

ISaGRAF

Version 3.4

USER'S GUIDE

CJ INTERNATIONAL

Information in this document is subject to change without notice and does not represent a commitment on the part of CJ International. The software, which includes information contained in any databases, described in this document is furnished under a license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the license or nondisclosure agreement. No part of this manual may be reproduced in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of CJ International.

© 2000 CJ International. All rights reserved.

Printed in France by CJ International.

2 Rue Hector BERLIOZ

F-38600 FONTAINE

Phone: 33 (0)4 76 26 87 30

Fax: 33 (0)4 76 26 87 39

ISaGRAF is a registered trademark of CJ International.

MS-DOS is a registered trademark of Microsoft Corporation.

Windows is a registered trademark of Microsoft Corporation.

Windows NT is a registered trademark of Microsoft Corporation.

OS-9 and ULTRA-C are registered trademarks of Microware Corporation.

VxWorks and Tornado are registered trademarks of Wind River Systems, Inc.

All other brand or product names are trademarks or registered trademarks of their respective holders.

Table of contents

A.	USER'S GUIDE	A-11
A.1	Getting started	A-12
A.1.1	Installing ISaGRAF	A-12
A.1.2	Using on-line information	A-14
A.1.3	A sample application	A-15
A.2	Managing projects	A-19
A.2.1	Creating and working with projects	A-19
A.2.2	Working with several groups of projects	A-21
A.2.3	Options	A-21
A.2.4	Tools	A-22
A.3	Managing programs	A-23
A.3.1	The components of a project	A-23
A.3.2	Working with programs	A-25
A.3.3	Running the code generation tools	A-28
A.3.4	Other ISaGRAF tools	A-29
A.3.5	Adding commands to the Tools menu	A-29
A.3.6	Simulating and debugging the application	A-30
A.4	Using the SFC editor	A-32
A.4.1	SFC language main topics	A-32
A.4.2	Entering an SFC chart	A-34
A.4.3	Working on an existing SFC chart	A-36
A.4.4	Entering the level 2 programming	A-37
A.4.5	Using the SFC gallery	A-41
A.5	Using the Flow Chart editor	A-42
A.5.1	Basics of the FC language	A-42
A.5.2	Entering a Flow Chart	A-43
A.5.3	Working on an existing chart	A-46
A.5.4	Entering level 2 programs	A-46
A.5.5	Programming level 2 with Quick LD	A-47

A.5.6	Display options	A-48
A.6	Using the Quick LD editor	A-49
A.6.1	Basics of the LD language	A-49
A.6.2	Entering an LD diagram	A-51
A.6.3	Working on an existing diagram	A-54
A.6.4	Display options	A-55
A.7	Using the FBD/LD editor	A-57
A.7.1	Basics of the FBD/LD languages	A-57
A.7.2	Entering an FBD diagram	A-59
A.7.3	Working on an existing diagram	A-61
A.7.4	Display options	A-62
A.7.5	Styles and modification tracking	A-63
A.8	Using the text editor	A-65
A.8.1	Editing commands	A-65
A.8.2	Options	A-65
A.9	More about program editors	A-67
A.9.1	Calling other ISaGRAF tools	A-67
A.9.2	Parameters of the program	A-67
A.9.3	Other commands of the "File" menu	A-68
A.9.4	Updating the program diary	A-69
A.9.5	Selecting a variable from dictionary	A-69
A.9.6	The output window	A-70
A.10	Using the dictionary editor	A-72
A.10.1	The dictionary main window	A-74
A.10.2	Managing variables	A-74
A.10.3	Description of objects	A-76
A.10.4	Quick declaration	A-77
A.10.5	Modbus SCADA addressing map	A-78
A.10.6	Exchanging information with other applications	A-79
A.11	Using I/O connection editor	A-83
A.11.1	Defining I/O boards	A-84
A.11.2	Setting board parameters	A-85
A.11.3	Connecting I/O channels	A-85
A.11.4	Directly represented variables	A-85

A.11.5	Numbering	A-86
A.11.6	Setting individual protections	A-87
A.12	Creating conversion tables	A-88
A.12.1	Main commands	A-88
A.12.2	Entering points of a table	A-88
A.12.3	Rules and limits	A-89
A.13	Using the code generator	A-90
A.13.1	Main commands	A-90
A.13.2	Compiler options	A-91
A.13.3	Producing C source code	A-93
A.13.4	Viewing information	A-93
A.13.5	Defining resources	A-94
A.14	Cross References	A-99
A.15	Using the graphic debugger	A-101
A.15.1	The debugger window	A-101
A.15.2	Controlling the application	A-102
A.15.3	Options	A-104
A.15.4	"Write" commands	A-104
A.15.5	On line modification	A-106
A.15.6	DDE exchanges	A-109
A.16	Spying Lists of variables	A-110
A.17	Debugging ST and IL programs	A-112
A.18	Debugging with SpotLight	A-113
A.18.1	Building the graphic layout	A-113
A.18.2	The list layout	A-115
A.18.3	Defining the item style	A-115
A.18.4	Commands of the "File" menu	A-116
A.18.5	Note for ISaGRAF V3.2 users	A-117
A.19	Uploading applications	A-118
A.19.1	Uploading a project	A-118
A.19.2	Communication settings	A-118
A.19.3	Preparing a project for upload	A-119

A.19.4	How zipped source is stored in the target	A-119
A.19.5	Memory requirements on the target	A-120
A.19.6	About uploaded project	A-120
A.19.7	Compatibility issues	A-120
A.20	Using the Diagnosis tool	A-121
A.21	Using the ISaGRAF simulator	A-122
A.21.1	Links with the debugger	A-122
A.21.2	I/O simulation	A-122
A.21.3	Library components	A-123
A.21.4	Options	A-123
A.21.5	Saving and restoring input states	A-124
A.21.6	The cycle profiler	A-124
A.21.7	Simulation scripts	A-125
A.22	Using the Library Manager	A-133
A.22.1	Managing library elements	A-133
A.22.2	I/O configuration	A-135
A.22.3	I/O complex equipment	A-136
A.22.4	I/O board	A-137
A.22.5	Functions and blocks written in IEC languages	A-138
A.22.6	"C" Functions and function blocks	A-140
A.22.7	Conversion functions	A-140
A.23	Using the Archive utility	A-141
A.23.1	Calling the archive manager	A-141
A.23.2	Options	A-142
A.23.3	Backup and restore	A-142
A.23.4	Archive files	A-142
A.24	Printing a complete document	A-144
A.24.1	Customising the table of contents	A-144
A.24.2	Options	A-145
A.25	Password protection	A-147
A.26	Advanced programming techniques	A-150
A.26.1	More about ISaGRAF tools	A-150
A.26.2	Locked I/Os and virtual I/Os	A-150

A.26.3	PC-PLC link validation	A-153
A.26.4	ISaGRAF directories	A-153
A.26.5	Application symbols	A-155
A.26.6	Limits of ISaGRAF "LARGE" (WDL) workbench	A-159

B. LANGUAGE REFERENCE B-163

B.1 Project architecture B-164

B.1.1	Programs	B-164
B.1.2	Cyclic and sequential operations	B-164
B.1.3	Child SFC and FC programs	B-165
B.1.4	Functions and sub-programs	B-165
B.1.5	Function blocks	B-166
B.1.6	Description language	B-167
B.1.7	Execution rules	B-168

B.2 Common objects B-169

B.2.1	Basic types	B-169
B.2.2	Constant expressions	B-169
B.2.3	Variables	B-171
B.2.4	Comments	B-174
B.2.5	Defined words	B-175

B.3 SFC language B-177

B.3.1	SFC chart main format	B-177
B.3.2	SFC basic components	B-177
B.3.3	Divergences and convergences	B-179
B.3.4	Macro steps	B-181
B.3.5	Actions within the steps	B-182
B.3.6	Conditions attached to transitions	B-187
B.3.7	SFC dynamic rules	B-189
B.3.8	SFC program hierarchy	B-190

B.4 Flow Chart language B-191

B.4.1	FC components	B-191
B.4.2	FC complex structures	B-194
B.4.3	FC dynamic behaviour	B-195
B.4.4	FC checking	B-195

B.5 FBD language B-196

B.5.1	FBD diagram main format	B-196
B.5.2	RETURN statement	B-197
B.5.3	Jumps and labels	B-197
B.5.4	Boolean negation	B-198
B.5.5	Calling function or function blocks from the FBD	B-198
B.6	LD language	B-200
B.6.1	Power rails and connection lines	B-200
B.6.2	Multiple connection	B-201
B.6.3	Basic LD contacts and coils	B-202
B.6.4	RETURN statement	B-207
B.6.5	Jumps and labels	B-208
B.6.6	Blocks in LD	B-209
B.7	ST language	B-210
B.7.1	ST main syntax	B-210
B.7.2	Expression and parentheses	B-210
B.7.3	Function or function block calls	B-211
B.7.4	ST specific boolean operators	B-212
B.7.5	ST basic statements	B-214
B.7.6	ST extensions	B-219
B.8	IL language	B-225
B.8.1	IL main syntax	B-225
B.8.2	IL operators	B-226
B.9	Standard operators, function blocks and functions	B-233
B.9.1	Standard operators	B-233
B.9.2	Standard function blocks	B-253
B.9.3	Standard functions	B-270
C.	TARGET USER'S GUIDE	C-311
C.1	Introduction	C-312
C.2	Installation	C-313
C.3	Getting started with ISaGRAF DOS target	C-314
C.3.1	Running ISaGRAF: ISA.EXE	C-314
C.3.2	Specific features	C-315

C.4	Getting started with ISaGRAF OS9 target	C-318
C.4.1	Running the ISaGRAF single task: isa	C-318
C.4.2	Running the ISaGRAF multitasks: isaker, isatst, isanet	C-319
C.4.3	Specific features	C-323
C.5	Getting started with ISaGRAF VxWorks target	C-327
C.5.1	The system resource manager: isassr.o	C-327
C.5.2	Common features to isa.o, isakerse.o and isakeret.o	C-327
C.5.3	Running the ISaGRAF single task: isa.o	C-328
C.5.4	Running the ISaGRAF multitasks: isakerse.o and isakeret.o	C-330
C.5.5	Specific features	C-334
C.6	Getting started with ISaGRAF NT target	C-338
C.6.1	Running ISaGRAF	C-338
C.6.2	General information on options	C-338
C.6.3	Specific features	C-342
C.6.4	User interface	C-346
C.7	"C" programming	C-352
C.7.1	Overview	C-352
C.7.2	"C" Conversion functions	C-353
C.7.3	"C" Functions	C-358
C.7.4	"C" FUNCTION BLOCKS	C-365
C.7.5	Compiling and linking techniques	C-381
C.8	Modbus link	C-387
C.8.1	MODBUS network and protocol	C-387
C.8.2	ISaGRAF implementation	C-388
C.9	Power fail management	C-393
C.9.1	Basics	C-393
C.9.2	Application variables backup	C-394
C.9.3	Program state backup	C-397
C.10	Appendix: Error list and description	C-399
D.	GLOSSARY	D-409
E.	GENERAL INDEX	E-417

A. User's guide

A.1 Getting started

This chapter covers the installation of the ISaGRAF workbench. It also includes a short example of an ISaGRAF application, giving the user a brief outline of its main features and enabling the immediate use of ISaGRAF.

A.1.1 Installing ISaGRAF

This chapter covers the installation of the ISaGRAF Workbench and how to set up the computer for application development.

Hardware and software requirements

The ISaGRAF Workbench can be installed on any computer meeting the minimum qualifications for Windows Version 3.1. However, the following hardware is recommended for application development:

- A personal computer using an 80486 or higher microprocessor (Pentium processor recommended)
- 8 megabytes of conventional and extended memory (16 megabytes recommended)
- One 3.5-inch (1.44 megabyte) disk drive
- One hard disk with at least 20 megabytes of available space
- A graphic VGA or SVGA adapter and compatible monitor
- A mouse (required for graphic development tools)
- A parallel LPT1 port (required for protection key)

Before installing the ISaGRAF workbench, the following software should already be included on the system:

- Windows Version 3.1 running in 386 enhanced mode
- Windows 95
- Windows NT Version 3.51 or 4.00



Using the installation program

The ISaGRAF workbench is installed by using INSTALL, the ISaGRAF installation program. This program copies the ISaGRAF software from the ISaGRAF CD-ROM or disks onto the user's hard disk. INSTALL also adds the group "ISaGRAF" to the Program Manager window and creates an initialisation file named "ISA.ini" in the installed **EXE** sub-directory.

INSTALL is a Windows program, which must be run from the Windows Program Manager or the Run command of the Start menu of Windows 95. To install ISaGRAF, the following steps must be performed:

- Insert ISaGRAF CD-ROM or floppy disk #1 into the appropriate drive
- From the Program Manager or the Start menu, run "SETUP.EXE" on the root folder of the CD-ROM, or "A:\INSTALL.EXE" in the case of floppy disks.

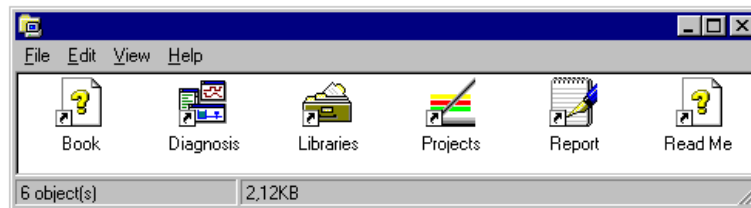
- Follow the on-line instructions to complete the installation. It is recommended that the ISaGRAF Workbench be installed on a new directory to avoid confusing files with files from other ISaGRAF versions.

INSTALL will ask whether the following components are required:

- ISaGRAF executable programs
- On line information and help files
- ISaGRAF standard libraries
- ISaGRAF sample applications

It is highly recommended that when installing ISaGRAF for the first time all components be included. Further components can, however, be added at a later date by re-installing the ISaGRAF Workbench.

The default name for the ISaGRAF main directory is "ISAWIN". This allows ISaGRAF for Windows to be easily installed on the same disk as a version of ISaGRAF for MS-DOS. Refer to the "ISaGRAF directories" section in the "Advanced techniques" chapter for more about ISaGRAF disk architecture. Once all the ISaGRAF files have been copied, the following group is added to your Program Manager Window:



Here are the main ISaGRAF icons:

- Projects:** Project management
- Libraries:** Library management
- Book:** On-line information about ISaGRAF
- Diagnosis:** Diagnosis tool for end user
- Read Me:** Information about the ISaGRAF new version
- Report:** Standard Bug report form

In case you encounter a problem, use the standard bug report form. Open it, fill the items requested and use the File/Save As menu command to save it with a given file name. Then send this file to CJ International, using Fax or e-mail.

Updating system files

Once installation is complete, the CONFIG.SYS file needs to be updated before restarting the computer. The ISaGRAF directory pathname does not have to be inserted in the PATH variable.

ISaGRAF does not use any MS-DOS environment variable. However, the following statements can be added to the CONFIG.SYS file:

```
files=20  
buffers=20
```

The ISaGRAF Workbench uses a serial port to communicate with the ISaGRAF target PLC. The default serial port for ISaGRAF is COM1. If the mouse also uses a serial port, choose COM2 for the mouse, so the default COM1 specification will be valid for any new ISaGRAF applications.

After updating the CONFIG.SYS file, it is necessary to restart the computer for the changes to take effect.

⇒ **Important for Windows NT user:**

When the Workbench is used under Windows NT 3.51 or 4.00, the following line has to be inserted in [WS001] section of ISA.ini file in \ISAWIN\EXE directory:

```
[WS001]  
NT=1  
Isa=C:\ISAWIN  
IsaExe=C:\ISAWIN\EXE  
IsaApl=C:\ISAWIN\APL1  
IsaTmp=C:\ISAWIN\TMP
```

This is absolutely required for RS communication.

⇒ ***The protection key***

A hardware key protects the ISaGRAF software against illegal copies. However, most functions of the ISaGRAF workbench are still available when the key is not inserted. The protection key also defines the option of the ISaGRAF Workbench, and defines the maximum size of developed applications. When the key is not inserted or not properly connected, some of the ISaGRAF Workbench functions will not run. This is NORMAL behaviour. To ensure that the key is properly connected, select the "**About...**" choice of the "**Help**" menu in any ISaGRAF window. The available option of the ISaGRAF workbench is displayed.

The key can be connected to any parallel port on the computer. If the machine has more than one parallel port, it is preferred to connect the key and the printer to different ports. For some PC/printer configurations, the key may not be recognised when its output is connected to an "off-line" printer. In this case, disconnect the printer, or start it in the "on-line" state, and restart the ISaGRAF Workbench.

Note that no key is needed for the **ISaGRAF-32** Workbench.

⇒ **Important for Windows NT user:**

On Windows NT systems, the Sentinel/Rainbow™ Driver has to be installed in order for the protection key to be seen. A separate diskette is provided.

A.1.2 Using on-line information

On-line information is installed with the ISaGRAF workbench, for the following topics:

- ISaGRAF languages reference

- Complete user's guide (for any ISaGRAF tool)
- Technical note for elements in the libraries

From any ISaGRAF window, select the choices of the **"Help"** menu to display online information.

A.1.3 A sample application

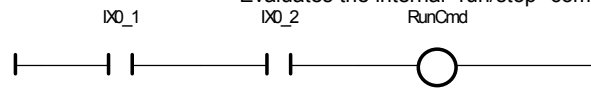
This chapter explains, step by step, all the basic operations required to make, design, generate and test a short but complete multi-language application.

Below are the complete specifications of this application, mixing LD and SFC representations:

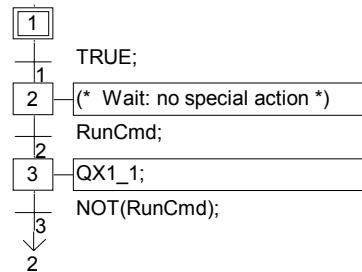
Boolean variables:

IX0_1, IX0_2:	input variables for process command
RunCmd:	internal "run/stop" command
QX1_1:	output variable: status of the process

Program Command:	Cyclic begin section - LD language Evaluates the internal "run/stop" command
------------------	---



Program RunStop:	Sequential section - SFC language Controls the process
------------------	---



Start **Running the ISaGRAF workbench**

To run the ISaGRAF Workbench, run the "Projects" command, in the "ISaGRAF" group, from the Start menu of Windows.



Creating the project

Create the project (called "RunStop") using the "New" command of the "File" menu or the New button. In the open dialog box:

Enter project name: **"RunStop"**
Select I/O configuration: **"Sim_Boo"**

Press the "**OK**" button.
The project has now been created.



Opening the project

The programs of the project are defined by opening the ISaGRAF program management window. Use the "Open" command of the Project management window, or double click the mouse on the name of the project or use the Edit button.



Creating the programs

The Program Management window is now open and empty (no programs defined). The first program is created using the "New" command of the "File" menu or the "New" button. In the open dialog box:

Enter the name of the program: "**Command**".
Select the "**Quick LD**" language.
Select the "**Beginning of cycle**" section.
Press the "**OK**" button to create the program.

The same operation must be repeated for the second program:
Use the "New" command of the "File" menu, or the "New" button. In the open dialog box:

Enter the name of the program: "**RunStop**".
Select the "**SFC**" language.
Select the "**Sequential**" section.
Press the "**OK**" button to create the program.

The programs are now created. They appear in the Program Management window.



Declaring the variables

Before entering the programs, the internal variable to be used in the programming must be declared. This is done using the command "Dictionary" of the "File" menu or the Dictionary button. I/O variables are automatically declared when the project is created.



The dictionary window is now opened. With the menu "File", the Sub-menu "Other", the Sub-menu "Global variables" and then the command "Booleans", select the "Global" boolean dictionary. Buttons Global objects and Boolean can be used for the same effect.



The "New" command of the "Edit" menu is used to create new boolean variables. You can also use the Insert objects button. In the open dialog box, enter the description of the internal variable:

name: **RunCmd**
comment: **Run/Stop command: internal**
attribute: Select the "**Internal**" attribute
Press the "**Store**" button: the variable is created.
Press the "**Cancel**" button to exit the dialog box.

Finally, exit the dictionary editor and save the modifications entered: Menu "**File**" - Command "**Exit**". Click on "**YES**" to save modifications.



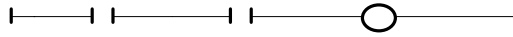
Editing the Quick LD program

To start editing the "Command" LD program, double click on its name in the Program Management window or use the Edit button.



The ISaGRAF Quick LD Editor window is now open. To increase the working area, resize the window to use the full screen size.

F2 F3 Press F2 and F3 key:
(*')



Associate variables to the LD symbols: move the cursor using the keyboard arrows. Place the cursor on each symbol and press Enter key. The variable section dialog box is opened.

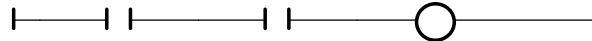
For the first contact, type in the variable selection box: IX0_1 then Enter.

For the second contact, type in the variable selection box: IX0_2 then Enter.

For the coil, type in the variable selection box: RunCmd then Enter.

The program is now complete. Here is the result:

IX0_1 IX0_2 RunCmd



Exit from the editor, and save the modifications entered: Menu "File" - Command "Exit". Click on "YES" to save modifications.



Editing the SFC program

To start editing the "RunStop" SFC program, double click on its name in the Program Management window or use the Edit button.



The SFC Editor window is now open. To increase the working area, resize the window to use the full screen size:



The initial step already exists and is selected. Press the "Down" keyboard arrow to select the empty cell after the initial step (0,1)

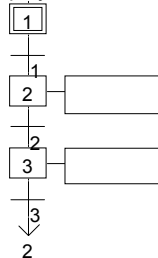
F4 F3 Press F4 then F3 to insert a step and a transition.


F4 F3 Press F4 then F3 to insert one more step and transition.

F5 Press F5 to insert a jump to a step and select GS2 as the destination of the jump.



The chart is now complete. Press the "Zoom" button in the toolbar to increase size of cells and give space to display level 2 instructions. Here is the chart:



-  To enter the programming of transition "2", select it using the keyboard arrows and press "Enter" key. The Level 2 programming window is open. Enter level 2 programming for transition 2:
RunCmd;
- ^TAB** Press "Control + Tab" keys to move focus back to the SFC chart, move selection on step 3, and press "Enter" key to edit its level 2 text:
QX1_1;
And do the same to enter text of transition 3:
Not (RunCmd);
- ^F4** Press "Control + F4" keys to close the level 2 window.
The SFC program is now complete. Exit from the editor with Menu "**File**" and Command "**Exit**", and save the modifications entered clicking on "**YES**".



Building the application code

Use the "**Make**" menu and command "**Make Application**" from the Program Management window to build the application code or the button in the Toolbar.

When the code generation is complete, a dialog box appears, which asks you to exit the code generation **now** or to **continue** working with it: Press the button "**Exit**".



Simulation

Use the "**Debug**" menu and command "**Simulate**" from the Program Management window to run the ISaGRAF kernel simulator or the button in the Toolbar.

When the Simulator window appears, the application can be tested. In this example, both inputs 1 and 2 (green buttons) must be pressed to run the process (output red LED lights).

Close the **Debugger** window to exit from simulation: Menu "**File**" - Command "**Exit**".

A.2 Managing projects

To run the ISaGRAF project management tool, double click the mouse on the "Projects" icon, in the ISaGRAF group. The "Project Management" window is then opened. A project corresponds to one PLC loop run on a target PLC. The upper window contains the list of the existing projects. The text descriptor of the selected project is displayed in the lower window.



Resizing windows

Just click on the separator (splitter) between project list and descriptor to resize corresponding windows. The descriptor window cannot be completely hidden. It always contains at least one line of text.



Inserting separators

A separator line can be inserted before any project name. This allows grouping some projects attached to the same application in the list layout. Use the "**Edit / Toggle separator**" command to insert or delete a separator before the selected project.



Moving projects in the list

To move a project in the list, you first have to select (highlight) it. Then click on its name and drag it to a new location in the list. When dragging the project, a small arrow on the left margin indicates where it will be placed. You can also use the "**Move**" commands of the "**Edit**" menu to move the selected project line by line. Note that if a separator is placed before the selected project, it is moved with the project.

A.2.1 Creating and working with projects

The commands of the project manager menu are used to create new projects, edit them and manage existing projects.



Creating a new project

To create a new project, first enter its name. An empty project is then created, with no object in it. An I/O configuration can be attached to the new created project. This I/O configuration must be defined in library. If a configuration is chosen, ISaGRAF will automatically set-up the I/O connection and declare the corresponding I/O variables in the new project dictionary. When creating or renaming a project, you have to conform the following naming rules:

- name cannot exceed **8** characters
- the first character must be a **letter**
- the following characters can be **letters**, **digits** or underscore character
- the project's name is case insensitive

When a project is created, use the "**Edit / Set comment text**" command to enter the text to be displayed with the project name in the list.



Editing the project descriptor

The "**Project / Project descriptor**" command is used to edit the project text descriptor. This document fully identifies the project from the others on the project list. The project descriptor can also be used to record any remarks during the project lifetime.



Editing project

The "**File / Open**" command opens the Program Management window for the selected project. From this window, all the contents (programs, application parameters...) of the project, can be managed. It is also possible to double click on a project name, to edit it.



The history of modifications

"The ISaGRAF system stores any modification relative to a component of a project in a history file. Each modification is identified in the history by a title, a date and a time. The history file contains the last **500** modifications. There is one history file for each project. The history of modifications for the project is the complement of the "diary" files attached to the programs of the project. The "**Project / History**" command allows the user to view or print the history of modifications for the selected project. The user can select one or more items in the main list, and press the following buttons:

OK..... closes this window
Print sends the contents of the list to the printer
Help..... displays help about this dialog box
[erase] Selected . removes (deletes) the selected lines from the list
[erase] All..... clears the complete list
Find..... finds a pattern in the list

The input box above the "**Find**" button is used to enter a search pattern. This function is case insensitive. When the search reaches the bottom of the list, it continues from the top of the list to the starting position.



Printing a complete document

The "**Project / Print**" command allows the user to build and print a complete document about the selected project. This document can group any component (program, variable, parameters...) of the selected project. To build a specific (non-complete) document, the user only has to define its table of contents.



Password protection

The "**Project / Set password**" command enables the user to define password protection for tools and data of the selected project. Refer to the "**Password protection**" section, at the end of the first part in this manual for further information about password levels and data protection. Passwords are only relative to the selected project. They have no influence on other projects and ISaGRAF libraries.

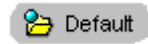
A.2.2 Working with several groups of projects

An ISaGRAF project corresponds to one directory on the disk, where all the project files are store. A "Project Group" corresponds to a list of project directories grouped together under the same root directory. A project group is identified by a name. As default, ISaGRAF creates two project groups:

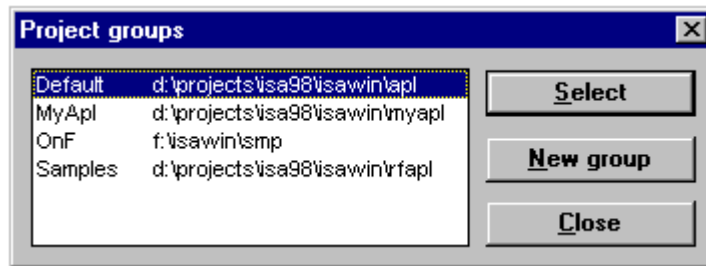
"Default" on "ISAWIN\APL": your working area

"Samples"..... on "ISAWIN\SMP": sample applications delivered with ISaGRAF workbench

The name of the currently selected project group is written in the toolbar, close to the button used to select a project group:



You can also run the "**File / Select project group**" to select an existing group or create a new one. The following dialog box is open:



Select a group in the list and press "**Select**" to activate it in the project management list. You can also double click on its name to select it. Use the "**New group**" command to create a new group. This command can be used either to assign a group name to an existing directory, or to create a new group with a new directory.

