.3.1 Begin Section ... 13
3.3.2 Sequential section ... 14
3.3.3 End section ... 15
3.3.4 Functions and Function Blocks ... 15
3.3.5 Procedure for modifying the baseline procedure ... 16
4 Data dictionary of the reversible heat pump application ... 17
4.1 Nomenclature ... 17
4.2 Control structure definition ... 18
4.3 Global and support variables ... 20
4.3.1 Vector KOMP_CIR_EV ... 20
4.3.2 Vector CirPresence ... 21
4.3.3 Vector CirEv ... 22
4.3.4 Vector CirKomp ... 22
4.3.5 Vector EvPresence ... 24
4.3.6 1.3.6 Vector EvCir ... 24
4.3.7 Vector KOMP_STEP ... 25
4.3.8 Vector KompCir ... 26
4.3.9 KompFans (Part of IV_Komp) ... 26
4.3.10 Variables EvNo, CirNo, KompNo ... 26
4.3.11 Checks of consistency and cycles on objects ... 27
4.3.12 Parameter FANS_NO (Part of IV_Fans) ... 31
4.3.13 CIR_FANS (Part of IV_Plant) ... 32
4.3.14 1.3.14 FansCir (Part of IV_Fans) ... 32
4.3.15 1.3.15 FansNo (Part of IV_Fans) ... 34
4.3.16 Parameter PUMP_NO ... 34
4.4 Constants (Defined words) ... 34
5 Description of the baseline reversible heat pump application ... 37
5.1 IniVar ... 37
5.1.1 CheckCon ... 37
5.1.2 IV_Plan ... 38
5.1.3 IV_Ev ... 38
5.1.4 IV_Cir ... 38
5.1.5 IV_Komp ... 38
5.1.6 IV_Fans ... 38
5.1.7 IV_Pump ... 38
5.1.8 IV_Def ... 38
5.2 Phy2Log ... 38
5.2.1 P2L_xxx ... 38
5.3 AlHnd ... 39
5.3.1 AHFans ... 40
5.3.1.1 AHFansTh ... 40
5.3.2 AHKomp ... 40
5.3.2.2 AHKompEr ... 40
5.3.2.3 AHKompTh ... 40
5.3.2.4 AHKompDis ... 41
5.3.3 AHCir ... 41

5.3.3.5 AHCirEr ... 41
5.3.3.6 AHCirHPr and AHCirLPr ... 41
5.3.3.7 AHCirPD ... 42
5.3.4 AHEv ... 42
5.3.4.8 AHEvEr ... 42
5.3.4.9 AHEvAf ... 42
5.3.5 AHPumpG ... 42
5.3.5.10 AHPumpTh ... 42
5.3.6 AHDef ... 42
5.3.7 AHPlan ... 42
5.3.7.11 AHPlanEr ... 42
5.3.7.12 AHPlantHT ... 42
5.3.7.13 AHPlantLT ... 42
5.4 AvaCalc ... 43
5.4.1 Status variables ... 43
5.4.2 AC_Plan ... 43
5.4.3 AC_Ev ... 43
5.4.4 AC_Cir ... 44
5.4.5 AC_Komp ... 44
5.4.6 AC_Def ... 44
5.4.7 AC_Fans ... 44
5.4.8 AC_PumpG ... 44
5.4.8.1 AC_Pump ... 44
5.5 DefReg ... 44
5.5.1 Status variables ... 44
5.6 CompeReg ... 44
5.6.1 Status variables ... 44
5.7 IntReg ... 44
5.7.1 Status variables ... 44
5.8 ThermReg ... 44
5.8.1 ThermReg ... 44
5.8.1.1 DynSet ... 44
5.8.2 Status variables ... 45
5.9 CtrlCalc ... 45
5.9.1 Status variables ... 45
5.9.2 CC_Plan ... 45
5.9.3 CC_Ev ... 45
5.9.3.1 CC_Def ... 46
5.9.3.2 CC_Cir ... 46
5.9.4 CC_Pump ... 46
5.9.5 Komp: control ... 46
5.9.5.3 Initial state ... 47
5.9.5.4 OFF Status ... 48
5.9.5.5 GT2 transition ... 48
5.9.5.6 ON Not Ready ... 48
5.9.5.7 GS3 step ... 49
5.9.5.8 GT3 transition ... 49
5.9.5.9 GT6 transition ... 49
5.9.5.10 ON status ... 50
5.9.5.11 GT4 transition ... 50
5.9.5.12 OFF Not Ready ... 50
5.10 Fans (Single condensation) ... 51
5.11 Liquid injection ... 51
5.12 Log2Phy ... 52
5.12.1 L2P_Plan ... 52
5.12.2 L2P_Ev ... 54
5.12.3 L2P_Cir ... 54
5.12.4 L2P_Komp ... 54
5.12.5 L2P_Fans ... 55
5.12.6 L2P_Pump ... 55
5.13 BbxDrv ... 55
5.14 KbdDrv ... 56
6 Specification of the function blocks of the reversible heat pump ... 57
7 Specification of the reversible heat pump functions ... 58

8 Specification of the library function blocks of the reversible heat pump 59
8.1.1 KompDrv 59
8.1.2 PI 59
9 Use of the device 60
9.1 Permitted Use 60
9.2 Responsibility and residual riscks 60
10 Disclaimer 61
11 Inalitic Index 63

1 USE OF THE 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 AppMaker potential and constraints

The aims of this chapter are to identify the potential and constraints of AppMaker for the purpose of specifying the methods for optimal use in the development of air-conditioning applications.

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

- Present on the market since the early 1990s.
- It is widely referenced.
- It is the only application providing a professional version for distributed control for over five years now.
- It has one of the most sophisticated development workbenches available on the current market.
- It has total opening to porting and customisations (the target sources can be purchased from ICS Triplex).

2.1.1 Limit of development philosophy of the Applications

The limits are mainly tied to the fact that in the PLC world (and therefore SoftPLC), there is a scarce adoption of code re-usability, usually producing applications with a low level of parameterisation potential (with the exception of the approach, now increasingly used, to produce super-configurators in vertical application sectors, able to automatically generate applications that can then be reprocessed with their standard workbenches: for example, automation application IEC 61131 and relative SCADA type HMI) . The immediate consequence is that the adopted programming model above all gives priority to easy and speed of development, and not the re-use or parameterisabilty of all that is generated.
However, in the world of air-conditioning there is the need to produce units that combine both high levels of programmability and equally high levels of parameterisability.

2.2 Use of languages IEC-61131-3

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

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

Textual languages
IL : used very little as it is highly cryptic, in fact similar to an “old ” assembly language for 8 bit microprocessors.
ST : its similarity to languages such as “C” or “Pascal” this language can be easily used by all those with experience in these common programming languages.
Graphic languages
LD : Other than being the language most used by PLC programmers, this language offers the benefits of using “Quick LD editor” to enable the development of parts of codes that can be allocated dynamically. Its use therefore enables a range of benefits along with high computational speed.
FBD : used when graphic/visual comprehension is important.
SFC : very powerful and enables traceabiluty (enables the placing of break-points and display at a glance of the status of automata); and is therefore worth using, above all when a relation between the algorithm and one or more automata in finished states is evident;
FC : As this is an extension available only on AppMaker and not part of the standard IEC 1131, use of this language would preferably be avoided.

2.3 Libraries and function blocks

Among the manifold possibilities offered by AppMaker there is that of including specific functional algorithms in the core of the controller, while enabling use from the workbench. In general “function blocks” included in the controller BIOS are those requiring “Real Time” execution times or for which development in “C” language is deemed more convenient. While in the first case, in principle, it is not possible for the developer of Energy XT-PRO to obtain the same behaviour of a function block included in the BIOS with a IEC 1131 “program unit”, in the second case there is no restriction for the IEC 1131 software engineer to develop alternative “program units” to those supplied as a basic library for the chiller application. However, to maintain the correct architecture, it will be important that the I/O interfaces of the new program unit developed (that which in language “C” is commonly known as the “functional prototype”) are identical to that of the library function block.

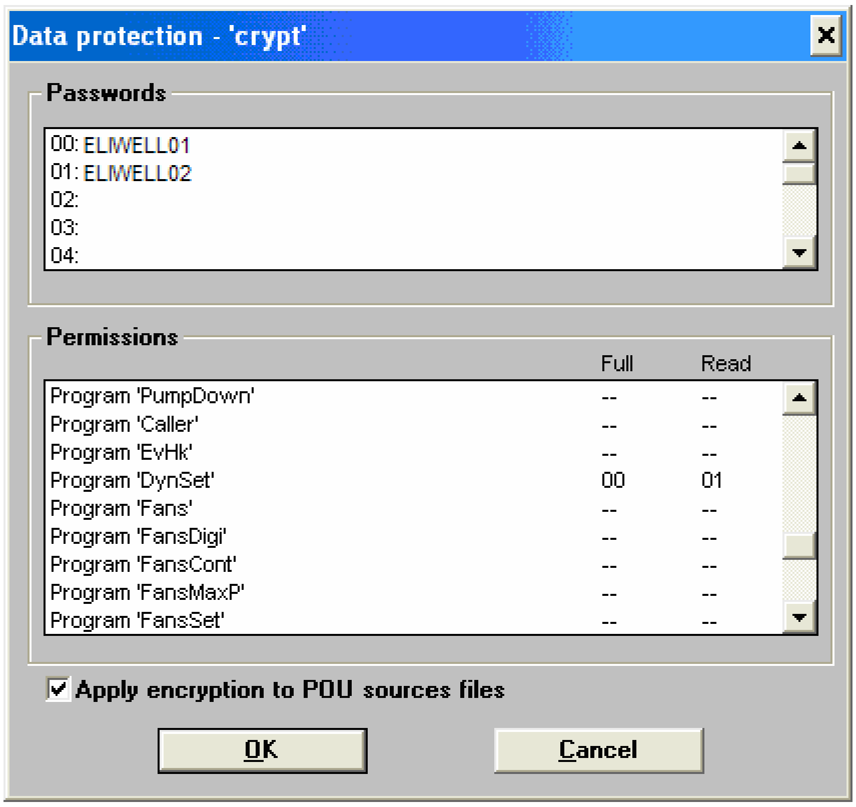
2.4 Possibilities of application parameterisation

Although in the development of the chiller “baseline” application the approach of a “programmable application” is adopted, there remains the objective of making an application that can also be “parameterised” within specific limits.

The non “structural” machine parameters may in any event be modified or set from the user interface to obtain the required application from the range of possible applications. However, once compiled and downloaded into the developed XT-PRO application this domain is finite and limited and parameters cannot be entered over their physical limit. Therefore it will not be possible to set the number of compressors to a value exceeding the number of compressors compiled and wired in the application.
Another example that highlights a limit to the possibility of paramterising the application is that related to the “wiring” of the physical I/O. This type of baseline application does not enable modifications to the physical wiring scheme. Therefore the I/O basket of AppMaker will adopt fixed wiring, modifiable only via the workbench of AppMaker.

2.5 AppMaker application implementation possibilities

AppMaker provides a series of access levels, protected by means of "passwords", to enable controlled release of the application sources.
There are 16 levels available (00..15). A specific password can be entered for each level. Access is hierarchical, i.e. the priority descends from level "n" to level "n+1" (i.e. the highest level is 00). For example, to modify the password of level 03 the user enters using the passwords for levels 00..03.
The figure below shows the protection window after access on level 01.



Data protection - 'crypt'

Passwords

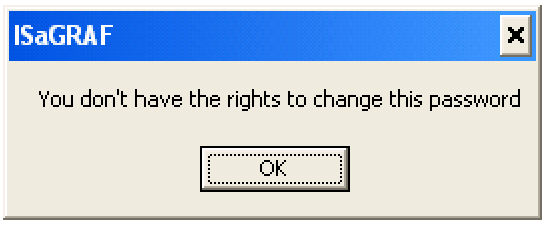
00: ELIWELL01
01: ELIWELL02
02:
03:
04:

Permissions

	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

The figure shows that the highest level (00) is concealed, in other words if the user attempts to modify it APPMAKER notifies the user that he does not have the rights.

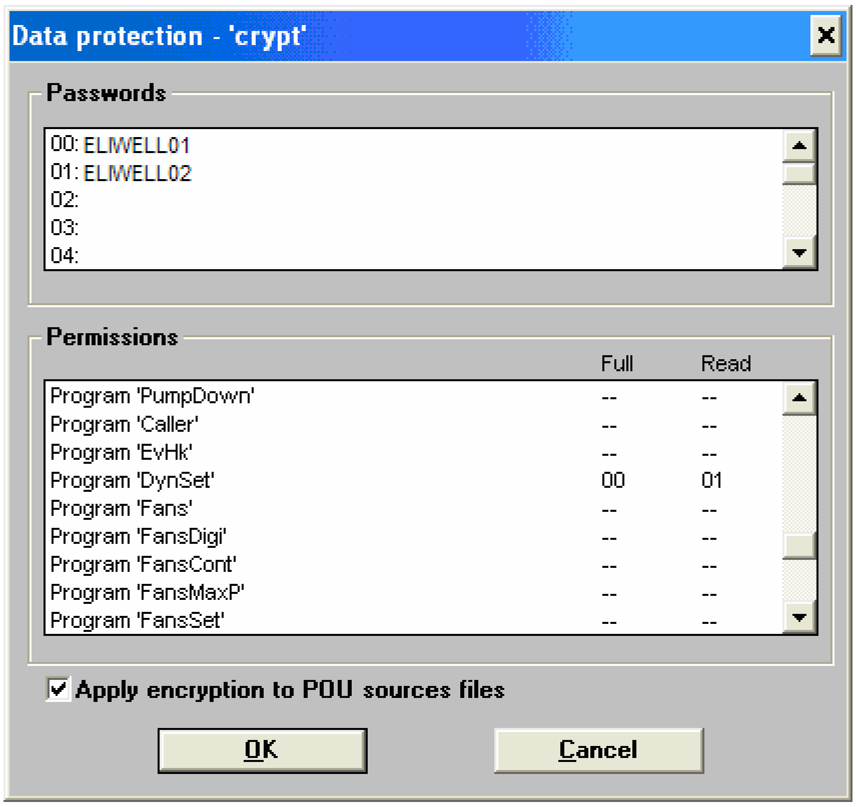


ISaGRAF

You don't have the rights to change this password

The passwords for all lower access levels are displayed and are modifiable. The protection levels can then be associated with single files (programs) making up the application, and can be associated with more general objects such as the I/O connections, global variables, etc.
In the case of most objects[1], including the developed programs, a "Full" access level can be associated, i.e. read/write, and a "Read" level, i.e. read-only.
In the following example the program *DynSet* is accessible in read-only mode, on entry of a level 01 password and in read-write mode on entry of a level 00 password.