Note: No group can be selected or created when other ISaGRAF windows (program manager, editors...) are open.

A.2.3 Options

The commands of the "**Options**" menu are used to display or hide the toolbar, select the character font for text, and set the Project Manager "auto close" mode. The character font selected is the one used to display the project descriptor, and is also used by all ISaGRAF text editors.

When the "**Keep Project Manager open**" option is removed, the Project Manager window is automatically closed when a project is entered.

A.2.4 Tools

The commands of the "**Tools**" menu are used to run other ISaGRAF applications. The "**Tools / Archive Projects**" command runs the ISaGRAF archive manager to save or restore projects. The "**Tools / Archive Common data**" command is used to save or restore files used by all projects (such as common defined words).

The "**Tools / Libraries**" command runs the ISaGRAF library manager in a separate window.

The "**Tools / Import IL program**" can be used to import a project described as a single IL program in a text file, according to PLC Open file exchange format.

A.3 Managing programs

The Program Management window shows the programs (also called modules or programming units) of the application and groups into its menus the available commands, to create the project architecture, run editors, compiler and debugger. This window is the workbench kernel during the development of an application. The Program Management window opens when running the "Open" command in the Project Management window.

A.3.1 The components of a project

The components of a project are called **programs**. A program is a logical entity that describes one part of the control execution. Global variables (such as I/O variables) can be used by any program in the application. Local variables may be used by only one program. Programs are listed in a **hierarchy tree**, divided into different **logical sections**. The window shows the programs and the links between them. The "**Top level**" programs appear on the left side of the hierarchy tree.

▣ **Top level programs**

The top-level programs appear on the left side of the hierarchy tree. Top level programs of the three first sections are always active, and are executed in the following order, during the run time cycle (scan):

- (Read inputs)
- Execute the top level programs of the **BEGIN** section
- Execute the top level programs of the **SEQUENTIAL** section
- Execute the top level programs of the **END** section
- (Refresh outputs)

The programs of the "**Begin**" or "**End**" sections describe cyclic operations. They are not dependent on Time. The programs of the "**Sequential**" section describe sequential operations, where the Time variable explicitly appears to distinguish basic operations. The main programs of the "**Begin**" section are systematically executed at the beginning of each run time cycle. The main programs of the "**End**" section are systematically executed at the end of each run time cycle. The main programs of the "**Sequential**" section are executed on the basis of the **SFC** or **FC** rules and must be written in **SFC** or **FC** language. The programs of the cyclic sections cannot be described in the **SFC** or **FC** language. Any program of any section may own one or more **sub-programs**.

▣ **Functions and function blocks**

The programs of the "**Functions**" section can be called by any program of any section in the project. A function is an algorithm that processes one output value from several input values. A function algorithm only works with volatile intermediate variables, erased from one call to the other. This implies that a function should never call a function block. A program of the "**Functions**" section cannot be described in the **SFC** or **FC** language.

Unlike functions, "**Function blocks**" associate an algorithm working on input values with hidden static data, which are copied (instanced) by the system on each different use of the function block. The programs of the "**Function Blocks**" section can be called by any program of any section in the project. They cannot be programmed in **SFC** or **FC** language.

⇒ **Sub-programs**

Sub-programs are functions dedicated to one (SFC, FC or other) father program. A sub-program can be executed (called) by its parent program only. Each program of each section may have one or more sub-programs. Any language apart from **SFC** and **FC** can be used to describe a sub-program.

⇒ **Child SFC and FC programs**

A **child SFC program** is a parallel program that can be started or killed by its parent program. The parent program and child program must both be described in **SFC** language.

When a parent program starts a child **SFC** program, it puts a **SFC token** into each initial step of the child program. When a parent program kills a child **SFC** program, it clears all the tokens existing in the steps of the child.







Any **FC** program of the sequential section may control other **FC** sub-programs. An **FC** father program is blocked (waits) during execution of an FC sub-program. It is not possible that simultaneous operations are done in father FC program and one of its FC sub-programs.

⇒ **Links between programs and sub-programs:**

Sub-programs and child programs are linked to their parent program by a line in the hierarchy tree. An arrow ends a link between an SFC program and an SFC child program. Note that such a link represents **parallel** operations.

⇒ **Programming languages**

Each program is described in only one **language**. This language, selected when the program is created, cannot be changed afterwards. However, **FBD** diagrams may include parts in **LD**, and **LD** diagrams may include function block calls. Available graphic languages are **SFC** (Sequential Function Chart), **FC** (Flow Chart) **FBD** (Functional Block Diagram) and **LD** (Ladder Diagram). Available literal languages are **ST** (Structured Text) and **IL** (Instruction List). **SFC** and **FC** languages are reserved for main and child programs of the sequential section. The language of each program is shown as an icon beside the program name in the Program Management window. Below are the icons used to represent the languages:

	SFC	Sequential Function Chart
	FC	Flow Chart
	FBD	Functional Block Diagram
	LD	Ladder Diagram (entered with Quick LD editor)
	ST	Structured Text
	IL	Instruction List

A.3.2 Working with programs

The **"File"** menu groups all the commands used to create, update or modify programs. It also launches appropriate editors to enter the contents of application programs.



Creating a new program

The **"New"** function of the **"File"** menu allows the creation of top level, child or sub-programs into each program section. The first piece of information to be entered is the name of the new program according to the following naming rules:

- the maximum length of a name is **8** characters
- the first character must be a **letter**
- the following characters must be **letters, digits** or **'_'** character
- the naming of a program is case insensitive

Next, select the editing language chosen to describe the new program:

SFC..... Sequential Function Chart
 FC Flow Chart
 FBD..... Functional Block Diagram (may include parts in LD)
 LD Ladder Diagram entered with Quick LD editor
 ST Structured Text
 IL..... Instruction List

Finally, select an execution style for the program:

Begin..... top level of the "Begin" section
 Sequential top level of the "Sequential" section
 End top level of the "End" section
 Function in the "Functions" section
 Function block in the "Function Blocks" section
 Child of..... SFC or FC child or sub-program of an existing program

By selecting one of the first five choices, the program is put at the top level of a **Begin, End, Sequential, Functions** or **Function Blocks** section. The selection of the latter indicates that the new program is an **SFC** child program or an **FC** sub-program or a sub-program. Remember that a top-level sequential program must be described in the **SFC** or **FC** language, and that the **SFC** and **FC** languages cannot be used for cyclic programs and their sub-programs.



Entering comments for each program

ISaGRAF allows you to attach a description text to each program of the project. This comment text is displayed with smaller character font beside the name of the program. Use the **"File / Program comment text"** command to enter or change the comment attached to the selected program.



Editing the contents of a program

This command allows the modification of a program's contents. The editor used to enter a program depends on the language chosen for that program. Program editing

is carried out in individual windows, so that it is possible to edit more than one program through parallel windows. Pressing the **ENTER** key allows the editing of the highlighted program. The user can also double click with the mouse on the name of the program to edit it.



Editing the "diary" file

A **diary file** is attached to each program. This is a text file, which contains all the notes about the modifications made to the program during its lifetime. The diary file can be edited, freely modified or printed at any time. When leaving the editor used to modify the source code of a program, a window is automatically opened to enter notes for the diary list. Such notes are inserted with the correct date and time into the diary file.



The dictionary of variables

The "**File / Dictionary**" command runs the dictionary editor, where are declared the variables of the project. Variables may be global (known by any program in the project) or local to the selected program. The dictionary editor may also be used to declare **defined words**, which are semantic aliases, used to replace a name or an expression in the source code of a program.



Parameters of a function, sub-program or function block

The "**File / Parameters**" command allows the user to define the call and return parameters of the selected sub-program, function or function block. This command has no effect if a main program of the "**Begin**" or "**End**" section, or an SFC program is selected in the Program Management window.

Sub-programs, functions or function blocks may have up to **32** parameters (input or output). A function or sub-program always has one (and only one) return parameter, which must have the same name as the function, in order to conform to ST language writing conventions.

The list in the upper left side of the window shows the parameters, in the order of the calling model: first the calling parameters, last the return parameters. The lower part of the window shows the detailed description of the parameter currently selected in the list. Any of the ISaGRAF data types may be used for a parameter. The return parameters must be located after calling parameters in the list. Naming parameters must conform to the following rules:

- the length of the name cannot exceed 16 characters
- the first character must be a letter
- the following characters must be letters, digits or underscore character
- naming is case insensitive

The "**Insert**" command is used to insert a new parameter before the selected parameter. The "**Delete**" command is used to erase the selected parameter. The "**Arrange**" command automatically rearranges (sorts) the parameters, so that the return parameters are put at the end of the list.



Moving a program in the hierarchy tree

The "**Rename/move**" command of the "**File**" menu is used to change the name of a program, or to move it into another section of the hierarchy tree. However the

description language of an existing program cannot be changed. When running this command, the same window as the one used for creating programs is opened, and all fields are set up with the attributes of the selected program. The name of a program can be modified, and another section or parent program selected to move it into the hierarchy tree.

The "**Arrange programs**" command of the "**File**" menu is used to give an explicit order between a list of programs with same level and father. If the selected program is at the top level, the command is used to arrange the top-level programs of the selected section. If the selected program is at a lower level, the command arranges only the SFC children and sub-programs which have the same father as the selected one. When the "**Arrange programs**" dialog box is opened, select the program you want to move, and press the "**Up**" or "**Down**" button to move it in the list.



Copying programs

To make a copy of a program, select the source program from the list of programs, and run the "**File / Copy**" command. When running this command, the same window as that used for creating programs is opened, with all fields set up with the attributes of the selected program. Enter the name of the destination program and its location in the sections of the hierarchy tree. If the destination program does not exist, it is created at the specified location. If the destination program already exists, it is overwritten. All the local declarations and defined words are copied with the program. The description language of the destination program must be the same as the one used for the source program. Press the "**OK**" button to copy the program.

The "**Copy to other project**" command of the "**File**" menu copies the selected program into another project, with the same name. The child SFC programs and sub-programs of the selected program can be copied with it. The names of the selected program and its children must not be used in the target project. Programs cannot be overwritten by this command. All the attached local declarations and defined words are copied with the programs.



Deleting programs

To delete a program, first select it from the list of programs, and then run the "**File / Delete**" command. A program owning child or sub-programs cannot be deleted. In order to delete a program with child or sub-programs, the child or sub-programs must be deleted first. All the local declarations and defined words are deleted with the program.



Importing function or function block from library

The "**Tools / Import from library**" command is used to copy a function or a function block written in IEC language described in the library to the "**Functions**" or "**Function blocks**" section of the open project. Local variables and defined words attached to the imported function are copied with it. When a function has been correctly imported from the library, it can be placed in another section or another location in the hierarchy tree, using the "**File / Rename/Move**" command. In order to avoid naming clashes, the imported function or function block must be renamed when imported in the project area. Don't forget to rename also the return parameter in the case of a function.

☰ **Exporting function or function block to library**

The "Tools / Export to library" command is used to send a program of the "Functions" or "Function blocks" section (in the open project) to the appropriate library. Local variables and defined words attached to the exported function or block are copied with it. The exported function or block will have to be re-compiled (verified) from the ISaGRAF Library Manager, to ensure that it can be used in a library environment. Functions and function blocks of the library cannot use global variables.

A.3.3 Running the code generation tools

The commands of the "Make" menu are used to run the code generator, and to enter options and additional data used when producing the application code. Refer to the chapter "Using the code generator" in this document for further information about these tools.



Make the application code

The "Make" command starts the project code generation. The options for target code generation must be set correctly before running this command. Before generating the target code, any program that is still not verified is checked to detect the syntax errors. ISaGRAF includes an incremental compiler, which does not re-compile programs, which have already been compiled.



Verify the selected program

The "Verify" command allows the user to verify the syntax of the program currently selected in the list. When a program is verified, with no error detected, it is not re-verified during the code generation until its contents or dependent defined words or variables change.

☰ **Simulating a modification**

The "Touch" command simulates a modification of each program so that all of them will be compiled again during the next code generation.



Application run-time options

This command opens a dialog box where are entered the main run-time parameters for the execution of the application. This includes the cycle timing programming, run time error management, the starting mode and the hardware implementation of retained variables. Refer to the chapter "Using the Code Generator" in this document for more explanations about this command.

☰ **Compiler options**

This command is used to set-up the options used by the ISaGRAF Code Generator to produce and optimise target code. Refer to the chapter "Using the Code Generator" in this document for more explanations about this command.

☰ **Defining resources**

A "**resource**" is a user defined data (for example a file) which has to be merged with the target code so it can be downloaded with it. Refer to the section "Using the Code Generator" in this document for more explanations about the format of the resource definition file.

A.3.4 Other ISaGRAF tools

The "**Project**" menu groups the commands that run ISaGRAF tools for the selected project. Refer to the corresponding chapters in this document for more information about these tools.



Wiring I/O variables

The "**IO connection**" command runs the ISaGRAF I/O variable connection editor. This tool is used to establish the relationship between I/O variables declared in the project dictionary and corresponding I/O hardware.



Running the cross reference editor

The "**Cross references**" command allows the user to calculate, to view or to print the cross references of the project. The cross-references show the user all the occurrences of each variable in the source code of the programs, in the entire project. This function is very useful to detect an access to a variable or any global resource, or to list all the occurrences of a global variable in the source code.



Entering the project descriptor

The "**Project descriptor**" command is used to edit the project text descriptor. This document fully identifies the project from the others on the project list. The project descriptor can also be used to record any remarks during the project lifetime. The project descriptor is the one displayed in the Project Manager window.



Printing a complete document

The "**Print project document**" command allows the user to build and print a complete document about the selected project. This document can group any component (program, variable, parameters...) of the selected project. To build a specific (non-complete) document, the user only has to define its table of contents.



History of modifications

This command opens a dialog box where is displayed the history of modifications for the project. Refer to the chapter "Managing projects" in this document for more explanations about this command.

A.3.5 Adding commands to the Tools menu

ISaGRAF provides the way to insert other commands in the "**Tools**" menu. User defined commands to be added are described in "**ISAWIN\COM\ISA.MNU**" text file. You can add up to 10 commands. Comments may be inserted on any line, beginning with ";" character. Each command is described on two lines of text, according to the following syntax:

```
M=menu_string  
C=command_line
```

The menu string is the text to be displayed in the **"Tools"** menu. The command line is any MS-DOS or Windows executable, and can be completed with arguments. In command line, you can use **"%A"** characters to replace the name of the open project, and **"%P"** characters to replace the name of the selected program. The following example runs "Notepad" to edit the selected program (to be used with ST and IL programs):

```
M=Edit with Notepad  
C=Notepad.exe \isawin\apl\%A\%P.lsf
```

A.3.6 Simulating and debugging the application

The command of the **"Debug"** menu are used to run the ISaGRAF graphic debugger, either in simulation mode or in real connected mode.



Simulation

The **"Simulate"** command opens the debugger in simulation mode. In this mode, another window appears, called the simulator. This command is very useful to test any application when the target machine is unavailable. Starting the simulator closes the Program Management window. The Program Management window is then re-opened in debug mode after both debugger and simulation windows are opened. The simulator cannot be started if the target code has not been generated. The simulator cannot be started when child windows (editors, code generation, I/O connection...) are opened. Each of them must be closed before running this command. This command is also available from menus of ISaGRAF editors.



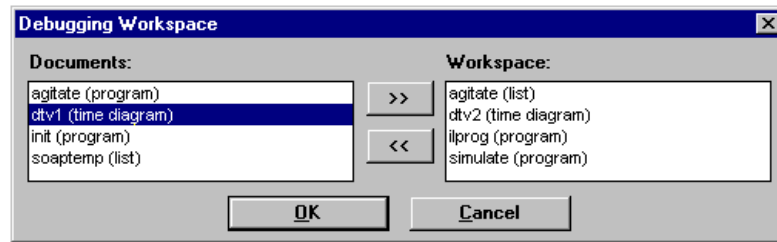
Real debugging

The **"Debug"** command opens the debugger main window, and closes the Program Management window. The Program Management window is then re-opened in debug mode as soon as communication is established between the debugger and the target application. The debugger cannot be started if the target code has not been generated. The debugger cannot be started when child windows (editors, code generation, I/O connection...) are opened. Each of them must be closed before running this command. This command is also available from menus of ISaGRAF editors.



Preparing the debug workspace

The **"Debug / Workspace"** command enables you to define a list of documents for initial workspace. Such documents can be programs, SpotLight graphics, and lists of variables. Graphics and lists of time diagrams from previous ISaGRAF versions are also listed with project documents. Documents defined in the initial workspace are automatically opened when simulation or On Line monitoring is launched.



The dialog box shows the existing documents of the project on the left, and documents selected for the initial workspace on the right. Use ">>" and "<<" push buttons to move documents from one list to the other. Each project has its own list of documents for initial workspace.



Link set-up

The "**Link set-up**" command enables the user to define the parameters of the link used for communication between the debugger on the host PC and the target ISaGRAF system.

The "**Slave number**" identifies the target ISaGRAF system or task. It can be from **1** to **255**. Refer to the target supplier manual for the slave number of the target system used.

The "**Communication port**" identifies the hardware media between ISaGRAF workbench and target. It can be either the name of a serial port, or "**Ethernet**", reserved TCP-IP communication using the "Winsock" Version 1.1.

The "**Time out**" is the time left to the target system for its communication operations between the end of a debugger question and the beginning of its response. This time is set as a number in **milliseconds**. The "Retries" field is the number of automatic trials the debugger executes for a communication operation before detecting a communication error.



Serial link set-up

When a serial port (COM1..4) is selected, the "**Set-up**" button is used to access to other serial link communication parameters.

The transmission baud rate, parity and format may be selected. When the "**hardware**" choice is selected for "**Flow Control**", the ISaGRAF Workbench controls the CTS and DSR lines to enable hardware handshaking during exchanges.



Ethernet link set-up

When "Ethernet" is selected as a communication port, the "**Set-up**" button is used to enter the "Internet Address" and "Internet port" number, for TCP-IP communication.

These fields use the standard formats defined by the Socket interface. The Workbench uses the WINSOCK.DLL Version 1.1 library for TCP-IP communications. This file must be correctly installed on the hard disk. "**1100**" is the default port number used if not specified when running the ISaGRAF target.

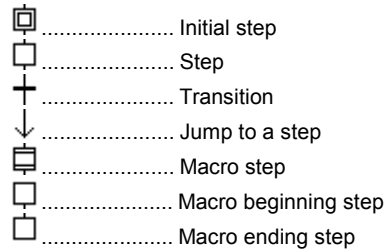
A.4 Using the SFC editor

The SFC language is used to describe operations of a sequential process. It uses a simple graphic representation for the different steps of a process, and conditions that enable the change of active steps. An SFC program is entered by using the ISaGRAF graphic SFC editor. SFC is the core of the IEC 1131-3 standard. The other languages usually describe the actions within the steps and the logical conditions for the transitions. The ISaGRAF graphic SFC editor allows the user to enter complete SFC programs. It combines graphic and text editing capabilities, thus allowing the entry of both the SFC chart, and the corresponding actions and conditions.

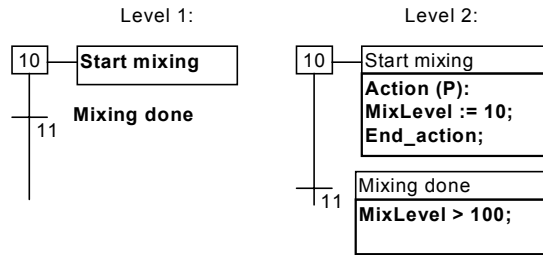
A.4.1 SFC language main topics

The SFC language is used to represent sequential processes. It divides the process cycle into a number of well-defined successive **steps** (self-contained situations), separated by **transitions**. Refer to the ISaGRAF Languages Reference Manual for more details on the SFC language.

SFC components are joined by **oriented lines**. The default orientation of a line is **up to down**. These are the basic graphic components used to build an SFC chart:



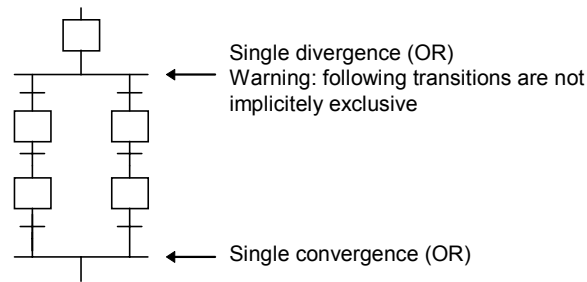
The SFC programming is usually separated into two different levels: The **Level 1** shows the graphic chart, reference numbers of the steps and the transitions, and comments attached to the steps and the transitions. The **Level 2** is the **ST** or **IL** programming of the actions within the steps, or the conditions attached to the transitions. Actions or conditions may refer to **sub-programs** written in other languages (**FBD**, **LD**, **ST** or **IL**). Below is an example of level 1 and level 2 programming:



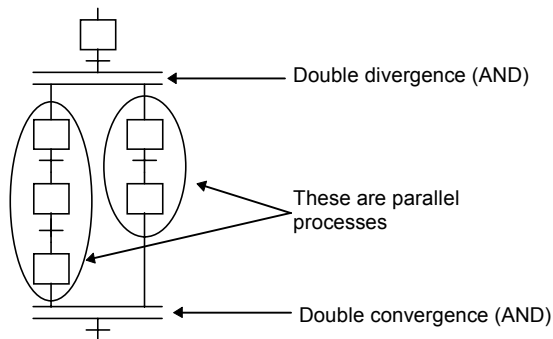
The level 2 programming of a step is entered in a text editor. It can include action blocks programmed in ST or IL. The level 2 programming of a transition can be entered either in IL or ST text languages, or with Quick LD editor.

≡ **Divergences and convergences**

Divergences and convergences are used to represent **multiple links** between steps and transitions. Simple divergences or convergences represent different **inclusive** possibilities between different sub parts of the process.

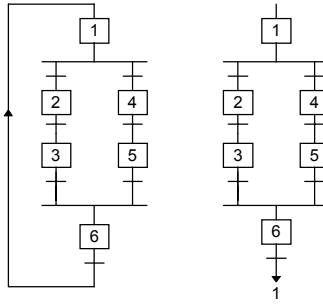


Double divergences represent **parallel** processes.



≡ **Jump to a step**

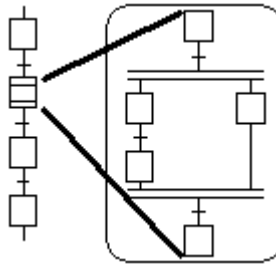
The SFC editor only allows the user to draw links in the **up** to **down** direction. A **jump** to a step can be used to represent a link to an upper part of the chart. Following charts are equivalent:



Jump to a transition is forbidden, and must be explicitly represented as a double (AND) convergence.

⇒ **Macro steps**












A macro step is a **unique** representation of a **stand-alone** group of steps and transitions. A macro step begins with a **beginning step** and terminates with an **ending step**.



The detailed representation of a macro step must be described in the same SFC program. The macro-step symbol must have the same **reference number** as the macro beginning step. A macro step description may contain another macro step.

A.4.2 Entering an SFC chart

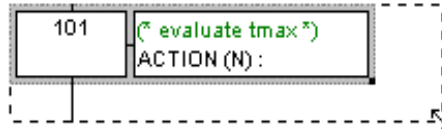
To draw an SFC chart, the user simply has to introduce the significant components of the chart. All the single lines joining two elements (horizontally or vertically) are drawn automatically by the SFC editor. To place an SFC component on the chart, the user has to move the selection to appropriate location and select the type of the component in the editor toolbar. The symbol is inserted at the current position. The following keyboard sequences can also be used:

- F2:  Insert an initial step
 F3:  Insert a single step
 F4:  Insert a transition
 F5:  Insert a jump to a step
 F6:  F7:  Insert an OR divergence or convergence / Add branches
 †F6:  †F7:  Insert an AND divergence or convergence / Add branches
 F8:  Insert a macro step
 F9:  †F9:  Insert begin or end step for the body of a macro step

(The "†" symbol indicates a combination with SHIFT key)





The **editing grid** shows **matrix cells**. An editor option allows the user to show or hide the grid during chart input. The grid is very useful for initial layout of SFC chart, or selecting sub-parts of the chart. Use the "**Options / Layout**" command to display or hide the grid.

The ISaGRAF SFC editor always shows the current position in the matrix. The focused cell is marked in grey. The small square on its bottom right corner can be used to freely resize the cells. The X/Y ratio of the cells can also be changed this way.







Creating a divergence or convergence

Divergences and convergences are always drawn **from the left to the right**. To draw a divergence or a convergence, its **left branches** has to be placed on the chart area. The type of drawing (simple or double) is set by selecting one of these buttons in the toolbar.

- F6:  F7:  Insert an OR divergence or convergence / Add branches
 †F6:  †F7:  Insert an AND divergence or convergence / Add branches

Adding branches to divergences

The **start** and **stop** position of each **auxiliary branch** is placed on the divergence or convergence line using these buttons in the toolbar. The left corner of the divergence or convergence must be present before inserting new branches. The right corners have the same style (simple or double) as the main left corner. Right corners cannot be placed if the main left corner has not been added.

- F6:  F7:  Insert an OR divergence or convergence / Add branches
 †F6:  †F7:  Insert an AND divergence or convergence / Add branches



Inserting a macro step

This button is used to insert a macro step in the main chart. The body of the macro step must be entered elsewhere in the same SFC program.



Body of a macro step

Macro steps must be described in the same SFC program as the main chart. A macro step must start with a **beginning step** and stop with an **ending step**. The sub-chart described as the macro implementation must be **self-contained**. The macro beginning step must have the same **reference** as the macro-step symbol of the main branch.

A.4.3 Working on an existing SFC chart

You can use either the mouse or keyboards arrows to select a rectangle area in the chart. The whole selected area is marked in grey. The commands of the "Edit" menu can then used:



Cut / copy / delete / paste commands

The following commands are available from the "Edit" menu when the "arrow" button is selected in the editor toolbar:

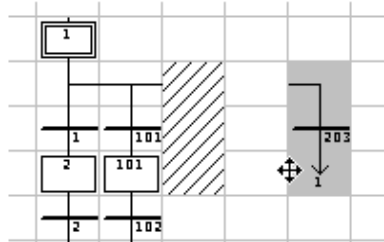
- Cut Move selected rectangle from the screen to the SFC clipboard
- Copy..... Copy selected rectangle from the screen to the SFC clipboard
- Delete..... Clear (delete) selected rectangle
- Paste..... Insert contents SFC clipboard at the current position

The "Edit / Paste" copies SFC clipboard to the screen. Copy / Paste commands work on both SFC chart and step/transition level 2 programming. It is also possible to copy a chart in a program and paste it in another SFC program. Elements are inserted before the currently selected position.



Move elements

When SFC elements are selected in the SFC chart, you can move them to another location of the chart by dragging the selection with the mouse. While you drag the selection, the initial location of selected elements is hatched.



The destination area for moved elements must be empty. No insertion is possible while moving SFC symbols.

☰ **Renumbering steps and transitions**

Each step or transition is identified by a logical number in the SFC chart. The "**Edit / Renumber**" command allows the user to automatically set up numerically sequential reference numbers for any of the steps and the transitions of the currently edited SFC program. When a step number is changed, all the jumps to this step are automatically updated with the new reference number. (This also applies to macro steps and beginning steps)



Direct access to a step or transition

The "**Edit / Go to**" command allows the user to access an existing step or transition. The scrolling position is automatically adapted so that the step or transition is visible.

☰ **Find and replace texts**

The "**Edit / Find Replace**" command can be used to find or replace text strings in the complete program (all steps and transitions). The Find/Replace dialog box is used to enter a searched text and directly open the level 2 programming section where text occurs.

A.4.4 Entering the level 2 programming

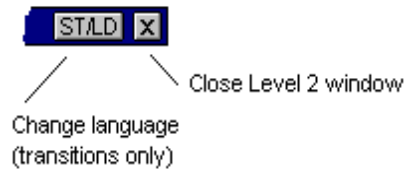
To enter the Level 2 text, the user must double click on the step or transition symbol. The level 2 programming is displayed on the right of the SFC window. The separation line between SFC chart and level 2 programming can be freely moved.

You can open one or two level 2 areas at the same time. The following commands are available from keyboard, mouse or the "Edit" menu:

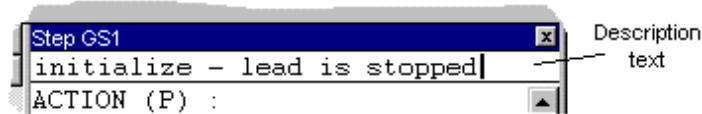
	<i>Keyboard</i>	<i>Mouse</i>	<i>"Edit" menu</i>
Open in last default window	Enter	Double Click	Edit level 2
Open in separate window	Ctrl+Enter	Ctrl + DoubleClick	Edit Level 2 in separate window

When two level 2 windows are visible, the separation between them can be freely moved. The button on the right of the level 2 title bar is used to close a level 2 window.

The default language for Level 2 programming is **ST** (Structured Text). For transitions, level 2 programming can also be entered with **Quick LD** editor. Use the "**ST/LD**" button in level 2 title bar to change the active language. This command is valid only if the level 2 programming window is empty.



A single line edit box appears at the top of the level 2 window. It is used to enter a short description text. This text will be displayed as an IEC comment in drawing of SFC symbols. It is very useful as it is used by other commands such as "Go To..." and also in the SFC printout to document SFC steps and transitions.



The "Options / Refresh" command can be used at any time when level 2 windows are open to refresh the main SFC chart with modified level 2 programs.



Inserting a variable name

When programming in text language, press this button to select a variable declared in the project dictionary and insert its name at the current position of the caret. When programming in Quick LD, press this button to select the variable to be attached to the selected contact or block I/O parameter.



Inserting a Pulse action block in step

When programming the level 2 of a step, press this button to insert the template of a Pulse action block at the current position of the caret. Below is the format of a Pulse action block:

```

Action (P) :
  ST statement;
  ...
End_Action;

```

Pulse actions are instructions, which are executed only once when the step becomes active. Refer to the ISaGRAF language reference for further details on SFC programming.



Inserting a Non stored action block in step

When programming the level 2 of a step, press this button to insert the template of a Non stored action block at the current position of the caret. Below is the format of a Non stored action block:

```

Action (N) :
  ST statement;
  ...
End_Action;

```

Non stored actions are instructions which are executed on every PLC cycle when the step is active. Refer to the ISaGRAF language reference for further details on SFC programming.

P0 P1 *New P0 and P1 action qualifiers*

ISaGRAF supports new **P0** and **P1** action qualifiers. When programming the level 2 of a step, press these buttons to insert the template of a P0 or P1 action block at the current position of the caret. Below is the format of such blocks:

```

Action (P0) :           Action (P1) :
  ST statement;         ST statement;
  ...                  ...
End_Action;           End_Action;

```

P1 actions are instructions which are executed only once when the step becomes active (same as Pulse). P0 actions are instructions, which are executed only once when the step becomes inactive. Refer to the ISaGRAF language reference for further details on SFC programming.

Boolean actions

Other text semantics are available to directly act on a boolean variable according to the step activity. Such actions consist of attaching the **step activity signal** to an internal or output boolean variable. This is the syntax of the basic boolean actions:

```

<boolean_variable> (N);           assigns the step activity signal to the variable
<boolean_variable>;              same effect (N attribute is optional)
/ <boolean_variable>;            assigns the negation of the step activity signal
                                to the variable

```

Other features are available to set or reset a boolean variable, when the step becomes active. This is the syntax of set and reset boolean actions:

```

<boolean_variable> (S);           sets the variable to TRUE when the step
                                activity signal becomes TRUE
<boolean_variable> (R);           resets the variable to FALSE when the step
                                activity signal becomes TRUE

```

SFC actions

Other text semantics are available to control the execution of a child SFC program. An SFC action is a child SFC sequence, started or killed according to the condition of the step activity signal. An SFC action can have the **N** (Non stored), **S** (Set), or **R** (Reset) qualifier. This is the syntax of the basic SFC actions:

```

<child_program> (N);              starts the child sequence when the step
                                becomes active, and kills the child sequence
                                when the step becomes inactive
<child_program>;                 same effect as the preceding syntax (N
                                attribute is optional)

```

<child_program> (S);	starts the child sequence when the step becomes active - nothing is done when the step becomes inactive
<child_program> (R);	kills the child sequence when the step becomes active - nothing is done when the step becomes inactive

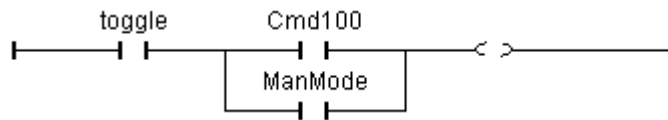
The SFC sequence specified as an action must be an existing **child SFC program** of the currently edited program, created with the ISaGRAF program manager.

▣ **Transitions written in ST**

The level 2 of a transition is a boolean expression. To program it in ST language, just enter the boolean condition according to the ST syntax. Optionally, a semicolon may be added at the end of the expression.

▣ **Transitions written in Quick Ladder**

Quick LD editor is available to program the level 2 condition of a transition. In this case, the diagram is made of just one rung, with only one coil, which represents the transition. The name of the transition is not repeated with the coil symbol. Below is an example of transition condition programmed in Quick LD.



When programming in Quick LD, use the keyboard arrows to move the selection in the programming logical grid, and then use the following shortcuts to insert symbols:

F2:..... insert a contact after the selected symbol / initiate the rung

F3:..... insert a contact before the selected symbol

F4:..... insert a contact in parallel with the selected symbol

F6:..... insert a block after the selected symbol

F7:..... insert a block before the selected symbol

F8:..... insert a block in parallel with the selected symbol

You can also click on the function key bar at the bottom of the level 2 window instead of hitting function keys.

Hit RETURN when the selection is on a contact or a block I/O parameter to select a variable or enter a constant value. Hit RETURN when the selection is on a function block to select the type of the function block. You can also double click on a symbol for the same effect.

Hit SPACE bar when a contact is selected to change the type of contact (direct, negated or with pulse detection). Refer to the chapter "Using the Quick LD editor" in this document for more details about Quick LD capabilities.

A.4.5 Using the SFC gallery

The ISaGRAF SFC editor manages an SFC gallery: it is a collection of SFC structures that can be inserted in any SFC chart. Elements of the SFC gallery can optionally embed the level 2 programming of steps and transitions. Use the following commands of the "**Tools**" menu:

Copy to SFC gallery copy selected elements to SFC gallery
Paste from SFC gallery paste an SFC gallery element at the current location

When copying to SFC gallery (i.e. creating a new SFC gallery element), you can optionally ask to embed level 2 programming of selected SFC symbols.

A.5 Using the Flow Chart editor

The ISaGRAF Flow Chart graphic editor allows the user to enter complete FC (Flow Chart) programs, with actions and tests (decisions) programmed in either ST, IL or Quick LD language. Flow Chart is a decision diagram, which can also be used to describe sequential operations as it enables some advanced features such as non-blocking backward jumps.

A.5.1 Basics of the FC language

Flow Chart (FC) is a graphic language used to describe sequential operations. A Flow Chart diagram is composed of Actions and Tests. Between Actions and tests are oriented links representing data flow. Below are graphic components of the Flow Chart language:



Beginning of FC chart: A "begin" symbol must appear at the beginning of a Flow Chart program. It is unique and cannot be omitted. It represents the initial state of the chart when it is activated.



Ending of FC chart: An "end" symbol must appear at the end of a Flow Chart program. It is unique and cannot be omitted. It is possible that no connection is drawn to the "End" symbol (always looping chart), but "End" symbol is still drawn anyway at the bottom of the chart. It represents the final state of the chart, when its execution has been completed.



FC flow links: A flow link is a line that represents a flow between two points of the diagram. A link is always terminated by an arrow. Two links cannot start from the same source connection point.



FC actions: An action symbol represents actions to be performed. An action is identified by a number and a name. Two different objects of the same chart cannot have the same name or logical number. Programming language for an action can be ST, LD or IL. An action is always connected with links, one arriving to it, one starting from it.



FC tests: A test represents a boolean condition. A test is identified by a number and a name. According to the evaluation of attached ST, LD or IL expression, the flow is directed to "YES" or "NO" path. When programmed in ST text, the expression may optionally be followed by a semicolon. When programmed in LD, the unique coil represents the condition value.



FC sub-program: The system enables the description of a hierarchised structure of FC programs. FC programs are organised in a hierarchy tree. Each FC program can call other FC programs. Such a program is called a child program of the FC program, which calls it. FC programs, which call FC sub-programs, are called father program. FC programs are linked together into a main hierarchy tree, using a

"father - child" relation. A sub-program symbol in a Flow Chart represents a call to a Flow Chart sub-program. Execution of the calling FC program is suspended till the sub-program execution is complete.



FC I/O specific action: An I/O specific action symbol represents actions to be performed. As other actions, an I/O specific action is identified by a number and a name. The same semantic is used on standard actions and I/O specific actions. The aim of I/O specific actions is only to make the chart more readable and to give focus on non-portable parts of the chart. Using I/O specific actions is an optional feature. I/O specific blocks have exactly the same behaviour as standard actions.



FC connectors: Connectors are used to represent a link between two points of the diagram without drawing it. A connector is represented as a circle and is connected to the source of the flow. The drawing of the connector is completed, on the appropriate side (depending on the direction of the data flow), by the identification of the target point (generally the name of the target symbol). A connector always targets a defined Flow Chart symbol. The destination symbol is identified by its logical number.



FC comments: A comment block contains text that has no sense for the semantic of the chart. It can be inserted anywhere on a free space of the Flow Chart document window, and is used to document the program.







A.5.2 Entering a Flow Chart

To enter a chart, you have to place elements (actions, decision tests, connectors...) in the graphic area, and draw flow links between them.






Inserting objects

To insert an object in the diagram, select the corresponding button in the toolbar and click in the graphic area, where you want to insert it. You can either put the element on an empty area, or insert it in a flow by clicking on a flow link. Insertion on a link is allowed for top to bottom vertical links only. You can insert the following basic elements:

 action programmed in ST, IL or Quick LD
 I/O specific action (highlights a particular non-portable action)
 test (decision) programmed in ST, IL or Quick LD
 connector
 call to an FC sub-program
 comment (description text)

The ISaGRAF Flow Chart editor also proposes you a list of classical Flow Chart structures. Such structures can only be inserted on an existing flow link. They cannot be put in an empty area:

-  If / Then / Else (binary selection)
-  Repeat until (waits for a condition)
-  While (loops while a condition is true)



Selecting objects

Selecting graphic objects is needed for most of the editing commands. The ISaGRAF FC graphic editor enables the selection of one or more objects existing in the diagram area. To select objects, the "select" (button with an arrow) choice must be checked in the editor toolbar. To select one object, the user only has to click on its symbol.

To select a list of objects, drag the mouse in the diagram to draw a rectangle area. All graphic objects in the selection rectangle are marked as "selected".

A selected object is drawn in dark blue colour, with little black squares around its graphic symbol. It is also possible to add or remove one object to a multiple selection, by clicking on its symbol with **Shift** or **Ctrl** key pressed.

By making a new selection, selection of all objects previously selected is removed. To remove the existing selection, simply click with the mouse in an empty area, outside of the rectangle which borders the selected objects.

For single selection, it is possible to use keyboard arrows to move selection from one object to the other in the chart. Flow links can also be selected.



Inserting comments

Comments may be inserted anywhere in an empty part of the diagram. Comments have no influence on the program execution. They allow a higher readability of the diagram. To insert a comment block, select the corresponding button in the toolbar, and click in the diagram where comment must be put. Double click on a comment to enter / change its text. No special leading or trailing characters such as "(" and ")" are needed when entering the text of a comment block. A comment block may be resized by dragging the corners of its border when it is selected.



Drawing flow links

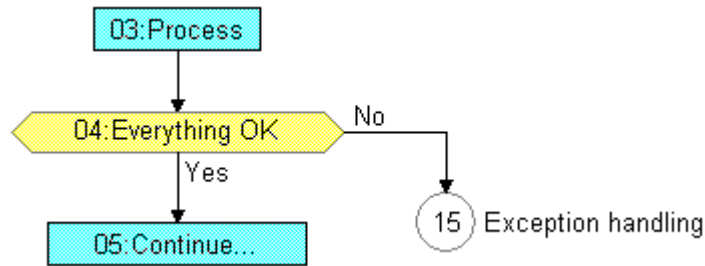
Select this button in the toolbar to draw a flow link between existing elements. A link must always be drawn in the direction of the flow. First select a non-connected output point of an FC element, and drag the mouse to the destination point to insert the link. The destination point can either be the top (input point) of a non-connected FC element, or any location on an existing link. Convergence points between links are shown with small grey circles in the Flow Chart. Convergence points can also be selected and moved in order to arrange the diagram.



Using connectors

The ISaGRAF Flow Chart editor enables the use of graphic connectors, as a replacement of a visible flow link. Connectors can be very useful to avoid very long links and increase chart readability. A connector cannot be used to establish a link with another FC program.

A connector is put in the chart as other FC objects. It is represented by a circle containing the numerical reference of targeted element (destination of the flow link). The short description text of the target element is displayed close to the connector circle.



Moving objects

To move objects in the chart, you have to select them, and drag the mouse to move them within the chart. You can either move a single element or a multiple selection. Elements cannot be overlapped when moving them. Moving elements cannot be used to connect them to an existing link.

When a single element (action, test...) is moved, the ISaGRAF Flow Chart editor automatically moves with the selected element all objects placed below and connected to it. This feature does not operate in the case of a multiple selection.



Resizing objects

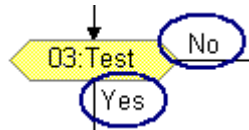
Any graphic element of a flow apart from "Begin", "End" symbols and connectors can be resized freely. To resize an element, you first have to select it. Then drag with the mouse the small squares drawn on its border to change its size.

When an element is connected to a flow link, resizing it horizontally acts on both left and right borders, so that the element is still correctly centred on the link when resized.



Swapping the outputs of a test

You can swap locations of YES / NO outputs on a test (decision). To do that, simply double click on either "Yes" or "No" marks displayed close to the test symbol.




A.5.3 Working on an existing chart

The commands of the "Edit" menu are used to change or complete an existing diagram. Most of these commands act on the elements currently selected in the diagram.

 **Correcting a chart**

The DEL key can be used to remove the selected elements. Pending links are deleted with selected elements. Use "Edit / Undo" command to restore elements after a DEL command. The DEL command can also be applied to a group of elements selected in the diagram. The "Cut", "Copy", "Paste" commands of the "Edit" menu are used to move or copy selected elements.

 **Find and replace**

The "Edit / Find Replace" commands can be used to find or replace text strings in the complete program (all actions and tests programmed in ST, IL or Quick LD). The Find/Replace dialog box is used to enter a text to be searched and to directly open the programming section where the text is found.

 **Direct access to an element**

The "Edit Go to" command allows the user to access a graphic element existing in the chart. The scrolling position is automatically adapted so that the element is visible. The element, when reached, is selected.

 **Renumbering elements**

The "Edit / Renumber" command is used to renumber elements of the Flow Chart. Any FC element put in the chart is identified with a unique reference number. Reference numbers are allocated by the editor each time new elements are inserted. The "Renumber" allows you to re-adjust element numbering according to their location in the chart. Growing numbering is performed from top to bottom and from left to right

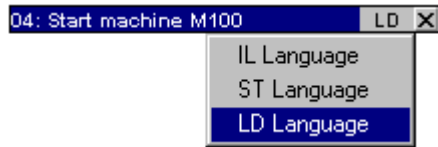
A.5.4 Entering level 2 programs

To enter the level 2 program, the user must double click on the action or test symbol. The level 2 programming is displayed on the right of the FC window. The separation line between FC chart and level 2 programming can be freely moved. You can open one or two level 2 areas at the same time. The following commands are available from keyboard, mouse or the "Edit" menu:

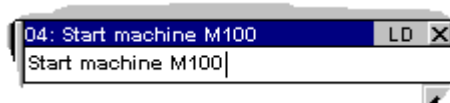
	<i>Keyboard</i>	<i>Mouse</i>	<i>"Edit" menu</i>
Open in last default window	Enter	Double Click	Edit level 2
Open in separate window	Ctrl+Enter	Ctrl + DoubleClick	Edit Level 2 in separate window

When two level 2 windows are visible, the separation between them can be freely moved. The button on the right of the level 2 title bar is used to close a level 2 window.

The default language for Level 2 programming is **ST** (Structured Text). The programming language can also be **IL** or **Quick LD**. The name of the selected language is displayed in a small box in the level 2 title bar. Run the "**Options / Set Level 2 language**" command from menus or click on that box to change the active language. This command is valid only if the level 2 programming window is empty.



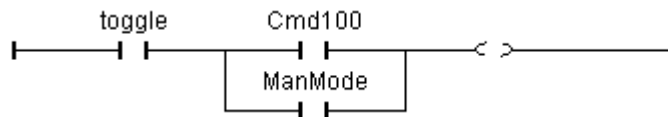
A single line edit box appears at the top of the level 2 window. It is used to enter a short description text. This text will be displayed as an IEC comment in the drawing of FC symbols. It is very useful as it is used by other commands such as "Go To..." and also in the FC printout to document FC actions and tests.



The "**Options / Refresh**" command can be used at any time when level 2 windows are open to refresh the main FC chart with modified level 2 programs.

A.5.5 Programming level 2 with Quick LD

Quick LD editor is available for level 2 programming. In the case of a decision test, the LD diagram is made of just one rung, with only one coil, which represents the decision. The name of the test is not repeated with the coil symbol. Below is an example of a test programmed in Quick LD.



When programming in Quick LD, use the keyboard arrows to move the selection in the programming logical grid, and then use the following shortcuts to insert symbols:

- F2:..... insert a contact after the selected symbol / initiate the rung
- F3:..... insert a contact before the selected symbol
- F4:..... insert a contact in parallel with the selected symbol
- F5:..... add a coil in parallel with the selected one (not for tests)
- F6:..... insert a block after the selected symbol
- F7:..... insert a block before the selected symbol
- F8:..... insert a block in parallel with the selected symbol

F9:..... add a jump symbol in parallel with the selected coil (not for tests)

A jump leads to a rung name. The name of a rung can be entered by hitting ENTER when selection is on the rung head. The ISaGRAF editor keeps the memory of the rung labels you already entered, whether it has been specified for a rung name or a jump operation. The "Jump/Label" dialog box gives you the possibility either to enter a new label, or to select an existing one. If you enter a new name, it will automatically be added to the list. The "**Remove**" button is used to remove the selected name from the list. It does not remove the label on the rung you selected in the diagram. To do this, just press **OK** when the edit box is empty.

You can also press buttons in the LD toolbar instead of hitting function keys.

Hit ENTER when the selection is on a contact or a block I/O parameter to select a variable or enter a constant value. Hit ENTER when the selection is on a function block to select the type of the function block. You can also double click on a symbol for the same effect.

Hit Control + SPACE bar when a contact is selected to change the type of contact or coil (direct, negated). Refer to the chapter "Using the Quick LD editor" in this document for more details about Quick LD capabilities.

A.5.6 Display options

The "**Options / Layout**" command opens a dialog box where are grouped all the parameters and options concerning the editor workspace and the drawing of the diagram. Use the check boxes in the "Workspace" group to display or hide editor toolbars and status bar. Option of the "Document" group allow you to show or hide points of the editing grid and to display chart either in black and white or with colours.



Use the "Zoom" button of the toolbar to change current zoom ratio. This command is also available when working on a Quick LD program attached to an action or a test.



Use the "Grid" button of the toolbar to show or hide points of the editing grid. This command is also available when working on a Quick LD program attached to an action or a test.

Use the "**Options / Font**" command to select the name of the character font to be used in all ISaGRAF documents. When called from an ST or IL block, you can specify size of the font. When selecting font for a graphic view (FC or Quick LD), font style and size are not relevant and do not need to be specified. ISaGRAF graphic editors always calculate the font size according to the current zoom ratio.

A.6 Using the Quick LD editor

The LD language enables graphic representation of boolean expressions. Boolean AND, OR, NOT operators are explicitly represented by the diagram topology. Boolean input variables are attached to graphic contacts. Boolean output variables are attached to graphic coils. The ISaGRAF Quick LD editor provides easy LD diagram entering using either keyboard or mouse. Elements are automatically linked and arranged on rungs by the Quick LD editor. No connection is drawn manually by the user. The Quick LD editor also arranges rungs in the diagram so that the space filled by the diagram is always optimised.

A.6.1 Basics of the LD language





An LD program is expressed as a list of **rungs** where contacts and coils are arranged. Below are the basic components of an LD diagram:

Rung head (left power rail)

Each rung begins with a left power rail, which represents the initial "TRUE" state. ISaGRAF Quick LD editor automatically creates the left power rail when the first contact of the rung is placed by the user. Each rung may have a logical name, which can be used as a label for jump instructions.







Contacts

A contact modifies the boolean data flow, according to the state of a boolean variable. The name of the variable is displayed upon the contact symbol. The following types of contacts are supported by ISaGRAF Quick LD editor:

-  direct contact
-  negated contact
-  contact with positive (rising) edge detection
-  contact with negative (falling) edge detection

Coils

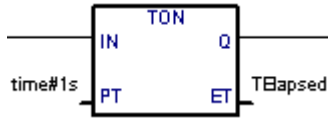
A coil represents an action. The rung state (state of the link on the left of the coil) is used to force a boolean variable. The name of the variable is displayed upon the coil symbol. The following types of coils are supported by ISaGRAF Quick LD editor:

-  direct coil
-  negated coil
-  "set" action coil
-  "reset" action coil
-  coil with positive (rising) edge detection
-  coil with negative (falling) edge detection



Function blocks

A block in an LD diagram can represent a function, a function block, a sub-program or an operator. Its first input and output parameters are always connected to the rung. Other input and output parameters are literally written outside of the block rectangle.



Rung end (right power rail)

A rung ends with a right power rail. Using the Quick LD editor, the right power rail is automatically inserted when a coil is placed by the user.



Jump symbol

A jump symbol always refers to a rung label, i.e. the name of a rung defined somewhere in the same LD diagram. It is placed at the end of a rung. When the rung state is TRUE, the execution of the diagram directly jumps to this target rung. Note that backward jumps are dangerous as they may lead to a blocking of the PLC loop in some cases.



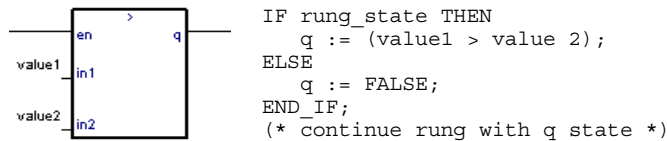
Return symbol

A return symbol is placed at the end of a rung. It indicates that the execution of the program must be stopped if the rung state is TRUE.



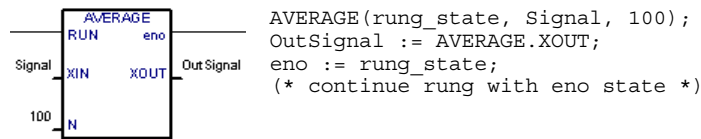
The "EN" input

On some operators, functions or function blocks, the first input does not have boolean data type. As the first input must always be connected to the rung, another input is automatically inserted at the first position, called "EN". The block is executed only if the EN input is TRUE. Below is the example of a comparison operator, and the equivalent code expressed in ST:

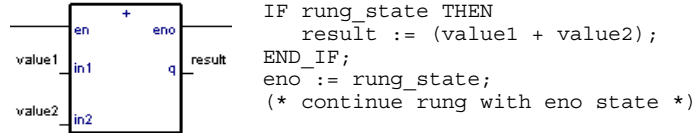


The "ENO" output

On some operators, functions or function blocks, the first output does not have boolean data type. As the first output must always be connected to the rung, another output is automatically inserted at the first position, called "ENO". The ENO output always takes the same state as the first input of the block. Below is an example with AVERAGE function block, and the equivalent code expressed in ST:



On some cases, both **EN** and **ENO** are required. Below is an example with an arithmetic operator, and the equivalent code expressed in ST:



HHH Limitations of Quick LD editor

The ISaGRAF Quick LD editor does not allow to continue a rung (insert other contacts or coils) on the right of a coil. If several outputs have to be made on the same rung, the corresponding coils must be drawn in parallel.

A.6.2 Entering an LD diagram

All the editing commands of the Quick LD editor may be achieved either with the keyboard or with the mouse.



The editing grid

The LD diagram is entered in a logical matrix. Each cell of the matrix may contain up to one LD symbol. Use the arrows of the keyboard, or click on a cell to move the current selection. The selected cell is marked in reverse. For some cut/copy/paste operations, it is possible to select several cells. To do that with the mouse, just drag the mouse cursor in the diagram. With keyboard, use arrow keys with SHIFT key pressed.



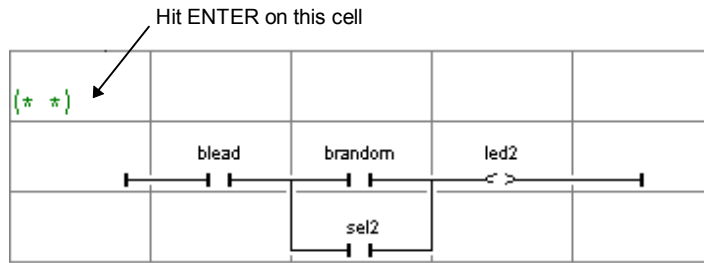
Starting a new rung

To add a new rung to a diagram, move the selection after the last existing rung and insert a contact (hit F2 or press the corresponding button in the LD toolbar). A new rung with one contact and one coil is created.



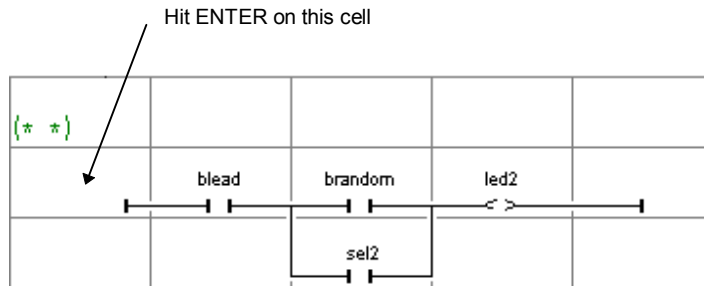
Entering the rung comment

Each rung may be documented with up to two lines of text. To enter a rung comment text, move the selection on the cell upon the rung and hit ENTER key, or double click on this cell with the mouse:



⊞ **Entering the rung label**

Each rung may be identified by a name. This name can be used as a target label for jump operations. To enter or change the label of a rung, move the selection on rung head and hit ENTER key, or double click on this cell with the mouse:



The ISaGRAF Quick LD editor keeps the memory of the rung labels you already entered, whether it has been specified for a rung name or a jump operation. The "Jump/Label!" dialog box gives you the possibility either to enter a new label, or to select an existing one.

If you enter a new name, it will automatically be added to the list. The **"Remove"** button is used to remove the selected name from the list. It does not remove the label on the rung you selected in the diagram. To do this, just press **OK** when the edit box is empty.