[1] For some types of object, such as the possibility of creating (adding) programs to an application, only one type of access is envisaged; these objects are those for which it does not make sense to envisage read-only type access.



```
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 in terms of access by means of the WorkBench (in other words if the files have passwords, this is requested on an attempt to open the file) but they are in any event saved in their original format and remain legible by any "external" editor (e.g. a text editor). This means that the source would therefore be available and copyable by a client.
Outside this context, we can envisage that some Program Units become simply containers for calls to *Functions and Function Blocks* encoded in C and implemented in the BIOS. At present, it is maintained that the most significant parts of the application are those that implement the thermoregulator and resource allocation logics (saturation, balancing, advanced policies).
Even without studying how to use the level hierarchies to provide different levels of visibility with a single source pack (providing individual clients with the suitable level password), "cloned" applications can be generated easily and quickly, in which the level of visibility can be customised, protecting the parts which for the specific client should remain hidden (protected) and revealing those which should be visible.

2.6 Using the Arrays

The use of "arrays" enables the user to have functions that operate on "data sets" declared as single dimension vectors and identified by a different index for each instance or copy of Program Unit that implements the same function.
An example can be seen in a general Program Unit that implements management of a compressor. To understand the great potential of using single dimension arrays, it is enough to consider that if this data structure is not available, the user would be obliged to have N versions of the same program, differentiated both in name (and therefore individually modifiable) and also where use is necessary of global variables that would have to be different for each compressor. The problem obviously does not apply to local variables of the Program Unit as they have ***"scopes"*** of the Program Unit in which they are declared.
On the other hand, the ability to define vectors of global variables means that each copy of a Program Unit will have one "index" variable only, to which a different value can be assigned. In this way the entire code remains identical in the different copies of the Program Unit enabling use of global_variable[index] type notations.
As neither **MenuMaker PRO** nor**TabMaker** are able to manage the arrays, it must be taken into account that these data structures are 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 when using vectors, wither because use is not possible or because improper use would impair correct operation of the application.

A solution to adopt for solving the limits tied to the use of vectors is that of using static I/O variables both for inputs and outputs. This solution, (with the only contraindication of memory waste due to the fact that the quantity of I/O variables allocated must be equal to the maximum possible dimensions of the two I/O vectors), enables access of MenuMaker PRO to the I/O, as well as enabling the MODBUS protocol to read specific I/Os with a fixed MODBUS address.

2.7 Design criteria and system optimisation

This paragraph is particularly important as it provides the guidelines used in the design and those to follow in the development phase for system optimisation.

The first choice regards the languages to be used when not set by AppMaker as in the case of SFC for a number of sequential parts.
The decision was made to privilege the ST language as it enables increased legibility of the code and enables the “C” programmer rapid comprehension of the code and quick customisation. In other points, priority was given to the use of languages such as FBD and QLD, which, with respect to textual versions, enable increased legibility to users familiar with working with graphic languages. Furthermore, the use of QLD with the “in-line” option becomes necessary when ***nesting*** of functions is required with *status variables*, but this language, if its use is widespread, leads to a significant increase in code dimensions.
The alternative is to use a textual language (ST) transforming the *status variables* into vectors of global variables. In this way the code dimensions are reduced (reducing execution speed) but the price to be paid is that the “interface pins” between functions are lost.
As a borderline case we could have all global variables placed in a single common area with no explicit interface between modules.

On the other hand there are some inevitable constraints:

the application must be “containable” in the memory available
the execution time must be compatible with the dynamics of the system to be controlled
the system must perform all specified tasks

In the development of a new application, it is always good practice to envisage alternative solutions, including the decision, in the development phase, which enable the user to economise the resource which, on a case by case basis is the most critical, i.e. space or time.

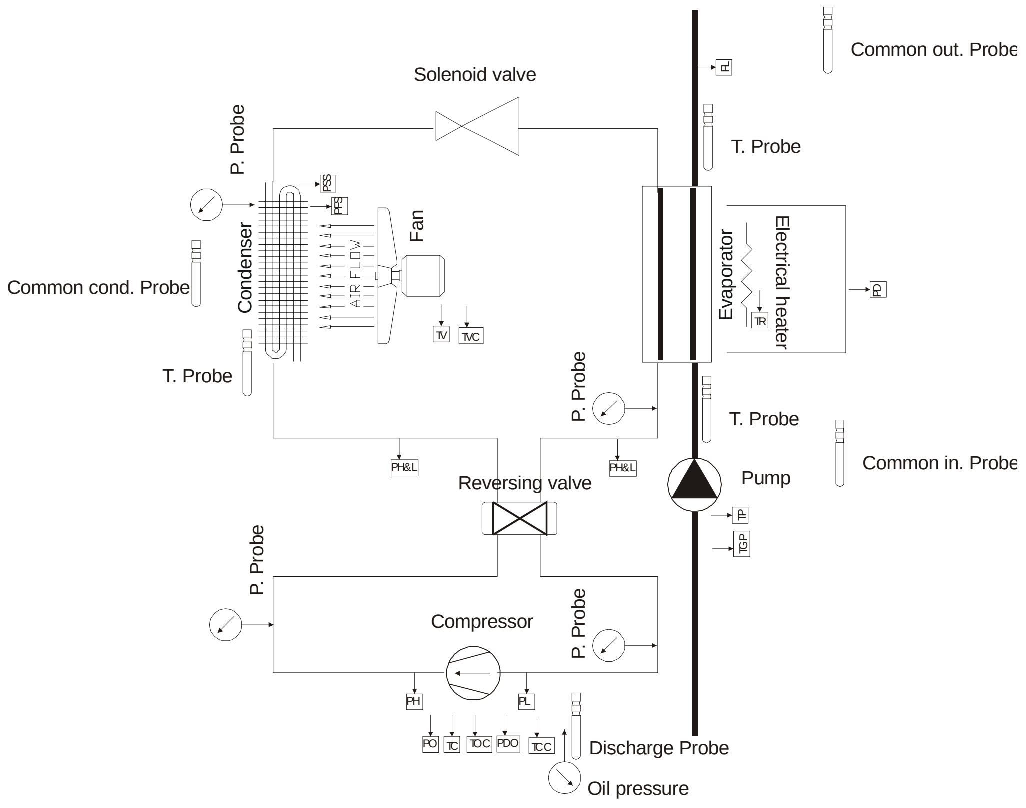
Therefore the correct approach to development should be the following:

development with graphic languages where compatible with specified constraints
alternative development in textual language
profiling of performance with AppMaker simulation tools
optimisation, if necessary, of space by “compacting” from FBD in ST
optimisation if necessary in time by increasing use of memory
possible processing of steps 3, 4, 5.

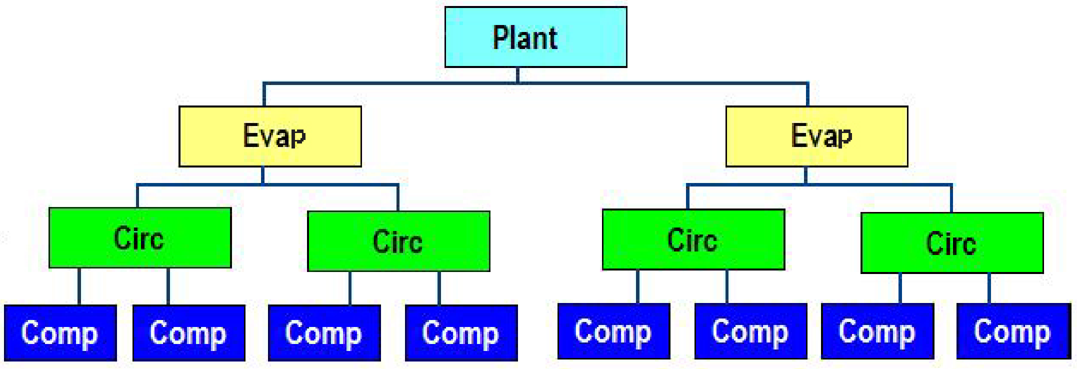
3 GENERAL ARCHITECTURE OF THE REVERSIBLE HEAT PUMP APPLICATION

3.1 Introduction

The chiller is a system comprising a circuit through which a refrigerant fluid is conveyed by means of a compressor; the compressed (and therefore heated) fluid is delivered to a condenser where heat is dispersed via an expansion vessel thus cooling, after which it passes through an evaporator from which it extracts heat.
In a water-air "chiller" the evaporator cools water, while an air-air version cools air.
In a reversible machine, in other words able to operate both in *chiller* mode and *heat pump* mode, the condenser and evaporator exchange roles depending on the operating mode, thus enabling the generation of both cold and heat. The fact that the system operates in chiller mode (thus generating cold) or heat pump mode (generating heat) depends on the status of the inversion valve, used to determine the direction of the refrigerant fluid in the circuit and thus setting the thermo-dynamic operation mode.
Water-air machine layout



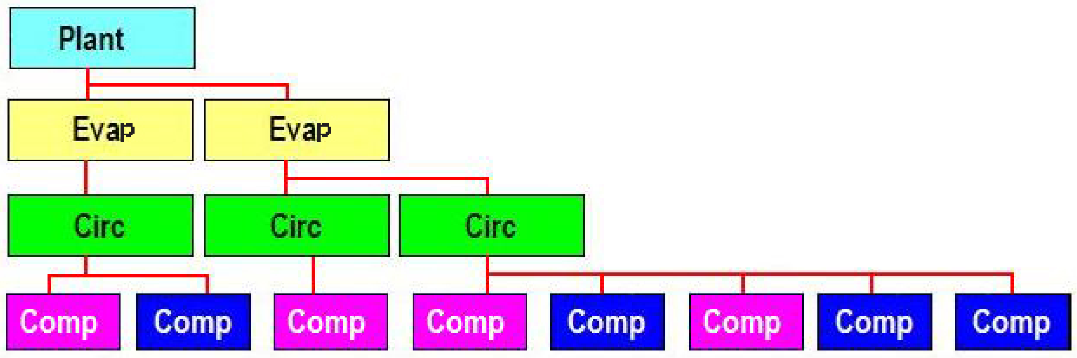
The Energy XT-PRO application is based on a hierarchical structure, which may be either symmetrical or asymmetrical,



homogeneous or inhomogeneous.

For example this structure imposes the restraint that each node on the same level must have the same number of child elements. Each evaporator should therefore control an equal number of circuits, and in the same way each circuit should control an equal number of compressors.

The structure of Energy XT-PRO overcomes these restraints, requiring the developer to define only the number of elements of each type. The links between the elements can be managed later according to a “data driven” scheme.
An example of such a structure is provided below:



Modularity is achieved by using parameters to define the number of child elements belonging to each father element: evaporators per machine, circuits per evaporator and compressors per circuit. There is also no restraint that the type of compressor must be the same. The graph shows the different types of compressor with colour coding.

This example can apply also to all elements making up the machine, and therefore this enables evaporators, each with a different number of circuits, or fan coils each with a different number of fans.

The main advantages of this type of structure are:

- possibility of offsetting machines: one circuit with two compressors and one with three;
- possibility of generating inhomogeneous machines: one circuit with several compressors, different from one another;
- extreme ease in standard management of asymmetrical elements: a fan block that serves two condensers on two different circuits.

3.2 Main principles of architecture

For the sake of clarity, a number of general features of an AppMaker application are listed below:

- Each application is divided into the following *Sections*
  Begin
  Sequential
  End
  Function and Function Block local to the application;
  bear in mind that use of the sections is optional: if required, the entire application can be inserted in the section Begin.
- Each section comprises a number of *Program Units* sized as required (compatibly with the physical memory available).
- There are no restraints or limitations in the selection of languages used in the development of the Program Units, remaining within the restraints of the AppMaker system.
- The *Function Blocks* are autonomous application units that can be invoked in different program units: they encapsulate frequently used logics.
- The data are stored in a *Data Dictionary* separately from the program logic. The data have ‘scopes’ as in modern program languages, i.e. the contents of these data during the debugging phase may be displayed and modified during program execution.

With the above in mind, the main principles of the software architecture described in this document are illustrated below:

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

Section	Program Unit Contents	Languages used
Begin	Contains the program units dedicated to the initialisation of the application, those dedicated to pre-processing of input signals and their conversion from physical to logical, and the computation of thermoregulation algorithms.	Quick LD/FBD, ST
Sequential	Contains the program units that encapsulate the status logics of the compressors.	SFC for automata, and ST, Quick LD/FBD for auxiliary logics
End	Contains additional program units related to the management of condensation and fans, and the conversion of signals from logical to physical.	Quick LD/FBD, ST
Function	Contains non-library functions, and in general all functions auxiliary to the computation and structures of fundamental data.	Quick LD/FBD, ST, or C.
Function Block	Contains the non-library FBs, and in general all functions auxiliary to the computation and structures of fundamental data.	Quick LD/FBD, ST, or C.

2. *Modularity*
   Modularity regards various aspects:
   a. Program Unit: the division of the code into program units must enable easy identification of a function within the application. As far as possible, a function will be enclosed in a specific program unit. However, take into account that "function" does not refer to the entire control function but a homogenous series of operations. For example, the core of the compressor 1 governing logic will be in a program unit of the *Sequential section*, but the pre-processing of its input signals and processing of the alarm conditions will be in other program units of the *Begin section*, and the post-processing from logic implementation or physical implementation will be in another program unit of the *End section*. This does not violate the principle of mapping functions-program units, as it is effectively possible to have cases in which the application maintains the same management logic, but modifies that of implementation etc.
   b. Function Blocks: these are typically elementary functional units and therefore extremely small. They are normally catalogued in libraries, but there may be some “application specific” FBs which are therefore allocated in the last section of the application.
   c. Complex data structures: Represented by management of the “one-dimensional arrays”. With suitable provisions, used also in the baseline application, two-dimensional structures can be managed through simple one-dimensional arrays.

3. *Names of variables and program units*
   These must be, as far as possible, highly ‘significant’. In the case of tool restraints (for example the length of program unit names) the use of adequate comments will be required.

4. *Data driven code*
   As far as possible the code will be data driven type: in other words there will be data structure which, depending on the parameterisation (default value for fixed parameters, or value overridden via keyboard/serial port for dynamic parameters) will enable modifications of important attributes of the application: for example the presence (or not) of compressor n, its association with circuit p, etc. Obviously in doing this there are the objective limitations of the characteristics of AppMaker (see Chapter 1).

5. *Readability of code*
   Also this aspect has various implications.
   a. Use of descriptive strings in the configuration of parameters: it is clearly useful that a parameter indicating the configuration of a compressor is not assigned the value 0 or 1 but is rather ‘inhibited" or ‘active", also in the case of the variable that indicates the alarm status, which should be "normal" or "alarm" rather than 0 or 1 as previously.
   b. Extended use of indentation: for ST code.
   c. Extended use of comments, notes and scrolls.
   d. Use of a clearly defined graphic layout: for FDB code.
   e. Use of statuses that enable immediate comprehension of the current situation of the logics, including auxiliary statuses such as those in which an non-configured or dynamically excluded object is entrapped: for SFC code.

The main principles illustrated here not only have been observed in the development of the baseline reversible heat pump application, but should also be observed in the application of future modifications. This applies not only to “entropize” the implementation, but also to prevent a number of architectural mechanisms from being broken down by the *introduction* of modifications not consistent with the basic philosophy.

3.3 General structure

The following section describes the *general structure* of the application. Note that the aim of this chapter is to provide guidelines only to the architecture. Therefore Chapter 5 should be consulted for more specific information on the program units shown in the figures below, and also the structure adopted to form the program unit.
The criteria adopted is to implement, in the various program units, the algorithms necessary for the control of the reversible heat pump. In more detail, the functions can be identified in the various sections:

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

- pre-processing of the auxiliary logics to give core logics a totally updated photo;
- management of the passage on input from physical to logical;
- Management of logic alarms;
- Calculation of availability;
- Management of defrosting;
- Management of integration resistances;
- Computation of thermoregulation;
- Computation of control;

- **sequential** : logics related to the compressors

- **end** : “Special” algorithms and conversion of sizes from logical to physical.

- Management of Condensation;
- Management of *liquid injection*;
- Logical-physical conversion of thermoregulation results;
- Management of Black Box;
- Management of data exchange with keyboard.

3.3.1 Begin Section

This section mainly implements purely combinatorial parts of logic. These are often program units without memory in which the data produced depend exclusively on the data on input.
There are exceptions to this rule, such as the logic for generation of alarms which activate the Function Blocks containing the automata of evolution of the alarms themselves.

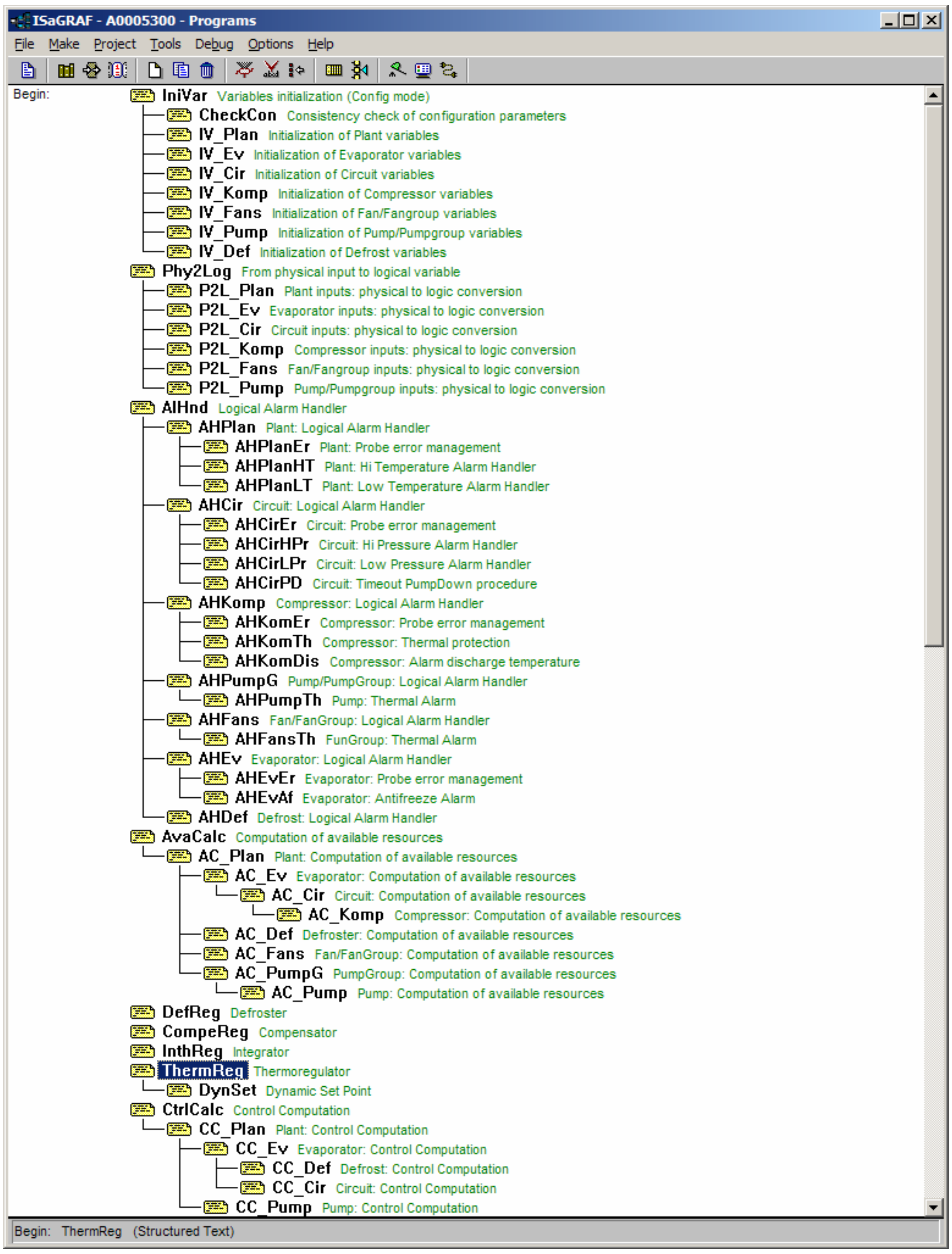
The main program units contained in this section are:

1. Configuration validation
2. Variable initialisation
3. Converter of physical input on variable work logics
4. Logic alarm generator
5. Calculation of availability
6. Defrost management (with relative compensation phase)
7. Management of integration resistances
8. Management of algorithm for the dynamic Set Point
9. Calculation of power requested (thermoregulator)
10. Calculation of control

Proceeding in top-down mode, each program unit can be broken down into sub-program units to obtain improved modularisation and encapsulation of the various functions.

The implementation language will be FBD/QLD where maximum legibility is required to simplify the modification for a user used to working with these programming languages and ST in parts where compactness of the code is to be optimised or where the situation requires implementation of cycles difficult to manage with graphic languages.

The following diagram illustrates how the section is divided[2]


ISaGRAF - A0005300 - 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
          └─IV_Def  Initialization of Defrost 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
          │   └─AHPlanLT  Plant: Low 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
          └─AHDef  Defrost: Logical Alarm Handler
        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_Def  Defroster: 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
        DefReg  Defroster
        CompeReg  Compensator
        InthReg  Integrator
        ThermReg  Thermoregulator
          └─DynSet  Dynamic Set Point
        CtrlCalc  Control Computation
          └─CC_Plan  Plant: Control Computation
              ├─CC_Ev  Evaporator: Control Computation
              │   ├─CC_Def  Defrost: Control Computation
              │   └─CC_Cir  Circuit: Control Computation
              └─CC_Pump  Pump: Control Computation
Begin:  ThermReg  (Structured Text)
```

3.3.2 Sequential section

In this section, the automata are implemented, which describe the evolution of the "compressor" system from a dynamic point of view.
The language used must be SFC reducing textual sections to a minimum and referring, where possible, executive parts to *functions and function blocks* developed with graphic languages and when necessary ST.

The program units implemented will fundamentally be the compressor management automata (possibly in several copies).

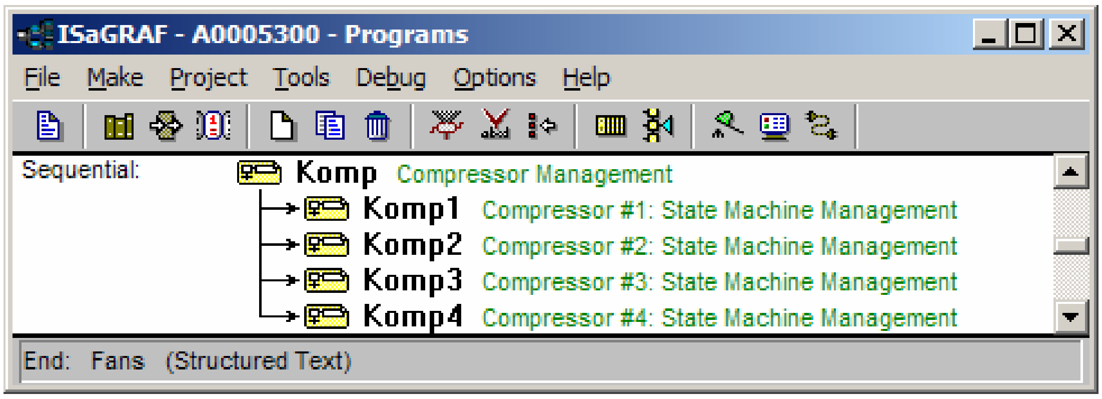
Note that no explicit hierarchical relation is envisaged (see a detailed explanation of this subject in paragraph 4….) and therefore the implementation scheme will be of the following type:

1. Compressor 1 management automata
2. Compressor 2 management automata
3. …
4. Compressor n management automata

[2] The diagram uses the nomenclature as described in the next chapter

As a general rule, the program units will contain compressor management automata: these are N copies of the same program that operate on a set of local variables and on the i-th set of global variables allocated in vectors with the dimension N. These automata are delegated with the compressor activation logic in observance of the block times for the various operations and maintaining the current availability data of the compressor.

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


ISaGRAF - A0005300 - Programs
File Make Project Tools Debug Options Help

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

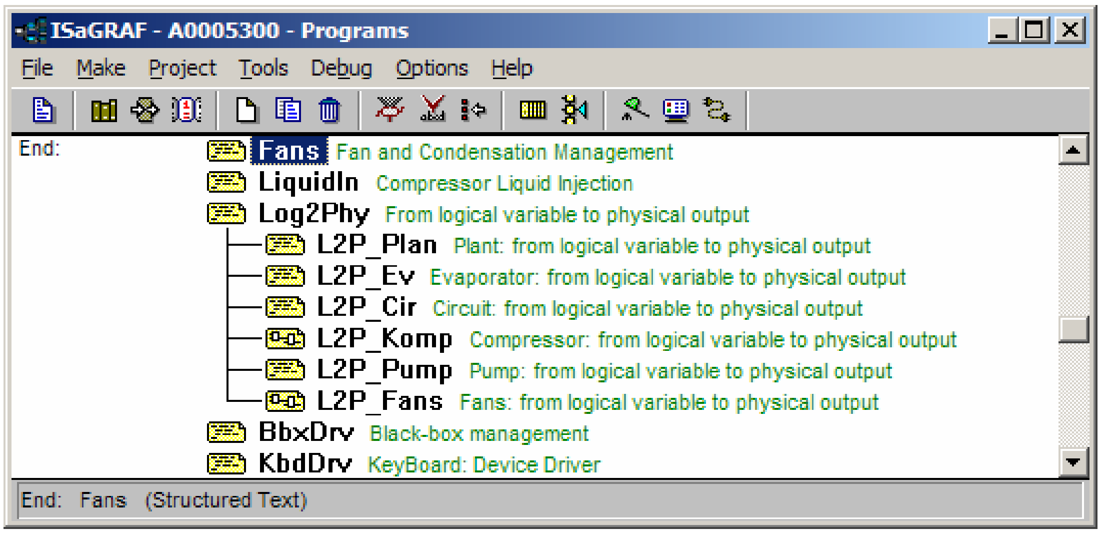
End: Fans (Structured Text)

3.3.3 End section

The main program units contained in this section are:

- Management of condensation and fans;
- Management of algorithm for *liquid injection* into compressors;
- Conversion of variables from logical to “physical output ” variables;
- Management of Black Box;
- Management of data exchange with keyboard.

Proceeding in top-down mode, each program unit can be broken down into lower level program units to obtain improved modularisation and encapsulation of the various functions.
The following diagram illustrates how the section is divided


ISaGRAF - A0005300 - Programs
File Make Project Tools Debug Options Help

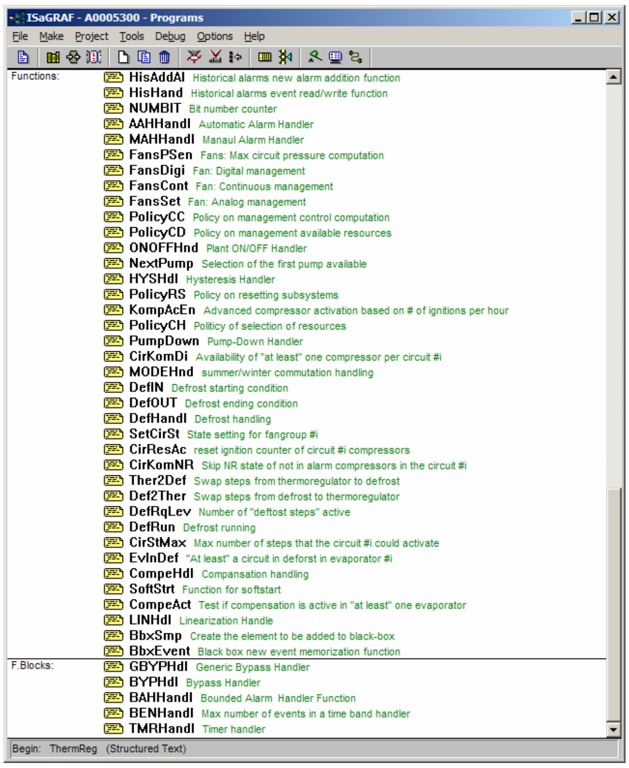
End: Fans Fan and Condensation Management
LiquidIn Compressor Liquid Injection
Log2Phy From logical variable to physical output
- L2P_Plan Plant: from logical variable to physical output
- L2P_Ev Evaporator: from logical variable to physical output
- L2P_Cir Circuit: from logical variable to physical 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

BbxDrv Black-box management
KbdDrv KeyBoard: Device Driver

End: Fans (Structured Text)

3.3.4 Functions and Function Blocks

In this section *functions and function blocks* are implemented for the calculation of thermoregulation algorithms. As well as the hysteresis and bypass algorithms, the *end section* also contains all algorithms commonly used and those requiring personalisation (such as the “bit counter”), and algorithms for the selection of “strategic” cooling resources which can neither be modified nor displayed (such as the Program Unit PolicyCC). When personalised algorithms for the selection of cooling resources are required, the user needs to cancel the program units in the baseline application and then generate a new program.


ISaGRAF - A0005300 - Programs
File Make Project Tools Debug Options Help

Functions:	HisAddAl	Historical alarms new alarm addition function
	HisHand	Historical alarms event read/write function
	NUMBIT	Bit number counter
	AAHHandl	Automatic Alarm Handler
	MAHHandl	Manaul Alarm Handler
	FansPSen	Fans: Max circuit pressure computation
	FansDigi	Fan: Digital management
	FansCont	Fan: Continuous 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
	MODEHnd	summer/winter commutation handling
	DefIN	Defrost starting condition
	DefOUT	Defrost ending condition
	DefHandl	Defrost handling
	SetCirSt	State setting for fangroup #i
	CirResAc	reset ignition counter of circuit #i compressors
	CirKomNR	Skip NR state of not in alarm compressors in the circuit #i
	Ther2Def	Swap steps from thermoregulator to defrost
	Def2Ther	Swap steps from defrost to thermoregulator
	DefRqLev	Number of "deftost steps" active
	DefRun	Defrost running
	CirStMax	Max number of steps that the circuit #i could activate
	EvInDef	"At least" a circuit in deforst in evaporator #i
	CompeHdl	Compansation handling
	SoftStrt	Function for softstart
	CompeAct	Test if compensation is active in "at least" one evaporator
	LINHdl	Linearization Handle
	BbxSmp	Create the element to be added to black-box
	BbxEvent	Black box new event memorization function
F.Blocks:	GBYPHdl	Generic Bypass Handler
	BYPHdl	Bypass Handler
	BAHHandl	Bounded Alarm Handler Function
	BENHandl	Max number of events in a time band handler
	TMRHandl	Timer handler

Begin: ThermReg (Structured Text)

3.3.5 Procedure for modifying the baseline procedure

The following section provides a general overview of the procedure that a developer could follow to obtain a new application, starting from the baseline application described in this document. Although the use of the workbench AppMaker enables the free execution of modifications to the application, we strongly recommend observing the following simple rules:

1. If the modification implies a change to the wiring scheme, for example with the aim of adding an output signal, the “I/O basket” must be modified. This modification will have little impact on the program logic as the I/O variables have logical names and therefore movement into the basket do not implicate changes to the logics. Therefore it will be sufficient to localise the program units and FBs involved in the modifications after evaluating possible changes in the global data structures for parameterisation and status.
2. If changes are made to the global data structures, the necessary modifications to the application “dictionary” must be made, in respect of the philosophy of the data structure that supports the baseline application.
3. If changes to a FB are required, it must be rewritten and its instances in the application will have to be replaced with those of the newly created FB.
4. Lastly, the code of the program unit involved in the modification will need to be changed/extended. In fact in some cases, the *introduction* of changes may lead to “side effects” also in program units not directly involved in the modifications: these cases should be adequately rare thanks to the optimal modularisation of the application and strong mapping between functionalities and software modules (program units).
5. Then a new TC must be generated (compilation).
6. The new TIC must then undergo validation by means of a preliminary simulation (validation on PC) and then by debugging (validation on Target) to ensure that the modifications entered obtain the required procedure and do not generate undesired effects (regression testing). Regression testing will in turn be significantly limited by the strong mapping between functionalities and software modules (program units).
7. Produce the project documentation in the updated version by means of the special functionalities of the workbench AppMaker.