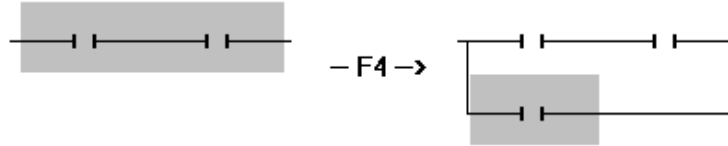
⊞ **Inserting symbols on a rung**

The insertion of symbols (contacts, coils, blocks...) on an existing rung is always made according to the current selection. You have to select a valid cell position within the rung and hit one of the following function keys to insert:

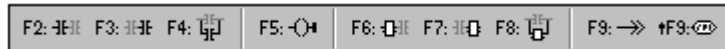
- F2..... a contact before the selected symbol (on the left)
- F3..... a contact after the selected symbol (on the right)
- F4..... a contact in parallel with the selected symbol
- F6..... a block before the selected symbol (on the left)
- F7..... a block after the selected symbol (on the right)
- F8..... a block in parallel with the selected symbol

The following commands are valid when the selection is on the rung output (coil):
 F5..... add a coil in parallel with the selected one
 F9..... add a "Jump" symbol in parallel with the selected one
 Shift + F9 add a "Return" symbol in parallel with the selected one

For parallel insertion (F4/F8), if several contacts of a rung are selected together, the symbol is inserted in parallel with the group of selected elements. Below is an example:



To insert symbols in the diagram, you can also use the commands of the **"Insert"** menu. With the mouse, you can click on the LD toolbar, on the type of symbol you want to insert:



Entering symbols

To associate a variable symbol to a contact or a coil, select it and hit ENTER. With the mouse, double click on the contact or coil. A variable selection box appears. Refer to chapter "More about program editors" in this document for further information about how to use this box. To associate a function, function block or operator to a block, hit ENTER when the selection is on the inside its rectangle. To associate a variable symbol to an input or output block parameter the selection must be on the corresponding location, **outside** the rectangle of the block.

Dialog boxes including variable or block selection lists are normally used for text input. If the **"Manual keyboard input"** mode is checked in the **"Options"** menu, variable symbols and block names are entered directly in a single text edit box. Enter new text and hit **"Enter"** key to validate it, or hit **"Escape"** key to exit modification and close the text editing box. The text edit box used in "manual keyboard input" mode cannot be closed with the mouse.



Changing the type of contacts and coils

The **"Edit / Change coil/contact type"** changes the type of the selected contact or coil. A contact may be direct, negated, with positive or negative edge detection. A coil may be direct, negated, set or reset, with positive or negative edge detection. Hitting the SPACE bar has the same effect.



Inserting a rung in a diagram

The **"Edit / Insert rung"** command insert a new rung in the diagram, before the selected one. The rung is initiated with one contact and one coil.

A.6.3 Working on an existing diagram

The commands of the "Edit" menu are used to change or complete an existing diagram. Most of these commands act on the elements currently selected in the diagram.

▣ **Correcting a diagram**

The DEL key can be used to remove the selected elements. It is not possible to remove a coil, a jump or return symbol when it is the only output of a rung. Use "Edit / Undo" command to restore elements after a DEL command. The DEL command can also be applied to a group of elements selected in the diagram. The DEL command can be used when selection is on the rung comment text to reset it. The DEL command, used when the selection is on the rung head, removes the entire rung.

▣ **Copying symbols**

The "Cut", "Copy", "Paste" commands of the "Edit" menu are used to move or copy selected elements. These commands do not act on rung comments. The "Edit / Paste special" command gives you the choice to insert the pasted elements:

- before the selected element (on the left)
- after the selected element (on the right)
- in parallel with the selected element

▣ **Managing entire rungs**

All editing commands (delete, copy, cut...) act on the entire rung if the selection is on the rung header (left power rail). This provides an easy way to arrange rungs in the diagram, just by moving the selection in the first column. It is also possible to extend the selection vertically so that it includes several rung headers. In this case edition commands may be applied to a list of entire rungs.

▣ **Find and replace**

The "Edit / Find" and "Edit / Replace" menu commands are used to find and replace texts in the diagram. Only complete names can be found. Search acts on contacts, coils, block names, block parameters and run labels. It cannot be used to find a string in a rung comment. The Replace command cannot be used to change the type of a block. The research can be made upward or downward, starting at position of the current selection. It "loops" when the limits of the diagram are reached. The following shortcuts are also available for quick research of variable names:

ALT+F2 finds the next element with the same variable name as the element currently selected. This feature can also be applied to function blocks and rung labels.

ALT+F5 finds the next coil with the same variable name as the element currently selected. This feature is mainly used in debug mode, to quickly find out the rungs which forces a suspicious variable.

A.6.4 Display options

The commands of the "**Options**" menu are used to customise the drawing of the LD diagram on the screen, and to hide or display some types of information.

▣ **Rung comments**

Use the "**Options / Rung comments**" command to hide or display the rung comments in the whole diagram. Hiding the rung comments can be required to have a more condensed view on a huge diagram, as each comment consumes one row in the editing matrix. This option does not affect the contents of the existing rung comments, and can be swapped at any time.

▣ **Names and aliases**

Each variable, when associated to a contact, a coil or a block I/O parameter is identified by its symbolic name. The ISaGRAF Quick LD editor also introduces the notion of "**alias**" for each variable. The alias of the variable is the variable comment text, truncated before the first ':' character, and limited to 16 characters. Below are examples:

```
variable comment:           alias:
short text                  short text
long text with no separator long text with n
short text: long description short text
```

Aliases have no effect on the execution of the LD diagram and should be considered as comments for the syntactic point of view. A variable alias is automatically extracted from the variable comment when the name is selected in the variable list. It cannot be changed manually. Use the "**Options / Contacts and coils**" commands to select a display mode for variable identification. The following modes are available:

- display only the variable names
- display only the variable aliases
- display both names and aliases

Quick LD editor does not automatically updates LD documents when variable aliases are changed in the dictionary. Use the "**Options / Contacts and coils / Update aliases**" command to update all aliases in edited diagram. You can also set the "**Always update on Open**" option from "**Options / Contacts and coils**" to ask ISaGRAF to automatically update all used aliases each time a Quick LD program is open. Warning: Setting this option may significantly increase the time spent to open a program.

▣ **Drawing options**

The "**Options / Layout**" command opens a dialog box where are grouped all the parameters and options concerning the editor workspace and the drawing of the graphic LD diagram.

Use the check boxes in the "Workspace" group to display or hide editor tool bar, status bar and LD toolbar. Options of the "Document" group allow you to show or hide points of the editing grid, and to enable/disable the use of colours for the drawing.



Options of the "Zoom" group allow you to select a main zoom ratio. You can also use the "zoom" button in the editor toolbar to swap between default zoom ratios.



You can also customise the X/Y aspect ratio of cells in the editing grid. This last option can be used to reduce the default cell width, if you commonly use short names for variables. You can also use the "width" button in the editor toolbar to change the X/Y aspect ratio without entering the Layout dialog box.

Use the "**Options / Font**" command to select the name of the character font to be used in all ISaGRAF graphic documents. When selecting font, font style and size are not relevant and do not need to be specified. ISaGRAF graphic editors always calculate the font size according to selected zoom ratio.

A.7 Using the FBD/LD editor

The ISaGRAF FBD/LD graphic editor allows the user to enter complete FBD programs, which may include parts in LD. It combines graphic and text-editing capabilities, so both diagrams and corresponding inputs and outputs can be entered. As this editor is more dedicated to FBD language, pure LD diagrams should rather be entered using the ISaGRAF Quick LQ editor.

A.7.1 Basics of the FBD/LD languages

The **FBD** language is a graphic representation of many different types of equations. **Operators** are represented by rectangular function boxes. Function inputs are connected to the left side of the box. Function outputs are connected to the right side. Diagram inputs and outputs (**variables**) are connected to the function boxes with **logical links**. An output of a function box may be connected to the input of another box.

The **LD** language enables graphic representation of boolean expressions. Boolean **AND**, **OR**, **NOT** operators are explicitly represented by the diagram topology. Boolean input variables are attached to graphic **contacts**. Boolean output variables are attached to graphic **coils**. Contacts and coils are connected together and to left and right power rails by **horizontal lines**. Each line segment has a boolean state of **FALSE** or **TRUE**. The boolean state is the same for all the segments directly linked together. Any horizontal line connected to the left **vertical power rail** has the **TRUE** state.

LD and FBD diagrams are always interpreted from the left to the right, and from the top to the bottom. Refer to the ISaGRAF Language reference Manual for more details about LD and FBD languages. These are the basic graphic components of the LD and FBD languages, such as supported by the FBD/LD editor:



Left power rail

Rungs must be connected on the left to a **left power rail**, which represents the initial "TRUE" state. ISaGRAF FBD editor also allows connecting any boolean symbol to a left power rail.



Right power rail

Coils may be connected on the right to a **right power rail**. This is an optional feature when using the ISaGRAF FBD/LD editor. If a coil is not connected on the right, it includes a right power rail in its own drawing.







LD vertical "OR" connection

LD vertical connection accepts several connections on the left and several connections on the right. Each connection on the right is equal to the OR combination of the connections on the left.



Contacts





A contact modifies the boolean data flow, according to the state of a boolean variable. The name of the variable is displayed upon the contact symbol. The following types of contacts are supported by ISaGRAF FBD/LD editor:

-  direct contact
-  negated contact
-  contact with positive (rising) edge detection
-  contact with negative (falling) edge detection



Coils

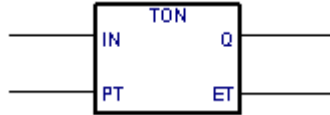
A coil represents an action. It must be connected on the left to a boolean symbol such as a contact. The name of the variable is displayed upon the coil symbol. The following types of coils are supported by ISaGRAF FBD/LD editor:

-  direct coil
-  negated coil
-  "set" action coil
-  "reset" action coil



Function blocks

A block in an FBD diagram can represent a function, a function block, a sub-program or an operator. Inputs and outputs must be connected to variables, contacts or coils, or other block inputs or outputs. Formal parameter names are displayed inside of the block rectangle.



Labels

Labels can be placed everywhere in the diagram. Labels are used as targets for jump instructions, to change the execution order in the diagram. Labels are not connected to other elements. It is highly recommended to place labels on the left of the diagram, in order to increase the diagram readability.



Jumps

A jump symbol always refers to a label placed elsewhere in the diagram. Its left connection must be linked to a boolean point. When the left connection is TRUE, the execution of the diagram directly jumps to this target label. Note that backward jumps are dangerous as they may lead to a blocking of the PLC loop in some cases.



Return symbol

A return symbol is connected to a boolean point. It indicates that the execution of the program must be stopped if the rung state is TRUE.



Variables

Variables in the diagram are represented inside small rectangles, connected on the left or on the right to other elements of the diagram.



Connection links

Connection links are drawn between elements put in the diagram. Links are always drawn from an output to an input point (in the direction of the data flow).



Connection links with boolean negation

Some boolean links are represented with a small circle on their right extremity. This represent a boolean negation of the information transported by the link.



User defined corners

User defined points may be defined on links. They allow the user to manually control the routing of a link. If no corner is placed, the ISaGRAF FBD/LD editor uses a default routing algorithm.

A.7.2 Entering an FBD diagram

To enter a diagram, you have to place elements (blocks, variables, contacts, coils...) in the graphic area, and draw links between them.



Inserting objects

To insert an object in the diagram, select the corresponding button in the toolbar and click in the graphic area, where you want to insert it.



Selecting objects

Selecting graphic objects is needed for most of the editing commands. The ISaGRAF LD/FBD graphic editor enables the selection of one or more existing objects in the diagram area. To select objects, the **"select"** (button with an arrow) choice must be checked in the editor toolbar. To select one object, the user only has to click on its symbol. To select a list of objects, drag the mouse in the diagram and select a rectangle area. All the graphic objects that intersect the selection rectangle are marked as **"selected"**. A selected object is drawn with little black squares around its graphic symbol. By making a new selection, all previously selected objects are unselected. To remove the existing selection, simply click with the mouse on an empty area, outside of the rectangle which borders the selected objects.



Inserting comments

Comments may be inserted anywhere in the diagram. Comments have no influence on the program execution. They allow a higher readability of the diagram. To insert a comment block, select this button in the toolbar, and drag the mouse to select the rectangle area where comment must be drawn. Then enter the text of the comment. No special leading or trailing characters such as **"(*"** and **"*)"** are needed when entering the text of a comment block. A comment block may be resized by dragging the corners of its border when it is selected.



Moving objects

To move objects in the diagram, you have to select them, and drag the mouse to move the selected area in the diagram. To move connected objects, the user simply has to move the graphic symbols put on the diagram. The ISaGRAF LD/FBD editor will automatically redraw the connection lines between the objects that were moved, based on their new location.



Drawing links

Select one of these buttons in the toolbar to draw a link between connection points of existing elements. If you draw a link from a connection point to an empty location in the diagram, a user-defined corner automatically terminates it, so that you can continue drawing another segment.



Changing link drawing

The "**Tools / Move line**" command is used when a link is selected in the diagram to change its automatic routing. This command has no effect when the link is connected to a user-defined corner. When a link is drawn as three segments, this command changes the position of the second segment. Below are examples:



Changing the type of a link

You can easily change the type of link (with or without Boolean negation) by double clicking with the mouse on its right extremity.



Drawing LD rungs

To draw a new LD rung, first insert the left power rail. Then place a coil: it will be automatically linked to the power rail. Other contacts and vertical OR connections may be directly inserted on the rung line, without drawing any new connection link.



When a new LD contact or coil is inserted in an empty space of the editing area, the new horizontal rung line is automatically drawn from the new inserted element to the existing power rails on the left and on the right. This line is not automatically drawn if the new contact or coil is not placed between power rails. The new inserted contact or coil can then be freely moved on the drawn rung. The horizontal lines created by the editor while inserting an LD contact or coil symbol can be selected and deleted. You can insert a new LD contact or coil symbol on the horizontal line of an existing rung. The editor automatically cuts the rungs and connects it to the left and right connection points of the new inserted contact or coil.



Multiple connections

A multiple connection can be created on the right of any **output** point. It means that the information is **broadcasted** to several other points in the diagram. The same state is propagated on each extremity on the right. The number of lines drawn at the right of an output connection point is not limited. Two connection lines cannot have

their right extremity connected on the same **input** point, except for the following LD symbols:

-  right power rail
-  multiple connection on the left (OR) operator

These LD symbols can have an unlimited number of inputs.

A.7.3 Working on an existing diagram

The commands of the **"Edit"** menu are used to change or complete an existing diagram. Most of these commands act on the elements currently selected in the diagram.

Correcting a diagram

The DEL key can be used to remove the selected elements. Pending links are deleted with selected elements. Use **"Edit / Undo"** command to restore elements after a DEL command. The DEL command can also be applied to a group of elements selected in the diagram. The **"Cut"**, **"Copy"**, **"Paste"** commands of the **"Edit"** menu are used to move or copy selected elements.

Find and replace

The **"Edit / Find"** and **"Edit / Replace"** menu commands are used to find and replace texts in the diagram. Only complete names can be found. Research acts on contacts, coils, block names, variables and labels. It cannot be used to find a string in a comment text. The Replace command cannot be used to change the name of a block. The research can be made upward or downward, starting at the current selection position. It "loops" when the limits of the diagram are reached.

Displaying the execution order

When an FBD diagram includes backward loops, the execution order cannot follow the single left to right / top to bottom method. In order to avoid confusion, use the **"Tools / Show execution order"** command or press **Control + F1** keys to display the execution order that will be used at compiling time. Tags numbered from 1 to N are displayed close to symbols that lead to an action (coils, set variables and function blocks).



Entering symbols and texts

Double click with the mouse on an element to enter the associated symbol or text. This applies to variables, contacts and coils, comment texts and labels. When used on a contact or coil, this also allows to change its type (direct, negated...).

Dialog boxes including variable or block selection lists are normally used for text input. If the **"Manual keyboard input"** mode is checked in the **"Options"** menu, variable symbols and block names are entered directly in a single text edit box. Enter new text and hit **"Enter"** key to validate it, or hit **"Escape"** key to exit modification and close the text editing box. The text edit box used in "manual keyboard input" mode cannot be closed with the mouse.

If the **"Auto input"** mode is checked in the **"Options"** menu, the variable symbol must be entered immediately each time a new contact or coil is inserted. The symbol must always be entered immediately when a variable or label is inserted.



Selecting function block type

Double click with the mouse on a block is used to change its type. The block type is selected from the list of available operators, functions and function blocks. This command also allows changing the number of input points in the case of a commutative operator. (e.g. AND, OR, ADD, MUL...)



Getting free space

When you press the right button of the mouse in the FBD drawing area, a popup menu is displayed. It contains the following commands that can be used to insert or remove free space at the location of the mouse cursor:

Insert rows:..... This command inserts horizontal free space, made of 4 rows according to the grid step, starting at the position of the mouse cursor where popup menu is open.

Delete rows:..... This command removes unused horizontal space (rows) starting at the position of the mouse cursor where popup menu is open. This command cannot be used to remove FBD elements.

When popup menu is open, a grey line in the FBD drawing area indicates where empty space will be inserted or removed.

A.7.4 Display options

The commands of the **"Options"** menu are used to customise the drawing of the FBD diagram on the screen.



Layout customisation

The **"Options / Layout"** command opens a dialog box where are grouped all the parameters and options concerning the editor workspace and the drawing of the graphic diagram. Use the check boxes in the **"Workspace"** group to display or hide editor toolbars and status bar. Option of the **"Document"** group allows you to show or hide points of the editing grid.



Options of the **"Zoom"** group allow you to select a main zoom ratio. You can also use the **"zoom"** button in the editor toolbar to swap between default zoom ratios.

Use the **"Options / Font"** command to select the name of the character font to be used in all ISaGRAF graphic documents. When selecting font, font style and size are not relevant and do not need to be specified. ISaGRAF graphic editors always calculate the font size according to selected zoom ratio.

A.7.5 Styles and modification tracking

The ISaGRAF LD/FBD editor enables you to assign a graphic style to any component of an LD/FBD diagram. A style is mainly defined as a special diagram colouring. But ISaGRAF also uses styles to enable modification tracking in diagrams for version control purpose.

Note that styles are not visible during simulation or on-line debug, as colours (red and blue) are used in that mode to highlight TRUE / FALSE states of spied variables.

▣ **Predefined styles**

The following styles are pre-defined:

- Normal**..... Default drawing (black). For modification tracking, "normal" style indicates that elements having that style are part of the original diagram. "Normal" style elements are normally scanned during execution.
- Modified** Elements marked as "modified" are painted in pink. For modification tracking, the "modified" style is used to highlight elements that have been added or changed after the original release of the diagram. "Modified" style elements are normally scanned during execution.
- Deleted** Elements marked as "deleted" are painted in grey, with dashed lines. Such elements are not taken into account for the execution of the diagram. This style is used to keep a track of elements removed after the original release when version control is required.
- Custom**..... In addition to predefined style, ISaGRAF LD/FBD editor allows you to select any colour to be applied to a part of the diagram. Such elements are considered as having a "Custom" style. The use of "Custom" style has no effect on the diagram execution at run time.

Use the commands of "**Style**" sub-menu in "**Edit**" menu to manually apply a style to selected elements.

▣ **Modification tracking**

The use of styles and the availability of the "Deleted" style allow automatic modification tracking in an existing diagram. Use the "**Mark modifications**" command in "**Edit/Style**" menu to enable or disable modification tracking.

When the "Mark modifications" option is set, all elements changed in or added to the diagram are automatically set with "Modified" style. When an element is deleted, using "Delete" or "Cut" commands, they are not visually removed from the diagram, but simply marked with "Deleted" style". This enables the user to automatically keep a trace of all modifications entered in the diagram.

Use the "**Edit/Style/Remove all deleted items**" to actually remove all elements marked with "Deleted" style from the LD/FBD diagram. This command does not take care of the current selection, and always applies to the entire diagram.

To "restore" one element marked with the "**Deleted**" style, select the desired element and apply to it the "**Normal**" style, the "**Modified**" style or any "**Custom**" style. Such operation may lead to invalid connections (more than one link connected to the same input point) that will be detected during next program verification.

A.8 Using the text editor

This chapter only describes features and commands of the ISaGRAF text editor, particularly when used to enter the source code of ST and IL programs.

A.8.1 Editing commands

The commands of the "Edit" menu are used to work on the edited text. Most of these commands act on the characters currently selected in the diagram, or perform an action at the current location of the caret.



Cut and paste

The DEL key can be used to remove the selected text. Use "Edit / Undo" command to restore elements after a DEL command. The "Cut", "Copy", "Paste" commands of the "Edit" menu are used to move or copy text in the program, or to insert pieces of texts copied in the clipboard by other applications.



Find and replace

The "Edit / Find" and "Edit / Replace" menu commands are used to find and replace texts in the program. Any character string can be found. Research can be performed upward or backward, starting at the current location of the caret. It does not "loop" when the limits of the program are reached.



Go to line

The "Edit / Go to line" command is used to move the caret to a specific line number. This can be very useful to have access to a line with an error detected by the ISaGRAF compiler in an ST or IL program, and referenced by a line number.



Insert symbol from dictionary

Use the "Edit / Insert variable" command to insert at the caret position the symbol of a variable or object declared in the project dictionary. Symbol is selected through the common variable selection box described in chapter "More about program editors" in this document.



Insert file

The "Edit / Insert file" command inserts the whole contents of a file at the current location of the caret. Note that only pure ASCII text files can be handled by this command.

A.8.2 Options

The commands of the "Options" menu are used to display or hide editor toolbars, and select the character font. The selected character font will be used for any text editing in all ISaGRAF Workbench.

When used to enter the source code of an ST / IL program, the "**Options / Show keywords**" command is used to show or hide a toolbox that groups the most common keywords of ST or IL language. Click on a button in the toolbar to insert the corresponding keyword or operator at the current location of the caret.

A.9 More about program editors

This chapter contains useful information about editing features which are common to all the ISaGRAF program editors. This mainly concerns links with other ISaGRAF tools and common ISaGRAF dialog boxes.

A.9.1 Calling other ISaGRAF tools



Verify (compile) the program

The "**File / Verify**" command runs the ISaGRAF code generator to verify the programming syntax of the currently edited program. In case of SFC language, both level 1 and 2 are checked. When syntax verification is complete, the code generator window must be closed to continue work on the program. If there is only one program in the application (the edited one) the application code is generated if no syntax error is detected. The "**Options / Compiling options**" command is used to set compiling and optimising parameters. Refer to chapter "Using the code generator" in this document for further information about compiling and code generation.



Simulate or debug the application

The "**File / Simulate**" and "**File / Debug**" commands run the ISaGRAF graphic debugger either in simulation or real connected mode, and re-opens the edited SFC program in debug mode. Used in debug mode, no modification can be entered in the program.



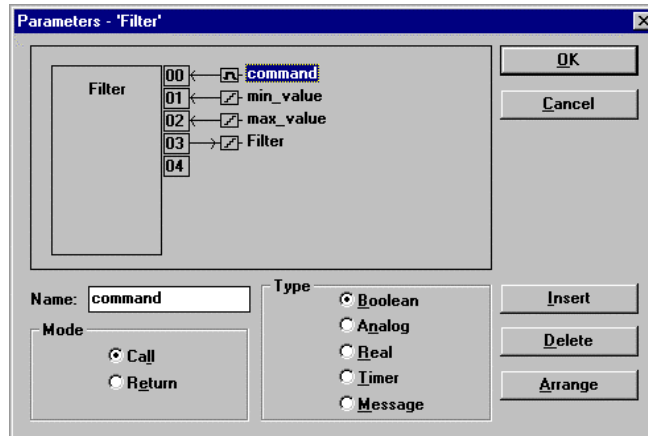
Editing the dictionary of variables

The "**File / Dictionary**" command is used to edit the dictionary of variables for the current application and the current program. It also contains the entry points to edit the user-defined words. The **local** declarations or defined words relate to the currently edited program.

A.9.2 Parameters of the program

When the edited program is a function, a function block or a sub-program, the "**File / Parameters**" command is used to define its calling and return parameters. This command has no effect if the edited program is an SFC or top level program from section **Begin** or **End**.

Sub-programs, functions or function blocks may have up to **32** parameters (input or output). A function or sub-program always has one (and only one) return parameter, which must have the same name as the function, in order to conform to ST language writing conventions. The following dialog box is used to describe the parameters of the sub program:



The list in the upper left side of the window shows the parameters, in the order of the calling model: first the calling parameters, last the return parameters. The lower part of the window shows the detailed description of the parameter currently selected in the list. Any of the ISaGRAF data types may be used for a parameter. The return parameters must be located after calling parameters in the list. Naming parameters must conform to the following rules:

- the length of the name cannot exceed 16 characters
- the first character must be a letter
- the following characters must be letters, digits or underscore character
- naming is case insensitive

The "Insert" command is used to insert a new parameter before the selected parameter. The "Delete" command is used to erase the selected parameter. The "Arrange" command automatically rearranges (sorts) the parameters, so that the return parameters are put at the end of the list.

A.9.3 Other commands of the "File" menu

The following commands are available in the "File" menu of all program editors:



Open another program

The "File / Open" command allows the user to close the currently edited program and start editing another program of the current project with the same language. This function cannot be used to edit a program written in another language. The new selected program replaces the current one in the editing window.



Printing the program

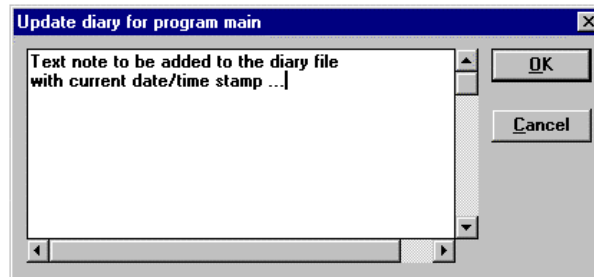
The "File / Print" command outputs the edited program on printer. This command automatically runs the ISaGRAF document generator to print out the edited program and attached local variables.

For some graphic programs (SFC, FBD and Quick LD) You can also use the **"Edit / Copy drawing"** command to copy in the clipboard the drawing of the chart in metafile format, so that it can be pasted in other applications such as word processors. For SFC programs, only the level 1 information (chart, numbering and level 1 comments) appears on the copied metafile.

A.9.4 Updating the program diary

The diary file attached to the edited program may be manually entered using the **"File / Diary"** command. The diary file is automatically updated with syntax checking output messages each time the program is compiled. Compiling outputs are completed with the compiling date / time stamp.

- ☰ If the **"Update diary"** mode is selected in the **"Options"** menu of program editors, the following dialog box is opened each time the program is saved on disk.

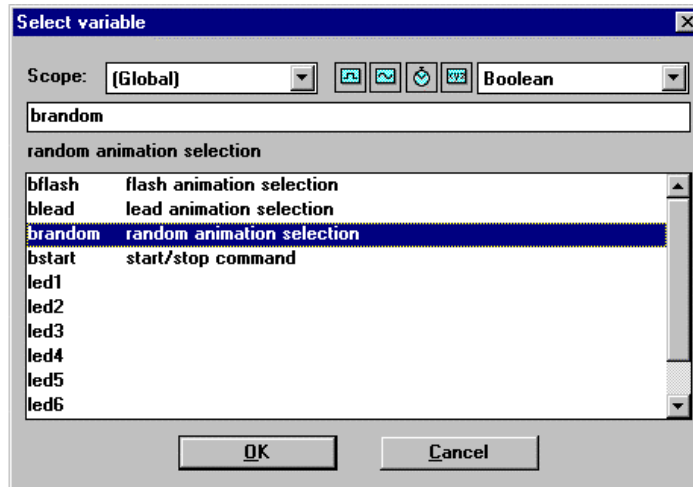


If OK button is pressed, the entered text note is then stored at the end of the diary file with current date / time stamp. This feature is very useful for maintenance of complete programs, as it provides useful help about the program life cycle.