Note that the conceived architecture will enable considerable changes to the structure of the application by simply re-using program units (codes), accompanied by suitable modifications/extensions of the code and structure of the global data of parameterisation and status (dictionary). For more details and explanations of this mechanism, refer to the next two chapters.

Note lastly that many changes, even significant, to the application structure (such as the movement of a compressor from one circuit to another) may not require any effective software modifications, as they are obtained by pure and simple re-parameterisation of the baseline application.

4 DATA DICTIONARY OF THE REVERSIBLE HEAT PUMP APPLICATION

This chapter describes the global data areas, accessible to each program, while the local data areas of each program will be described in chapter 4.
Global data play a central role; they constitute to all effects the software interfaces between the various programs, and thus completely represent the mechanisms for communication and synchronisation between the various programs.

The data on which the application works are classified in different areas, divided where necessary into sub-areas.
The hierarchical division is as follows:

1. Parameters: data that define, on various levels, operation of the application. The application uses these data accessing them in read-only mode. Using the classification already used, the parameters are divided into three categories:

   a. FIXED parameters (referred to in the rest of this manual as Defined Words): this category contains read-only values (not modifiable by the user) identifying the maximum machine domain, such as:

      •∴ταβ maximum number of evaporators;
      •∴ταβ maximum number of circuits;
      •∴ταβ maximum number of compressors;

   b. COLD parameters: these are modifiable by the user but implicate in general the need to stop and restart the plant. The following belong to this category:

      •∴ταβ number of compressors enabled;
      •∴ταβ regulation algorithm (proportional, *PI*)
      •∴ταβ resource selection algorithm (saturation/balancing)

   c. HOT parameters: these are operating parameters modifiable during operating conditions without special provisions, such as:

      •∴ταβ minimum compressor off time
      •∴ταβ minimum time between start-up of two compressors
      •∴ταβ temperature set-point

2. Variables: these are data that define, on various levels, machine status. The application produces and uses these data, accessing them in read and write mode. The classification is by functional area or physical component and by ***"scope"*** in other words differentiating between global variables of the application and local variables of the different programs and copies of the same program. This section will also specify the different areas, specifying the global variables while local variables will be dealt with in chapter 4 in the paragraphs related to the various function areas.

   2.1 plant variables
   2.2 condenser variables
   2.3 circuit variables
   2.4 compressor variables
   2.5 fan set variables
   2.6 pump unit variables
   2.7 defrosting variables

4.1 Nomenclature

To facilitate interpretation of the various objects present in the dictionary, the following conventions are adopted:

1. object names: these are made up of an acronym identifying the physical area of classification, the function to which the object refers or the object type:

   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 unit area
   1.6 PUMP: pump / pump unit 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 (e.g. a pressure switch)
   1.18 SENS: analogue sensor (analogue input)

when applicable, the physical area of classification must be used as a pre-fix for each acronym (e.g. for the circuit pressure sensor, the total 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:	parameter or variable of I/O or defined word (constant)
2.2	XxxYyyy:	internal global or local variable; name of function or function block
2.3	xxx_yyy:	internal support variable, indexes, temporary etc.

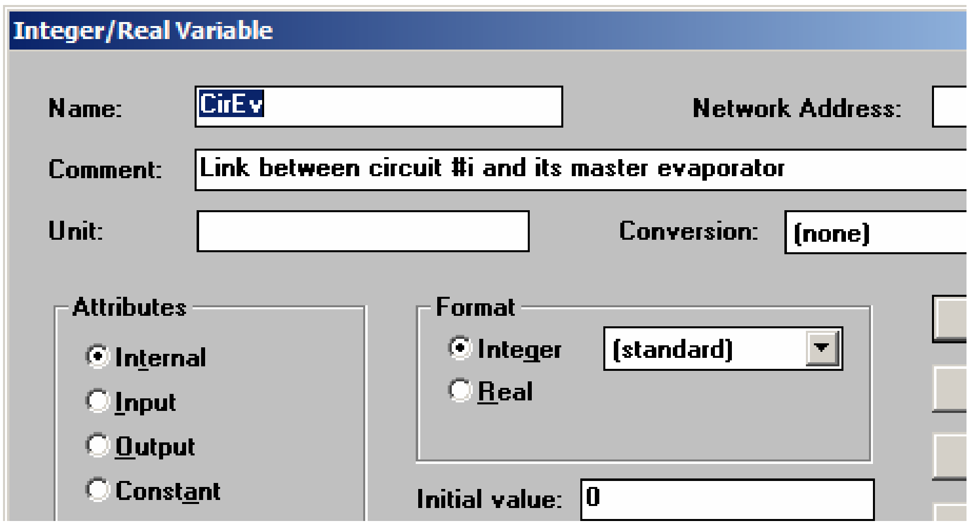
Note that what has been previously defined as "parameter" uses the syntax "XXX_YYY", i.e. the ID comprises only upper case and the names/acronyms are separated by the character "_" (underscore), while that previously defined as "variable" uses the syntax "XxxYyy, i.e. the ID comprises upper case (first character of the name/acronym) and lower case (remaining characters of the name/acronym) and the names/acronyms are separated by the change between upper and lower case.
Note also that variables in any event related to the I/O and constants use the syntax of "parameters". The first because in some way they are also considered external with respect to the application AppMaker (the input is read-only and the output is write-only), while the second because it is common practice to use upper case characters to define the constants.

4.2 Control structure definition

The application AppMaker must be developed from the start taking into consideration the maximum limits of machine expandability. This means that the sizing of the control structure (management arrays) must be made taking into account the "maximum of maximums".
In particular in the example below the set structural limits are:

Maximum number of machine circuits	8
Maximum number of machine evaporators	4
Maximum number of machine compressors	8
Maximum number of machine pumps	2
Maximum number of machine fan coils	8
Maximum number of machine fans	16
Maximum number of compressors per circuit	8
Maximum number of circuits per evaporator	4
Maximum number of fans per fan unit	8
Maximum number of circuits per fan unit	8

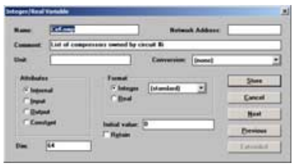
The dimensioning of vectors must be performed not only on the basis of these elements but also according to the specific use of the vector; therefore it cannot be assumed, for example that all vectors that have circuits as the classification have the dimension 8.
A clear example can be seen in the fact that CirEv is dimension 8 because it indicates, for each circuit, the relative evaporator as classification.



Name: CirEv
Network Address:
Comment: Link between circuit #i and its master evaporator
Unit:
Conversion: (none)
Attributes: [✓ Internal] Input Output Constant
Format: [✓ Integer] Real (standard)
Initial value: 0

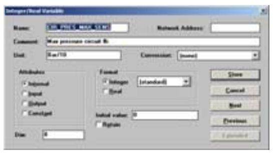
On the contrary, the vector CirKomp must have the dimension 64.
In fact this indicates which circuit each compressor belongs to: it may occur that all 8 compressors belong to the first circuit such as in the case in which only one compressor is present for each of the 8 circuits.
Therefore CirKomp must have the dimension of:

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


Integer/Real Variable

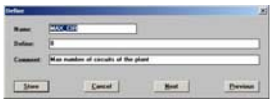
Name: [illegible]
Network Address:
Comment: List of compressors owned by circuit i-th
Unit:
Conversion: (none)
Attributes: Internal [✓] Input Output Constget
Format: Integer [✓] (standard) Real
Initial value: 0
Retain
Dim: 64

Since AppMaker does not enable dimensioning of the vectors using the IDs of the constants (“Defined words” in the dictionary AppMaker), as would happen in a high level programming language, vectors must be sized using a numerical value in the field “Dim” in the parameter definition window in the dictionary, as shown in the example below:


Integer/Real Variable

Name: CIR_PRES_MAX_SENS
Network Address:
Comment: Max pressure circuit i-th
Unit: Bar/10
Conversion: (none)
Attributes: Internal [✓] Input Output Constget
Format: Integer [✓] (standard) Real
Initial value: 0
Retain
Dim: 8

Obviously, though having *nomenclature* that can be defined as “self-identifying” by virtue of the acronyms used, the fact of using the constants in the windows for variable definition could lead to ambiguity or difficulties in identification of the vectors, as most of the maximum limits are generally 8 (e.g. circuits, compressors) or 4 (e.g. evaporators).
In this case, the only possibility is to make an explicit statement in the vector type variable comment field (“Comment” field of AppMaker) to what it refers and how many elements (e.g. “i-th circuit in the example above) must be envisaged for the field “Dim”, within the limits of the 60 characters envisaged by AppMaker for comments on variables.
However, as an aid to compilation of the ST code, and to enable execution of so-called “parametric” loops (in the sense of loops limited by constant identifiers and not by “magic numbers” entered in the ST code), the insertion is envisaged in


Define

Name: MAX_CIR
Define: 8
Comment: Max number of circuits of the plant

AppMaker also of the definition of the constants as “Defined words”, as shown in the example below:

All the “Defined words” used to define these maximum limits will have to have the ID of (“Name” field of AppMaker) a string type “MAX_XXX”, where “XXX” , states the object, the maximum of which is to be defined, and obviously a suitable comment.
Any loops of the AppMaker programs written in ST will thus be able to be written as:

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

in this way the resulting architecture is:

a) dimensioned on the envisaged maximum and in any event able to work (without any modifications) on machines with smaller "physical " dimensions (down-sized)
b) potentially ready to be expanded even over the limits currently envisaged. For example, to increase the number of circuits management, it would be sufficient to simply change the size of all circuit vectors (variables CIR_XXX) and at the same time modify the value of the relative "Defined word" (the "Define" field of MAX_CIR must take on the new maximum value for the number of circuits).

4.3 Global and support variables

This chapter and relative sub-chapters, list the *global and support variables*, of primary importance for the proposed machine architecture.

For AppMaker, the division between parameters and variables in reality does not exist, as the database envisages variables (analogue and digital), timers, messages (strings), FB instances and constants ("Defined words").
As a general rule, the presence of two types of variable can be defined; global and support.
The first are the "working" versions that can be defined "really global", in the sense that they serve to monitor the behaviour and status of the plant.
They contain configuration and set-up information that make the application AppMaker able to be parameterised and data driven (e.g. KOMP_CIR_EV), and information on states, analogue or digital, used (as inputs) by the algorithms to ensure the correct reactions (output generation) (e.g. CIR_PRES_MAX_SENS[]). In practice these are the parameters of the application and the virtual and physical I/O. Details and descriptions of the main parameters will be given below, i.e., those that have an impact on the AppMaker application architecture.

The choice to insert all parameters or not has been made taking into the consideration that:

a) Parameter entry is reasonably quick
d) the overall number of parameters is quite considerable (more than three hundred)
e) it is simpler to follow the evolution of the variables of an application if the total number of variables is smaller
f) AppMaker enables the generation of both dedicated and general "spy and connected to single programs

The second versions (support variables) are those which, though also considered "working" variables, serve to implement, facilitate and accelerate execution of the algorithms.
This category includes both the constructed link tables, on the basis of the configuration parameters, during initialisation, and the other variables not directly related to parameters. Both, from many points of view, can thus be considered real and actual "support variables".
Some of them, such as the look up tables, are global as they are used by different algorithms, even if for completely independent purposes, which however require an external support (for example the pumpdown function of a circuit needs to know which compressors belong to this specific circuit, and therefore a look up table needs to be used).
These variables are described in detail in the following sub-chapters as considered general.
Very probably many other variables will have to be "global" in the sense of "not local" to a single application. However, these variables, unlike the ones dealt with in this chapter, related to the "*Data dictionary of the reversible heat pump application*", are to be considered part of specific applications. Even if these variables will have to be used in several applications (and must therefore be "global"), a clearly defined producer-consumer (or producer-consumers) relationship will remain.

The "local" variables of the single applications are not deemed pertinent to a document on architecture and therefore are not described.

4.3.1 Vector KOMP_CIR_EV

(Part of *IV_Plan*)

KOMP_CIR_EV[8]: integer type vector of 8 elements; uses a decimal 2-digit identifier to associate compressors with circuits and circuits with evaporators. The first digit of each element identifies the evaporator to which the circuit belongs and the second digit identifies the circuit to which the compressor belongs. Only the first n elements can be initialised with n <= 8. The value 0 indicates the corresponding compressor is not present. **This vector is configured manually**.
A single analogue value enables the association of a single compressor both with its relative circuit (decimal value module 10) and the evaporator in which the circuit is located (decimal value divided 10).

Note that the values to be entered as the numbers 1..n, in other words for reasons of clarity the use of the value 0 should be avoided.
The vector is therefore the entry point that enables the generation of a different family of machines, potentially also not symmetrical and unbalanced.

For example, in the case of the BASELINE APPLICATION machine, the vector must be initialised as follows, taking care that there are no gaps and that numbering is monotone, ascending from left to right:

KOMP_CIR_EV for machine 2-4-4 (Baseline Application)					
Index in vector	**Compr.**	**Contained**	**(Evaporator)**	**(Circuit)**	**Notes**
0	1	11	1	1	First compressor, compressor of first circuit of first evaporator
1	2	12	1	2	Second compressor, compressor of second circuit of first evaporator
2	3	21	2	1	Third compressor, compressor of first circuit of second evaporator
3	4	22	2	2	Fourth compressor, compressor of 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* initialised as follows:

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

4.3.2 Vector CirPresence

(Part of *IV_Cir*)

CirPresence[8]: Boolean type vector with 8 elements: defines the presence of the i-th circuit. Calculated by KOMP_CIR_EV[ ].

The *vector KOMP_CIR_EV* is scanned in the initialisation phase, and for each element with a value other than zero sets the presence value of the relative circuit to true, with the index

((<value> MOD 10) – 1) + offset

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

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

CirPresence				
Index in vector	**Circuit**	**Contained (2-4-4) (BASELINE APPLICATION)**	**Contents (2-4-8)**	**Contents (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 Vector CirEv

(Part of *IV_Cir*)
`CirEv[8]`:integer type vector of 8 elements, one per circuit; lists the relative evaporator for the i-th circuit (with i = 0..7), calculated by KOMP_CIR_EV[ ].
Used when, on a circuit level, information must be acquired regarding the evaporator to which the circuit belongs.

The *vector KOMP_CIR_EV* is scanned in the initialisation phase, and for each index element "k" with a value other than zero sets the CirEv element to "**`(<value> DIV 10) - 1`**" with the index

((<value> MOD 10) – 1) + offset

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

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

CirEv				
Index 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
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 Vector CirKomp

(Part of *IV_Cir*)

`CirKomp[8*8]`: integer type vector of 8*8 elements per circuit; lists the compressors that are part of the i-th circuit (with i = 0..7) Calculated by KOMP_CIR_EV[].
The support of AppMaker for single dimension arrays only leads the use of a single vector in place of a two dimension matrix (which would be the more logical solution for this variable).

Given that each circuit can have at the most 8 compressors, and a maximum of 8 circuits is possible, this vector is sized for 64 elements, even though in this case the overall limit of 8 compressors implies that the vector is any event filled partially (many elements, those unused, will count as "-1").

For the "i-th" circuits (with 0 <= i <= (MAX_CIR – 1)), the pertinent starting index will be obtained from the expression:

i * (MAX_KOMP4CIR) (representing 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 *vector KOMP_CIR_EV* is scanned in the initialisation phase, and for each index element "k" with a value other than zero sets the CirKomp element to "k" with the index

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

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

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

CirKomp					
Index in vector	**Circuit**	**Contents (2-4-4) (BASELINE APPLICATION)**	**Contents (2-4-8)**	**Contents (2-8-8)**	**Contents (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)
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)

Note that the use of this type of vector leads to the use of several variables (and therefore more memory) but has substantial benefits in the execution phase.
For example in this case an integer type vector of only 8 elements would have been sufficient (one per compressor, each element would be the circuit to which the compressor belongs) but this vector would still have to be scanned from the first to the last element whenever something needs to be done on the compressors in the specific circuit.
However, this vector enables a simple multiplication and cycle interruptible on the first element found with the value “-1” to significantly reduce the processing times required for run-time, as will be outlined in the subsequent chapters.

4.3.5 Vector EvPresence

(Part of *IV_Ev*ap)

EvPresence[4 (MAX_EV)]: Boolean type vector with 4 elements: defines the presence of the i-th evaporator. Calculated by KOMP_CIR_EV[ ].
The *vector KOMP_CIR_EV* is scanned in the initialisation phase, and for each element with a value other than zero sets the presence value of the relative evaporator to true, with the index

```
((<value> DIV 10) – 1)
```

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

EvPresence				
Index in vector	**Evaporator**	**Contents (2-4-4) (BASELINE APPLICATION)**	**Contents (2-4-8)**	**Contents (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 1.3.6 Vector EvCir

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

Also in this case, the support of AppMaker for single dimension arrays only leads the use of a single vector in place of a two-dimension matrix (which would be the more logical solution for this variable).
Given that each evaporator can have at the most 4 circuits, and a maximum of 8 evaporators is possible, this vector is sized for 16 elements, and also in this case the vector is filled partially (many elements, those unused, will count as “-1”).
For the “i-th” evaporator (with 0 <= i <= (MAX_EV – 1)), the pertinent starting index will be obtained from 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 *vector KOMP_CIR_EV* is scanned in the initialisation phase, and for each index element “k” with a value other than zero sets the EV_CIR element to “(<value> MOD 10) ” 1 + offset” with the index

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

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

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

EvCir					
Index in vector	**Evap.**	**Contents (2-4-4) (BASELINE APPLICATION)**	**Contents (2-4-8)**	**Contents (2-8-8)**	**Contents (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)

Also in this case, note that the use of this type of vector leads to the use of several variables (and therefore more memory) but has substantial benefits in the execution phase.
For example in this case an integer type vector of only 8 elements would have been sufficient (one per circuit, each element would be the evaporator to which the circuit belongs) but this vector would still have to be scanned from the first to the last element whenever something needs to be done on the circuits of the specific evaporator.
However, this vector enables a simple multiplication and cycle interruptible on the first element found with the value “-1” to significantly reduce the processing times required.

4.3.7 Vector KOMP_STEP

(Part of *IV_Komp*)

KOMP_STEP[8]: maximum number of capacity steps for each compressor. Admissible range 0..4.

KOMP_STEP					
Index in vector	**Compr**	**Contents (2-4-4) (BASELINE APPLICATION)**	**Notes (2-4-4)**	**Contents (2-4-8)**	**Notes (2-4-8)**
0	1	3	On/off plus two capacity steps (0-33-66-100%)	1	Compressors type on/off for circuit I (0-100%)
1	2	3	On/off plus two capacity steps (0-33-66-100%)	1	Compressors type on/off for circuit I (0-100%)
2	3	3	On/off plus two capacity steps (0-33-66-100%)	2	Compressors type on/off plus one capacity step for circuit II (0-50-100%)
3	4	3	On/off plus two capacity steps (0-33-66-100%)	2	Compressors type on/off plus one capacity step for circuit II (0-50-100%)
4	5	0	(not used)	4	Compressors type on/off plus three capacity steps for circuit III (0-25-50-75-100%)
5	6	0	(not used)	4	Compressors type on/off plus three capacity steps for circuit III (0-25-50-75-100%)
6	7	0	(not used)	4	Compressors type on/off plus three capacity steps for circuit IV (0-25-50-75-100%)
7	8	0	(not used)	4	Compressors type on/off plus three capacity steps for circuit IV (0-25-50-75-100%)

4.3.8 Vector KompCir

(Part of *IV_Komp*)

KompCir[8]: integer type vector of 8 elements, one per compressor; lists the relative circuit for the i-th compressor (with i = 0..7), calculated by KOMP_CIR_EV[ ].
Used when, on a compressor level, information must be acquired regarding the circuit to which the compressor belongs.
The *vector KOMP_CIR_EV* is scanned in the initialisation phase, and for each index element "k" with a value other than zero sets

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

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

the element KompCir with the index "k".

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

KompCir				
Index 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 vector of 8 elements, one per compressor; lists the fan unit for the i-th compressor (with i = 0..7) Calculated by CIR_FANS[] and KompCir[].
Used when, on a compressor level, information must be acquired regarding the fan coil to which the compressor belongs.

4.3.10 Variables EvNo, CirNo, KompNo

(Part of *IV_Komp*)

These variables are not considered mandatory as the *"look up tables"* defined in the previous chapters (with the values "stage" "-1") are (and must be) in reality sufficient to guarantee correct access to all vectorised objects.
The decision whether to implement parameter/variable types "XxxNo" basically depends on the type of consistency checks to be implemented, to then be executed in *"one shot"* mode during initialisation.
Also these parameters, which define:

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

can in any event be calculated by KOMP_CIR_EV[].
The *vector KOMP_CIR_EV* is scanned in the initialisation phase, and for each index element "k" with a value other than 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 taken into consideration
c) increases the number of circuits (CirNo) if, and only if, the circuit
"(<value> MOD 10) - 1"
has not already been taken into consideration

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

These variables could become local on the initialisation phase (and therefore not visible to the rest of the application) depending on the selections made for consistency checks.

4.3.11 Checks of consistency and cycles on objects

Preliminary note

PLC applications do not typically contain consistency checks on the contents of the parameters on which the algorithms work. In fact it is supposed that it is task of MMI to perform the operations to check the configuration input made by an operator, refusing illegal parameters when necessary. If this is supposed to be true also for applications developed by the final user, some guidelines are provided in this section with the aim of highlighting any critical aspects that may have a negative influence on the behaviour of the machine. The consistency checks described in this chapter (and also in others) can therefore be considered as requirements for the operator interface management software (MenuMaker PRO).

Requirements for initialisation of KOMP_CIR_EV

- ∴ταβ The array is filled leaving no gaps;
- ∴ταβ It must always be filled starting from evaporator 1;
- ∴ταβ The evaporator index must be monotone in ascending order (max increment of 1);
- ∴ταβ For each evaporator the circuit index must be monotone in ascending order (max increment of 1);

An incorrect configuration is illustrated below (no evaporator 2).

KOMP_CIR_EV for machine 2-4-4 (configuration error)					
Index in vector	**Compr.**	**Contents**	**(Evaporator)**	**(Circuit)**	**Notes**
0	1	11	1	1	First compressor, compressor of first circuit of first evaporator
1	2	12	1	2	Second compressor, compressor of second circuit of first evaporator
2	3	31	3	1	Third compressor, compressor of first circuit of third evaporator
3	4	32	3	2	Fourth compressor, compressor of 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 must be considered wrong is open
to dispute. The decision to make these types of configuration possible or not (defined as "spreaded") must be made before the start of development as it has an impact on:

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

Requirements for initialisation of CIR_FANS

- ∴ταβ The array is filled leaving no gaps;
- ∴ταβ The fan unit circuit index must be monotone in ascending order (max increment of 1);

If the *"spreaded"* type configurations are considered possible, variable types "XxxNo" must NOT be present, as they could generate incorrect reactions in these configurations. In these configurations, the check cycles (see below) must be performed analysing all possible elements of the vectors through to their maximum (vector limit).
Vice versa, if *"spreaded"* type configurations are not considered possible, variable types "XxxNo" can be used by the applications, following a consistency check during initialisation, and the control cycles can be significantly optimised.

The complexity of a general conditioning machine very often involves "cross-references" between the various components making up the unit. The term "cross-reference" refers both to the obvious evaporator-circuit-compressor (top-down) hierarchy, but also the possible necessity to return through the hierarchical structure to upper objects to take information pertinent to the upper objects themselves. Normally this concept is also applicable to the "transversal" objects such as the fans and functions such as pump-down.
For example, with single condensation, each circuit still has its own sensor/digital input for pressure/temperature, even if several circuits are then combined on a single fan coil.
Control of the fan unit must therefore scan (creating the need for a cycle) all circuits referring to the specific fan unit to read the pressure/temperature values on the basis of which fan control is operated.
The fact that the number of these "cross-references" is not exactly limited, together with the fact that the user could decide to move up and down through the hierarchy in indented mode (from a cycle onto circuits to check something related to the circuit compressors, but in turn the control on compressors imposes the use of something related to the circuit), means that this problem must not be underestimated.
To avoid time-wasting processes and pointless calculations, the variables can be used in the cycles in the place 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 certainly less efficient than a second cycle type (also not admissible 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;
```

which is interrupted as soon as the first circuit set as "not present" is detected[3].
Note that in the first cycle the construct "for" has been used to execute the cycle itself and "(MAX_CIR - 1)" to identify the final limit of the scan, in this case on the circuits, while in the second case it uses the construct "while"[4].
A variant of the first cycle, without a doubt simpler and therefore clearer for non programmers, would be:

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

which still uses the construct "for" but uses "(NumCir - 1)" to identify the final limit of the scan, and which could be used if the "spreaded" configurations are not considered possible.
The resulting optimisation from the second cycle is significant and not to be underestimated: for a BASELINE APPLICATION type machine (four circuits) the time required for the scan of the second cycle is approximately half of that required for the first cycle (which in any event scans all eight circuits to see if they are present).
However, there is the problem of understanding whether there is a need to enable aspects such as the complete inhibition of a circuit or evaporator, while maintaining others enabled, for example "there are three circuits and the second is to be disabled".
In fact in this case the code of the second cycle would be inadvertently interrupted on the second circuit (which is disabled), preventing progress to the third which is still active.
For these cases, the proposed solution is to insert a new variable, or rather a new vector, to notify of the fact that the object may be enabled or disabled, leaving the presence flag on "present".

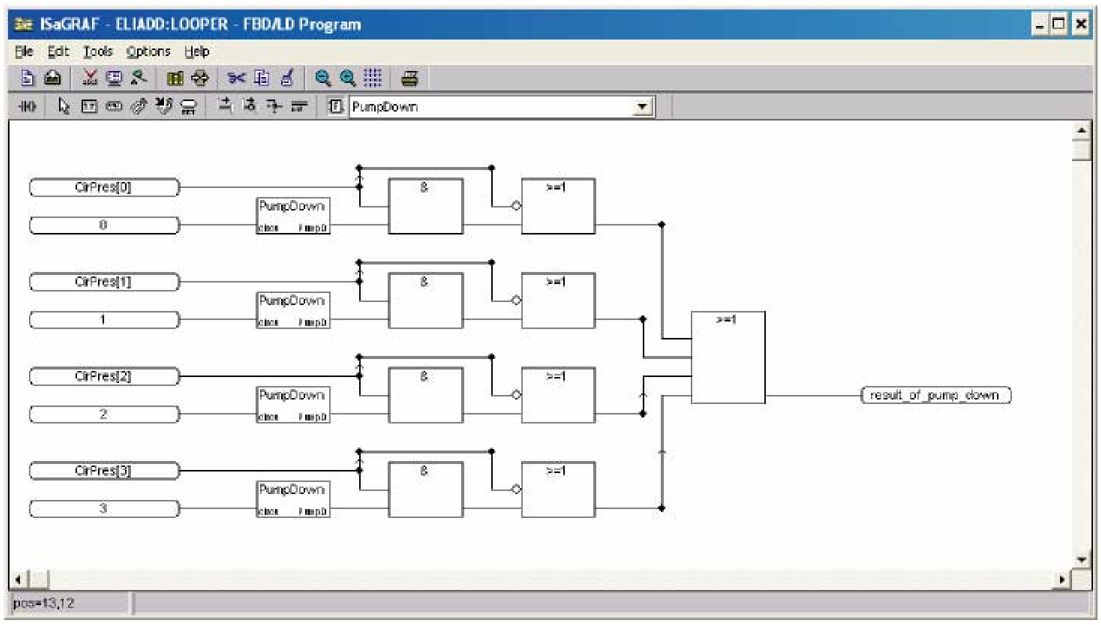
Execution of the algorithms would depend, where necessary, on a dual condition:

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

...
end_if;
```

where the variable CirActive[] indicates the "possibility" of the control using the i-th circuit and is therefore modifiable by means of MMI, whereas this is not possible for the variable CirPresence[] which defines a structural characteristic of the plant.

[3] Again this construct can only be used if the *"spreaded"* configurations are considered possible.
[4] AppMaker provides the construct "while … do … end_while" for perform conditioned loops, but generates a warning in compilation to notify of use of a potentially hazardous statement given that an incorrect condition on exit from the loop could influence execution times of the PLC cycle.

Another fact to bear in mind is the management of cycles on objects (compressors, circuits, evaporators) outside the ST code.
While in the case of a program written in “Structured Text” the choice of the type of cycle to perform is not totally critical (it can be modified very quickly), in other types of program (e.g. FBD/SFC) management of a while cycle could be complicated if not impossible, or would at least involve the *introduction* of additional programming level making the structure more rigid.
The simple ST cycle described above would not be feasible as FBD, unless feedback is introduced, which as managed in various PLC cycles[5] would make the application drastically slow with unforeseeable reactions; in this case the FBD should necessarily be “exploded in line” as in the following case, referred to 4 circuits.



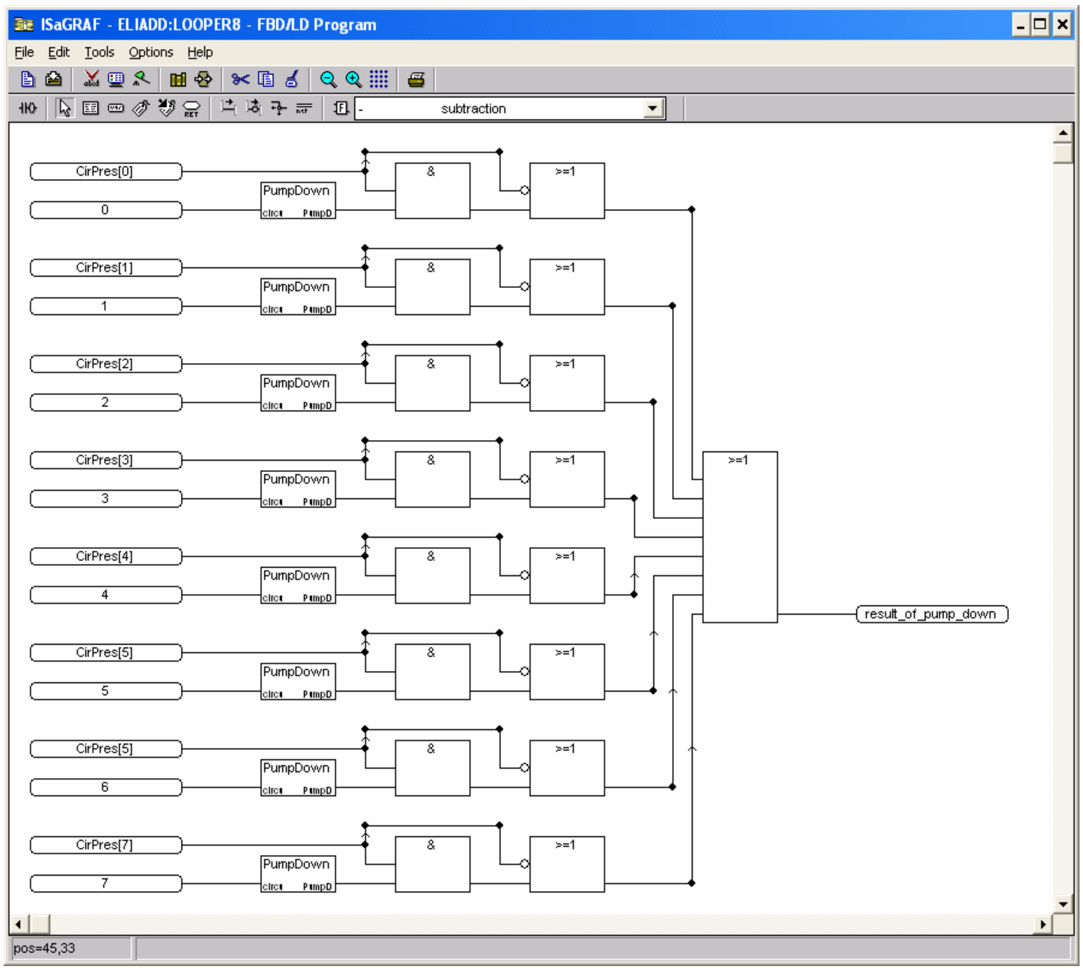
Note that when the circuits are exploded in line, there is no need for the use of the variables “XXX_NO” or “MAX_XXX”.

Implementation of the same function, for 8 circuits instead of 4, would required the production of a similar scheme but differing in some aspects (for example the final OR would be with 8 input). If on the one hand the work is simple and feasible also be personnel without specific high level programming backgrounds, the high number of copy-paste-modify actions subject to potential errors makes modifications a much more critical operation (for example the following FBD contains an error).

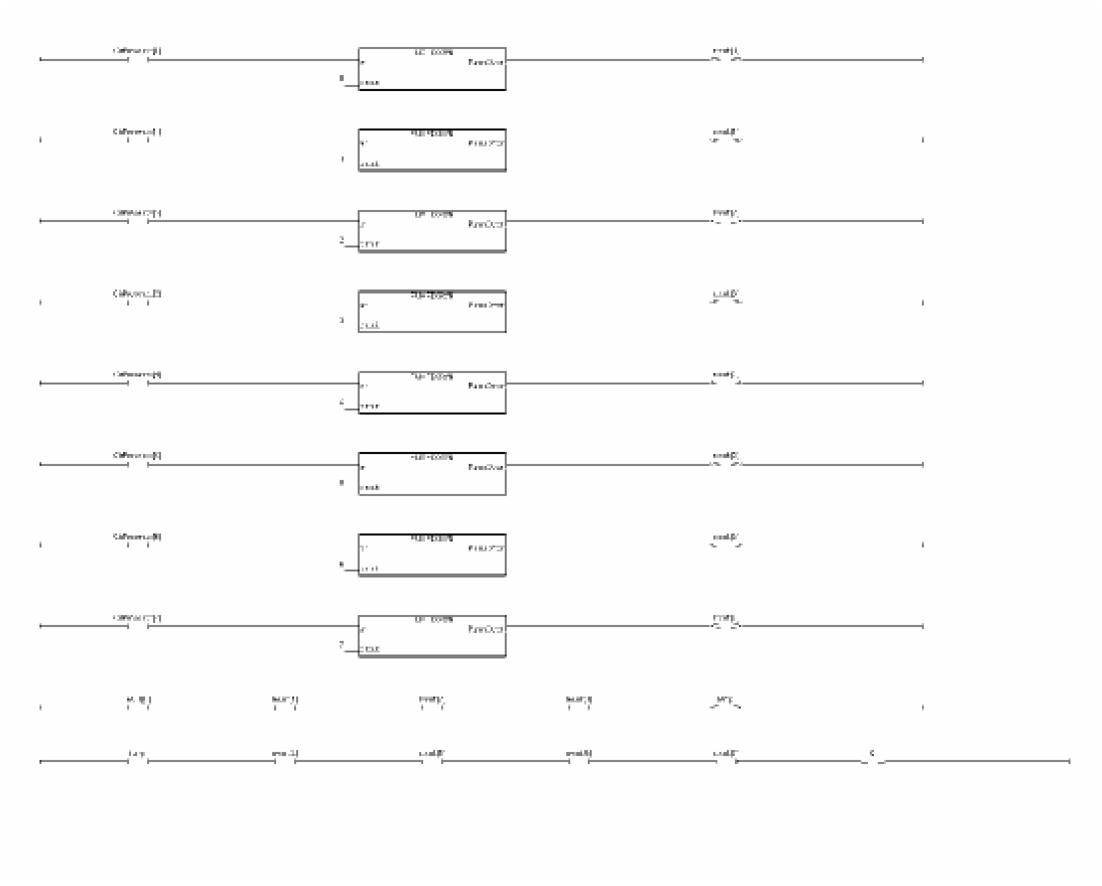
It must also be remembered that the contents code in the FBD would still be executed regardless of whether the objects on which it works are present or not (in the example the function “PumpDown” would always be called for all 8 circuits).
There could be the objection that execution within a function could be avoided by verifying, in the function itself, whether the specific object is effectively present, but in this case there would still be a pointless overhead for calling the function (and note that this overhead increases proportionally, the smaller the machine to be produced).
Alternatively, the use of Ladder Diagrams could avoid the execution of the code, by using the object presence flag in the contact to the left of the function.

[5] On each PLC cycle a single turn would be performed and the output used as input (feedback) would be taken into consideration only on the next PLC cycle

This hypothesis, though valid, would involve the use of additional local variables and *"rungs"* to retrieve the result of the function execution, as shown in the example below.



In the example the function that implements the circuit pump down is only executed for the circuits in which the presence flag (CirPresence) is true. Note that to obtain the overall result of the function there was the need to introduce the local vector "result", a temporary variable "temp" and two *"rungs"* to queue (and) and obtain the results of the single calls in a "legible" and printable manner.
Though more efficient than FBD, the LD must also be modified if the maximum number of objects to be treated is changed.
For these reasons the treatment of the loops is only recommended for the language ST.
The proposed architecture and variables referred to in this chapter lead to the recommendation to implement the control logics on two levels. The lower level must contain the "pure" control algorithm to be performed on the single object (for example the logic of pump-down of the single circuit), while the higher level, written in ST, must contain the loop to be performed on all objects present.
Both algorithms would be part of a library. In this way, the user would then be able to use the low level / high level, configure to client specifications the low level and use it for all circuits / configure to client specifications the high level and use it only for specific circuits etc., with the maximum freedom of configuration.



4.3.12 Parameter FANS_NO (Part of IV_Fans)

FANS_NO[8]: integer vector of 8 elements; contains the number of fans of each fan coil. Initialised once only in the start-up phase by parameters FANS_NO_1…8. Admissible range 1…4.
This vector is configured manually.
For example, in the case of the BASELINE APPLICATION machine, the vector must be initialised as follows, taking care that there are no gaps:

FANS_NO			
Index in vector	**Fan Coil**	**Contents (BASELINE APPLICATION)**	**Notes (2-4-4)**
0	1	3	Three fans in first fan coil
1	2	3	Three fans in second fan coil
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 vector of 8 elements; contains the association between circuit and fan coil. Initialised once only in the start-up phase by parameters CIR_FANS_1...8. Admissible range 0...2, the value 0 indicates that no fan coil is associated with the specific circuit.
This vector is configured manually.
For example, in the case of the BASELINE APPLICATION machine, the vector must be initialised as follows, taking care that there are no gaps and that numbering is monotone, ascending (max. increment of 1):

CIR_FANS			
Index in vector	**Circuit**	**Contents (BASELINE APPLICATION)**	**Notes**
0	1	1	The first circuit belongs to the first fan coil
1	2	1	The second circuit belongs to the first fan coil
2	3	2	The third circuit belongs to the second fan coil
3	4	2	The fourth circuit belongs to the second fan coil
4	5	0	(not used)
5	6	0	(not used)
6	7	0	(not used)
7	8	0	(not used)

4.3.14 1.3.14 FansCir (Part of IV_Fans)

FansCir[8*8]: vector of MAX_FANGROUPS * MAX_CIR4FANGROUP integer elements; lists the circuits belonging to the i-th fan unit (with i = 0..7) Calculated by CIR_FANS [].
The support of AppMaker for single dimension arrays only leads the use of a single vector in place of a two dimension matrix (which would be the more logical solution for this variable).

Given that each circuit can have at the most 8 circuits , and a maximum of 8 fan coils is possible, this vector is sized for 64 elements, even though in this case the overall limit of 8 circuits implies that the vector is either way filled only partially (many elements, those unused, will count as "-1").

For the "i-th" coil (with 0 <= i <= (MAX_FANGROUPS – 1)), the pertinent starting index will be obtained from the expression:

`i * (`MAX_CIR4FANGROUP`)` (representing an offset in the vector)

where "MAX_CIR4FANGROUP" is the constant that defines the maximum number of circuits per fan coil; the valid elements for this coil will therefore be MAX_CIR4FANGROUP.
The vector CIR_FANS is scanned in the initialisation phase, and for each index element "k" with a value other than zero sets the FansCir element to "k" with the index

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

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

FansCir			
Index in vector	Fan Coil	Contents (BASELINE APPLICATION)	Notes
0	1	0	The first circuit belongs to the first fan coil
1	1	1	The second circuit belongs to the first fan coil
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 coil
9	2	3	The fourth circuit belongs to the second fan coil
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 1.3.15 FansNo (Part of IV_Fans)

This variable is not considered mandatory as the “look up tables” defined in the previous chapters (with the values “stage” “-1”) are (and must be) in reality sufficient to guarantee correct access to all vectorised objects.
The decision whether to implement parameter/variable types “XxxNo” basically depends on the type of consistency checks to be implemented, to then be executed in “one shot” mode during initialisation.
This parameter defines the total number of fan coils to be controlled (FansNo, with 0 < FansNo <= MAX_FANGROUPS) and can in any event be calculated by CIR_FANS[].
The vector CIR_FANS is scanned in the initialisation phase, and for each index element “k” with a value other than zero it increases the number of coils (FansNo) if, and only if, the coil “<value>” has not already been taken into consideration.

In the Baseline application FansNo=2.

4.3.16 Parameter PUMP_NO

This parameter is used to set the number of pumps to be managed by the plant.
The range of this parameter must be 0..MAX_PUMP, where MAX_PUMP is the constant that defines the maximum number of pumps manageable by the application.
For the BASELINE APPLICATION, the maximum number of pumps, equivalent to the number of pumps to be managed, is two,
The code for management for all that related to the pumps depends on the fact that the number of pumps is “greater than zero”, as shown in the example below in “ST” code :

```
if PUMP_NO > 0 then
        (* pump management *)
end_if;
```

Obviously cycles operating on vector types PumpsXxx[MAX_PUMP] will also be possible. These cycles will have the logic condition of termination “index < (PUMP_NO – 1) :

```
for pump_idx := 0 to (PUMP_NO – 1) do
(* management of pumps of i-th (with the index “pump_idx) *)
…
end_for;
```

4.4 Constants (Defined words)

Even though it may seem excessive go this far on a definition level in an architecture document, this section provides a preliminary list of the main constants.
Obviously the list is not totally comprehensive but serves to identify the areas for which constants are deemed necessary.