A.9.5 Selecting a variable from dictionary



When editing a text program (ST or IL) the **"Edit / Insert variable"** allows the selection of a declared variable name to be inserted at the current position of the caret. When editing LD or FBD programs, variable selection is required for the description of contacts, coils, block I/O parameters or FBD variable boxes. In both cases, the following dialog box is open to select a declared variable:



The "**Scope**" selection box is used to select between global and local variables. The selection box on the right allows the selection of the data type. Small icons beside the type selection box are buttons that can be used as shortcuts to select most current data types:

-  Boolean
-  Integer / Real
-  Timer
-  Message

To select a variable, click on its name in the list. Its name and comment are then displayed on the top of the list. Then press the "**OK**" button to confirm its selection. It is also possible to directly enter a variable name in the edit control without using the list.

A.9.6 The output window

The following commands are available in the Tools menu of all language editors. They are used to display information in a small text list at the bottom of the editing window, and use it for program browsing.

- | | |
|-------------------------------|--|
| "Show compiler output" | display in the output window the error messages from the last compiling of the edited program. |
| "Find in..." | find occurrences of a text in the whole edited program, and list them in the output window. For SFC and FC languages, this command searches in all level 2 programs. |

"Hide output window" close the output list window

When error messages or occurrences are displayed in the output window, double click on a line to directly move selection to the corresponding location. For SFC and FC languages, this command opens corresponding level 2 programming window.



A.10 Using the dictionary editor

The ISaGRAF dictionary is an editing tool for the declaration of the internal variables, I/O variables, function block instances, and "defined words" of the application. The dictionary groups together the declared variables and function block instances of the application, and the words defined as constant strings.





Variables, function blocks and defined words must be declared in the dictionary before using them in source code. Variables and defined words can be used with any of the automation languages: SFC, FBD, LD, ST and IL. Function blocks used in FBD language do not have to be declared, because the ISaGRAF FBD and Quick LD editors automatically declare the instances of the used blocks.

Variables

The variables are sorted according to their **range** and **type**. Only variables of the same type and the same range can be entered on the same input grid. These are basic ranges for variables:

-  **GLOBAL** can be used by any program of the current project
-  **LOCAL** can be used by only one program

These are basic types of variables:

-  **BOOLEAN** ... true/false binary values
-  **ANALOG** real or integer values
-  **TIMER** time values
-  **MESSAGE** ... character strings




A variable is identified by a name, a comment, an attribute, a network address and other specific fields. Here are the basic variable attributes:

- INTERNAL** memory variable
- INPUT** variable linked to an input device
- OUTPUT** variable linked to an output device
- CONSTANT** read only internal variable (with initial value)

Note: **Timers** are always **internal** variables. **Input** and **Output** variables always have the **GLOBAL** range.

Defined words

A defined word is an alias that can be used in any language to replace a text string. The replaced text can be a variable name, a constant expression or a complex expression. Defined words are sorted according to their range. Only defined words of the same type and the same range can be entered on the same input grid. Here are basic ranges:

-  **COMMON** can be used by any program of any project
-  **GLOBAL** can be used by any program of the current project
-  **LOCAL** can be used by only one program

A defined word is identified by a name, a well-defined block of ST text equivalence and a free comment.



Function block instances

The instances of the function blocks used in the ST and IL languages must be declared in the dictionary. Because a function block has internal "hidden" data, each copy of a function block must be identified. The following example shows the function block "R_TRIG" (rising edge detection) defined in the library, used for edge detection on different variables. Each copy of the block must be identified by a unique name. Naming the type of block and definition of its parameters is made by using the library manager:

Block name: R_TRIG
Parameters: Input=CLK
Output=Q

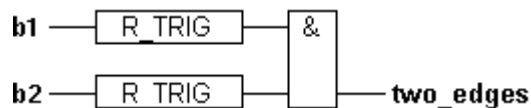
Naming the instances is made by using the dictionary editor:

Instance name: TRIG_B1 **Block name:** R_TRIG
Instance name: TRIG_B2 **Block name:** R_TRIG

The declared instances may be used in ST programs:

```
TRIG_B1 (b1);
edge_b1 := TRIG_B1.Q;  (* b1 variable edge detection *)
TRIG_B2 (b2);
edge_b2 := TRIG_B2.Q;  (* b2 variable edge detection *)
```

Declared function block instances may be **GLOBAL** (known by any program in the project), or **LOCAL** to one program. Function blocks used in FBD or LD languages do not have to be declared, because the ISaGRAF FBD editor automatically declares the instances of the used blocks.



(* the function blocks always have the name of the block defined in the library. The ISaGRAF FBD and Quick LD editors automatically declare an instance each time a block is inserted in the diagram *)

Function block instances automatically declared by the FBD and Quick LD editors are always **LOCAL** to the edited program.

Network addresses

Network addresses are **optional**. A variable with a non-zero network address can be **spied** by an external system (for example a process visualisation system) at run time. More generally, the network address provides an identifying mechanism for each run time communication system that cannot handle symbolic names. A network address may be entered for each variable, during its complete description, when the variable is created or modified.

A.10.1 The dictionary main window

The dictionary editing window shows a list of variables with same type and range. The type and range of edited variables is always displayed in the title bar.



The editing window shows only main fields of variable description: name, attribute and network address, and text comment. The full description of the selected variable is always displayed in the status bar. Use the following buttons in the toolbar to select the range of variable to be edited:



- COMMON**can be used by any program of any project
- GLOBAL**can be used by any program of the current project
- LOCAL**can be used by only one program

Use the "Tab" control displayed with title bar to select the type of object to be edited:



Use the text-input field on the left of the toolbar to search for a variable prefix name. In this case, research is processed on the entire list, from the beginning, based on the current selection. The **"Edit / Find"** command is also available to search a text string in variable names and comments, and to move the selection to this variable. Search is always **case insensitive**.

A.10.2 Managing variables

The available **"Files"** menu commands work on the entire selected class of variables, function block instances or defined words. Use the **"Other"** command to select the type and range of objects to be edited.



Printing variables

Use the **"Files / Print"** command to print the currently edited list of variables or defined words, on a standard Windows™ printer device. Printing is made using the ISaGRAF document generator. The printout includes the complete description of each variable or defined word of the currently edited type.



Creating new variables

The "**Edit / New**" command allows the user to create new variables, function block instances or defined words for the selected range and type. New variables are inserted just before the variable currently pointed to by the selection bar. When this command is run, an input box is opened to enter the variable description. When the description is complete, pressing the "**Store**" button puts it onto the list. The input box is automatically re-opened, so the user can enter other variables with the same "**Edit**" command. Pressing the "**Cancel**" button of the dialog box breaks the variable creation process.



Modifying existing variables

The "**Edit**" command of the "**Edit**" menu allows the user to modify the description of the variable currently pointed at by the selection bar. When this command is run, an input box is opened to modify the variable description. When description is complete, pressing the "**Store**" button enables modification. The user also can press "**Next**" and "**Previous**" buttons to extend the modification command to adjoining variables. Pressing the "**Cancel**" button closes the dialog box without storing any modification.



Cut and paste

The ISaGRAF dictionary editing tool enables **multiple-line selection**. Many commands are available to work on the currently edited list of variables. Below are available "**Edit**" menu commands:

COPY..... Copy the selected group of variables to the dictionary clipboard
CUT..... Copy the selected group of variables and remove it from the edited list
CLEAR..... Remove the selected group of variables from the edited list
PASTE Insert the dictionary clipboard before the selected variable

Copy/Cut/Paste functions can be used from one list of variables to another. They cannot be used between list of different object types.



Sorting variables

The "**Tools / Sort**" command sorts the variables or defined words of the currently edited list. The sorting order is given by the attributes of the variables:

- first the internal variables
- then the input variables
- finally the output variables

Variables with the same attribute are sorted into alphabetical order. Defined words are always sorted into alphabetical order.



Setting network addresses

Network addresses are **optional**. A variable with a non-zero network address can be **spied** by an external system (for example a process visualisation system) at run time. A network address may be entered for each variable, during its complete description, when the variable is created or modified. The "**Tools / Renumber addresses**" command allows the user to set up network addresses of an entire

group of variables. When this command is run, it acts on the group of variables currently selected on the list. Entering a hexadecimal **basis address** (address for the first variable of the group) results in network addresses of the variables of the group being set with **consecutive addresses**. Entering a null basis address resets to zero the network address of all the selected variables.



Importing boolean "true/false" strings

When editing defined words, the "**Tools / Import true/false definitions**" allows the user to automatically define as language keywords the strings attached to boolean variables to represent TRUE and FALSE states. Such strings are normally defined for debug formatting. They have to be specified as defined words if they are to be used in programs. This command searches for boolean true/false strings in the declarations with the same range as the one currently selected for the editing of the defined words.

A.10.3 Description of objects

A complete description must be entered for each variable, function block instance, or defined word. Description fields are different for each type of object. The following fields are common for any type of variables:

- Name**Name of the variable: first character must be a letter, following characters must be letters, digits or '_'.
- Network address**Hexadecimal network address (optional). When this field is non-zero, the variable can be spied by external systems at run time.
- Comment**Free comment for variable description.
- Retain**This option indicates that the variable must be saved on backup memory.



These are other description fields for a **boolean** variable:

- Attribute**Specifies an internal, constant, input or output variable.
- "False" string**.....String used for false value at debug time.
- "True" string**String used for true value at debug time.
- Set to true at init**.....The initial value is TRUE if this option is checked, otherwise the initial value is FALSE.



These are other description fields for an **integer or real** variable:

- Attribute**Specifies an internal, constant, input or output variable.
- Format**.....Specifies an integer or real (floating) variable. Display format used during debug can be selected.
- Unit string**String used to identify the physical unit at debug time.
- Conversion**Name of the conversion table or conversion function attached to the variable (for input or output variables only)
- Initial value**Initial value of the variable (must have the same format as the variable). If not specified, the initial value is 0.



These are other description fields for a **timer** variable:

AttributeSpecifies an internal or constant variable.

Initial valueInitial value of the variable (time value). If not specified, the initial value is time#0s.



These are other description fields for a **message** variable:

AttributeSpecifies an internal, constant, input or output variable.

Maximum Length.....Specifies the maximum number of characters that can be stored in the message.

Initial valueInitial value of the variable (length cannot exceed the capacity of the message). If not specified, the initial value is the empty string.



These are the description fields for a **defined word**:

NameName used in ST source files: first character must be a letter, following characters must be letters, digits or '_'.

DefineString according to ST syntax that replaces the defined word during compiling. Example: Name = PI - Equivalence = 3.14159

CommentFree comment for defined equivalence



These are the description fields for a **function block instance**:

NameName of the instance, used in ST source files: first character must be a letter, following characters must be letters, digits or '_'.

Type.....Name of the corresponding function block in the library.

CommentFree comment for the function block instance.

A.10.4 Quick declaration

The "**Tools / Quick declaration**" command enables you to declare several variables at the same time. Variables created by quick declaration are named using a numbering convention. For that, you have to define:

- the index (number) of the first and the last variables,
- the text to be added before and after the number in variable symbols
- the number of digits used to express the number in variable symbols.

Additionally, you can specify basic attributes of created variables (internal, input or output...), plus some properties depending on the variable type ("Retain" attribute, integer or real format, message string maximum length).

You always need to define a text to be inserted before variable number, as a variable symbol cannot start with a digit. When the "number of digits" is set to "Auto", ISaGRAF formats the variable number on the minimum needed number of digits. When number of digits is specified, ISaGRAF formats all numbers to the specified length by adding leading '0' characters. Setting a fixed number of digits for

variable numbers can be very useful to prevent bad lexicographic sorting. Below are some examples.

Example: This setting for quick declaration:

Numbering:	
From: <input type="text" value="9"/>	To: <input type="text" value="11"/>
Digits: <input type="text" value="auto"/>	
Symbol:	
Name: <input type="text" value="Var"/> ## <input type="text" value="xx"/>	

will create the three following variables:

Var9xx Var10xx Var11xx

Example: This setting for quick declaration:

Numbering:	
From: <input type="text" value="1"/>	To: <input type="text" value="100"/>
Digits: <input type="text" value="3"/>	
Symbol:	
Name: <input type="text" value="MyVar"/> ## <input type="text"/>	

will create 100 variables with names from **MyVar001** to **MyVar100**

A.10.5 Modbus SCADA addressing map

ISaGRAF "network addresses" are often used to establish a link between ISaGRAF system and a SCADA based on Modbus communication. In that case, the SCADA is a Modbus master and ISaGRAF target acts as a Modbus slave. Network addresses are used to create a virtual Modbus map for all ISaGRAF variables that must be controlled from the SCADA. The "**Tools / Modbus SCADA addressing map**" is a powerful tool to quickly create a Modbus virtual map with variables of the application.

The mapping tool shows two lists. The upper one is a segment (4096 locations) of the Modbus map, showing mapped variables (the ones having a network address). The lower list shows unmapped variables (without network address defined). The "0" address cannot be used to map a variable.

Use the "**Map**" and "**Remove**" commands of the "**Edit**" menu to move a variable from one list to another, and thus build the map. Same actions can be performed by double clicking on a variable symbol in a list, to send it to the other list. At any

moment, you can use the "Segment" drop down list to view another segment of the map.

The commands of the "**Options**" menu can be used at any moment to display addresses either in decimal or in hexadecimal.

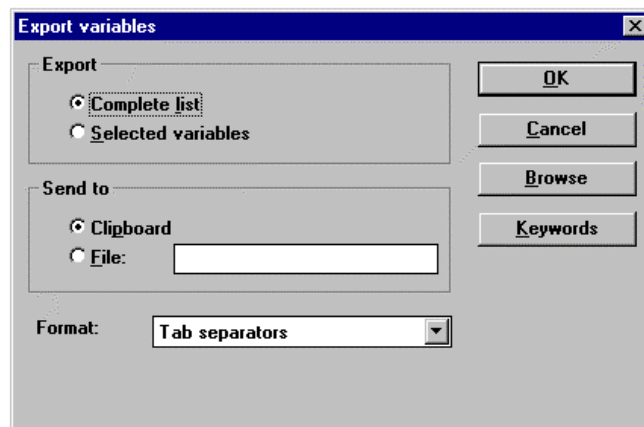
The "**Edit / Find**" commands is used to search for a declared variable, whether it is already mapped or not.

A.10.6 Exchanging information with other applications

The ISaGRAF dictionary editing tool offers import/export functions in order to exchange information with other applications, such as word processors, spreadsheets, data base managers... These commands are grouped in the "**Edit**" menu. The "**Export text**" command builds a pure ASCII text description of the fields describing a set of edited objects, and stores this text either in the Windows clipboard or in a file. Such information is typically used by another application. The "**Import text**" command imports variable declaration description fields, described in pure ASCII text format, stored either in the Windows clipboard or in a file, and updates the currently edited list with imported fields. Such information is typically produced by another application.

Exporting data

The following dialog box appears when the "**Export text**" command is run. It enables the user to control the export mechanism.



Checking the "**Complete list**" choice indicates that the complete edited list has to be exported. The current selection is ignored in this case. Checking the "**Selected variables**" choice indicates that only highlighted variables will be exported.

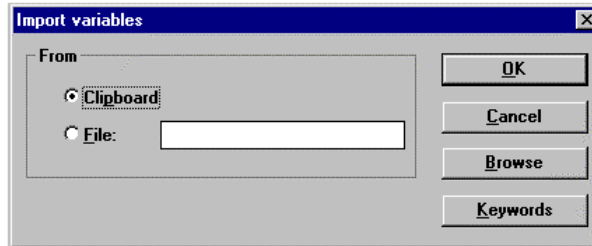
If the "**Clipboard**" option is checked, the exported information is stored, in pure ASCII text format, in the Windows clipboard. The text is then available for "paste" commands in other applications. If the "**File**" option is checked, the exported text is stored in an ASCII file. The complete pathname of this file has to be entered. The "**Browse**" command may be used to find an existing pathname.

Then the user chooses a format for the exported text. The available formats are described in further sections. Pressing the **"OK"** button runs the export function. Pressing the **"Cancel"** button closes the dialog box and escapes from the export command.

All the fields of the selected objects are stored in the exported text, in the standard declaration order. The first line of the exported text contains the name of the fields. Each object is described on one line of text. The "end of line" separator is the standard MS-DOS sequence **"0d-0a"**. The names used to identify the fields in the first exported line may be changed, by pressing the **"Keyword"** button. This command is described in further sections.

☰ **Importing data**

The following dialog box appears when the **"Import text"** command is run. It enables the user to control the import mechanism.



If the **"Clipboard"** option is checked, the imported information is taken from the Windows clipboard, in pure ASCII text format. If the **"File"** option is checked, the exported text is read in an ASCII file. The complete pathname of this file has to be entered. The **"Browse"** command may be used to find an existing pathname.

The import function automatically recognises the format (separators) used in the imported text. The available formats are described in further sections. Pressing the **"OK"** button runs the import function. Pressing the **"Cancel"** button closes the dialog box and escapes from the import command. The names used to identify the fields in the first imported line may be changed, by pressing the **"Keyword"** button. This command is described in further sections.

The first line of the text must contain the name of the fields, according to the order used in the following lines. Each object must be described on one line of text. The "end of line" separator is the standard MS-DOS sequence **"0d-0a"**. Fields can appear in any order. If some fields are missing, they are automatically filled in the imported object description by default values. If an imported object already exists in the edited list, the user has to confirm that it will be overwritten. The object description is then updated with imported fields. If some fields are missing, they are not updated in the object description.

☰ **Available text formats**

Below is the list of available formats for export command. The import command automatically recognises these formats.

- tab separators

Description: Fields are separated by tab characters.

Example:

Name	Attribute	Comment
level	internal	internal calculated water level
alm1	output	main alarm output

- comma separators

Description: Fields are separated by commas.

Example:

Name,Attribute,Comment
level,internal,internal calculated water level
alm1,output,main alarm output

- semicolon separators

Description: Fields are separated by semicolons.

Example:

Name;Attribute;Comment
level;internal;internal calculated water level
alm1;output;main alarm output

- commas and quotes

Description: Fields are separated by commas. Each field is written between quotes.

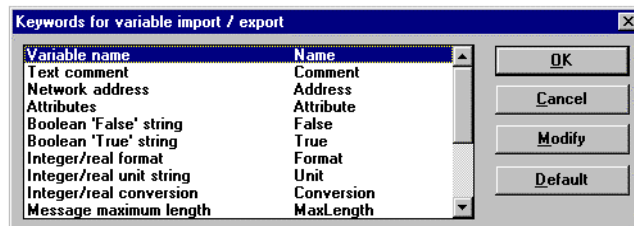
Example:

"Name","Attribute","Comment"
"level","internal","internal calculated water level"
"alm1","output","main alarm output"



Keywords

The names used to identify the fields in the first imported or exported line may be changed, by pressing the "**Keyword**" button. This command opens the following dialog box:



The window shows the list of object fields, and the associated keywords. To modify a keyword, the user must select a field in the list and press the "**Modify**" button.

Pressing the "**Default**" button restores the original list of keywords. Naming the keywords must conform to the following rules:

- the name cannot exceed **16** characters
- the first character must be a **letter**
- the following characters can be **letters**, **digits** or **'_'** character
- the same name cannot be used for different keywords

Below are the standard keywords found in ISaGRAF:

Object name.....	Name
Text comment	Comment
Network address	Address
Attributes (internal, input, output).....	Attribute
Boolean 'False' string	False
Boolean 'True' string.....	True
Analog format (real or integer).....	Format
Analog unit string	Unit
Analog conversion name	Conversion
Message maximum length.....	MaxLength
Function block library type.....	Library
Defined word equivalence	Equivalence
Internal attribute	Internal
Input attribute	Input
Output attribute	Output
Constant attribute.....	Constant
Real analog format.....	Real
Integer analog format	Integer








A.11 Using I/O connection editor

The aim of the I/O connection operation is to establish a logical link between the I/O variables of the application and the physical channels of the boards existing on the target machine. To make this link the user has to identify and set-up all the boards of the target machine, and place I/O variables on corresponding I/O channels.




The list on the left shows the rack of the target machine, with **board slots**. A slot may be free, or used by one I/O board or complex equipment. Each slot is identified by an **order number**. The rack may contain up to **255** boards. The list on the right shows the board's parameters and the variables connected on the selected board. A board may have up to **128** I/O channels. The total number of single I/O boards (including single equipments and boards of complex equipments) cannot exceed **255**.

Icons




The icons displayed on the front face indicate the type and attributes of variables that may be connected to the board channels. The ISaGRAF system does not allow the connection of variables of different types on the same board. This is the meaning of the used icons:

	boolean type
	integer/real type (both types of variables may be connected)
	message type
	inputs - no channel connected
	outputs - no channel connected
	inputs - at least one channel connected
	outputs - at least one channel connected

Below are the icons used to show the type of I/O device installed on a slot:

	complex I/O equipment
	real I/O board
	virtual I/O board

Below are the icons used to draw a parameter or a channel:

	board parameter
	free channel
	connected channel

Moving boards in list

Use these buttons in the toolbar or "**Edit / Move board up/down**" menu commands to move the selected I/O board one line up or down in the main list. The "**Edit / Insert slot**" command inserts an empty slot at the current position.

A.11.1 Defining I/O boards

The "**Edit**" menu contains basic commands to define the selected board (set-up its parameters), and to connect I/O variables to its channels.



Selecting I/O board type

Before connecting I/O variables to a board, the board identification must be entered. A library of pre-defined boards is available on the ISaGRAF workbench. This library may have been compiled by one or more I/O device suppliers. The "**Edit / Set Board/Equipment**" command is used to set-up board identification. This command can be used to select either a single board, or complex I/O equipment from the ISaGRAF library. It is also possible to double click on a slot to set the corresponding board or equipment.

All the channels of a single board have the same type (boolean, integer/real or message) and direction (input or output). Real and integer variables are not distinguished during I/O connection. A complex I/O equipment represents an I/O device with channels of different types or directions. A complex I/O equipment is represented as a list of single I/O boards. It uses only one slot in the rack list.





Removing a board

The "**Edit / Clear slot**" command is used to remove the currently selected board or I/O equipment. If variables are already connected to the corresponding channels, they are automatically disconnected when clearing the slot.



Real boards and virtual boards

The "**Edit / Real/virtual board**" command sets the validity of the selected board or complex I/O equipment. The following icons are displayed in the rack list to show the validity of a board:

 real I/O board
 virtual I/O board

In **Real Mode**, I/O variables are directly linked to the corresponding I/O devices. Input or output operations in the application program tie directly to corresponding input or output conditions of the actual field I/O devices. In **Virtual Mode**, I/O variables are processed exactly as internal variables. They can be read or updated by the debugger, so that the user can simulate the I/O processing, but no real world connection is made.



Technical notes

The "**Tools / Technical note**" command displays the on-line user's guide of the selected board or complex equipment. The board technical note is written by the hardware supplier of the I/O board. It contains all the information about I/O board management. It also describes the meaning of its parameters.



Removing connected variables


The "**Tools / Free board channels**" command disconnects all the I/O variables already connected on the selected board.

☰ **Defining comments for free channels**

The "**Tools / Free board channels**" command disconnects all the I/O variables already connected on the selected board.

A.11.2 Setting board parameters



To set the value of a board parameter, the user has to double click on its name in the list on the right. It is also possible to select (highlight) it and choose the "**Set channel/parameter**" command of the "**Edit**" menu. Parameters are listed at the beginning of the list. The following icon is used to represent them in the list:

 board parameter

The meaning and input format of the parameter are designed by the supplier of the corresponding I/O board or equipment. Use the "**Tools / Technical note**" command or refer to your hardware manual for more information about board parameters.

A.11.3 Connecting I/O channels

To set the connection of a channel, the user has to double click on its location in the list on the right. It is also possible to select (highlight) it and run the "**Edit / Set channel/parameter**" command. The following icons are used to represent channels in the list:

 free channel
 connected channel

The list contains all the variables which match with the selected board type and direction. Only variables which are not yet connected are listed here. The "**Connect**" button connects the variable selected in the list to the selected channel. The "**Free**" button removes (disconnects) the variable from the selected channel. "**Next**" and "**Previous**" buttons are used to select another channel of the board. The location of the selected channel is always displayed in the title of the dialog box.

A.11.4 Directly represented variables

Free channels are the ones which are not linked to a declared I/O variable. ISaGRAF enables the use of **directly represented variables** in the source of the programs to represent a free channel. The identifier of a directly represented variable always begins with "%" character.

Below are the naming conventions of a directly represented variable for a channel of a single board. "**s**" is the slot number of the board. "**c**" is the number of the channel.

%IXs.c free channel of a boolean input board
 %IDs.c free channel of an integer input board
 %ISs.c free channel of a message input board

%QXs.c free channel of a boolean output board
%QDs.c free channel of an integer output board
%Qss.c free channel of a message output board

Below are the naming conventions of a directly represented variable for a channel of a complex equipment. "s" is the slot number of the equipment. "b" is the index of the single board within the complex equipment. "c" is the number of the channel.

%IXs.b.c free channel of a boolean input board
%IDs.b.c free channel of an integer input board
%ISs.b.c free channel of a message input board
%QXs.b.c free channel of a boolean output board
%QDs.b.c free channel of an integer output board
%Qss.b.c free channel of a message output board

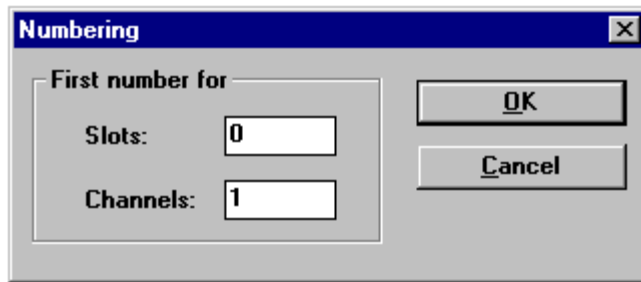
Below are examples:

%QX1.6 6th channel of the board #1 (boolean output)
%ID2.1.7 7th channel of the board #1 in the equipment #2 (integer input)

A directly represented variable cannot have the "real" data type.

A.11.5 Numbering

Use the "Options / Numbering" command to set numbering conventions. You can specify the number used for the first slot and the number used for the first channel of each board in the following dialog box:



As default, slot numbering starts at index "0", and channel numbering starts at index "1".

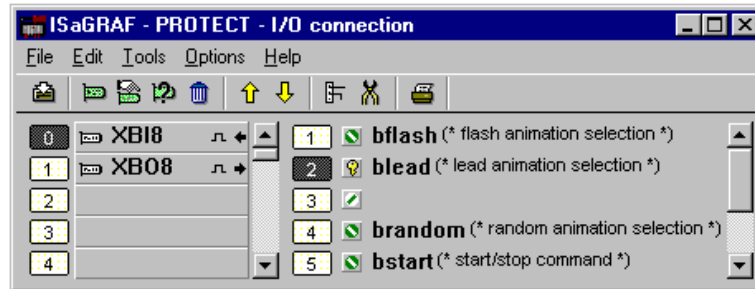
Warning: be very careful while changing numbering conventions as it has effect on symbols used for directly represented variables and may lead to compiling errors if directly represented I/O variables are used in existing programs.

A.11.6 Setting individual protections

The ISaGRAF workbench provides a complete data protection system based on hierarchised passwords. I/O connection can be globally protected by a password. Additionally, ISaGRAF enables you to set individual protection to any I/O channel. This assumes that:

- passwords are already defined in the password definition system (use the "**Project / Set password**" command of the Project Management window) so that protection levels are available for individual protection.
- you use protection levels with higher priority for individual protection compared to global I/O protection.