Name	Equivalence	Comment
MAX_CIR	8	Maximum number of machine circuits
MAX_EV	4	Maximum number of machine evaporators (2 CVM)
MAX_KOMP	8	Maximum number of machine compressors
MAX_PUMP	2	Maximum number of machine pumps
MAX_FANGROUPS	8	Maximum number of machine fan coils (2 CVM)
MAX_FANS	16	Maximum number of machine fans (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 unit (4 CVM)
MAX_CIR4FANGROUP	8	Maximum number of circuits per fan unit (4 CVM)
KIM_STANDARD	0	Standard compressor start-up
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 sub-system
P_CIR	1	circuit sub-system
P_EV	2	evaporator sub-system
P_PLAN	3	Plant sub-system
P_DEF	4	Defroster sub-system
SATURATION	false	definition for saturation policy
BALANCING	true	definition for balancing policy
KOMP_OFF	0	*OFF status* of i-th compressor
KOMP_ON_NR	1	*ON Not Ready* status of i-th compressor
KOMP_ON	2	*ON status* of i-th compressor
KOMP_OFF_NR	3	*OFF Not Ready* status of i-th compressor
ENTRY_SENS	0	definition for water inlet sensor
EXIT_SENS	1	definition for water outlet sensor
TREG_*PI*	2	Thermoregulation P.I.
TREG_TIMEPROP	1	Time-proportional thermoregulation
TREG_PROPORTIONAL	0	Proportional thermoregulation
SEMIERMETICO	0	semi-hermetic type compressor
VITE	1	screw type compressor
PLAN_OFF	0	*OFF status* of plant
PLAN_SHUTDOWN	1	SHUTDOWN status of plant
PLAN_ON	2	*ON status* of plant

PLAN_MODE_CHILLER	FALSE	Plant in CHILLER mode via keyboard
PLAN_MODE_POMPA	TRUE	Plant in PUMP mode via keyboard
PUMPGROUP_OFF	0	*OFF status* of pump unit
PUMPGROUP_GOING_UP	1	*ON status* of pump unit from plant ON
PUMPGROUP_ON	2	*ON status* of pump unit
PUMPGROUP_GOING_DOWN	3	*ON status* of pump unit 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
DTSET_CURR	2	Dynamic Set point enabled in current
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 of the same power
PDA_START	0	start pump-down on start-up
PDA_HANDLE	1	management of pump-down on start-up
PDS_START	2	start pump-down on shutdown
PDS_HANDLE	3	management of pump-down on shutdown
PD_RESET	4	pump-down management reset
solenoid_open	false	solenoid valve open
solenoid_close	true	solenoid valve closed
PD_NONE	0	pump-down not supported
PD_ONSTART	1	pump-down on start-up
PD_FULL	2	pump-down on start-up and shutdown
pd_max_press_reached	false	pump-down pressure switch indicates max. pressure reached
pd_min_press_reached	true	pump-down pressure switch indicates min. pressure reached
NOT_IN_PD	0	not in pump-down
PDA1	1	pump-down on start-up phase 1
PDA2	2	pump-down on start-up phase 2
PDS	3	pump-down on shutdown
FINISH_PDA	4	pump-down on start-up terminated
FINISH_PDS	5	pump-down on shutdown terminated
TMR_IDLE	0	idle command to timer
TMR_RESET	1	reset command to timer
TMR_START	2	start command of timer
TMR_SUSPEND	3	timer count suspension command
TMR_OFF	0	*OFF status* of timer
TMR_RUNNING	1	timer in active count status
TMR_SUSPENDED	2	timer suspended in count status
TMR_EXPIRED	3	expired timer status
DEF_IDLE	0	defrost not active status
DEF_ENTER_DOWN	1	circuit shutdown status on defrost input
DEF_PRE_INV_VALVE	2	inversion valve standby status
DEF_POST_INV_VALVE	3	circuit in defrost activation standby status
DEF_GOING_UP	4	circuit in defrost activati*on status*
DEF_STABLE	5	stable defrost status
DEF_GOING_DOWN	6	circuit in defrost shutdown status
DEF_WAIT_DRIP	7	circuit off on standby for drip status
DRIP_PRE_INV_VALVE	8	drip on standby for inversion valve status
DRIP_POST_INV_VALVE	9	drip on standby for drip end status
inversion_valve_in_chiller	false	Inversion valve in chiller mode status
inversion_valve_in_heatpump	true	Inversion valve in heat pump mode status
MAX_VALUE	false	requests maximum
MIN_VALUE	true	requests minimum
SKIP	true	Skips statuses OFF_NR and ON_NR of compressors
NOT_SKIP	false	Does not skip statuses OFF_NR and ON_NR of compressors
DEF_STANDARD	4	Standard defrosting

DEF_NONE	5	Defrosting not enabled
COMPE_IDLE	false	compensation not active status
COMPE_GOING_ON	true	compensation active status
FANS_DIGI	1	Fan control in digital mode
FANS_CONT	0	Fan control in continuous mode
MAX_HISTORY_ELEMENTS	50	Maximum number of alarms in alarm history
HISTORY_WRITE	false	element writing in alarm history
HISTORY_READ	true	element reading in alarm history
HISTORY_LOCKED_TIME	t#60s	Alarm history download timeout
PLANTEMPINWATERSENSERR_COD	0	Water inlet temperature sensor error code
PLANTEMPOUTWATERSENSERR_COD	1	Water outlet temperature sensor error code
PLANCURRDTSETSENSERR_COD	2	Sensor in current for dynamic set point error code
PLANHTEMPA_COD	3	Plant high temperature alarm code
PLANLTEMPA_COD	4	Plant low temperature alarm code
CIRPRESMAXSENSERR_COD	5	Maximum circuit pressure sensor error code
CIRHPRA_COD	6	Maximum circuit pressure alarm code
CIRLPRA_COD	7	Minimum circuit pressure alarm code
KOMPTEMPDISCHARGESENSERR_COD	8	Compressor discharge temperature sensor error code
KOMPTHERA_COD	9	Compressor thermal cut-out alarm code
KOMPDISA_COD	10	Compressor discharge temperature alarm code
PUMPTHERA_COD	11	Pump thermal cut-out alarm code
FLOWA_COD	12	Automatic and/or blocking flow switch alarm code
FANSTHERA_COD	13	Fan set thermal cut-out alarm code
EVTEMPOUTWATERSENSERR_COD	14	Evaporator water outlet temperature sensor error code
EVAFA_COD	15	Evaporator anti-freeze alarm code
MAX_BBX_ELEMENTS	20	Maximum number of elements for collection from black box
MAX_BBX_FILES_NUM	3	Maximum number of memorisable collections
BBX_STEP_0	0	Init MSF of black box
BBX_STEP_1	1	Black box header memorisation in FLASH
BBX_STEP_2	2	Black box sample memorisation in FLASH
BBX_STEP_3	3	End of writing collection of black box in FLASH
BBX_STEP_4	4	Entry of samples of collection in RAM
BBX_LOCKED_TIME	t#180s	Black box download timeout

Note that the constants have been divided into groups
The first group contains the constant types “MAX_XXX”, in other words the constants that "limit" the physical dimensions of the plant and take on values according to which the arrays have been dimensioned.
The other groups contain the definition of Boolean logic variables referable therefore only to “true” and “false” values[6] or the numbering of internal states of the machine.
The use of Defined Words makes the application mode legible and less ambiguous. For example, the fact that for a solenoid valve open, the status of the output line that controls it must be 1/on/active, i.e. “true” could be disputed and may be a source of incomprehension, while the use of a construct similar to the one below clarifies and specifies the required behaviour of the algorithm.

```
IF (PD_FUNCTION <> PD_NONE) THEN

        IF ((PdStatus[i] = PDA1) OR (PdStatus[i] = FINISH_PDA)) THEN
                CirSolenoidValve[i] := solenoid_open;
        ELSE
                CirSolenoidValve[i] := solenoid_close;
        END_IF;

        IF boo(CirLPrA[i]) THEN
                CirSolenoidValve[i] := solenoid_open;
        END_IF;

ELSE

CirSolenoidValve[i] := solenoid_open;

END_IF;
```

[6] These aliases are currently indicated in the document using lower case letters, but as they are constants, they may follow the nomenclature envisaged for constants (all upper case)

5 DESCRIPTION OF THE BASELINE REVERSIBLE HEAT PUMP APPLICATION

For each of the program units outlined in the previous chapter
details are provided of the main data structures and the control algorithm section.

5.1 IniVar

In this first section of the initial block all data structures describing the machine structure are initialised as described in chapter 3, as well as management of all functions that are normally present in each AppMaker project as inputs in configuration mode. Initialisation also "expands" any other "compressed" parameters set by the configurator/MMI , in the relative vectors used by the PLC applications. In this case we are talking of single parameters settable by the configurator/MMI valid for all elements to which they refer (a single threshold for all circuit alarms, a single flag to enable a specific type of sensor for all compressors etc.). When the configurator/MMI initially enables the entry of a single parameter, valid for all system elements (compressor, circuit, evaporator) although the PLC applications still need to be developed to enable inhomogeneous management (element by element) of this parameter, this program unit will provide for relative expansion with the code type:

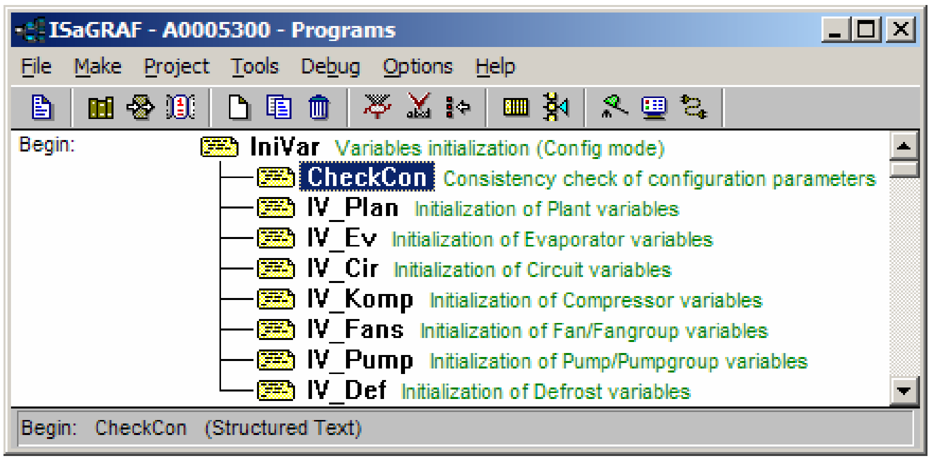
```
for cir_idx := 0 to (MAX_KOMP – 1) do
  CirHPresADelta[cir_idx] := A_MAX_DELTA_PRES;
end_for;
```

Once the configurator/MMI is updated, this expansion will no longer be necessary and may be removed. In this way the user can decide whether and which parameters will be differentiated for the various elements of the same type and if a single parameter will be implemented. The choice will be made on the basis of functional requirements (for example to have compressors with different numbers of capacity steps) within the limits set by the platform both in terms of memory dimensions and execution time.

It is important to note that since the reversible heat pump structure is asymmetrical and potentially unbalanced, the data vectors describing the machine structure must enable the "reconstruction", and therefore traceability of the entire hierarchical tree both in top-down mode (from the root to leaves) and down-top mode, i.e. from the leaves to the roots. This explains the presence of data structures that describe which are the "children" of each element , and structures that indicate the "father" (or fathers) of each node.

5.1.1 CheckCon

For this program unit no special support structures are envisaged, apart from the variable/parameter that enables one-shot execution and the result of consistency checks which inhibit any PLC operation.


ISaGRAF - A0005300 - Programs

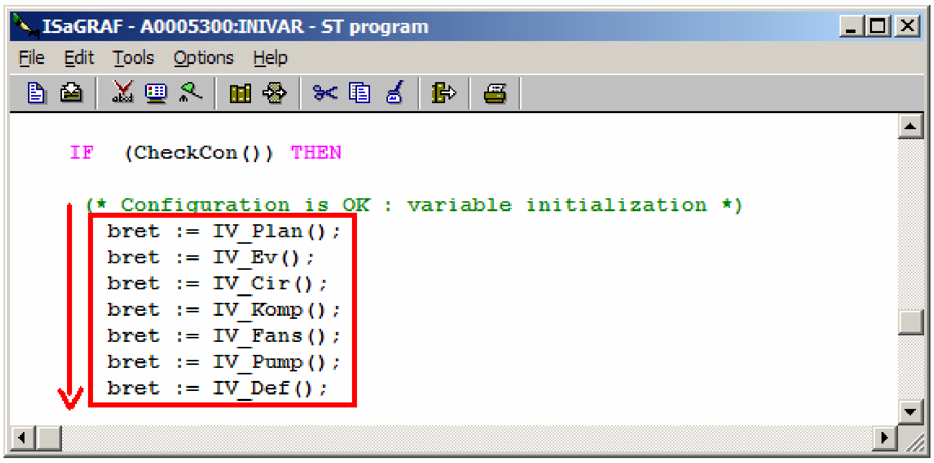
File Make Project Tools Debug Options Help

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
- IV_Def Initialization of Defrost variables

Begin: CheckCon (Structured Text)

In the BASELINE APPLICATION this function is "dummy" type, i.e. always returns a "true" value (*CheckCon* := true;). In fact it has been decided to leave the option to the application developer whether to personalise this function, depending on which are the "sensitive" points of its application.

It is extremely important to note that the order of execution of the single Program Units that are "daughters" of another Program Unit does NOT follow the order with which these are entered in the AppMaker project, but the order in which they are invoked within the father Program Unit.


ISaGRAF - A0005300:INIVAR - ST program
File Edit Tools Options Help

```
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();
  bret := IV_Def();
```

In the example shown in the figure, the two orders coincide, but should *IV_Plan*() and *IV_Ev*() change places in the ST program contained in InitVar, the effect would be that the Evaporator variables would first be initialised, followed by initialisation of the Plant variables.

5.1.2 IV_Plan

The “InitVar” of “Plant” has the main function of copying the I/O support variables onto the relative single dimension array structures, to then be involved in the architecture computation. This means first “filling” the *vector KOMP_CIR_EV*[] which as seen in chapter 3 represents the “family” relations between circuit compressors and evaporators. As this is a high level configuration, this will also acquire the vector that describes the relation between fan coils and circuits, in other words CIR_FANS[].All Plant alarms and timers are then reset.

5.1.3 IV_Ev

The initialisation of the Evaporator variables has the task of filling the structures of EvPresence[] and of EvCir[], i.e. the structures that define the presence (or not) of the evaporator (enable) and therefore the family relations with the circuits. It is these data structures that enable the specification of which circuit hierarchically belongs to which evaporator. All Evaporator alarms and timers are then reset.

5.1.4 IV_Cir

This describes all compressors present (enabled) to calculate the hierarchical links between circuits and upper structures (evaporators) and lower structures (compressors) . As already seen in chapter 3 the vectors CirPresence[], CirEv[] , CirKomp[] represent these links.
Also in this case all circuit alarms and timers are reset.

5.1.5 IV_Komp

As the architecture supports different types of compressor, the structure KOMP_STEP[MAX_KOMP] will need to be filled, which describes the characteristics (in terms of capacity steps) of each compressor. Subsequently the filling of the structure KompCir[] will describe the hierarchical relation between each compressor and its relative dependent circuit. All compressor alarms and timers are then reset.

5.1.6 IV_Fans

In this *program unit* the structure FANS_NO[MAX_FANGROUPS] is initialised, which as seen above describes the link between fans and circuits. All additional data structures to the fans are also initialised, i.e. FANS_CSTART_SET_PRES[] and FANS_CSTOP_DELTA_PRES[]. FansCir[] is calculated and then all fan set alarms and timers are reset.

5.1.7 IV_Pump

The two pumps do not require special initialisations, apart from the re-alignment of the use timers (in PumpHours[]) and reset of the relative alarm variables.

5.1.8 IV_Def

The two pumps do not require special initialisations, apart from the re-alignment of the use timers, reset of the relative alarm variables and the creation of the vector CirMaxPowerCir[] used for the algorithm of compensation in defrosting.

5.2 Phy2Log

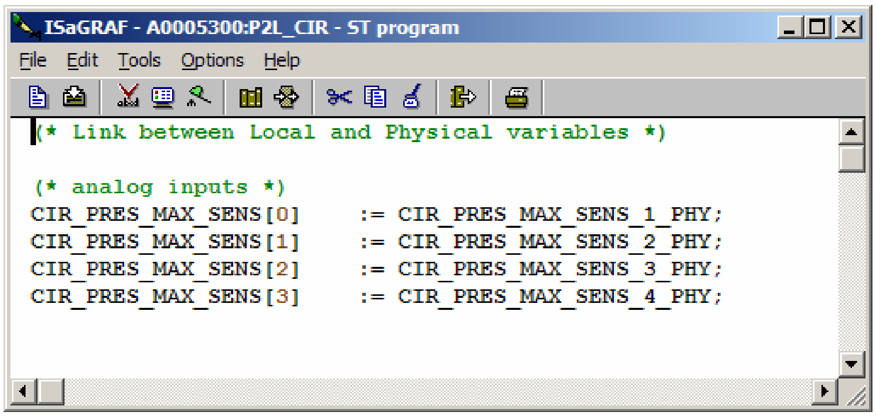
5.2.1 P2L_xxx

The task of this group of *program units* is to copy all physical dimensions of the system onto the same number of “logical” variables. As defined in the chapter on *nomenclature* of variables and parameters, all variables referring to a physical dimension have names terminating with the suffix _PHY. In the same way “HOT” type parameters are named with the suffix “HOT”.

Therefore there are the following program units, one per structural element of the reversible heat pump:

• *Phy2Log*	(Main Program unit)
• P2L_Plan	(Conversion from physical IN to logical IN on system level)
• P2L_Ev	(Conversion from physical IN to logical IN on evaporator level)
• P2L_Cir	(Conversion from physical IN to logical IN on circuit level)
• P2L_Komp	(Conversion from physical IN to logical IN on compressor level)
• P2L_Fans	(Conversion from physical IN to logical IN on fan unit level)
• P2L_Pump	(Conversion from physical IN to logical IN on pump unit level)

This type of operation (i.e. copying of physical variables to logical variables) is necessary as the controller terminals can only be assigned with specific variables of I/O which CANNOT be vectors. Therefore if it becomes necessary to use I/O variables defined on arrays; this has to be via the support variables which are then copied into the vectors used for algorithm computation.


ISaGRAF - A0005300:P2L_CIR - ST program
File Edit Tools Options Help

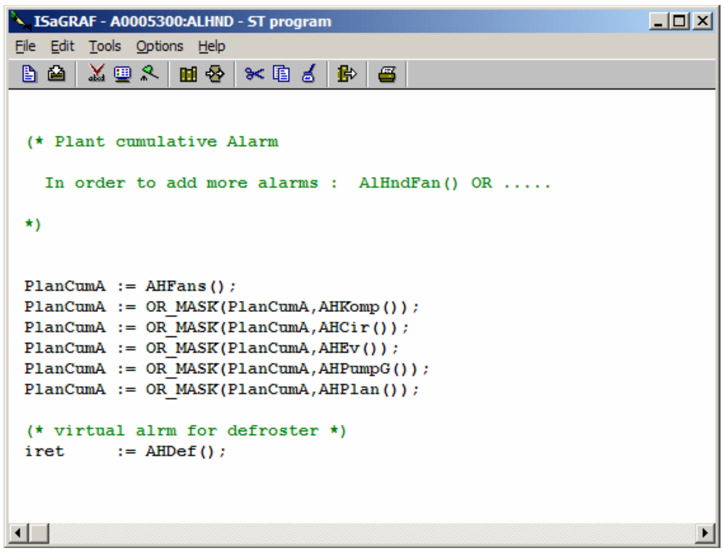
```
(* Link between Local and Physical variables *)

(* 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;
```

In order to maintain the code "clean" and "rational" , physical to logical conversions are performed on all subsystems of the *r*eversible heat pump, even in situations where this would not be necessary, as in AppMaker that which is not "updated" is not "refreshed".

5.3 AlHnd

For the elements of the system, plant, evaporator, circuit etc., but also for some functions (free cooling, recovery, anti freeze etc) there is a nominal behaviour, and also one or more anomalous conditions (alarms) which influence the behaviour to be implemented by the logics.

Given this fact, this program unit therefore has the task of verifying the alarm conditions of any type, and of generating possible block conditions which have a direct consequence on the control calculation and therefore on the outputs.


ISaGRAF - A0005300:ALHND - ST program
File Edit Tools Options Help

```
(* 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());

(* virtual alrm for defroster *)
iret      := AHDef();
```

As already explained, the process proceeds according to the order with which the various functions are invoked by the "father" Program Unit. In this case the first to be computed is the management of fan alarms (*AHFans*()).

It is extremely important to note that the alarm of a component can have a domino effect on the alarm status of other components. Due to this fact, it is fundamental to identify an alarm "hierarchy", and to evaluate the machine alarm status in a "cumulative" manner, from the least important element to the most important one in the hierarchy of the alarms itself.

The section below shows the order in which the various Alarm Handling program units are considered, according to this principle: the alarm status of the function *AH_componente* considered will depend on the alarm status of the component itself and the alarm status of all that examined up to that time.

5.3.1 AHFans

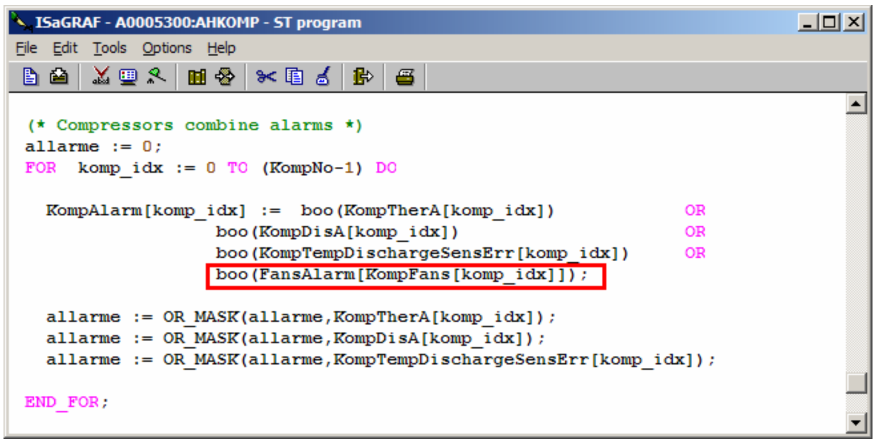
After evaluating the presence of a thermal cut-out alarm of the fan set, it is sufficient to analyse the status of each single fan, after which an "or" logic is implemented on all possible alarms.
Note that *AHFans* is executed first as the possible alarm status of the fan group depends exclusively on the fans themselves, and therefore the computation of the alarms does not need a check of whether other parts of the system are in alarm status or not.

5.3.1.1 AHFansTh

In this program unit a check is made for the presence of a fan coil thermal cut-out alarm on the coils present..

5.3.2 AHKomp

A check on the alarm status of the compressors depends on the status of the compressors themselves and possible fan alarm status.


ISaGRAF - A0005300:AHKOMP - ST program
File Edit Tools Options Help

```
(* 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 can be seen in the part selected within the red frame, the alarm status of each compressor is the sum of all potential alarm conditions as well as the alarm status of the fans, which therefore must have already been computed and thus be available.

5.3.2.2 AHKompEr

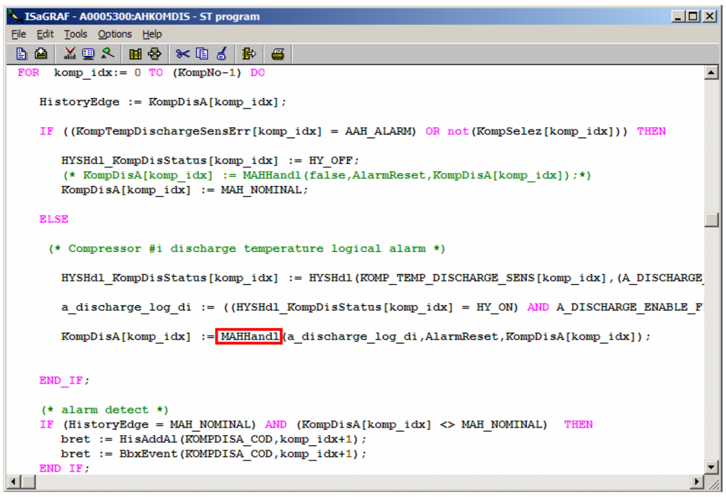
In this program unit a check is made for the presence of any discharge temperature probe errors. Note the presence of the function AAHHandl(), i.e. the function of managing the alarm "discharge temperature probe error" with automatic reset.

5.3.2.3 AHKompTh

In this program unit a check is made for the presence of a compressor thermal cut-out alarm only on the compressors present.

5.3.2.4 AHKompDis

In this program unit a check is made for the presence of any compressor discharge temperature errors.



ISaGRAF - A0005300:AHKOMDIS - ST program
File Edit Tools Options Help

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

   HistoryEdge := KompDisA[komp_idx];

   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_F

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


   END_IF;

   (* alarm detect *)
   IF (HistoryEdge = MAH_NOMINAL) AND (KompDisA[komp_idx] <> MAH_NOMINAL)  THEN
      bret := HisAddAl(KOMPDISA_COD,komp_idx+1);
      bret := BbxEvent(KOMPDISA_COD,komp_idx+1);
   END_IF;
```

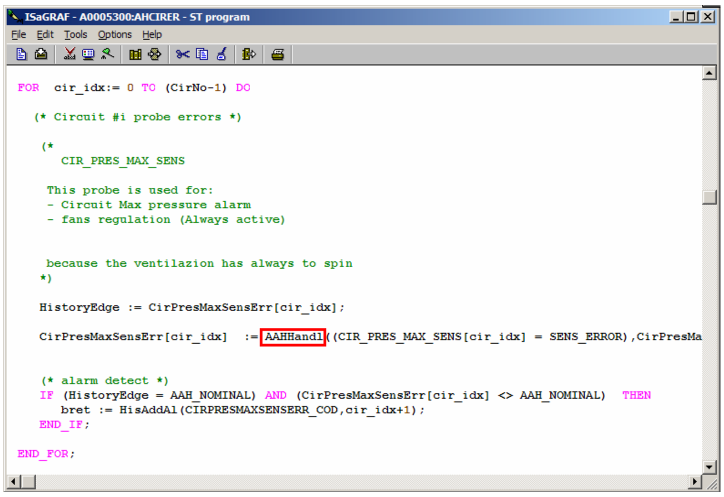
Note in the calculation of the i-th compressor discharge alarm, the presence of the function MAHHandl() is used for management of alarms with **manual reset**.