When an I/O channel has individual protection, a small icon is drawn close to its name in the I/O connection window:



Use the "**Set protection**" and "**Remove protection**" commands of the "**Edit**" menu to set or remove an individual protection for selected channel. Both commands ask you to enter a valid password so that a protection level can be attached to the channel. Then, each time you want to change connection to a channel having individual protection you must enter a password with sufficient priority level.

Warning: If a channel is protected with a level, and the corresponding password is removed from protection system, and if no higher level password is defined, connection to the channel cannot be changed anymore unless a new password with sufficient level is defined.

A.12 Creating conversion tables

The ISaGRAF workbench allows the user to create conversion tables. A conversion table is a set of points used to define an analog conversion. A conversion table can be attached to an analog input or output variable. A table creates a proportional relationship between electrical values (read on input sensor or sent to the output device) and physical values (used in application programming).

Conversion tables are edited through a dialog box run by the "**Tools / conversion**" command in the ISaGRAF dictionary window

A defined conversion table can be used to filter values of any input or output analog variable of the selected project. Attaching a conversion table to a variable is made using commands of the ISaGRAF dictionary, the variable declaration editor. An input or output analog variable must then be selected and its parameters edited. A variable cannot be attached to a conversion table that is not already defined.

A.12.1 Main commands

The "**Conversion tables**" dialog box shows the list of defined conversion tables, and contains push buttons for main commands, to edit an existing table (define its points), to create a new table, and also to rename or delete a table. Press OK to quit the "Conversion tables" dialog box and save them on disk.



Creating a new table

The "**New**" command allows the user to create a new conversion table. Up to **127** conversion tables can be created for each project. Only used tables (the ones attached to analog variables) are inserted in the application executable code. Naming a table must conform to the following rules:

- the name cannot exceed **16** characters
- the first character must be a **letter**
- the following characters can be **letters**, **digits** or **'_'** character
- the table name is case insensitive



Changing the contents of a table

The "**Edit**" command is used to enter the points of a table selected from the list. It is also possible to double click on the name of the table. The "**Edit**" command is automatically called when a new table is created. At least two points must be entered for each table.

A.12.2 Entering points of a table

The "**Edit**" dialog box allows the user to define the points of a conversion table. The box shows on the left side the list of points already defined. The lower right box shows the defined table as a graphic curve. The points are entered by using the box commands. The user must comply with the number rules for the definition of points, described at the end of this chapter. The box on the left always contains the list of existing points for the currently edited table. The column on the left shows the

electrical (external) value of the points. The column on the right shows the physical (internal) values. The user has to select a point on the list in order to modify its values or to clear (remove) it. The last choice of the list ("... ..") is used to define a new point. The box on the lower right shows the currently edited table as a graphical curve. No axes or co-ordinates are shown, as this is a proportional representation of the curve. This representation is useful as a quick check that the curve is properly defined.

⇒ ***Defining a new point***

When defining a new point, select the last entry ("... ..") on the list of points. This is also the default mode when starting to define a new conversion table. The user has to enter the electrical (external) and the physical (internal) values of each point. Values are stored as simple precision floating point numbers. Remember that at least **two points** have to be entered to define a curve. When both values are entered, pressing the "**Store**" button adds the point to the table. A maximum of **32** points can be defined for each conversion table.

⇒ ***Modifying a point***

To modify the values of an existing point, first select it from the list. The new electrical (external) and the physical (internal) values of the point can then be entered. Values are stored as simple precision floating point numbers. When both values are entered, pressing the "**Store**" button updates the point in the table.

⇒ ***Clearing a point***

An existing point is cleared by selecting it from the list and pressing the "**Clear**" button. Remember that at least **two points** must be entered to define a table.

A.12.3 Rules and limits

The rules shown below must be followed when defining a conversion table. The table can be used to convert both input and output analog variables:

- Two points cannot be defined with the same electrical value
- The curve must be continuously increasing or decreasing
- Two points cannot be defined with the same physical value

The following limits apply when defining conversion tables for a project:

- No more than **127** conversion tables can be defined in the same project
- No more than **32** points can be defined for the same conversion table.

A.13 Using the code generator

The code generation window is automatically opened by the "Verify" and "Make" commands of the other ISaGRAF Workbench windows. The code generation window is not automatically closed when the requested code generation operation ends, so that the user still has access to all the code generation commands and options from the window menu.

A.13.1 Main commands

The "Files" menu contains the commands for program syntax checking and code generation.

☰ **Make application code**

The "Make" command constructs the entire code of the project. Before generating anything, this command checks the syntax of the declarations and programs. Any error that cannot be detected during single program compiling is detected during code generation. This applies to tables of conversion, I/O variable connections and links with the libraries. The code generation halts the compiling of a program when errors are detected. This program must be corrected before continuing the code generation. Programs which have already been checked (with no error detected) and that have not been modified since their last "Verify" operation are not re-compiled. Variable declaration verification and application coherence checking are always processed. During program checking, the "Make" operation can be aborted by hitting the **ESCAPE** key.

Note: If the declaration of a local variable of a program has been modified, this program is verified. If a global variable has been modified, all the programs are verified.

☰ **Program syntax checking**

The "Verify program" command allows the user to verify only one program. The selected program is compiled even if it has not been modified since its last verification. The "Verify dictionary" command allows the user to verify the declarations of all the variables of the project.

The "Verify all programs" checks the syntax of all the programs of the project, even if some of them have not been modified. This command **does not** stop when an error is detected in a program. It can be used to produce a complete listing of all the errors remaining in programs of the project. This command may be aborted by hitting the **ESCAPE** key.

☰ **Simulating a modification**

The "Touch" command simulates a modification of all the project's programs, so that they are all verified during the next "Make" operation. The "Open" command is used to open the last verified program. This command is very useful to directly access a program where syntax errors have been detected.

A.13.2 Compiler options

The "**Compiler options**" command is used to set-up main parameters used by the ISaGRAF Code Generator to build and optimise the target code. The aim of this command is to select the type of code which has to be generated, according to corresponding ISaGRAF targets, and to set-up the optimiser parameters according to the expected compiling time and application run-time requirements.

The "**Upload**" button opens a second dialog box with other options that enable the embedding of zipped source code to downloaded code, in order to enable the "Upload" feature. Refer to "Upload" documentation for further explanations.

☰ **Selecting targets**

The upper list shows the list of available target codes that can be produced. The ">>" sign is used to indicate the selected target(s). The ISaGRAF Code Generator can produce up to **3** different codes in the same compiling operation. Use the "**Select**" and "**Unselect**" buttons to set the list of required target codes, according to your target hardware. Below are the standard ISaGRAF targets:

SIMULATE:..... This code is dedicated to the ISaGRAF Simulator on the Workbench. The simulator cannot be run if this target is not selected to produce the application code.

ISA86M:..... This is a TIC code (Target Independent Code) dedicated to ISaGRAF kernels installed on Intel based processors. The processor type only concerns byte ordering in the generated code.

ISA68M:..... This is a TIC code (Target Independent Code) dedicated to ISaGRAF kernels installed on Motorola based processors. The processor type only concerns byte ordering in the generated code.

SCC:..... Selecting this target leads ISaGRAF compiler to produce structured "C" language source code to be compiled and linked with ISaGRAF target kernel libraries to produce an embedded executable code.

CC86M:..... Selecting this target leads ISaGRAF compiler to produce non structured "C" language source code to be compiled and linked with ISaGRAF target kernel libraries to produce an embedded executable code. This selection is provided for compatibility with ISaGRAF versions before V3.23, when structured "C" code generation and integration were not supported.

Refer to your hardware manual to know the type of ISaGRAF target kernel installed on your PLC. Other target types (machine code, C source code...) may be supported in future releases of the ISaGRAF Workbench.

☰ **SFC processing**

Check the "**Use embedded SFC engine**" box to enable the use of the ISaGRAF SFC engine. ***This mode should be preferred as it leads to higher run time performances.*** However, the target engine may be missing on some particular implementations of the ISaGRAF target, of more commonly on customised targets based on ISaGRAF code post-processing. In this case you may have to remove this

option and let ISaGRAF compiler translate SFC charts into low level instructions. Refer to your hardware documentation for more information about the use of this option.

▬ **Optimiser options**

Below are the parameters, used by the ISaGRAF Code Generator to optimise the target code, that can be set from the "**Compiler options**" dialog box. The "**Default**" button is used to remove all optimising options, in order to reduce the compiling time.

- ◆ When the "**Run two optimiser passes**" option is set, the ISaGRAF Code Optimiser is run twice. Optimisations made during the second pass are generally less significant than the ones made in the first pass.
- ◆ When the "**Evaluate constant expressions**" option is set, constant expressions are evaluated by the compiler. For example, the numerical expression "**2 + 3**" is replaced by "**5**" in the target code. When this option is not set, constant expressions are calculated at run-time.
- ◆ When the "**Suppress unused labels**" option is set, the Optimiser simplifies the system of jumps and labels of the programs, in order to suppress unused target labels or null jumps.
- ◆ When the "**Optimise variable copying**" option is set, the use of temporary variables (used to store intermediate results) is optimised. This option is commonly used with the "**Optimise expressions**" option. When this option is set, the Optimiser re-uses the result of expressions and sub-expressions which are used more than once in the program.
- ◆ When the "**Suppress unused code**" option is set, the Optimiser suppresses the code which is not significant. For example, if the following statements are programmed: "**var := 1; var := X;**", the corresponding generated code is only: "**var := X;**".
- ◆ When the "**Optimise arithmetic operations**" option is set, the Optimiser simplifies arithmetic operations according to special operands. For example, the expression "**A + 0**" will be replaced by the "**A**". When the "**Optimise boolean operations**" option is set, the Optimiser simplifies boolean operations according to special operands. For example, the boolean expression "**A & A**" will be replaced by "**A**".
- ◆ When the "**Build binary decision diagrams**" option is set, the Optimiser replaces the boolean equations (mixing **AND**, **OR**, **XOR** and **NOT** operators), by a reduced list of conditional jump operations. The translation is operated only if the expected execution time of the jump sequence is less than the one expected for the original expression.

The following table summarises the expected optimisation and requested compiling time corresponding to each parameter:

	gain (performances)	compiling time
Run 2 passes	xxxx	(*)
Optimise constant expressions	xxxxxxxx	xxxx
Suppress unused labels	xxxx	xxxxxxxx
Optimise variable copying	xxxx	xxxxxxxx
Optimise expressions	xxxx	xxxxxxxx
Suppress unused code	xxxx	xxxxxxxx
Optimise arithmetic operations	xxxxxxxx	xxxx
Optimise boolean operations	xxxxxxxx	xxxx
Build binary decision diagrams	xxxxxxxxxxx	xxxxxxxxxxx

(*) The compiling time is also multiplied by 2.

A.13.3 Producing C source code

The ISaGRAF workbench enables the production of source code in "C" language. In this case, the whole contents of the application, including SFC chart description, data base definition and sequences of code are generated in "C" source code format. There are two possibilities, proposed as two styles of generated code:

CC86M..... (C source code - V3.04) produces non-structured "C" source code. This style should be selected if your target software is based on ISaGRAF release previous to 3.23.

SCC (structured C source code) produces a structured "C" source code. This style should be preferred if your target software is based on ISaGRAF release 3.23 or later.

The following two files are created in the project directory:

APPLI.C common source code of the application
APPLI.H common "C" language definitions

In the case structured "C" source code generation, a ".C" source file and a ".H" definition file are created for each program of the application, in addition to common "APPLI.C" and "APPLI.H" files. These files must be compiled and linked to the ISaGRAF target libraries in order to produce the final executable code. Refer to the "ISaGRAF I/O development toolkit User's Guide" for further information about recommended implementation techniques.

Note: Some debugging features such as application downloading, on line modification and breakpoints are no more available when the ISaGRAF application is "C" compiled.

A.13.4 Viewing information

The "Edit" menu contains the commands for viewing the different text files built during code generation or syntax checking operations on the code generator window. The code generation window is a text area that contains messages during code generation or syntax checking operations. All information is stored on the disk so it can be examined using the "Edit" menu commands.

☰ **Editing commands**

The "**Clear Screen**" command is used to clear the window text area. The window is automatically cleared before each code generation or syntax checking operation. The "**Copy**" command is used to copy the displayed text in the clipboard of Windows, so it can be used by other applications such as ISaGRAF text editors.

☰ **Viewing compiler output messages**

The "**Execution messages**" command shows all the messages displayed during the last "**Make**" or "**Verify**" operation on the window text area. This applies to all the error messages.

Other choices of the "**Edit**" menu allow the user to monitor auxiliary text files created during syntax verification and code generation. These files are not usually used for a common ISaGRAF project.

A.13.5 Defining resources

The "**Resources**" command of the "**Options**" menu allows the user to define resources. A resource is any user-defined data (network configuration, hardware setting...) of any format (file, list of values) which has to be merged with the generated code, in order to be downloaded with it in the target PLC. Such data is not directly operated by the ISaGRAF kernel, and is commonly dedicated to other software installed on the target PLC. Refer to your hardware manual for further information about available resources.

☰ **The resource definition file**

The resources are defined in a "**Resource definition file**" stored with other files of the ISaGRAF project. This is a pure ASCII text file, processed by the ISaGRAF Resource Compiler. This compiler is automatically run when the application code is built. This section explains the syntax of this file. The resource definition file uses lexical rules of the ST language. Comments, beginning with "(" and ending with ")" characters can be inserted anywhere in the text. Strings are delimited by single apostrophes. Refer to the second part of this manual for more explanations about the lexical formats used to enter numerical values.

☰ **Language reference**

Below is the list of keywords and statements used in a resource definition file.

ULONGDATA

Meaning: Specifies a resource which is a list of integer values. Values are stored in target code as unsigned 32 bit integers. Values are stored in the order specified in the resource definition file. Values must be separated by comas. The name of the resource cannot exceed 15 characters.

Syntax: **ULONGDATA** '<resource_name>'
BEGIN

```

    ...target_selection...
    ...list of values...
END

```

```

Example: ULongData 'MYDATA'
Begin
    ...
    0, -1, 100_000,          (* decimal *)
    16#A0B1, 2#1011_0101    (* hexadecimal, binary *)
End

```

VARLIST

Meaning: Specifies a resource which is a list of variable addresses. Variables are identified by their name in the resource definition file. Variable addresses are stored in target code as unsigned 16 bit integers. Addresses are stored in the order specified in the resource definition file. Variables must be separated by comas. The name of the resource cannot exceed 15 characters.

```

Syntax: VARLIST '<resource_name>'
BEGIN
    ...target_selection...
    ...list of variable names...
END

```

```

Example: VarList 'LIST'
Begin
    ...
    Var100, MyParameter, Command, Alarm
End

```

BINARYFILE

Meaning: Specifies a Binary File resource. The source data is stored in an MS-DOS file. The target resource definition is completed with a target pathname. End of line characters are not converted by the ISaGRAF Resource Compiler. The name of the resource cannot exceed 15 characters.

```

Syntax: BINARYFILE '<resource_name>'
BEGIN
    ...target selection...
    FROM '<source_pathname>'
    TO '<destination_pathname>'
END

```

```

Example: BinaryFile 'MYFILE'
Begin
    ...
    From 'c:\user\config.bin'
    To '/dd/user/appl/config.dat'
End

```

TEXTFILE

Meaning: Specifies a Text File resource. The source data is stored in an ASCII file. The target resource definition is completed with a target pathname. End of line characters are converted by the ISaGRAF Resource Compiler according to the target host system conventions. The name of the resource cannot exceed **15** characters.

Syntax: **TEXTFILE** '<resource_name>'
 BEGIN
 ...target selection...
 FROM '<source_pathname>'
 TO '<destination_pathname>'
 END

Example: TextFile 'MYFILE'
 Begin
 ...
 From 'c:\user\config.bin'
 To '/dd/user/appl/config.dat'
 End

TARGET

Meaning: Specifies the name of a target code that has to include the resource. Refer to the previous section (compiler options) for further information about handled targets. The "**Target**" statement can appear more than once in the same resource block, in order to select several targets. This statement cannot be used if the "**AnyTarget**" statement is specified.

Syntax: **TARGET** '<target_name>'

Example: BinaryFile 'MYFILE'
 Begin
 Target 'ISA86M'
 Target 'ISA68M'
 ...
 End

ANYTARGET

Meaning: Specifies that the resource must be merged to all the target codes built by the Code Generator. The ISaGRAF Code Generator can produce several target codes during the same "**Make**" command. This statement cannot be used if one or several "**Target**" statements are specified.

Syntax: **ANYTARGET**

Example: ULongData 'MYDATA'


```

Begin
  AnyTarget
  ...
End

```

FROM

Meaning: Specifies the source pathname (on the PC where the ISaGRAF Workbench is installed) of a **BinaryFile** or **TextFile** resource. The characters used to isolate the components of the pathname (drive, directory, prefix, suffix) must conform to the MS-DOS system conventions.

Syntax: **FROM** '<target pathname>'

Example: BinaryFile 'MYFILE'
 Begin
 ...
 From 'c:\user\config.dat'
 To '/dd/user/appl/config.dat'
 End

TO

Meaning: Specifies the destination pathname (on the target system) of a **BinaryFile** or **TextFile** resource. The characters used to isolate the components of the pathname (drive, directory, prefix, suffix) must conform to the target host system conventions.

Syntax: **TO** '<target pathname>'

Example: TextFile 'MYFILE'
 Begin
 ...
 From 'c:\user\config.dat'
 To '/dd/user/appl/config.dat'
 End

Example

Below is a complete example of a resource definition file:

```

(* resource definition file *)

ULongData 'DATA1'          (* list of values *)
Begin
  Target 'ISA86M'          (* for this target only *)
  1, 0, 16#1A2B3C4D, +1, -1 (* numerical values *)
End

VarList 'VLIST1'          (* list of variables *)
Begin
  Target 'ISA86M'          (* for this target only *)
  Valve1, StateX, Command, Alrm1 (* variable names *)

```

```
End

BinaryFile 'FILE1'          (* binary file resource *)
Begin
  AnyTarget                 (* dedicated to all targets *)
  From 'c:\user\updatef.bin' (* source file on PC *)
  To 'updatef.cfg'         (* target file on PLC *)
End

TextFile 'FILE2'           (* text file resource *)
Begin
  Target 'ISA68M'
  From 'c:\nw\nwbd.txt'    (* source file on PC *)
  To '/nw/dat/nwbd'       (* target file on PLC *)
End
```

▣ **Resource compiling**

If resources have been entered in resource definition file, a dialog box appears at the end of ISaGRAF code generation. Press the **"Start compile"** button to run resource compiler. Output messages and errors will be displayed in the main control. Press **"Exit"** to avoid resource compiling. In this case, resources will not be added to the ISaGRAF code.

▣ **Implementation**

The number of resources, the size of data rows and files are not limited by ISaGRAF. Resources are stored at the end of the generated code, with a resource directory. Below is the format (using C notations) of the resource directory format:

```
RESOURCE:
{
  long nbres;                /* number of defined resources */
  {
    char name[16];          /* resource name */
    long type;              /* resource data type */
    long size;              /* exact size of data block */
    uint32 data;
    uint32 path_offset;     /* points to a string */
  } /*nb of records */
}
```

Below are the possible values of the "type" field:

- 1 = binary file
- 2 = text file
- 3 = ulong data (path_offset field is not used in this case)
- 4 = variable list (path_offset field is not used in this case)

For text files, end of line characters are translated by the resource compiler, according to the target system conventions. All pointers are 32 bit offsets from the address of the corresponding structure. All resource names and pathnames are NULL terminated strings. Pathnames and data follow the resource directory.

A.14 Cross References

The ISaGRAF workbench includes a cross-reference editor which provides user with a total view of the declared variables in the project's programs, and where they are used. The aim of the cross reference is to list all the variables declared in the project, and to localise, at the source of each program the parts of source code where those variables are used. The cross-references are very useful for a global view of one variable life cycle. They help localise side effects, and reduce the time to understand the project during the maintenance. The cross-references may also be used for a global view of the complete dictionary of a project, so unused variables are easily found and the complexity of the project measured.

The list on the left shows the declared objects of the project (programs, variables and defined words), and the library elements (functions and function blocks) referenced in the project. The list on the right shows the occurrences in the programs of the object currently selected in the first list.

The description of an occurrence includes the program name, the number of the FC or SFC step, transition or test, plus line number for text languages or co-ordinates for LD or FBD diagrams. For quick LD diagrams, the description is completed with the number of the rung. If the variable is used as an output (on a coil) the rung number is followed by a star ("*") character.

Set the "**Show unused variables**" option from the "**Options**" menu to display also in main list variables that are not used in the application programs.

Object type selection

Because a project can group a huge number of declared objects, the combo box in the editor toolbar is used to select the type of objects which must be listed in the window. This allows the user to have access to selected information.

Each time the cross-references are re-calculated, the selection is reset to "**All objects**" in order to present the complete list.

Re-calculate cross-references

The "**File / Re-calculate**" command can be used at any time to update the cross references according to the modifications entered in other ISaGRAF editing windows.

Export cross-references

The "**Tools / Export**" command is used to write the complete listing of the cross-references in an ASCII text file. This file can then be opened with other applications such as Windows NotePad or word processors.



Dictionary errors

The "**Edit / Dictionary errors**" command displays in a dialog box the list of errors detected when the project dictionary was loaded.



Statistics

The "**Tools / Statistics**" command displays in a dialog box the number of objects and variables declared in the project, according to variable types and attributes. A

particular application of this command is to know the number of I/O variables declared in the project, in order to ensure that it can be compiled, if a limited version of the ISaGRAF Workbench is used.



Search in object list

The "**Edit / Search**" command allows the user to directly select an object in the editor list. The searched object cannot be found if it is not actually listed (when using a selected display). It is recommended, before searching for an object, to activate the "**All**" selection in the toolbar.



Open program

The list on the right contains the occurrences of the selected object in the source files and I/O connection of the open project. The "**Edit / Open program**" command enables the user to directly open a program where the object appears. It is also possible to double click the mouse on an occurrence (in the occurrence list) to open the corresponding program.

A.15 Using the graphic debugger

ISaGRAF includes a complete graphic and symbolic debugger. The "Debug" command of the program management window runs the debugger to control the application downloaded in the target PLC. In this mode, the debugger communicates with the target system via hardware link. The "Simulate" command of the program management window simultaneously runs the debugger and a complete target simulator. This enables the user to test his application when the target's I/O system is not yet complete. The debugger window contains the commands to control the entire application.

When the debugger starts, and if the application in the target PLC is the same as the one on the workbench, it automatically opens the **program management window**, in debug mode. Commands of this window may be used to open other ISaGRAF windows (graphic and text editors, dictionary, lists of variables, I/O connection...). All windows opened during a debug session operate in "**debug mode**", meaning that the editing command is disabled. Displayed program components (steps, transitions, variables...) are shown with their current run time status or value. Double clicking on an object changes its status or value in the target application.

When running the debugger in **simulation mode**, communication with the ISaGRAF target system is stopped. The debugger only communicates with the simulator window. Because the target system does not exist in this mode, the "**download**", "**stop**" or "**activate**" commands are not available on the debugger menu.

A.15.1 The debugger window

The debugger window only contains information about the complete application status. It is linked to other ISaGRAF windows creating a complete interactive debug system. Detected run time errors are displayed in the bottom area of the debugger window. Commands from the "**Options**" menu are used to hide, show or clear the list of errors.

The control panel (area under the debugger menu) shows the global status of the target application, and information about the execution cycle timing. The list of possible target status is as follows:

Logging:.....Debugger establishes communication with the target system.

Disconnected:.....Debugger cannot communicate with the target system. Ensure connection cable and communication parameters are valid.

No application:Connection is OK, but no ISaGRAF application currently exists in the target system. Download an application.

Application active:.....Connection is OK and an active application exists in the target system. Debugger is now establishing the communications with this application, if it is the same as the one on the Workbench.

RUN:Target application is in "Real Time" mode.

STOP:.....Target application is in "Cycle to Cycle" mode.
BreakPoint:Target application is in "Cycle to Cycle" mode, because a breakpoint is encountered.
Fatal Error:.....Target application failed because a serious error occurred.

Information on the run time cycle timing is as follows:

Allowed:programmed timing.
Current:exact timing of the last complete execution cycle.
Maximum:maximum timing detected since the application started.
Overflow:.....number of execution cycles detected with a timing greater than the allowed timing.

All time values are given in milliseconds. Time values are not displayed when debugger is used in simulation mode.

A.15.2 Controlling the application

The "**File**" and "**Control**" menus contain the commands for the installation and the control of the currently edited ISaGRAF application on the ISaGRAF target system.

Note: Some of these commands are not available during simulation, because the application processed by the simulator is automatically installed by the ISaGRAF Workbench.



Stop the target application

The "**File / Stop application**" command stops the execution of the application currently active in the ISaGRAF target system.



Activate the target application

The "**File / Start application**" command runs the application existing in the target system. When an application is downloaded, it is automatically started, so that the "**Start**" command does not have to be used. The "**Start**" command is typically used after a "**Stop**" command.

Note: the target application must be stopped (inactive) before it is possible to download a new application.



Download the application

The "**File / Download**" command is used to download the application code in the target system. Select the type of code to be downloaded, according to the target system processor and the application options.



Display version number

The "**File / Get version number**" command is used to display complete identification of both Workbench and target applications. The Workbench application is the one currently open on the ISaGRAF Workbench. The target application is the one executed in the target ISaGRAF PLC. The following items are displayed:

VERSION: This is the version number of the application code. This number has been calculated by the code generator.

DATE: This item shows the date and time when the code has been built.

CRC: This is a checksum calculated with the contents of the table of symbols. This number has been calculated by the code generator. This value depends on the contents of the dictionary of variables.

Note: The "**Get version number**" command is also available during simulation. In real debug mode, this command cannot be used if the target PLC is not connected.



On line modification

The "**File / Update application**" command enables the user to achieve "on line modification" of the running target application. This command is detailed in further sections of this chapter. It is not available when the debugger is used in simulation mode.



Real Time mode

The "**Control / Real time**" command is not available when no application is active. It sets the target application in normal "real time" mode: Normal mode: the execution cycles are triggered by the programmed cycle timing.



Cycle to Cycle mode

The "**Control / Cycle to cycle**" command is not available when no application is active. It sets the target application in normal "cycle to cycle" mode: In this mode, cycles are executed one by one, according to the "**Execute one cycle**" commands made by the user from the debugger menu.



Execute one cycle

When target is in cycle to cycle mode, the "**Control / Execute one cycle**" command runs the execution of one cycle.



The cycle timing

The "**Control / Change cycle timing**" command enables the user to modify the programmed cycle timing. This time is titled as "**Allowed**" in the debugger control bar window. The "**Cycle to cycle**" mode should be set before modifying the cycle timing. The cycle timing is entered as an integer number in milliseconds.



Remove all breakpoints

The "**Control / Clear all breakpoints**" command removes all the breakpoints currently installed (encountered or still active) in the whole application. Existing breakpoints are not automatically removed when the debugger window is closed.



Unlock I/O variables

The "**Control / Unlock all IO variables**" command unlocks all the I/O variables currently locked in the application. When an I/O variable is locked, no input or output status change is made to the corresponding I/O device. Variables attached

to the I/O can still be written by the application or by the debugger. Currently locked I/O variables are not automatically unlocked when the debugger window is closed.

A.15.3 Options

The "**Options**" menu contains the options to control the information displayed in the debugger window.

▣ **The communication parameters**

The communication timing parameters can be adjusted when the debugger is active. Only communication **time-outs** can be set here. Other communication parameters (baud rate, parity...) must be set from the "**Debug**" menu of the Program Management window.