5.3.3 AHCir

Circuit alarm management is in hierarchical order immediately after that of compressor management. Its logic is the same as that of the compressors.

5.3.3.5 AHCirEr

The aim of this program unit is to calculate any probe errors of the i-th compressor. The logic is the same as that of the compressors, but note also the presence of the function AAHHandl(), i.e. the function for managing "probe maximum pressure error" alarms with automatic reset.



ISaGRAF - A0005300:AHCIRER - ST program
File Edit Tools Options Help

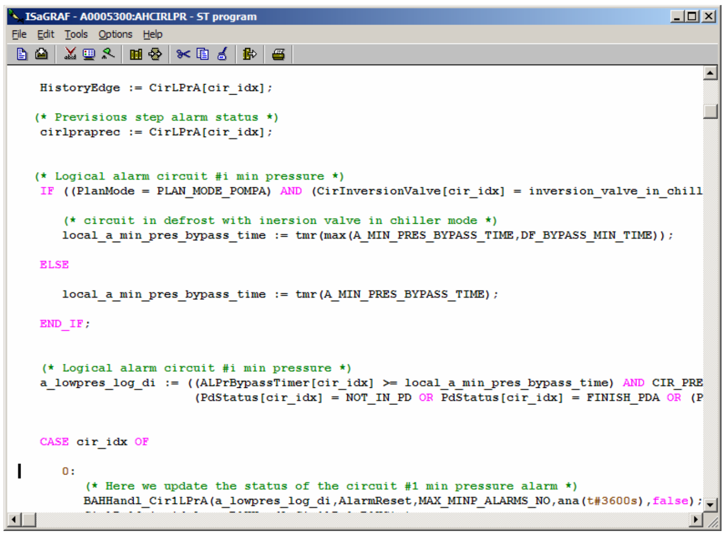
```
FOR  cir_idx:= 0 TO (CirNo-1) DO

  (* 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
   *)

   HistoryEdge := CirPresMaxSensErr[cir_idx];

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


   (* alarm detect *)
   IF (HistoryEdge = AAH_NOMINAL) AND (CirPresMaxSensErr[cir_idx] <> AAH_NOMINAL)  THEN
      bret := HisAddAl(CIRPRESMAXSENSERR_COD,cir_idx+1);
   END_IF;

END_FOR;
```

5.3.3.6 AHCirHPr and AHCirLPr

The aim of these two functions is to compute the presence of high pressure (AHCirHPr) or low pressure (AHCirLPr) alarms. Computation is particularly complex with respect to previous procedure, and the need to use "bounded" alarms, preventing the use of vectors, forces the use of artifices in code writing, replicating the same code for the maximum number of circuits. Note that if the user wishes to extend the DOMAIN of the BASELINE APPLICATION, increasing the number of possible circuits, this Program Unit will also have to be extended, modifying it accordingly to enable work on an extended domain. An immediate example is that which, given that the circuit index variable (cir_idx) can take on values greater than 8 (assuming an increase in the maximum number of circuits) the number of "cases" considered will have to be increased, adding code similar to that shown below:

5.3.3.7 AHCirPD

The aim of this function is to calculate the pumpdown alarm timeout.


ISaGRAF - A0005300:AHCIRLPR - ST program
File Edit Tools Options Help

```
  HistoryEdge := CirLPrA[cir_idx];

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


 (* Logical alarm circuit #i min pressure *)
  IF ((PlanMode = PLAN_MODE_POMPA) AND (CirInversionValve[cir_idx] = inversion_valve_in_chill

     (* circuit in defrost with inersion valve in chiller mode *)
     local_a_min_pres_bypass_time := tmr(max(A_MIN_PRES_BYPASS_TIME,DF_BYPASS_MIN_TIME));

  ELSE

     local_a_min_pres_bypass_time := tmr(A_MIN_PRES_BYPASS_TIME);

  END_IF;


   (* Logical alarm circuit #i min pressure *)
  a_lowpres_log_di := ((ALPrBypassTimer[cir_idx] >= local_a_min_pres_bypass_time) AND CIR_PRE
                      (PdStatus[cir_idx] = NOT_IN_PD OR PdStatus[cir_idx] = FINISH_PDA OR (P


  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);
```

5.3.4 AHEv

Management of the logic alarms of the evaporator do not have other features with respect to the ones described until now, and in fact involves managing potential errors due to the breakage of one or more probes (computed in *AHEvEr*), and the evaporator anti-freeze alarm (computed in *AHEvAf*).

5.3.4.8 AHEvEr

The function manages the probe error in the evaporator module.
Note the presence of the function AAHHandl(), i.e. the function of managing the “water outlet temperature probe error” alarm with automatic reset.

5.3.4.9 AHEvAf

In this Program Unit the evaporator anti-freeze alarm is managed.

5.3.5 AHPumpG

Management of the logic alarms of the pumps/”pump unit” integrating pump swap with the option of a flowswitch alarm.

5.3.5.10 AHPumpTh

The aim of this program unit is to evaluate the possible presence of a pump thermal cut-out.

5.3.6 AHDef

Management of the defrosting system alarms (defroster) must enable the combination of alarms related to all elements involved in defrosting. Therefore all compressor alarms must be checked, subordinate to each fan coil, and the latter for each circuit.

5.3.7 AHPlan

Management of the plant alarms must enable the combination of the plant alarms and the lower level alarms suitably combined to reach the “brain ” of the machine for use in the subsequent control phases.

5.3.7.11 AHPlanEr

The local plant errors are usually those tied to the breakage of probes, and in particular to the probes for regulation of water inlet, water outlet and that for set point management.

5.3.7.12 AHPlantHT

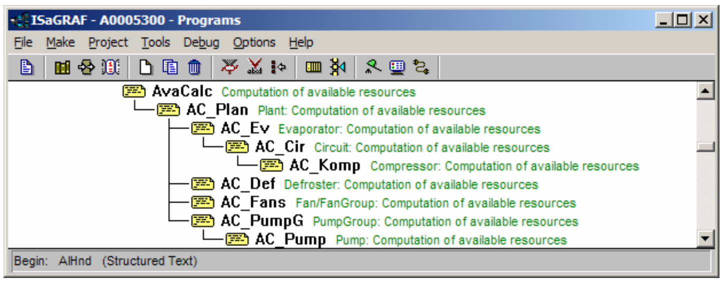
In this case the aim of the Program Unit is to evaluate computation of the plant high temperature alarm.

5.3.7.13 AHPlantLT

In this case the aim of the Program Unit is to evaluate computation of the plant low temperature alarm.

5.4 AvaCalc

After executing the alarm section management, a fundamental point in the thermoregulation process is to compute the of refrigeration resources (machine) "available".


ISaGRAF - A0005300 - Programs
File Make Project Tools Debug Options Help

```
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_Def  Defroster: 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:  AlHnd   (Structured Text)
```

In conceptual terms the calculation of availability is a process that starts from the less important machine elements which each inform their own "father element" of which and how many resources (cooling and physical) they are able to supply in the event of need.
Therefore each compressor should inform the circuit to which it belongs of how many capacity "steps " it is able to supply on request. In turn the circuit, after collecting data on all the availability of all compressors belonging to it, must inform the evaporator to which it is subordinate on how many resources it can supply on request. It is evident that this power will be the sum of the resources of each compressor which belongs to it. In the same way the power that each evaporator can supply as data available to the "plant" will be the sum of the powers available from each circuit that belongs to it.

It must be noted that the calculation of the resources available is not limited to a simple mathematical sum of the availability of the compressors, but must take into account any situations/states in which the system may find itself, which, although the compressors may be "available" to provide cooling power, one or more upper hierarchical components may not be able to do so.

5.4.1 Status variables

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

KompMinLevDin[8]	Minimum compressor dynamic availability
KompMaxLevDin[8]	Maximum compressor dynamic availability
CirMinLevDin[8]	Minimum circuit dynamic availability
CirMaxLevDin[8]	Maximum circuit dynamic availability
EvMinLevDin[4]	Minimum evaporator dynamic availability
EvMaxLevDin[4]	Maximum evaporator dynamic availability
PlanMinLevDin	Minimum plant dynamic availability
PlanMaxLevDin	Maximum plant dynamic availability
FansActivable[8]	Fan unit availability
PumpGMinLevDin	Minimum pump unit dynamic availability
PumpGMaxLevDin	Maximum pump unit dynamic availability
PlanPumpGMinLevDin	Min. dyn. pump unit availability on plant level
PlanPumpGMaxLevDin	Max. dyn. pump unit availability on plant level
DefMinLevDin[8]	Minimum fan coil (defroster) dynamic availability
DefMaxLevDin[8]	Maximum fan coil (defroster) dynamic availability

All vectors are allocated for the maximum number of elements of each type. This, where not rendered problematic by memory limitations, enables the development of a code in unvaried mode with respect to the number of evaporators, circuits and compressors.
In this case, on variation of the number of elements, the code would need to be modified. In the case of code developed in the language ST the modification is limited to changing the numerical value of a parameter, while in the case of encoding with a graphic language (FBD / QLD) blocks need to be added or removed from the relative program units.

5.4.2 AC_Plan

The aim of this *program unit* is to compute the availability of cooling resources for the entire plant, acquiring the availability of each and all system elements.

5.4.3 AC_Ev

To calculate the availability of the evaporator, after obtaining the availability of all circuits, it is sufficient to check for the presence of any alarms. The effective calculation is made by the function PolicyCD() located in the functions area of the AppMaker project.

5.4.4 AC_Cir

Also for the circuits, the availability is calculated only after computing the effective availability of the compressors, and therefore after checking for possible block or alarm conditions.

5.4.5 AC_Komp

The calculation of the compressor availability involves more complex algorithms that take into account both the time that a compressor remains in operation and the number of times it has been started up and shut down. It is obvious that starting from the BASELINE APPLICATION the resource calculation management algorithms can be personalised, and in particular in the choice of compressor availability calculation. For example, if wishing to use compressors of different capacities, it will be in this part of the program that modifications are to be implemented, to inform the plant (via circuits and evaporators) of the effective total of resources available.

5.4.6 AC_Def

The calculation of the defrosting availability (of the defrosters) follows the logic of the devices not tied to other plant elements and which therefore can be positioned hierarchically immediately below the plant.

5.4.7 AC_Fans

Although the fans are an element of the Plant, from a point of view of availability they are completely independent from other plant elements. This explains why the program unit *AC_Fans*, though hierarchically dependent on *AC_Plan*t is not correlated with other program units.

5.4.8 AC_PumpG

The calculation of the pump unit availability is the expression of the sum of availability of the individual pumps.

5.4.8.1 AC_Pump

The availability of the individual pumps is checked by examining for the presence (or not) of pump alarms and thus setting "no availability" accordingly.

5.5 DefReg

This *program unit* has the task of managing variations in power of the compressors of each fan coil during defrosting.

5.5.1 Status variables

DefReqLev[i]	Power required by i-th defroster

5.6 CompeReg

This *program unit* has the task of managing the defrost compensation algorithm. It requires the maximum cooling power from the compressors in the circuits alternative to those in defrosting.
The alternative circuits are those that belong to the same evaporator block that the defrosting circuit belongs to, but which do not have links with the fan coil involved in defrosting.

5.6.1 Status variables

CompeReqLev[i]	Power required by i-th compensator

5.7 IntReg

This *program unit* has the task of managing the algorithm for heat integration via the resistances positioned on the evaporator level.

5.7.1 Status variables

InthReqLev	Power required by the integration regulator

5.8 ThermReg

This *program unit* has the task of calculating cooling power requests according to the deviation between a measured temperature (feedback variable) and a set temperature (set point). The measured temperature may be that of the water on inlet to the evaporator or on outlet from the latter.
In the case of plants with multiple evaporators, the temperature on outlet can be obtained as an average of the temperatures on outlet from the various evaporators or from a single common sensor.

5.8.1 ThermReg

In this program unit, after checking for the possible presence of a dynamic set point (see next program unit), the value of the thermoregulation variable is calculated via an algorithm *PI* (or possibly P only) .

5.8.1.1 DynSet

This function modifies the set point of the operator on the basis of an analogue input signal.
The function calculates a delta which must be added to the regulation set-point, and as such must be performed by the thermoregulator, where the effective set point is modified, on the basis of which it is then decided to start up or shut down evaporators/circuits/compressors.

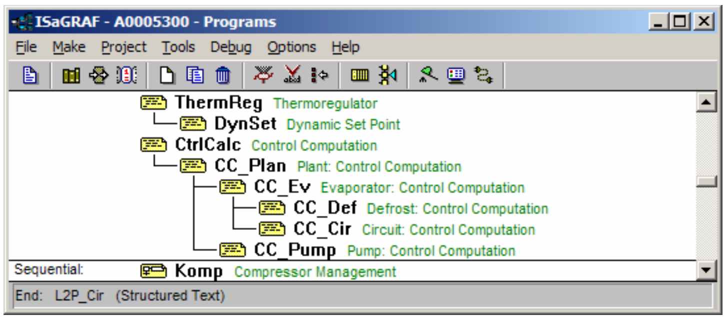
The function has a linear trend, with the selection of the type of dynamic set point to apply according to the type of sensor that influences the calculation, which must be made for the delta (available both in temperature and pressure).

5.8.2 Status variables

TregReqLev	Power required by the thermoregulator

5.9 CtrlCalc

After making the calculation of the availability and requirement of the thermoregulator, a fundamental point in the thermoregulation process is the computation of the control of cooling resources.



```
ThermReg Thermoregulator
└─DynSet Dynamic Set Point
CtrlCalc Control Computation
└─CC_Plan Plant: Control Computation
    ├─CC_Ev Evaporator: Control Computation
    │   ├─CC_Def Defrost: Control Computation
    │   └─CC_Cir Circuit: Control Computation
    └─CC_Pump Pump: Control Computation
Sequential:  Komp Compressor Management
End:  L2P_Cir  (Structured Text)
```

In conceptual terms the calculation of availability is a process that starts from the most important machine elements which each inform their own “child element” of which and how many resources (cooling and physical) require implementation.
The plant must then inform all evaporators on the basis of the resource selection policy, of how much power it requires.
In turn each evaporator, after defining the resources assigned to it, must inform all circuits on the basis of the resource selection policy, of how much power it requires.
In the same way, each circuit, after defining the resources assigned to it, must inform all compressors on the basis of the resource selection policy, of how much power it requires.

5.9.1 Status variables

The data structures on which this program unit acts are mainly used to maintain data of power requirements and power effectively implemented respectively by the evaporator, circuit and compressor.

PlanReqLev	Power required by plant
PlanOutLev	Power implemented by plant
EvReqLev[4]	Power required by each evaporator
EvOutLev[4]	Power implemented by each evaporator
CirReqLev[8]	Power required by each circuit
CirOutLev[8]	Power implemented by each circuit
KompReqLev[8]	Power required by each compressor
KompOutLev[8]	Power implemented by each compressor
DefReqLev[8]	Power required each defroster
DefOutLev[8]	Power implemented by each defroster
PumpGReqLev	Power required by pump unit
PumpOutLev[2]	Power implemented by each pump

All vectors are allocated for the maximum number of elements of each type. This, where not rendered problematic by memory limitations, enables the development of a code in unvaried mode with respect to the number of evaporators, circuits and compressors.
In this case, on variation of the number of elements, the code would need to be modified. In the case of code developed in the language ST the modification is limited to changing the numerical value of a parameter, while in the case of encoding with a graphic language (FBD / QLD) blocks need to be added or removed from the relative program units.

5.9.2 CC_Plan

The aim of this *program unit* is to compute the power required by the plant on the basis of thermoregulator requirements and plant availability.
This data is then used to implement the policy for assignment of resources to the evaporators.
The effective calculation is made by the function PolicyCC() located in the functions area of the AppMaker project. The power effectively implemented PlanOutLev coincides with PlanReqLev.
It also transfers the plant alarm to its subsystems.

5.9.3 CC_Ev

The aim of this *program unit* is to implement the resource assignment policy on the circuits that belong to each evaporator. The power effectively implemented EvOutLev coincides with EvReqLev.
It also transfers the alarm of each evaporator to its subsystems.

5.9.3.1 CC_Def

The aim of this *program unit* is to implement the resource assignment policy on the compressors that belong to each circuit also in relation to the management of all defrosting phases.

5.9.3.2 CC_Cir

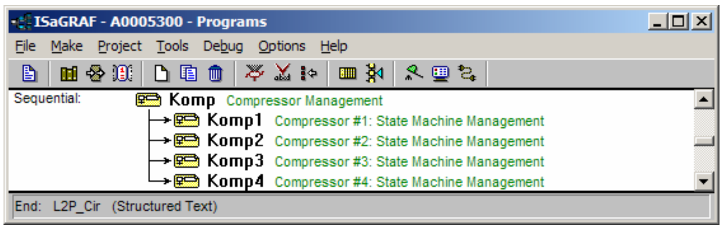
The aim of this *program unit* is to implement the resource assignment policy on the compressors that belong to each circuit also in relation to the activati*on status* of the pumpdown function. The power effectively implemented CirOutLev coincides with CirReqLev.
It also transfers the alarm of each circuit to its subsystems.

5.9.4 CC_Pump

The aim of this *program unit* is to manage evolution of the machine in stages of the pump unit and implement the pump selected accordingly.

5.9.5 Komp: control

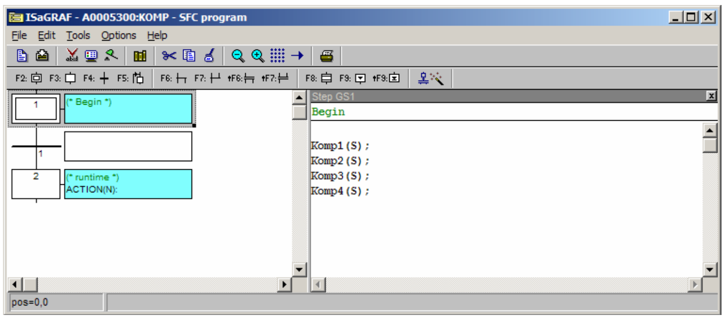
Compressor control can be generally comparable to a state machine. The program for management of the single compressor acts on a set of local variables and a set of global variables by means of which it interacts with the other parts of the control.


ISaGRAF - A0005300 - Programs
File Make Project Tools Debug Options Help

```
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
End:  L2P_Cir  (Structured Text)
```

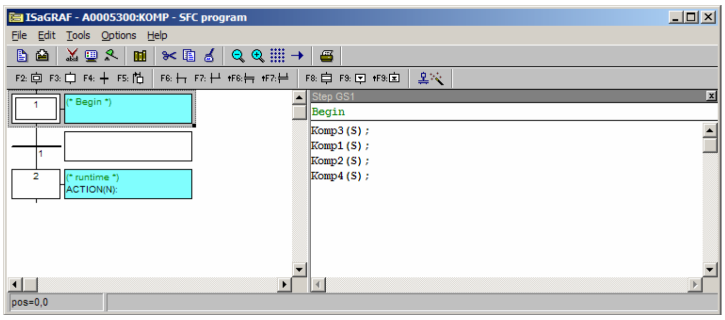
The objective laid down is to have a single code able to be cloned for the management of multiple compressors. For this purpose, the global variables are allocated on vectors with the same dimensions as the number NC of compressors present on the plant. The different copies of the compressor management program operate on a series of global variables with the index n with n variable from 0 to NC-1; this therefore results in a local variable of the compressor management program which identifies the series of global variables on which the n-th copy of the program will operate.

The automata of the single compressors are “independent” entities. The aim of the Komp program unit will be to invoke all the automata of the NC compressors.


ISaGRAF - A0005300:KOMP - SFC program
File Edit Tools Options Help
1
(* Begin *)
1
2
(* runtime *)
ACTION(N):
pos=0,0

```
Step GS1
Begin

Komp1 (S) ;
Komp2 (S) ;
Komp3 (S) ;
Komp4 (S) ;
```

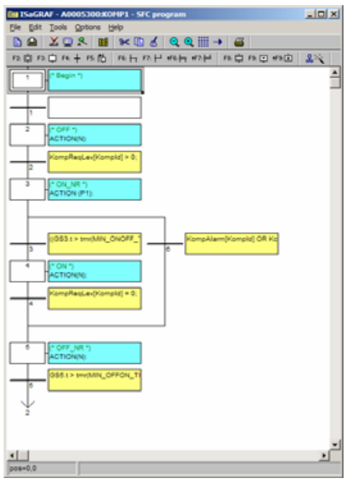
Note that the order with which the calls to automata KompX(S) are entered in the program unit Komp has NO relevance, in fact all automata are independent and are executed SIMULTANEOUSLY. Therefore a situation of this type:


ISaGRAF - A0005300:KOMP - SFC program
File Edit Tools Options Help
1
(* Begin *)
1
2
(* runtime *)
ACTION(N):
pos=0,0

```
Step GS1
Begin
Komp3 (S) ;
Komp1 (S) ;
Komp2 (S) ;
Komp4 (S) ;
```

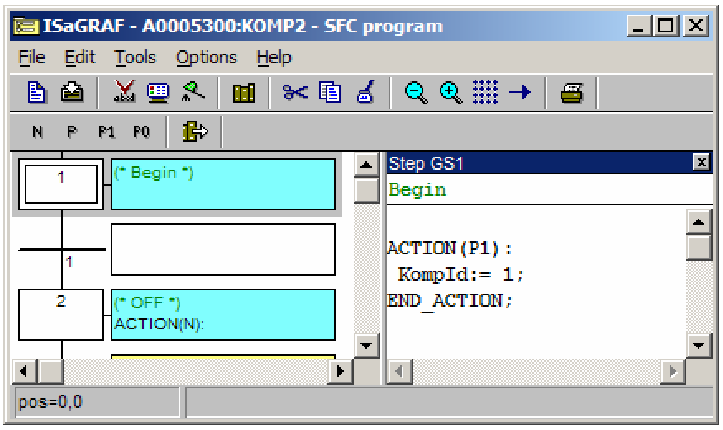
would be IDENTICAL to the previous one.

The SFC implementation of the compressor state machine is dealt with in detail below:



5.9.5.3 Initial state

In this state, all variables of the single compressor are initialised; in particular the index of the data set of global variables and the global variables are initialised. The base program initialises all local variables at the default value.