The "**Communication time-out**" is the time left for the target system to begin the answer to one workbench request. The "**Cyclic refresh duration**" is the time period required for the "**read**" requests to be sent by the debugger in order to refresh data in the opened windows.

All the time values are displayed and entered as integer numbers in **milliseconds**. The communication timing parameters cannot be set when the debugger is used in simulation mode.

▣ **Display options**

The "**Show cycle timing**" option enables the user to hide or show the **cycle timing** values in the debugger control bar. When this option is set, all the cycle timing components (allowed, current, maximum, overflows) are displayed and refreshed. Disabling this option reduces the debugger communication burden.

When the "**Show errors**" option is set, detected run time errors are listed in the bottom area of the debugger window. When this option is disabled, the error list is closed. Removing this option reduces the debugger display and communication burden. The "**Options / Clear errors**" command clears the list of run-time errors currently displayed in the debugger window.

The "**Options / minimise window**" command reduces the size of the debugger window so that it is shown as a small, always on top, panel containing only the application status and graphic buttons for most commonly used commands.

A.15.4 "Write" commands

The ISaGRAF symbolic debugger offers many commands to change the **value** or **status** of the application components. Selecting the component to be changed is done by **double clicking** on its name or its drawing in an editing window, when the debugger window is opened.

▣ **Variables**

A variable status is changed by double clicking on its name in one of the following windows:

- Dictionary
- Lists of variables or time diagrams
- LD or FBD Programs

- I/O connection

The following commands are offered in the debug dialog box:

- Write the variable to a new value
- **Lock** the variable (for I/O variables only)
- **Unlock** the variable (for locked I/O variables only)
- **Start** or **stop** a timer variable (set automatic refresh mode)

Symbolic values used to represent boolean **FALSE** and **TRUE** values are the strings defined for that specific boolean variable in the dictionary. The analog value specified for a **"Write"** command must be entered in an integer or real format, according to the variable definition in the dictionary. The string specified for a **"Write"** command for a message cannot be longer than the message capacity attached to that specific variable in the dictionary.

☰ **SFC objects**

To observe a control operation on an **SFC program** while debugging the application, commands of the **"File"** menu are used in the Program Management window. The SFC program must be selected from the list of programs. The following commands are available:

Start SFC program:Enables the selected program by putting an SFC token into each of its initial step.

Kill SFC program:Kills the selected program by removing all its existing tokens.

Freeze SFC program:Suspends the execution of the selected program.

Restart SFC program: ...Restarts a frozen (suspended) program.

For child programs, these commands correspond to the **"GSTART"**, **"GKILL"**, **"GFREEZE"** and **"GRST"** functions in the programming language.

A control operation can be seen in an **SFC step** when debugging the application by double clicking on its graphic representation in the SFC editing window. The following commands are available in the debug dialog box:

- Install a breakpoint on the step **activation**
- Install a breakpoint on the step **de-activation**
- **Clear** breakpoint added to the step

Note: Activation and de-activation breakpoints cannot be added to the same step.

A control operation can be seen in an **SFC transition** when debugging the application by double clicking on its graphic representation in the SFC editing window. The following commands are proposed in a debug dialog box:

- Add a **breakpoint** on the transition clearing
- **Clear** a breakpoint added to the transition
- Manually **clear** the transition (move or add tokens)

Conditional clearing: a token is created on the steps following the transition. The tokens existing in the preceding steps are removed. **Unconditional clearing:** a

token is created on the steps following the transition. The tokens existing in the preceding steps are not removed.

A.15.5 On line modification

The "On line modification" feature enables the user to modify the application while the process is running. This is sometimes necessary for chemical processes where any interruption may jeopardise production or safety. This function should be used **very carefully**. ISaGRAF may not be able to detect all possible conflicts generated by user defined operations as a result of these on-line changes.

▣ **Code sequences**

As ISaGRAF offers many possibilities for access to variables, programs or I/O boards from the debugger, the "On line modification" function described here applies only to the code sequences modification. A sequence of code is a complete set of ST, IL, LD or FBD instructions executed in a row. In a "beginning of cycle" or "end of cycle" program, a code sequence is the entire list of instructions written in the program. In an SFC program, a code sequence is the Level 2 programming of one step or transition. The "On line modification" consists of replacing one or more code sequences, without stopping the PLC execution cycle. As the control of the SFC tokens is very critical, **it is not possible to modify an SFC structure, to add, renumber or remove a step, a transition or an SFC program.**

▣ **Variables**

As the variable database is a very critical part of the application, it can be accessed at any time by other processes (on multitasking PLC). It is also possible to modify variable values from the debugger. Therefore, **ISaGRAF does not allow the user to add, rename or remove a variable** on line. Anyway, it is possible to modify the way a variable is used in the application. It is also possible to reserve "unused" internal or I/O variables in the first version of the application, so that future modifications can make use of them.

They are different styles of variables in ISaGRAF target database. Limitations act on all of them:

- Declared variables

They are the ones declared using the ISaGRAF dictionary. They cannot be changed and cannot be renamed for on line change. It is recommended that some extra variables are declared and initialised in the application even if not used today. Such extra variables will enable future modifications to work on without changing the application data checksum.

- Instances of function blocks

Each instance of "C" or IEC written function block corresponds to data stored in ISaGRAF target real-time database. When function block instances are added or removed, On Line change is no more possible. So it is better to work in ST with FB instances declared in dictionary, rather than adding blocks (that will correspond to new automatically declared instances) in Quick LD or FBD diagrams. Also, any

modification in the definition of available function blocks in the ISaGRAF library will lead to an impossible On Line change.

- *Steps*

Each SFC step corresponds to a piece of data where are stored SFC step dynamic attributes (its activity time and flag). Adding or removing SFC steps change the application database and is denied for On Line change.

- *Hidden variables allocated by compilers*

The ISaGRAF Compiler generates "hidden" temporary variables to solve complex expressions. In some case, the change of an expression may lead to a different set of non-visible temporary variables, and that leads to an impossible On Line change. To avoid this situation, you can add the following entries in ISA.INI file, in order to force a minimum number of temporary variables to be allocated for each program, even if not used for the compiling of the first application version. Values given here are examples:

```
[DEBUG]
MNTVboo=8      ; for booleans
MNTVana=4      ; for integers and reals
MNTVtmr=4      ; for timers
MNTVmsg=2      ; for messages
```

When such a setting is written in ISA.INI, the compiler outputs a warning message if a new compiling of the application leads to a greater number of allocated temporary variables.

⇒ **Inputs and outputs**

As the ISaGRAF I/O system is very open, required modifications should be implemented by the OEM, using specific features of the corresponding hardware. The ISaGRAF system **does not allow the user to add, connect or remove an I/O variable, or to modify the description of an I/O board** on line. Operations such as modifying board parameters and locking I/O channels are available using standard OEM features and the "**OPERATE**" function.

⇒ **Run time operations**

Modifying a running application consists of the following operations:

- modify the application source code on the workbench
- generate the new application code
- download the new application code using "**update**" command instead of "**download**"
- switch from the old application to the new one, in between PLC execution cycles using the "**Realise update**" command.

This procedure guarantees that the target PLC always has a complete and reliable running application, and enables the user to control the timing of the sample operations in a very safe and efficient way. It also enables the user to modify the project as often as possible. Regardless to the process itself, the "On line modification" is essentially the same as a normal "**stop, start and download**" set of

commands. The only differences are that no variable state is lost, and the switching time is very short (usually 1 or 2 cycle duration). During the switch, no variable is modified, and **all internal, input or output variables keeps the same value** before and after the application modification. During the switch, no action is performed, and **SFC tokens are not moved**.

☰ **Memory requirements**

In order to support the "On line modification" capability, the target PLC must have free memory space to enable the storage of the modified version of the application code. Both versions of the application code have to be stored in PLC memory during the switch operation.

☰ **Limitations**

As described before, only modifications to code sequences are allowed. Variable definition, application parameters and I/O connections cannot be modified. When downloading a modified version of the application, ISaGRAF makes a comparison between the modified application and the running one, in order to detect any unsafe change. If the switch seems dangerous or impossible, a download error is generated. One of the safeguards performed by ISaGRAF is to compare the symbol table checksum, so that any variable, program, or SFC element name change is detected. If a step is active when the switch occurs, its non-stored (N) actions are lost. The new step activation actions are not executed. Actions executed at the de-activation of the step are the ones carried over in the new application code. If a transition is valid when the switch occurs, its receptivity equation is updated. The new downloaded application code is not backed up on the PLC. The backup is the version which was previously downloaded with standard download commands.



Operations

To update the code of a running application, the following operations have to be performed:

- Before making any change on a running application, it is highly recommended to make a copy of the current project under another name. The modifications may be performed on the copies.
- Before editing any program, the user should check that the "**update diary**" option of the editing tools is set, to ease future program maintenance.
- When one or more sequences have been modified (without modifying SFC structures and program hierarchy), the code of the new application must be generated on the workbench before downloading.
- Using the debugger, from within the old project, the user must connect the target PLC and perform any operation which can make the application update faster or more safety.
- Using the debugger, from inside the new project, the user must connect the target PLC. If the application name is changed, the target database cannot be accessed. The user must run the "**File / Update**" command.
- The modified application is downloaded by selecting the "**update later**" option. This may slightly slow down the PLC during transfer.
- When download is complete, the user can run the "**File / Realise update**" command to enable the switch at the most adequate moment. The switch will have a 1 or 2 cycle duration.

- When the switch has been correctly performed, the programs of the modified running application are displayed. If not, the existing running application remains as is.

A.15.6 DDE exchanges

The ISaGRAF debugger includes a DDE (Dynamic Data Exchange) server. An advice loop can be installed between the ISaGRAF debugger and other applications, in order to dynamically display the current value of variables in non-ISaGRAF applications.

Only "advise" and "poke" transactions are supported by the ISaGRAF debugger DDE server. You can use "request" transaction only for variables already spied in an advice loop. Other DDE services such as "execute" are not available. When an advice loop is established on a variable, the value of this variable is updated in the client application each time it changes. Variables of any type can be spied. The identification of the dynamic link includes the following names:

Service name: "ISaGRAF"
Topic name: Name of the ISaGRAF project
Item name: Name of the variable

If the variable is local to a program, its name must be followed by the name of its father program, written between parentheses, with the following syntax:

variable_name(program_name)

The ISaGRAF debugger DDE server is dedicated to the ISaGRAF application currently spied by the debugger. Up to **256** variables can be spied by the ISaGRAF server. The DDE server may be used when the ISaGRAF debugger runs in either connected or simulation mode. The refresh duration is the one established for communication between the debugger and the ISaGRAF target system or simulator.

A.16 Spying Lists of variables

The "**Spy lists**" command in the "**Spy**" menu of the Debugger window enables the user to build non-contiguous lists of variables which are refreshed with their current values. Lists are built when debugging the application. The lists can be stored on the disk and opened again during other debug sessions. A list may contain up to **32** variables. Variables of different types may be mixed in the same list. Global and local variables can be inserted in a list. A list of variables is dedicated to one particular project. Lists of variables are very useful for the functional testing of an application. They allow the user to watch the changes of a limited part of the controlled process, independent of the corresponding source code in the application programs. Lists of variables are also useful while debugging ST and IL text programs. The user can easily group in a list the set of variables used in a program, in order to control or monitor the execution of the programmed instructions.

For each variable of the list, ISaGRAF displays its name, its current value and its comment text. Columns can be resized by dragging separation lines with mouse in the list title bar.

Saving lists on hard disk

The commands of the "**File**" menu are used to create, open and save the lists of variables. The number of lists for one project is not limited by ISaGRAF. While naming the lists of variables to be saved on disk, the rules shown below have to be followed:

- name cannot exceed **8** characters
- the first character must be a **letter**
- the following characters can be **letters**, **digits** or underscore character
- naming of lists is case insensitive

The list editor cannot display more than one list of variables at a time in the same window. However, the list editor can be run more than once, in order to spy different lists simultaneously.



Inserting variables in the list

The "**Edit / Insert**" command inserts another variable in the list. The variable name is selected in the list of objects defined in the project dictionary. This way the user does not have to manually enter the identifier. The variable is inserted before the variable currently selected in the list. The list cannot contain more than **32** variables. The same variable cannot appear more than once in the same list.



Changing the selected variable

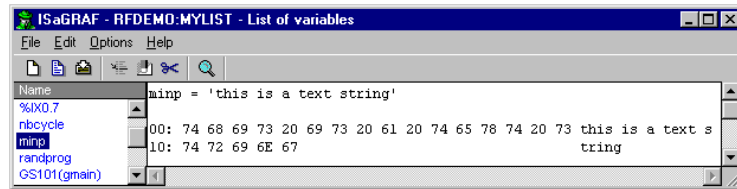
The "**Edit / Modify**" command replaces the selected variable by another variable. You can also use the "**Cut**" command to remove the selected variable from the list.



Dump display

At any time, you can swap viewing mode between list and "Dump" view. Press the "zoom" button in toolbar or use "**Options / Dump**" command to swap viewing mode.

In "Dump" mode, only one variable value is displayed. Its value is displayed in numerical/symbolic format at the top of the window, and is also displayed in binary "dump" format. This mode allows you to spy hexadecimal value of each byte in the variable value.



"Dump" display is very useful for spying and understanding message strings containing non-printable characters.

A.17 Debugging ST and IL programs

During simulation or On Line debugging of ST and IL program, no modification can be entered in the program text.

IL For IL programs, instructions are formatted in a list view. Current value of a variable used in an instruction is displayed on the same line. You can double click on an instruction to change the value of the corresponding variable.

ST For ST programs, a Spy List window is embedded in the editor window. You can resize views by dragging with the mouse the separation line between them.

For each variable of the list, ISaGRAF displays its name, its current value and its comment text. Columns can be resized by dragging separation lines with mouse in the list title bar.



Saving list on hard disk

The "**File / Save list**" command save the lists of variables on the disk, under the same name as the edited program. This list will be automatically re-loaded each time ST or IL program is open in debug mode. This list can also be freely open and modified using the Spy List tool run by the "**Spy / Spy list**" command of the debugger window.



Inserting variables in the list

The "**Edit / Insert variable**" command inserts another variable in the list. The variable name is selected in the list of objects defined in the project dictionary. This way the user does not have to manually enter the identifier. The variable is inserted before the variable currently selected in the list. The list cannot contain more than **32** variables. The same variable cannot appear more than once in the same list.



When the name of a variable is highlighted in ST text, press this button in the toolbar or run the "**Edit / Spy selection**" command to directly send the variable to embedded spy list.



Changing the selected variable

The "**Edit / Change variable**" command replaces the selected variable by another variable. You can also use the "**Cut variable**" command to remove the selected variable from the list.

A.18 Debugging with SpotLight

ISaGRAF SpotLight tool allows the user to define watch lists that can be displayed either as graphic pictures or as lists during debug. Graphic items must be linked to the variables of the ISaGRAF project. The graphic picture is both defined and animated "on line".

To force the value of a variable, double click on the corresponding item from graphic or list layout, or hit ENTER when it is selected.

You also can lock the document (deny any modification) using the "**File / Lock**" command. When a document is locked, you still can force variables by double clicking on their symbol.

A.18.1 Building the graphic layout

A chart is made of background pictures (bitmaps or metafiles), and a set of graphic items that will be animated during debug. To enter the chart, the following operations must be performed: Insert background pictures, insert graphic items, link objects to the variables of the project



Background pictures

The background pictures are "bitmap" (.BMP) or "metafile" (.WMF) files. Numbers of pictures included in the graphic layout is not limited. Pictures can be moved or resized in graphic layout. They do not appear in list layout. Pictures are built with other tools. SpotLight does not include a painting tool. The "**Options / Background colour**" command is used to select a solid colour for empty space in graphic layout.

Note: Bitmaps consume a large amount of memory. It is highly recommended to correctly size the picture, and limit the unused space inside the bitmap rectangle.



Single text display

A "single text" item is a text written in a rectangle. The text displayed is the value of the attached variable. Thus, such item can be linked to message string variable. The rectangle where text is displayed can be either filled with colour or transparent. The character font used to display text is adjusted to fit the height of the rectangle when item is resized.

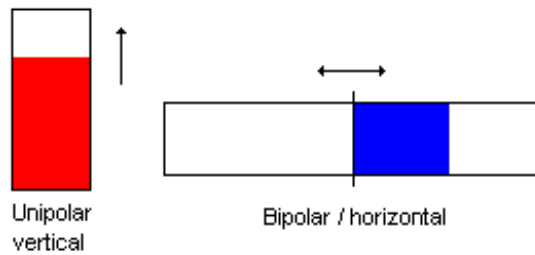


Unipolar and bipolar bar graphs

A bar graph is a rectangle with a coloured part that represents the numerical value of the attached variable. Optionally, the rest of the rectangle can be filled with colour. A bar graph can be either horizontal or vertical.

Unipolar bar graphs can grow in any direction: to the top, to the bottom, to the left or to the right.

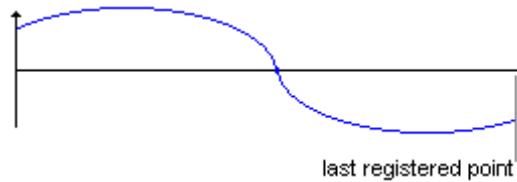
Bipolar bar graphs can grow either in positive or negative direction, according to the value of attached variable. In case of a bipolar bar graph, the maximum allowed value is the same for both negative and positive scales.



Curves

It is possible to insert a curve in a document. A curve shows the history of the attached variable. Although it is not a precise measurement tool, it can give useful debug information about synchronism between various variables.

A curve stores the 200 last values of a variable. The number of samples is not changed when the curve item is resized in the graphic layout.



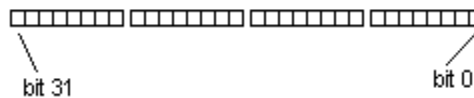
Boolean icons

A "boolean icon" item is used to display a binary state. One icon (.ICO) file is defined for FALSE or 0 value. Another icon is defined for all other non zero values. As SpotLight does not include an icon editor, icon files should be prepared with another tool.



Bit fields

A "bit field" item shows in a graphic panel the 32 bits of an integer value. The less significant bit is always displayed on the right. It is not recommended to use bit field for other data types such as real analog values, as the displayed information can lead to confusions.



Select, move or resize items

Selecting graphic objects is needed for most of the editing commands. SpotLight enables the selection of one or more existing objects in the chart area. To select objects, the **"select"** (button with an arrow) choice must be checked in the editor toolbar. To select one object, the user simply has to click on its symbol. To select a list of objects, drag the mouse in the drawing area to select a rectangle area. All the graphic objects that intersect the selection rectangle are marked as **"selected"**. A selected object is drawn with little black squares around its graphic symbol.

By making a new selection, any previously selected objects are unselected. To remove the existing selection(s), simply click with the mouse on an empty area outside of the rectangle which borders the selected objects.

To move objects, you first have to select them. Then place the mouse cursor on the border of the selected item and drag it to other location.

To resize an object, you first have to select it. Then place the mouse cursor on one of the small rectangles displayed in the selection border, and drag it in appropriate direction to resize the object. Pictures can also be resize. In such case, the corresponding bitmap or metafile is stretched to fit the new specified item rectangle.



Group items / dissociate groups

You can group items together so that they are managed as one item. To make a group, select items in graphic layout and run the **"Edit / Group"** command. The **"Edit / Dissociate"** command is used to restore items of the selected group as separated ones.

A group may contain a picture. A group may also contain another group.

When items are grouped, their style cannot be changed anymore. Items of the group are still displayed, but cannot be used (with double click) to modify the value of attached variables.

A group appears at just one line in the list layout.

A.18.2 The list layout



At any time, you can swap between graphic and list layout, by pressing this button. You can also use the **"Options / List - Graphic layout"** command.

In the list layout, items are shown in a classical list box. The height of each item is calculated according to its drawing style. Pictures (bitmaps and metafile) are not visible from the list layout. A selection is available in list layout, and should be used to set item style or change the value of a variable. Multiple selection and commands using it are not available in this mode.



You can re-order the items in the list using the **"Edit / Move in list"** commands. The item to be moved should be selected in the list.

A.18.3 Defining the item style

The graphic style and settings of an existing item can be modified, by double clicking on its symbol in the graphic area, or by running the **"Edit / Set style"** command when item is selected in graphic or list layout. The "Style" dialog box is also opened when a new item is added to the document. It groups the following pieces of information to be selected by the user:

- ☐ **Graphic style and settings:**

The display style (single text, bar graph, curve...) of an item can be changed dynamically. When foreground and background colours are used, they can be customised using the corresponding boxes. When style is "boolean icon", the pathname of corresponding .ICO files has to be specified. Use "..." buttons close to these controls to browse icon files existing on the disk.
- ☐ **Scale:**

This is the maximum value that can be displayed in bar graphs and curves. For bipolar bar graphs and curves, the same absolute value is used for both positive and negative axis.
- ☐ **Variable name:**

When the "Name" field is the active field, pressing the "..." button close to edit control enables the user to find the names of the variables already declared in the project dictionary.
- ☐ **Caption:**

A caption can be displayed closed to the graphic item in graphic layout. You can customise the location of the caption text (top, bottom, left or right) and its contents. Caption can be any combination of the variable name and its value formatted as text. Caption customisation has no effect on list layout.
- ☐ **Command variable:**

If the "Command variable" option is set, the user can modify the value of the linked variable during debug by double clicking on the item graphic symbol.

A.18.4 Commands of the "File" menu

The "File" menu contains commands that allow the user to manage the complete document.



The "New" command of the "File" menu starts the editing of a new document. The number of documents defined for a project is not limited by ISaGRAF. Before editing the new chart, the previously opened chart is closed. The SpotLight cannot be used to edit several charts at once. However, multiple SpotLight windows can be opened simultaneously with each used to edit a different document.



The "Open" command of the "File" menu allows the user to close the currently edited document and to start editing another document of the current project. The new selected document replaces the current one in the editing window. When selecting the new document, the "Delete" button can be used to delete an existing file, in order to clean up the project directory. Icon and bitmap files referenced in a chart are not erased when the chart is deleted.



The "**Save**" command of the "**File**" menu stores the currently edited document on the disk. If it is a new untitled document, the user must give it a name before saving it. Naming a document must conform to the following rules:

- The length of the name cannot exceed **8** characters
- The first character must be a **letter**
- The following ones must be **letters, digits** or **underscore** characters
- Naming is case insensitive

The "**Save as**" command of the "**File**" menu allows the user to store the currently edited document under another name.

A.18.5 Note for ISaGRAF V3.2 users

Spotlight can read graphics and lists of time diagrams built with the tools of ISaGRAF V3.0 or V3.2. Such files appear in the "**Open**" dialog box, with the description of their origin. Files can be read and freely modified with SpotLight.

When opening an ISaGRAF V3.2 graphic, the document is automatically marked as "Locked". Remove the "Lock" option from the "**File**" menu if you want to make changes in the graphic.

When an ISaGRAF 3.2 graphic or list of time diagram is open, SpotLight always proposes to save it in native SpotLight format. The "**Save As**" dialog box is systematically open when closing such a document.

A.19 Uploading applications

ISaGRAF supports the uploading of the application stored in the target. The upload procedure communicates with the target to load the embedded zipped source code (EZS) and then restore the loaded project in the workbench environment.

The project running on the connected target system can be uploaded if the target version is V3.22 or later, and if zipped source code have been embedded with the application. Embedding source code for upload is an optional feature.

A.19.1 Uploading a project

The **"Upload"** dialog box is run from the **"Files"** command of the ISaGRAF Project Manager. Upload does not refer to an existing project on the Workbench. The currently selected project in project management list has no relationship with upload mechanism. To upload the application running on the target you must:

- 1- ensure that the target is properly connected
- 2- set-up the communication parameters according to the connection link
- 3- press the **"Run"** button

Uploading embedded zipped source (EZS) and decompressing them may take few seconds. Messages in the dialog box will inform you when upload is complete, or in case of error.

The name used to create the ISaGRAF project is the one read in the target through communication. If this name is already used for an existing project in the workbench, you will be prompt to either overwrite it or select an unused name. You cannot cancel the registration of loaded sources as a project when upload is complete. The uploaded project is now ready and can be opened.



Possible errors

The following errors may occur when uploading a project. You are informed of the error in the "Upload" dialog box.

- communication cannot be established with the target
- connected target is an ISaGRAF system before version 3.22
- there is no application running in the target
- there is no EZS embedded in the target

A.19.2 Communication settings

Pressing the **"Set-up"** button enables the user to define the parameters of the link used for communication for upload between ISaGRAF workbench and the target ISaGRAF system. You have to ensure that the configured parameters match to the connected target before running upload.

A.19.3 Preparing a project for upload

You have to inform the ISaGRAF Code Generator that zipped source code must be embedded with the application code if you want to enable upload later. For this, press the "Upload" button in the "Compiler options" dialog box. Another dialog box enables you to check, as an option, the embedding of zipped source code. In this case, only minimum required source files will be embedded. Use other check boxes to embed also optional files.

Important note: Libraries are not downloaded with embedded source code. This includes functions and function blocks and I/O boards and equipments.

Optional files

In addition to the minimum required source code, the following files can also be embedded. They are options as their selection leads to extra memory requirement on the target.

Project descriptor: If not embedded, the project descriptor after upload will just indicate the upload date.

Password protection: Upload function is not protected by a password. If you want the uploaded project protected, you have to embed its password protection system with source code.

Comments for not connected I/O channels: ISaGRAF gives you the possibility to enter description text for non-connected I/O channels. Do not check this option is you work with connected I/Os only.

History of modifications: This is the global history of modifications for the project.

Diary files: Diary file of each program contains user written notes plus the history of compiler output messages referring to the program. Embedding diary files may consume a lot of memory in target.

Lists of variables and time diagrams: These are the files created during debug, and containing lists of variable names for list or time diagram monitoring.

Graphics, icons and bitmaps: This includes ISaGRAF graphics, plus all attached icon and bitmap files, if they are located in the project directory. Warning: embedding diary files may consume a lot of memory in target.

A.19.4 How zipped source are stored in the target

Embedded zipped source (EZS) is stored in generated code with resources. The generated resource is called "EZS". If source code embedding is selected, you cannot choose this name for another resource. Embedding source code does not imply any limitation in resource definition. The user written resource definition file is not affected by source embedding.

Please refer to the ISaGRAF documentation about the Code Generator for further details and information about resources.

A.19.5 Memory requirements on the target

Embedded zipped source (EZS) code requires extra memory to be stored with application code in the target. A general rough estimation is that minimum EZS (no extra option selected for source embedding) has one and a half the size of the executable code. This means that the embedding of EZS will multiply the size of downloaded code by 2.5.

Special limitation may appear on some target system based on segmented memory. As EZS are stored as resources in generated code, they must be stored in the same data segment as the application code.

A.19.6 About uploaded project

The uploaded project contains all the files and data required for re-compiling. Depending on the options selected during its previous compiling, it may also contain auxiliary files such as project descriptor and program diary files.

You have to compile (make) the project before debugging or monitoring it. **Warning:** as ISaGRAF uses the compiling date stamp to compare applications, you will be informed when opening the debugger that workbench and target applications have different version codes.

Important note: Libraries are not downloaded with embedded source code. You have to ensure that the appropriate library functions and function blocks are installed with your ISaGRAF workbench before re-compiling the uploaded application.

A.19.7 Compatibility issues

Upload is supported by ISaGRAF target and workbench version 3.22 or later. Extensions have been made to the communication protocol to support upload.

There is no restriction in embedding zipped source code (EZS) in a target based on ISaGRAF systems version 3.03 to 3.21, as EZS are stored in application code as standard resources. But embedded information cannot be uploaded in this case, as such target does not support required communication services.