ISaGRAF - A0005300:KOMP2 - SFC program
File Edit Tools Options Help
N P P1 P0
1
(* Begin *)
1
2
(* OFF *)
ACTION(N):
pos=0,0

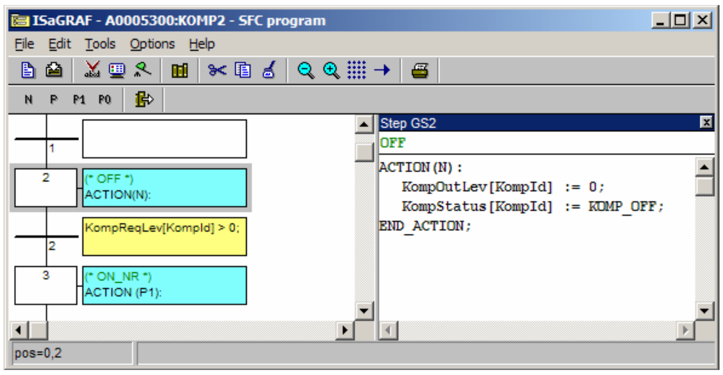
Step GS1

```
Begin

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

Analysing the structure of the block Komp1 it can be noted that its “KompId” (dataset) is initialised at “1”.

5.9.5.4 OFF Status

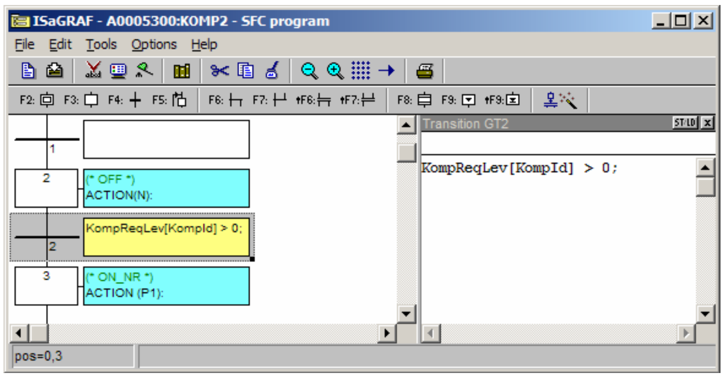
In this state the work variables are initialised in relation to the state and power delivered of the single compressor; given that to be able to define these variables in indexed mode, the index of the compressor must already be defined, for cleaning of the code writing, the index is defined in the previous block. Also in this way all OFF blocks are identical in all various compressors.



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

5.9.5.5 GT2 transition

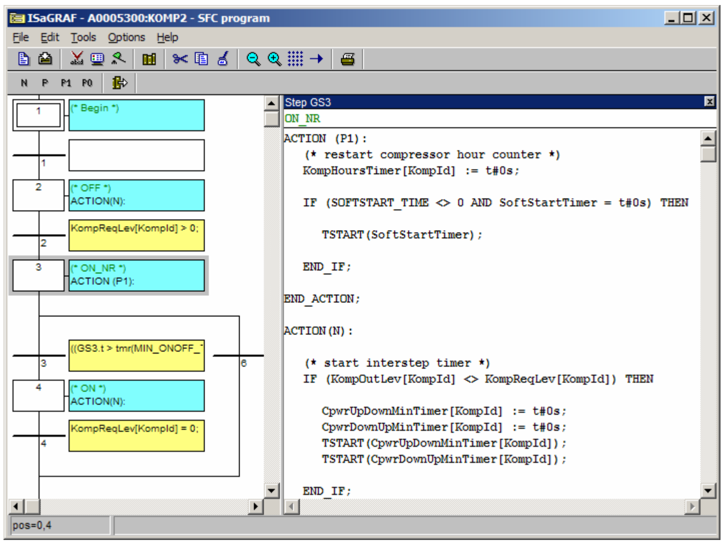
The transition from OFF to the subsequent status of ON_NR (*ON NOT Ready*) occurs in the event of a power request from the compressor.


ISaGRAF - A0005300:KOMP2 - SFC program
File Edit Tools Options Help
F2: F3: F4: F5: F6: F7: ↑F6: ↑F7: F8: F9: ↑F9:
1
2
(* OFF *)
ACTION(N):
KompReqLev[KompId] > 0;
3
(* ON_NR *)
ACTION (P1):
Transition GT2
ST LD
pos=0,3

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

5.9.5.6 ON Not Ready

In this phase all timers related to compressor management are initialised and any capacity steps are managed if relevant.



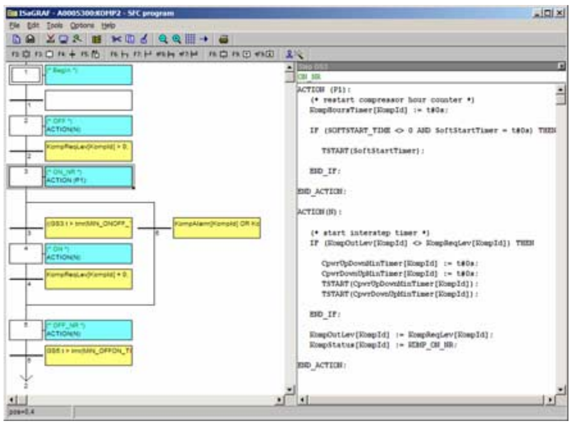
```
ACTION (P1):
   (* restart compressor hour counter *)
  KompHoursTimer[KompId] := t#0s;

  IF (SOFTSTART_TIME <> 0 AND SoftStartTimer = t#0s) THEN

     TSTART(SoftStartTimer);

  END_IF;

END_ACTION;

ACTION(N):

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

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

  END_IF;
```

If these times are configured (with management enabled) the compressor, activated at partial power for a time equal to the maximum time at reduced power, is brought to maximum power in observance of the minimum step up times, outside the control of the thermoregulator.
Once the maximum power is reached, the status is maintained, remaining outside thermoregulator control, for an interval equal to the minimum time at maximum power.
When this interval elapses, the compressor returns under control of the thermoregulator.
Special care must be taken regarding the possibility of assigning, and therefore managing, the value 0 for certain time intervals.
In principle the problem should not be envisaged due to the fact that a compressor can be restarted immediately after being shut down (min toff = 0) even when time 0 is treated as the cycle time of the PLC. If this simplification should lead to undesired reactions, then double transitions will have to be envisaged for t <> 0 and t = 0.

5.9.5.7 GS3 step

In this point the active compressor timer is reset and restarted and a check is then made whether an intermediate step situation needs to be managed or not.

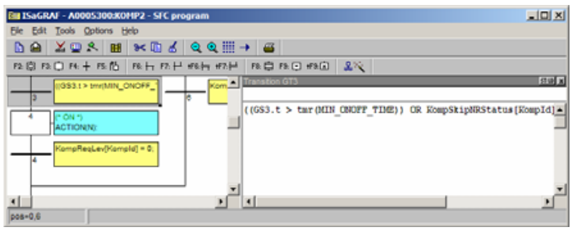
Once this point is overcome, two potential transitions have to be managed:



```
ACTION (P1):
  (* restart compressor hour counter *)
  KompHoursTimer[KompId] := t#0s;

  IF (SOFTSTART_TIME <> 0 AND SoftStartTimer = t#0s) THEN

    TSTART(SoftStartTimer);

  END_IF;

END_ACTION;

ACTION(N):

  (* start interstep timer *)
  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;
```

5.9.5.8 GT3 transition

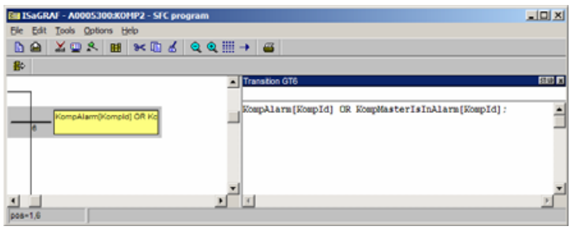
When the compressor protection time elapses and no alarm situations are active, the system is ready to change to the *ON status*.



```
((GS3.t > tmr(MIN_ONOFF_TIME)) OR KompSkipNRStatus[KompId]
```

5.9.5.9 GT6 transition

This transition serves to manage any alarm situations of the specific compressor. If an alarm situation of the compressor KompId occurs (that managed by the specific automata) the system skips immediately to OFF_NR.

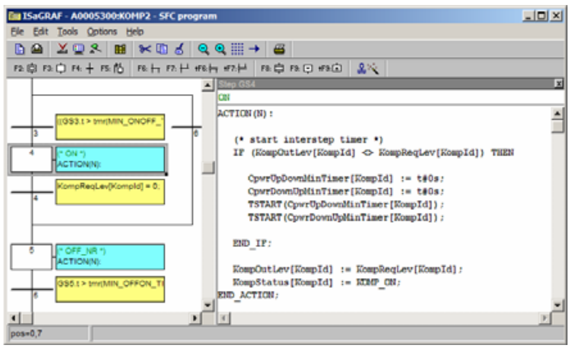


ISaGRAF - A0005300:KOMP2 - SFC program
File Edit Tools Options Help
KompAlarm[KompId] OR Ko
Transition GT6
pos=1,6

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

5.9.5.10 ON status

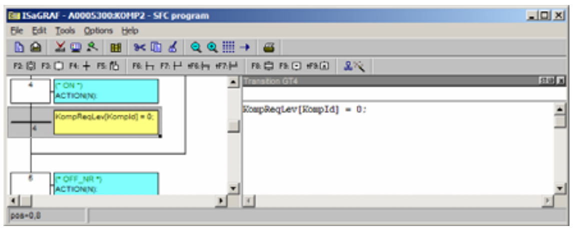
The *ON status* corresponds to the state in which the compressor operates normally. A check is therefore made to ensure that the power delivered corresponds to that requested, and if this is not so, after updating all timers the power delivered is updated.



```
ACTION(N):

   (* start interstep timer *)
   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;
```

5.9.5.11 GT4 transition

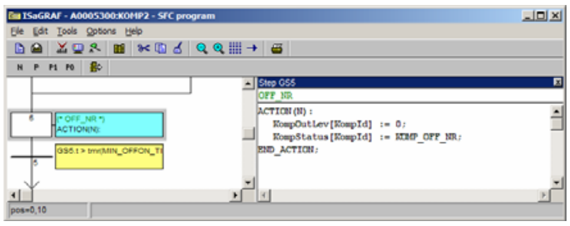
This transition is performed when the request for regulation is equal to zero, in other words the request is made to shut down the compressor.



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

5.9.5.12 OFF Not Ready

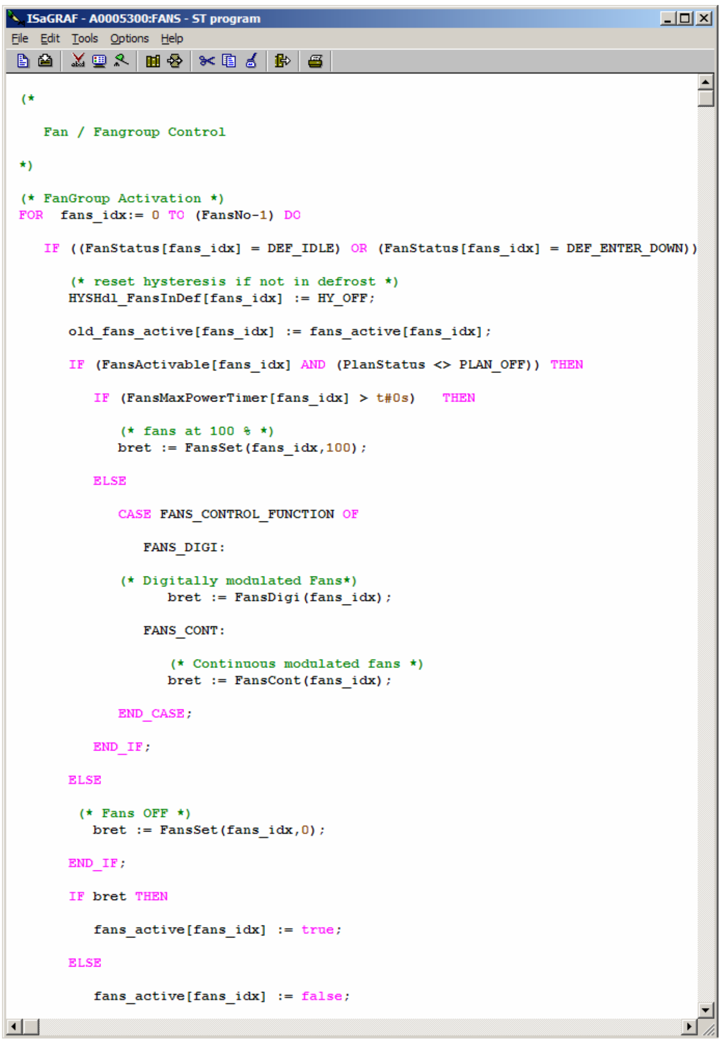
The compressor is shut down bringing its power to zero and setting its status to "*off not ready*". In fact it will be necessary to observe all protection time intervals before restart is enabled.



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

5.10 Fans (Single condensation)

This program unit is used for the management of condensation and fan control.

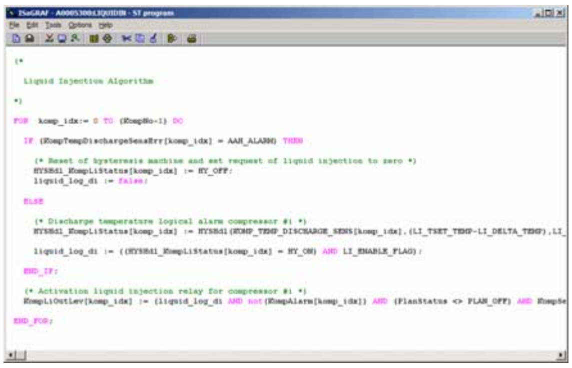

ISaGRAF - A0005300:FANS - ST program
File Edit Tools Options Help

```
(*

   Fan / Fangroup Control

*)

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

   IF ((FanStatus[fans_idx] = DEF_IDLE) OR (FanStatus[fans_idx] = DEF_ENTER_DOWN))

      (* reset hysteresis if not in defrost *)
      HYSHdl_FansInDef[fans_idx] := HY_OFF;

      old_fans_active[fans_idx] := fans_active[fans_idx];

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

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

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

         ELSE

            CASE FANS_CONTROL_FUNCTION OF

               FANS_DIGI:

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

               FANS_CONT:

                  (* Continuous modulated fans *)
                  bret := FansCont(fans_idx);

            END_CASE;

         END_IF;

      ELSE

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

      END_IF;

      IF bret THEN

         fans_active[fans_idx] := true;

      ELSE

         fans_active[fans_idx] := false;
```

Management of single condensation can be considered a high level functions that acts on the computation of the behaviour of the fans.

5.11 Liquid injection

This function enables the regulation of cooling on the compressor discharge temperature:


ISaGRAF - A0005300: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;
```

To all effects it is a function that can be called during regulation and as such can be considered an independent program unit.

5.12 Log2Phy

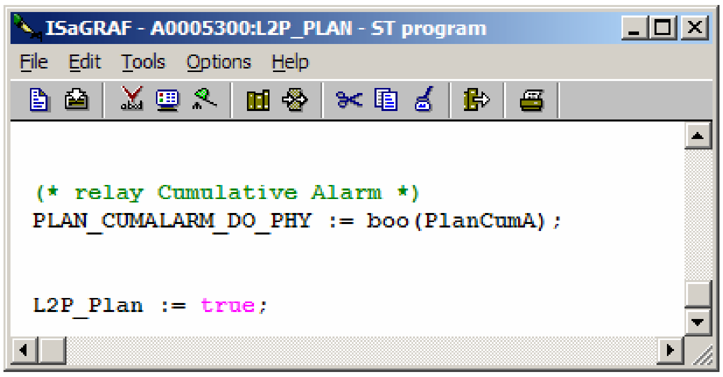
This program unit has the task of converting the logical variables calculated by the control into physical output variables. Once again the hypothesis has been made to divide these program units into sub-program units, each related to a specific area of control machines or type of component. This choice responds to a need for modularity and legibility of the program units rather than a functional requirement.
An important point is the level of abstraction and generalisation that is to be encapsulated in the program units, which needs to achieve a balance between the need to have program units and Function Blocks for general use (and therefore more complex) and at the same time simplicity and efficiency in terms of execution.
For example, consider the compressor implementation program unit: this could be sized for the maximum number of compressors manageable by the maximum application or resized according to the effective number (in the case of the BASELINE APPLICATION). The general program unit could have the implementation parts of the 4 compressors not present disabled by the relative presence flags of the compressors, but this nevertheless implicates an increased dimension of the application and longer execution times. On the other hand the re-sized program unit to 4 compressors should be modified to manage a different number of compressors.

The second point regards the library Function Blocks which consolidate the outputs. Also in this case we could decide to have the “maximum” FB able to manage all types of compressor (on/off, capacity step implementation at 1, 2, 3, step, analogue) or different FB each specialised for a different type of compressor. In the same way it is possible to manage the problems of start-up (simple, star/delta, part-winding) in an FB for general use or specialise the FBs with respect to the different management modes). The general FB has the advantage of not requiring modification of the application to manage different compressors and/or compressors with different start-up modes. However it is more complex and less efficient in terms of execution. The solution with specialised blocks inverts the pros and cons, meaning that the FBs are simpler and more efficient but require modifications to the applications if the user wants to change type of compressor.

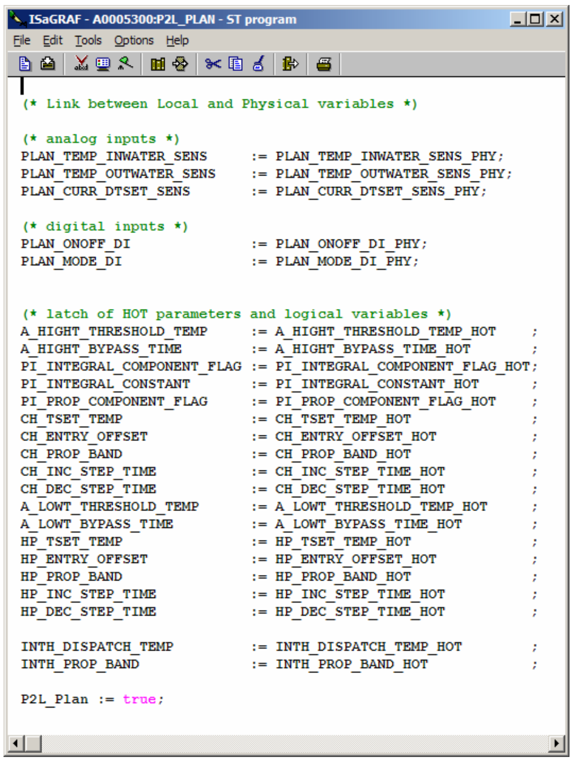
In the case of the BASELINE APPLICATION it has been decided to opt for the solution implementing “maximum” FBs able to manage multiple types of physical device (compressors, fans etc) leaving the possible alternative solutions as hypotheses for development.

5.12.1 L2P_Plan

In the implementation of the BASELINE APPLICATION of a reversible heat pump, the only physical output related exclusively to the plant is the general plant alarm, which activates a control relay of a luminous or acoustic indicator. From this viewpoint, the physical output will coincide with the logical OR of the plant alarms


ISaGRAF - A0005300:L2P_PLAN - ST program
File Edit Tools Options Help

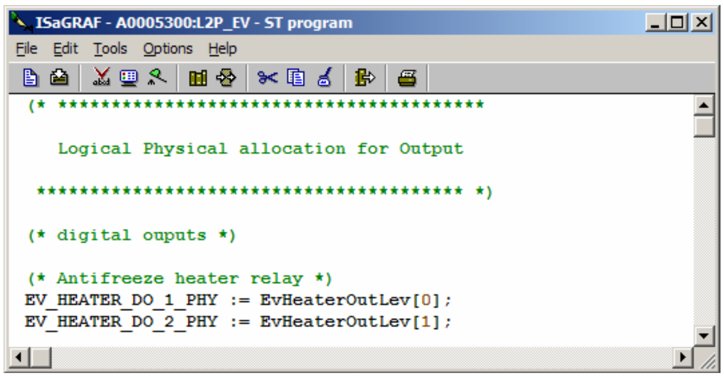
```
(* relay Cumulative Alarm *)
PLAN_CUMALARM_DO_PHY := boo(PlanCumA);


L2P_Plan := true;
```


ISaGRAF - A0005300:P2L_PLAN - ST program
File Edit Tools Options Help

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

(* analog inputs *)
PLAN_TEMP_INWATER_SENS       := PLAN_TEMP_INWATER_SENS_PHY;
PLAN_TEMP_OUTWATER_SENS      := PLAN_TEMP_OUTWATER_SENS_PHY;
PLAN_CURR_DTSET_SENS         := PLAN_CURR_DTSET_SENS_PHY;

(* digital inputs *)
PLAN_ONOFF_DI                := PLAN_ONOFF_DI_PHY;
PLAN_MODE_DI                 := PLAN_MODE_DI_PHY;


(* latch of HOT parameters and logical variables *)
A_HIGHT_THRESHOLD_TEMP       := A_HIGHT_THRESHOLD_TEMP_HOT    ;
A_HIGHT_BYPASS_TIME          := A_HIGHT_BYPASS_TIME_HOT       ;
PI_INTEGRAL_COMPONENT_FLAG   := PI_INTEGRAL_COMPONENT_FLAG_HOT;
PI_INTEGRAL_CONSTANT         := PI_INTEGRAL_CONSTANT_HOT      ;
PI_PROP_COMPONENT_FLAG       := PI_PROP_COMPONENT_FLAG_HOT    ;
CH_TSET_TEMP                 := CH_TSET_TEMP_HOT              ;
CH_ENTRY_OFFSET              := CH_ENTRY_OFFSET_HOT           ;
CH_PROP_BAND                 := CH_PROP_BAND_HOT              ;
CH_INC_STEP_TIME             := CH_INC_STEP_TIME_HOT          ;
CH_DEC_STEP_TIME             := CH_DEC_STEP_TIME_HOT          ;
A_LOWT_THRESHOLD_TEMP        := A_LOWT_THRESHOLD_TEMP_HOT     ;
A_LOWT_BYPASS_TIME           := A_LOWT_BYPASS_TIME_HOT        ;
HP_TSET_TEMP                 := HP_TSET_TEMP_HOT              ;
HP_ENTRY_OFFSET              := HP_ENTRY_OFFSET_HOT           ;
HP_PROP_BAND                 := HP_PROP_BAND_HOT              ;
HP_INC_STEP_TIME             := HP_INC_STEP_TIME_HOT          ;
HP_DEC_STEP_TIME             := HP_DEC_STEP_TIME_HOT          ;

INTH_DISPATCH_TEMP           := INTH_DISPATCH_TEMP_HOT        ;
INTH_PROP_BAND               := INTH_PROP_BAND_HOT            ;

P2L_Plan := true;
```

5.12.2 L2P_Ev

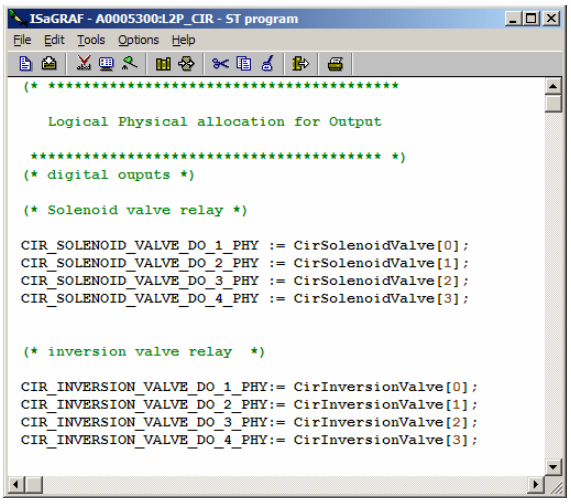
The only two evaporator variables which must be converted on a physical level are the two anti-freeze resistances, which in the AppMaker program are two elements of a vector. Therefore it will be necessary to assign each element of the vector (which note can only be internal variables) to two static I/O variables:


ISaGRAF - A0005300:L2P_EV - ST program
File Edit Tools Options Help

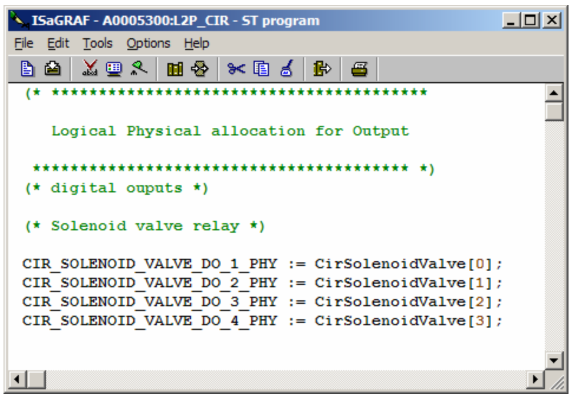
```
(* ******************************************

   Logical Physical allocation for Output

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

(* digital ouputs *)

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

5.12.3 L2P_Cir

The circuit also has physical elements that belong to it and these are the solenoid valves. Also as these are managed as internal vector elements they must be assigned to static I/O variables to enable connection to the I/O rack that determines physical allocation of the inputs and outputs.


ISaGRAF - A0005300:L2P_CIR - ST program
File Edit Tools Options Help

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

   Logical Physical allocation for Output

 ****************************************** *)
(* digital ouputs *)

(* Solenoid valve relay *)

CIR_SOLENOID_VALVE_DO_1_PHY := CirSolenoidValve[0];
CIR_SOLENOID_VALVE_DO_2_PHY := CirSolenoidValve[1];
CIR_SOLENOID_VALVE_DO_3_PHY := CirSolenoidValve[2];
CIR_SOLENOID_VALVE_DO_4_PHY := CirSolenoidValve[3];


(* inversion valve relay  *)

CIR_INVERSION_VALVE_DO_1_PHY:= CirInversionValve[0];
CIR_INVERSION_VALVE_DO_2_PHY:= CirInversionValve[1];
CIR_INVERSION_VALVE_DO_3_PHY:= CirInversionValve[2];
CIR_INVERSION_VALVE_DO_4_PHY:= CirInversionValve[3];
```

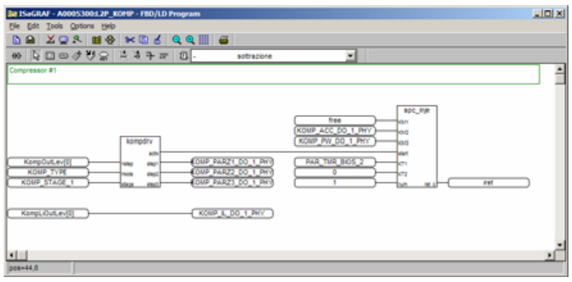

ISaGRAF - A0005300:L2P_CIR - ST program
File Edit Tools Options Help

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

   Logical Physical allocation for Output

 ****************************************** *)
(* digital ouputs *)

(* Solenoid valve relay *)

CIR_SOLENOID_VALVE_DO_1_PHY := CirSolenoidValve[0];
CIR_SOLENOID_VALVE_DO_2_PHY := CirSolenoidValve[1];
CIR_SOLENOID_VALVE_DO_3_PHY := CirSolenoidValve[2];
CIR_SOLENOID_VALVE_DO_4_PHY := CirSolenoidValve[3];
```

5.12.4 L2P_Komp

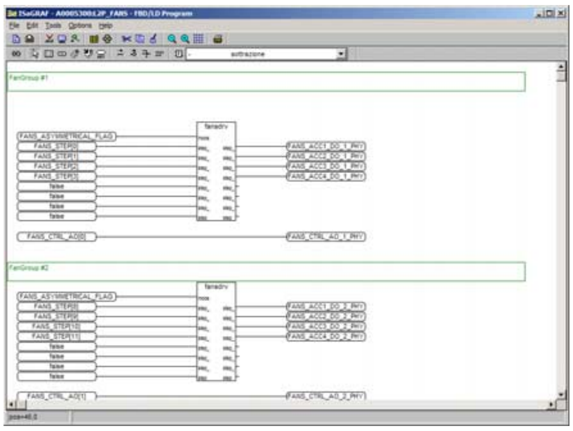
Management of the compressors has been developed using the language FBD. This code will have to be replicated the same number of times as the maximum number of compressors physically present on the plant.
As can be seen in the image describing the logical/physical conversion of a compressor, the function *KompDrv* is used, which as seen in the next chapter represents the compressor device driver. As already mentioned at the start of this

document, the fact of having a programmable device enables the availability of inhomogeneous components such as compressors within the same plant. This means that the same plant can host compressors not only of different outputs but with different start-up/control modes.
When the "physicity" of compressor start-up remains unchanged (in other words n relays are always required for start-up), but the sequence in which to act on the relays changes, intervention is required on the functions of the device driver as described later on in this document. On the other hand, if the use of a new compressor requires a different number of I/O for control, a new device driver will need to be implemented in the functions area, as well as modification of the program unit *L2P_KOMP*, as a different number of I/O will be used.



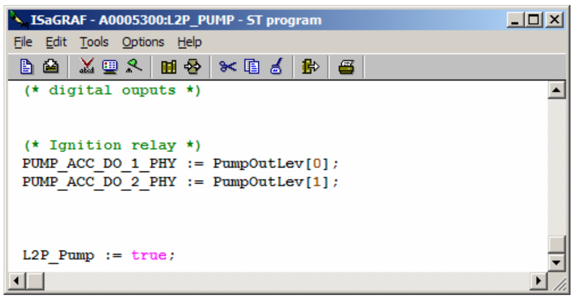
5.12.5 L2P_Fans

In the same way as described for the compressors, the management of conversion from logical to physical for the fans is implemented by means of a program unit in FBD. Also in this case inhomogeneous fans can be used, and different types of device driver may be present. Following this the same considerations as made for the compressors apply also in this case.
The image shown here highlights the fact that the DB code must be replicated the same number of times as the maximum number of fan units present on the plant.



5.12.6 L2P_Pump

In implementation of the BASELINE APPLICATION, two pumps have been installed, the control signal of which is in the vectors PumpOutLev[ ]. This therefore means that the values of this vector should be copied into two static I/O variables.


ISaGRAF - A0005300:L2P_PUMP - ST program
File Edit Tools Options Help

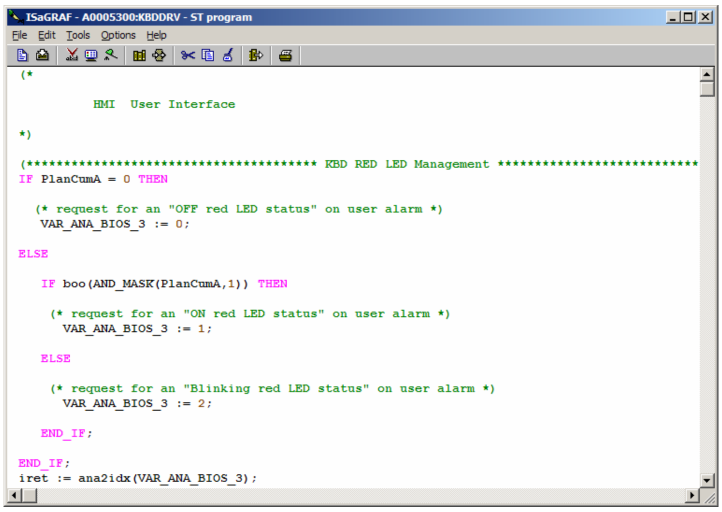
```
(* digital ouputs *)


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



L2P_Pump := true;
```

5.13 BbxDrv

This *program unit* has the task of managing an example of a "black box". The "black box", when enabled, is used in the event of an alarm or critical event to memorise all data related to the plant, useful for understanding the reasons leading to the malfunction or critical event. In the case of XT-PRO the black box memorises the data in the mass memory managed by means of the file system commands. As the quantity of memory is not unlimited, in the event of modifications to the black box or personalisation of operation, great care must be taken with regard to the physical limits of the memory.

5.14 KbdDrv

This program unit has the task of producing the information that the XTK Pro keyboard must display regarding the Baseline Application. In fact, although management of browsing is programmed and managed with the program MenuMaker PRO, management of the red led (commonly used to indicate an alarm), and the production of data for display is performed in this part of the code.


ISaGRAF - A0005300: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;

END_IF;
iret := ana2idx(VAR_ANA_BIOS_3);
```

6 SPECIFICATION OF THE FUNCTION BLOCKS OF THE REVERSIBLE HEAT PUMP

The last part of the project A0005300 deals with the library of the function blocks of the project. Among these are those commonly used as Bypass Handlers (general or not) and the Bounded Alarm Handler. It is evident that when a new driver of any type is to be implemented (for compressors or fans) this must be inserted in this part of the project.

1.1 Function GBYPHdl

This is a function developed to implement a General Bypass able to manage the transactions off <---> bypass_offon ---> on <---> bypass_onoff ---> off regulated by means of a timer. This function is widely used in alarm management. In particular the use of a general bypass (parameterised on a time interval T) means that the alarm becomes active after it has been present without interruptions for at least time T. (Transaction off->on). For the alarm to disappear, the condition that generated the alarm must no longer have been present for at least time "**T'**".

1.2 Function BYPHdl

This Bypass Function differs from the general bypass in that the transaction on/off, unlike the off/on transaction, is instantaneous. This type of bypass will therefore be used if the alarm should appear after the alarm condition has persisted for at least time T (set by parameter), but as soon as the alarm condition disappears, the bypass will immediately switch to off.

1.3 Function BAHHand

This function, as useful as the previous two in alarm management, enables an "active" output after at least the quantity Q of alarm "events" have been present in a time-based window definable by a parameter.
The return of the output to "not active" conditions has to be by operator intervention (manual reset).

1.4 Function BENHandl

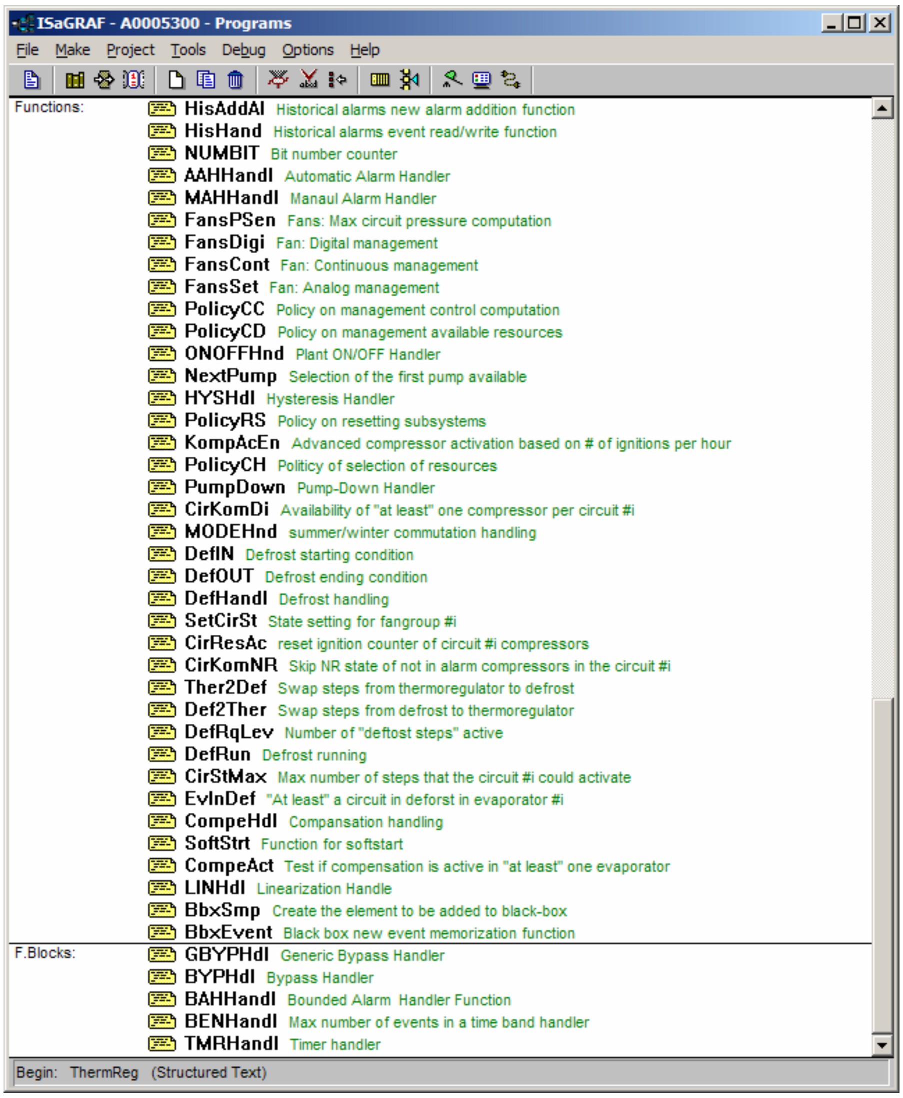
Also this Function has the additional aim of managing the alarm system. In this case the activation of the output is when at least a certain quantity of events are present in a time-based window.

1.5 TMRHandl

This innovative Function enables management of the TIMERs, and in particular enables not only the start-up and termination of the timers but also the suspension d and relative restart of a timer. In the baseline application the timers are used in defrost management as this function can be suspended in particular circumstances to then be resumed at a later time.

7 SPECIFICATION OF THE REVERSIBLE HEAT PUMP FUNCTIONS

The Functions section of the baseline project contain the functions commonly used in the program.


ISaGRAF - A0005300 - Programs
File Make Project Tools Debug Options Help

Functions:	HisAddAl	Historical alarms new alarm addition function
	HisHand	Historical alarms event read/write function
	NUMBIT	Bit number counter
	AAHHandl	Automatic Alarm Handler
	MAHHandl	Manaul Alarm Handler
	FansPSen	Fans: Max circuit pressure computation
	FansDigi	Fan: Digital management
	FansCont	Fan: Continuous 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
	MODEHnd	summer/winter commutation handling
	DefIN	Defrost starting condition
	DefOUT	Defrost ending condition
	DefHandl	Defrost handling
	SetCirSt	State setting for fangroup #i
	CirResAc	reset ignition counter of circuit #i compressors
	CirKomNR	Skip NR state of not in alarm compressors in the circuit #i
	Ther2Def	Swap steps from thermoregulator to defrost
	Def2Ther	Swap steps from defrost to thermoregulator
	DefRqLev	Number of "deftost steps" active
	DefRun	Defrost running
	CirStMax	Max number of steps that the circuit #i could activate
	EvlnDef	"At least" a circuit in deforst in evaporator #i
	CompeHdl	Compansation handling
	SoftStrt	Function for softstart
	CompeAct	Test if compensation is active in "at least" one evaporator
	LINHdl	Linearization Handle
	BbxSmp	Create the element to be added to black-box
	BbxEvent	Black box new event memorization function
F.Blocks:	GBYPHdl	Generic Bypass Handler
	BYPHdl	Bypass Handler
	BAHHandl	Bounded Alarm Handler Function
	BENHandl	Max number of events in a time band handler
	TMRHandl	Timer handler

Begin: ThermReg (Structured Text)

8 SPECIFICATION OF THE LIBRARY FUNCTION BLOCKS OF THE REVERSIBLE HEAT PUMP

8.1.1 KompDrv

Represents the driver for control of capacity step implementation (stage) of a SEMI-HERMETIC or SCREW compressor with "step control" according to the specifications below:
As described in the technical notes of the program unit, the parameters on which it operates are the compressor type, maximum number of capacity steps and the number of steps to implement.
The output corresponds to the activati*on status* (or not) and to that of the capacity step "relays" .
The fact that in this program unit two types of compressors are implemented and the fact that among the input variables there is one that identifies the actual type of compressor, enables the use of machines with inhomogeneous compressor types. Computation of the cooling output in fact is not influenced by the type of compressor used (even though this impacts calculations of resources). It therefore means that the right driver has to be invoked for each compressor (provided that it is a compressor with capacity step implementation).

1.2 FansDrv

Driver for control of fans according to SYMMETRICAL or ASYMMETRICAL mode. The device driver of the fans follows the same logic as that of the compressors. It is a function that receives in the form of an input the quantity of fan power to be delivered and the mode, and then commands the suitable outputs to activate the correct number of fans.
For example, to activate the fans not according to a sequential logic but according to a special combinatorial logic, the modifications to the code would be located at this point.

8.1.2 PI

Implements a *PI* type regulator in which, after supplying the set point values, of proportional band of maximum integral time, and sampling time as inputs (together with other parameters) these are able to independently compute the quantities Ki and Kp and to execute a *PI* control, producing an output (control) expressed as a whole number varying from 0 to 1000.

Note that this program unit is protected by passwords reserved for each user. This password is indispensable for the compilation of the program unit and for display of the interface/parameters. Also remember that this program unit must always be compiled on initial import in AppMaker to enable it to be recalled in any project.

9 USE OF THE DEVICE

9.1 Permitted Use

For safety purposes, it is important to make sure that the control device is installed and used in accordance with the instructions supplied and that no parts subject to dangerous voltage are accessible to users during ordinary operation.
The unit must be resistant to water and dust, depending on the application, and only be accessible using special tools. This unit can be fitted on domestic appliances and/or similar units used for air conditioning.
The use of the unit for applications other than those described is forbidden.

9.2 Responsibility and residual riscks

Eliwell 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.

10 DISCLAIMER

11 INALITIC INDEX

1

1.3.14FansCir (Part of IV_Fans) ***30***

1.3.15FansNo(Part of IV_Fans) ***32***

1.3.6 Vector EvCir ***22***

A

AC_Cir ***42***

AC_Def ***42***

AC_Ev ***41***

AC_Fans ***42***

AC_Komp ***42***

AC_Plan ***41***

AC_Pump ***42***

AC_PumpG ***42***

AHCir ***39***

AHCirEr ***39***

AHCirHPr and AHCirLPr ***39***

AHCirPD ***40***

AHDef ***40***

AHEv ***40***

AHEvAf ***40***

AHEvEr ***40***

AHFans ***38***

AHFansTh ***38***

AHKomp ***38***

AHKompDis ***39***

AHKompEr ***38***

AHKompTh ***38***

AHPlan ***40***

AHPlanEr ***40***

AHPlantHT ***40***

AHPlantLT ***40***

AHPumpG ***40***

AHPumpTh ***40***

AlHnd 37

AppMaker application implementation possibilities 5

AppMaker potential and constraints 4

AvaCalc 41

B

BbxDrv 53

Begin Section ***11***

C

Call-outs ***3***

CC_Cir ***44***

CC_Def ***44***

CC_Ev ***43***

CC_Plan ***43***

CC_Pump ***44***

CheckCon ***35***

Checks of consistency and cycles on objects ***25***

CIR_FANS (Part of IV_Plant) ***30***

CompeReg 42

Constants (Defined words) 32

Control structure definition 16

Cross references ***3***

CtrlCalc 43

D

DATA DICTIONARY OF THE REVERSIBLE HEAT PUMP APPLICATION 15

DefReg 42

DESCRIPTION OF THE BASELINE REVERSIBLE HEAT PUMP APPLICATION 35

Design criteria and system optimisation 6

DISCLAIMER 59

DynSet ***42***

E

End section ***13***

F

Fans (Single condensation) 49

Functions and Function Blocks ***13***

G

GENERAL ARCHITECTURE OF THE REVERSIBLE HEAT PUMP APPLICATION 8

GENERAL ARCHITECTURE OF THE REVERSIBLE HEAT PUMP APPLICATION:Introduction ***8***

General structure 11

Global and support variables 18

GS3 step ***47***

GT2 transition ***46***

GT3 transition ***47***

GT4 transition ***48***

GT6 transition ***47***

H

Highlighted icons ***3***

I

Initial state ***45***

IniVar 35

IntReg 42

Introduction 8

INTRODUCTION TO APPMAKER 4

IV_Cir ***36***

IV_Def ***36***

IV_Ev ***36***

IV_Fans ***36***

IV_Komp ***36***

IV_Plan ***36***

IV_Pump ***36***

K

KbdDrv 54

Komp: control ***44***

KompDrv ***57***

KompFans (Part of IV_Komp) ***24***

L

L2P_Cir ***52***

L2P_Ev ***52***
L2P_Fans.. ***53***
L2P_Komp.. ***52***
L2P_Plan ***50***
L2P_Pump.. ***53***
Libraries and function blocks 4
Limit of development philosophy of the Applications ***4***
Liquid injection 49
Log2Phy.. 50

M

Main principles of architecture.. 10

N

Nomenclature.. 15

O

OFF Not Ready..***48***
OFF Status..***46***
ON Not Ready..***46***
ON status ***48***

P

P2L_xxx ***36***
Parameter FANS_NO (Part of IV_Fans) ***29***
Parameter PUMP_NO..***32***
Permitted Use.. 58
Phy2Log.. 36
PI ***57***
Possibilities of application parameterisation 4
Procedure for modifying the baseline procedure....***14***

R

Responsibility and residual riscks 58

S

Sequential section ***12***
SPECIFICATION OF THE FUNCTION BLOCKS OF THE REVERSIBLE HEAT PUMP 55
SPECIFICATION OF THE LIBRARY FUNCTION BLOCKS OF THE REVERSIBLE HEAT PUMP..57
SPECIFICATION OF THE REVERSIBLE HEAT PUMP FUNCTIONS 56
SPECIFICATION OF THE REVERSIBLE HEAT PUMP FUNCTIONS:..***56***
Status variables.. ***41***; ***42***; ***43***

T

ThermReg 42

U

Use of languages IEC-61131-3 4
USE OF THE DEVICE 58
USE OF THE MANUAL.. 3
Using the Arrays.. 6

V

Variables EvNo, CirNo, KompNo..***24***
Vector CirEv..***20***
Vector CirKomp..***20***
Vector CirPresence..***19***
Vector EvPresence..***22***
Vector KOMP_CIR_EV..***18***
Vector KOMP_STEP..***23***
Vector KompCir..***24***

ELIWELL CONTROLS 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@invensyscontrols.com

Invensys Controls Europe
An Invensys Company