A.20 Using the Diagnosis tool

The "**Diagnosis Tool**" is a subset under the ISaGRAF debugger tool. It enables the end user to work on a predefined set of variables, in order to examine and control the process. The ISaGRAF debugger is a very powerful tool, which includes high level functions. The Diagnosis Tool provides a safe way to control the target application for final running operations or maintenance. The ISaGRAF Diagnosis Tool is run directly from the ISaGRAF group in Program Manager, by double clicking on the following icon:



Diagnosis

The list of existing projects is displayed in a dialog box. It enables the user to run the limited ISaGRAF debugger on an existing, already downloaded ISaGRAF application. Pressing the "**OK**" button starts the limited debugger on the selected project. Pressing the "**Cancel**" button closes the dialog box. The "**Set-up**" command is used to set-up the communication link between the ISaGRAF Workbench and the target PLC. Refer to the "**Managing programs**" chapter of this manual for more information about this command.

Note: The ISaGRAF Diagnosis Tool (limited debugger) cannot be used to download, stop or update the application running in the target PLC. No operation can be performed if the project selected in the Diagnosis Tool dialog box is not the same as the one installed and running in the PLC.

When the limited ISaGRAF debugger is run, and correctly connected to the target application, the following commands are available:

- Spy lists of variables
- Spy graphic documents with SpotLight

A.21 Using the ISaGRAF simulator




The ISaGRAF Kernel simulator is started with the debugger when the "Simulate" command of the "Debug" menu in the Program Management window is run. The kernel simulator is a complete ISaGRAF target system supporting ISaGRAF standard features and all the "C" functions and function blocks of the standard library delivered by CJ International. The I/O boards are graphically simulated in a window. Any type of I/O board can be simulated. The boards defined as "Virtual boards" during the I/O connection also appear in the simulation window.




A.21.1 Links with the debugger

The kernel simulator supports full communication with the ISaGRAF debugger, so any of the debug possibilities can be used during simulation. The kernel simulator always works on the current ISaGRAF application. During simulation, the debugger commands "**Start**", "**Stop**", "**Download**" or "**Update**" are no longer available. The simulator cannot be used if the "**SIMULATE**" target was not selected in compiler options before building the target code. Closing the simulator window implies that the debugger window (and any ISaGRAF window opened during the debug session) is also closed.

A.21.2 I/O simulation

I/O boards appear in the simulator window, titled by their name and slot number. Any of the ISaGRAF standard types of I/Os (boolean, analog or message) are handled. The channels of the input boards are displayed with special buttons and fields. The channels of the output boards are displayed with graphic status lights and data fields.

-  **Boolean inputs:** A boolean input is represented by a square green button. The number of the channel is displayed with the I/O button. The input value is TRUE when the button is pressed. Clicking on the button changes the corresponding I/O value. Use the right button of the mouse to set the input only when the button is pressed.
-  **Boolean outputs:** A boolean output is represented by a small circle. The number of the channel is displayed with the I/O. The output value is TRUE when the graphic symbol is highlighted.
-  **Analog inputs:** An analog input channel is a simple numerical field, where the value of the corresponding input can be entered. Clicking on the box displays the caret. A new value for the channel can then be entered. It is not necessary to use the **ENTER** key after input. Analog inputs can be entered in either decimal or hexadecimal base. Use up/down buttons to increase or decrease the current value.

-  **Analog outputs:** An analog output channel is a numerical output field. The output value can be displayed as either a decimal or hexadecimal number. No action can be performed by the user on an output channel.
-  **Message inputs:** A message input channel is a simple text field, where the value of the corresponding input is entered. Clicking on the box displays the caret. A new value for the channel can then be entered. It is not necessary to use the **ENTER** key after input.
-  **Message outputs:** A message output channel is a text output field. No action can be performed by the user on an output channel.

A.21.3 Library components

The ISaGRAF simulator fully supports the standard conversions, functions and function blocks, delivered by CJ International. Below is the list of supported objects:

▣ **Conversion functions:**

bcd, scale

▣ **Functions:**

abs, acos, ArCreate, ArRead, ArWrite, ascii, asin, atan, char, cos, delete, expt, find, insert, left, limit, log, max, mid, min, mlen, mod, mux4, mux8, odd, rand, replace, right, rol, ror, sel, shl, shr, sin, sqrt, tan, trunc

▣ **Function blocks:**

average, blink, cmp, ctd, ctu, ctud, derivate, f_trig, hyster, integral, lim_alm, r_trig, rs, sema, sr, stackint, tof, ton, tp

User defined conversions, "C" functions and function blocks are commonly not integrated with the ISaGRAF Simulator. Typically, such objects are designed to use software and hardware resources of the target system. Such resources are generally not available on the Windows system. The ISaGRAF Simulator provides the following standard behaviour for any user defined conversion, function or function block:

- When a new conversion is processed by the simulator, it is replaced by a "null" conversion. This means that the physical value of the analog variables is always equal to the electrical value (as entered or displayed on the Simulator panel).
- When a new "C" function or function block is run by the simulator, it does not process any operation. The result value is not set.

A.21.4 Options

The commands of the **"Options"** menu enable the user to control the display of I/Os in the simulator panel. The user can set or remove these options at any time during debug.

- ⇒ When the "**Colour display**" option is set, I/O channels are displayed as colour bitmaps. If colours cannot be distinguished on some LCD screens, the user should remove this option, to get pure black and white input and output graphics for I/O channels.
- ⇒ When the "**Variable names**" option is set, a sticker is displayed beside any I/O channel, with the name of the connected I/O variable. Removing this option enables the user to reduce the size of the simulator panel.
- ⇒ When the "**Hexadecimal values**" option is set, any input or output analog channel is displayed or entered in hexadecimal format.
- ⇒ When the "**Always on top**" option is set, the simulator window is always visible, even if the input focus is on another window.

A.21.5 Saving and restoring input states

Using the ISaGRAF simulator, input channels are forced through manual operations, acting on toggle buttons and edit controls of the simulation panel. You can at any time use the following commands of the "**Tools**" menu to save and restore the state of all input channels:

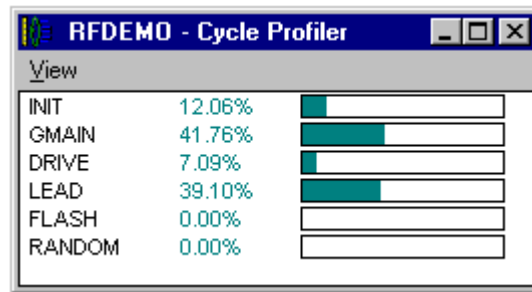
Load input scheme	Set values of input channels with values stored in a file that has been created on disk by "Save input scheme" command.
Save input scheme	Save state of input channels in a file so that they can be restored later using the "Load input scheme" command. File is stored in the project directory and thus is saved with other project files by the ISaGRAF archive utility.

Note: Only named input channels (the ones having a variable connected) are saved on disk.

A.21.6 The cycle profiler

The ISaGRAF Cycle Profiler is a powerful diagnostic tool that shows how cycle time is distributed between various programs, functions and function blocks of an application. This tool is very useful to have a quick diagnostic on the application performances, and leads the programmer to the parts of code which may need optimisations.

The Cycle Profiler is run by the "**Tools / Cycle Profiler**" command in the menus of the ISaGRAF Simulator window. It displays, for each program, function or function block, the percentage of the cycle time spent to execute it:



When the "**View / Average**" option is set, displayed information is an average of percentages calculated since the application has been started, or since the last time the "**View / Reset**" command has been run.

If the "**View / Average**" option is not set, displayed information shows measurements done during the execution of the last cycle. You can also use this feature when the application is in "**Cycle to Cycle**" mode to have a set of measurements depending on the application context.

Use the "**View / Copy**" command to copy program names and percentages to the Windows Clipboard in ASCII format. Then, data can be pasted into text documents or common SpreadSheets.

Important notes:

These are not precise measurements. Percentage calculation is based on TIC instructions counting, taking into account various instruction execution times. Calculation does not include the time spent in "C" functions and function blocks.

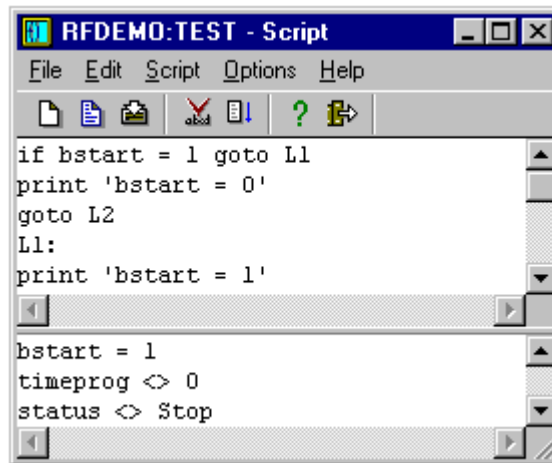
The value displayed for a function or a function block is the sum of all the "calling times" from the application programs in the same cycle.

Time calculation is based on TIC code and does not provide reliable information if the actual application code is generated in "C" language and built using a "C" compiler.

A.21.7 Simulation scripts

ISaGRAF simulator includes a tool to build and run simulation script. A script is described with an easy ST like text language, and is used to automate tests with the ISaGRAF simulator.

The simulation script editor is run by the "**Tools / Simulation scripts**" command of the Simulator window. Below is the frame of the script editor:



The upper window is a text editor where script instructions are entered. It is used as other ISaGRAF text editors and includes high level features such as mouse selection of a variable symbol. You can use commands of the "Options" menu to set-up tab width and select a character font.

The lower window shows all the messages output when the script is run. The separation line between windows can be freely dragged to resize windows. The output window can be hidden during script editing, but is automatically open each time a script is run.

Editing scripts

Use the commands of the "File" menu to manage script files:

- New** creates a new untitled script
- Open** loads an existing script from file
- Save** saves script text and contents of output window to disk, in project directory
- Save as** saves script under another name

Two files are created in the ISaGRAF project directory for each script:

- <scriptname>.SCCtext of the script (instructions)
- <scriptname>.SCOcontents of the output window

where <scriptname> is the name of the script. Both files are standard text files and can be open using any other text editor.




While editing a script, you can use the "Edit / Insert symbol" command to select a declared variable name to be inserted at the caret position.

Running scripts

Script must be checked and compiled before running it. If necessary, syntax checking is automatically performed on a "Run" command. Use the following commands of the "Script" menu:




- Check**check syntax and compile script

 **Run Script**.....start execution of the script currently edited

In the case of a new untitled script, it must be saved (and a name must be entered for it) before it is checked. In case of a named script, script is automatically saved to disk before syntax checking.

When script is running, its contents cannot be changed. A message is displayed when end of script is reached. You can also abort a running script using the following command of the "**Script**" menu:

 **Abort Script**.....terminates the running script

Script execution is performed between target cycles. In the case of an infinite loop programmed in the cycle, ISaGRAF simulator ensures that this loop is always broken so that ISaGRAF cycles are still executed and other ISaGRAF applications are not blocked. The ISaGRAF script interpreter decides to break script execution if the same "label" is encountered more than once in the same target cycle. Script execution can also be normally broken by "Cycle" or "Wait" instructions.

Script description language

Script description language is a very simple text language similar to ST, but where each instruction is entered on a separate text line, and does not need to be terminated by a semicolon. Use the following button of the toolbar to know the list of available instructions and to insert a keyword at the caret position:

 insert instruction (keyword and help as comments)

There are various types of instructions. First is the assignment (forcing) of a variable:

:= assignment

Other instructions allow the output of messages to the output window:

Print..... outputs a text string or a variable value

PrintTime outputs current time stamp

Other instructions are used to synchronise script instructions with ISaGRAF cycle:

Cycle..... let ISaGRAF simulator execute one cycle

Wait waits during a specified time

Other instructions are used to control instruction flow in the script:

Labels..... can be placed anywhere in the script

Goto..... unconditional jump to a label

If goto conditional jump to a label

End terminates script

Script language is not case sensitive. Comments can be inserted at the end of any text line. Comments can either be written according to ST conventions (between "(" and ")" characters), or prefixed by a ";" character.

":=" Assignment

Meaning: Force the value of an ISaGRAF variable. It can be an internal variable, an input channel or an output channel.

Syntax: `<varname> := <constant_expression>`
`<varname> = <constant_expression>`

Arguments: `<varname>` is a valid symbol of a declared application variable, or a directly represented I/O variable using "%" writing conventions.

`<constant_expression>` is a valid constant expression that matches the type of the specified variable. For booleans, "0" and "1" can be used instead of "FALSE" and "TRUE". For timers, the "T#" or "TIME#" prefix can be omitted.

Notes: Input variable forced by a script does not need to be locked. The drawing of the corresponding input channel is updated when input variable is forced by a script.

Warning: do not force input or output analog variable attached to a conversion, as script execution does not support conversion functions or tables.

Example: `MyBooVar := 1 (* same as TRUE *)`
`MyIntVar := 1234`
`MyRealVar := 1.2345`
`MyMsgVar := 'Hello'`
`MyTmrVar := t#12s`

Print

Meaning: Writes a string or the value of a variable in the output window. Text is output as one new line at the end of text already written in output window.

Syntax: `Print '<text>'`
`Print <varname>`

Arguments: `<text>` is any text string expressed between single quotes

`<varname>` is the valid symbol of a declared application variable, or a directly represented I/O variable using "%" writing conventions.

Notes: Output of variable values is always formatted according to IEC conventions.

Example: `Print 'Hello'`
`Print MyBooVar`

Output: Hello
MyBoovar = TRUE

PrintTime

Meaning: Writes the current time stamp in the output window. Text is output as one new line at the end of text already written in output window.

Syntax: **PrintTime**

Notes: Time stamp is formatted according to current setting of Windows System

Example: Print 'Time now is:'
PrintTime

Output: Time now is:
15:45:22

Cycle

Meaning: Suspends the execution of the script until the next ISaGRAF cycle is performed.

Syntax: **Cycle**

Notes: Script instructions are executed at the beginning of an ISaGRAF cycle. If the simulator is in "Cycle to Cycle" mode, the "cycle" instruction is immediately followed by a cycle. The following instructions of the script will be performed on the next "Execute one cycle" command from the debugger.

Example: (* the ISaGRAF program copies A to B *)
A := 0
Cycle
Print B
A := 1
Print B (* no cycle performed / B not set to 1 *)
Cycle
Print B

Output: B = 0
B = 0
B = 1

Wait

Meaning: Suspends the execution of the script until a delay is elapsed

Syntax: **Wait** <delay>

Arguments: <delay> delay expressed according to IEC conventions for time constant expression. The "T#" or "TIME#" prefix can be omitted. Delay value must be between 10 milliseconds and 1 hour.

Notes: Accuracy of the "Wait" instruction is not precise as it depends on the host Windows system. Also, the delay should be considered with an accuracy of plus or minus one ISaGRAF cycle. When a "Wait" instruction is reached, ISaGRAF cycles are performed until the delay is elapsed and before continuing the script execution.

Example:
PrintTime
Wait 2s
PrintTime

Output:
15:45:27
15:45:29

Labels

Meaning: Labels can be placed anywhere in the script. They are used as a destination by "Goto" instructions and allow flow control for script instructions.

Syntax: <labelname>:

Arguments: <labelname> unique name according to ISaGRAF variable naming conventions: limited to 16 characters, beginning with a letter, followed by letters, digits or underscore characters. When defined, label name should be followed by a ":" character.

Notes: No instruction should be placed on the line where a label is defined. Label name should not be the same as a declared ISaGRAF variable symbol

Example:
(* example of a script with an infinite loop *)
loop:
PrintTime
Wait 1s
Goto loop

Goto

Meaning: Unconditional jump to a label

Syntax: **Goto** <labelname>

Arguments: <labelname> is the name of a label defined in the script.

Notes: Backward jumps are allowed. In case of an infinite loop, script execution is automatically broken on each loop in order to preserve execution of ISaGRAF cycles.

Example:

```
Print 'Before Jump'
Goto MyLabel
Print 'Within Jump' (*never performed *)
MyLabel:
Print 'After Jump'
```

Output:

```
Before Jump
After Jump
```

If Goto

Meaning: Conditional jump to a label. The condition is either a comparison between two ISaGRAF variables, or a comparison between a variable and a constant expression.

Syntax:

```
If <var1> test <var2> Goto <labelname>
If <var1> test <constant_expr> Goto <labelname>
```

Available comparison **tests** are:

```
= true if both members have same value
<> true if members have different values
< true if first member is less than second
<= true if first member is less than or equal to
second member
> true if first member is greater than second
>= true if first member is greater than or equal
to second member
```

Arguments: <var1> <var2> are valid symbols of declared application variables, or directly represented I/O variables using "%" writing conventions.

<constant_expr> is a valid constant expression that matches the type of specified variable. For booleans, "0" and "1" can be used instead of "FALSE" and "TRUE". For timers, the "T#" or "TIME#" prefix can be omitted.

<labelname> is the name of a label defined in the script.

Notes: Backward jumps are allowed. In case of an infinite loop, script execution is automatically broken on each loop in order to preserve execution of ISaGRAF cycles.

Example:

```
(* This script loops until MyVar is TRUE *)
Loop:
If MyVar = TRUE Goto TheEnd
```

```
Print MyVar  
Goto Loop  
TheEnd:
```

End

Meaning: Terminates script

Syntax: **End**

Notes: It is not mandatory to place an "End" instruction on the last line of the script

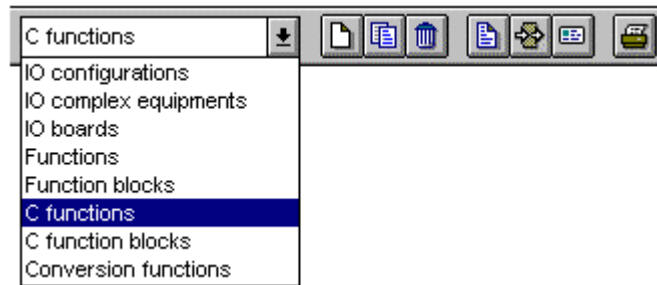
Example:

```
(* This script loops until MyVar is TRUE *)  
Loop:  
If MyVar = FALSE Goto Continue  
End  
Continue:  
Print MyVar  
Goto Loop
```

A.22 Using the Library Manager

The ISaGRAF libraries provide a standard interface between automation development and the software or hardware capabilities of the ISaGRAF target system. There is one library for each type of interface. The ISaGRAF Workbench Library Manager is dedicated to the hardware supplier, or to the software engineer. He uses the library manager to describe the ISaGRAF programming interface of the objects he creates.

The ISaGRAF Workbench Library Manager shows the elements of one of the ISaGRAF libraries. In the left area of the window is the **list of the elements** of the selected library. In the right area is the **technical note** (user manual) of the element currently selected on the element list. The menus of the library manager contain the commands to create, define or modify elements of the active library. The **"File / Other library"** command allows the selection of one of the ISaGRAF libraries. The combo box on the left of the toolbar can also be used to select a library:



A.22.1 Managing library elements

Use the commands of the **"File"** menu to create elements and work on existing ones in the open library



Creating a new element

The **"New"** command of the **"File"** menu creates a new element into the selected library. The name of the new element is entered, based on the following naming rules:

- the maximum length of a name is **8** characters
- the first character must be a **letter**
- the following characters must be **letters, digits** or **'_'** character
- the naming of a library element is case insensitive.

A text comment is associated to each library element. This comment is entered while creating the element. When a new element is created, the following must be entered:

- its definition for an I/O configuration,
- its parameters for an I/O board,

- its user interface for a function or function block.

When a "C" conversion, "C" function or "C" function block is created, a complete frame of its source code is automatically generated.

Working on existing elements

The "**File / Rename**" command allows the user to change the name or the comment of the element selected from the list of elements. The "**File / Copy**" command allows the user to copy the element highlighted in the active library on another element in the same library. If the destination element already exists, all its contents are overwritten. If the destination element does not exist, it is automatically created. The "**File / Delete**" command removes the currently selected element from the active library. The following components of the element are handled by "**Rename**", "**Copy**" and "**Delete**" commands:

- technical note
- complete definition for an I/O configuration
- parameters for an I/O board or complex equipment
- interface definition for a function or function block
- source code for function and function block written in IEC language
- source code for a C conversion, a function or a function block



If the element is a "C" conversion, "C" function or "C" function block, its name is not automatically updated in the attached source code by a "**Rename**" or "**Copy**" command.



If the element is a function written in IEC language, the return parameter name is not changed by a "**Rename**" or "**Copy**" command.

Setting password protection

The "**File / Set password**" command enables the user to define password protection for the selected element in the open library. Refer to the "**Password protection**" section, at the end of the first part in this manual for further information about password levels and data protection. Passwords are only relative to the selected element. They have no influence on other elements of ISaGRAF libraries.

Compiling functions and function blocks

When the library of functions or function blocks written in IEC languages is selected, the "**Verify (compile)**" command of the "**File**" menu is used to check the syntax of the selected element and create its object code. Functions and blocks written in IEC languages have to be compiled without errors before they can be used in ISaGRAF projects. This command has no effect if another library is selected.



Technical notes

The "**Edit / Technical note**" command allows the user to enter the technical note of the element selected in the active library. The technical note is entered with the ISaGRAF text editor. The technical note of an element is its **user guide**. It will be consulted by the user of the element during its integration in an ISaGRAF project. The technical note on how to use the element should contain the description of its

main function, the detailed explanation of its programming interface and parameters, and its context and limits.

The "**Tools / Standard note format**" command allows the user to define a standard text format for all the elements of the currently selected library. When editing the technical note for a new element, this format is used as a main frame. This allows the user to optimise technical note editing.



Parameters

The parameters of an element describe the **interface** between the computer operations provided by the element and the use of the element in an ISaGRAF application. Parameters have a different meaning for each type of library element.

The parameters of an I/O configuration define the complete set of I/O boards of the configuration, and default variable names used for I/O channels. The parameters of an I/O board or complex equipment define the physical and logical configuration of the board. The parameters of a function or function block define the interface of the element, according to ST language function calling conventions. There is no parameter for a conversion function because it uses a standard pre-defined interface.



Source code

The ISaGRAF Workbench allows the programmer to manage the source code of a library conversion, function or function block. The source code of a function or a block written in IEC language is a text or a diagram described with the language attached to the function. The source code of "C" components ("C" functions, "C" function blocks and conversion functions) is divided in two separate files: a **source header** that contains the exact definition of the **interface**, according to the element's parameter definition and a **source code** file that contains the element's operation implementation.

The ISaGRAF workbench generates the source code file when a new library element is created. It also creates and updates the source header, based on the parameter definition. The programmer can use the ISaGRAF text editor to complete the source code file.



Archiving library elements

The "**Tools / Archive**" menu command runs the ISaGRAF archive manager to save or restore library elements. You first need to select a library before running the "**Archive command**". The archive manager shows list of elements for only one library at a time.

A.22.2 I/O configuration

The ISaGRAF I/O configuration library provides an easy way to initialise new ISaGRAF projects with pre-defined I/O configuration. An I/O configuration defines:

- a set of I/O boards
- default values for I/O boards parameters
- default names for I/O channels

When a new ISaGRAF project is created with a library I/O configuration, the corresponding I/O connection is automatically set, and the I/O variables

corresponding to channel names are automatically declared in the project dictionary.



The definition of an I/O configuration is made with the ISaGRAF I/O Connection tool (the same tool used within a project). Refer to the "I/O Connection" section in this manual for further information about how to use this tool. When inserting a new I/O board in the configuration, all the channels of the new board are declared with standard default names. The standard default name of an I/O channel has the following format:

<direction><type><slot_number>_<channel_number>

The first character indicates the direction of the I/O channel:

"I" input channel
"Q" output channel

The second character indicates the type of the I/O channel:

"X" boolean
"D" analog
"M" message

Below are examples of a standard I/O channel names:

IX0_7 boolean input - board #0 - channel #7
QD2_4 integer output - board #2 - channel #4

The "Connect I/O channels" command of the I/O Connection Editor is used to modify the default name attached to an I/O channel.

A.22.3 I/O complex equipment

All the channels of a single board have the same type (boolean, analog or message) and direction (input or output). A complex I/O equipment represents an I/O device with channels of different types or directions. A complex I/O equipment is represented as a list of single I/O boards. It uses only one slot in the I/O connection rack list.



To define a complex I/O equipment, the user has to define the list of single boards which define the I/O equipment. He also has to enter the detailed parameters of each single board. The list of single I/O boards is entered through a dialog box.

Pressing the "**Append**" button allows the user to add a single board at the end of the current list. The "**Insert**" button is used to insert a new single board before the one currently selected in the list. The "**Delete**" button removes the selected single board from the list. The "**Rename**" and "**Parameters**" button are used to change the name and the parameters of the selected single board. Refer to the following section for a complete explanation of single board parameters. A complex I/O equipment can group up to **16** single I/O boards. The name of a single board (within an I/O equipment) cannot exceed **8** characters.

A.22.4 I/O board

The ISaGRAF I/O board library defines a standard interface between the application variables and the target hardware. During the description of the application, all the I/O variables are connected to the channels of the target I/O boards. An ISaGRAF I/O board is defined by a **name** and an "**OEM key code**" that identifies its **supplier**. Other I/O board parameters describe the I/O board topology (number of channels, channel direction and type), and its hardware or software configuration.



I/O board parameters

There are two different types of parameters for an I/O board: common parameters which are defined for any ISaGRAF library board, and OEM parameters which are specific to the board implementation, provided by the hardware supplier. Common parameters are entered in the upper part of the I/O board parameters definition box. These parameters (plus the I/O board name) identify the ISaGRAF standard I/O board interface.

The "**OEM key code**" is a simple number that defines the **hardware supplier**. All the boards defined by the same supplier must have the same OEM key code. The OEM key code is a **16 bit unsigned word**, entered in a **hexadecimal** format. The reserved OEM key code for **CJ International** is "**1**".

Main parameters define the topology of the I/O board. The **number of channels** defines the number of available channels on the board. The **type** of the board is the type of the variables that may be connected on the channels of the board. The **direction** defines whether variables connected on the board are **input** or **output** variables.

Note: I/O variables of different types or directions cannot be grouped on the same ISaGRAF I/O board. This feature should require a complex I/O equipment.



The OEM parameters

The OEM parameters are entered in the lower part of the I/O board parameters definition box. These parameters are defined by the I/O board hardware supplier and are specific to the board. There are at most **16** OEM parameters for a board. A board may have no OEM parameters. The ISaGRAF library manager allows the hardware supplier to define the identification and the format of each parameter, and the way the automation programmer enters it.

The box on the left contains the list of the OEM parameters. Each parameter is identified by a **name** and a logical **number**, from **0** to **15**. The area on the right contains the detailed description of the parameter selected on the list. A parameter is selected in the list in order to access to its complete description. Pressing the "**Clear**" button resets the parameter description, and removes it from the parameter list. **Warning:** this command cannot be "undone".

The name of a parameter is used to identify the corresponding input field during the I/O board connection if the field must be defined by the automation operator. The name of a parameter must conform to the following rules:

- the maximum length of a name is **16** characters
- the first character must be a **letter**
- the following characters must be **letters**, **digits** or **'_'** character

The type of a parameter defines the **internal format** of the parameter, and its **input format** during application I/O connection. Below is the list of available internal formats:

word unsigned 16 bit word
long unsigned 32 bit word
word hexa unsigned 16 bit word
long hexa unsigned 32 bit word
boolean unsigned 16 bit word (only lowest bit is used)
character unsigned 16 bit word (only lowest byte is used)
string array of 16 bytes containing a null-terminated string
float single precision 32 bit floating value

Below are available input formats:

word unsigned decimal word
long decimal long word
word hexa unsigned hexadecimal word
long hexa unsigned hexadecimal long word
boolean "true" or "false"
character single character
string ascii string (15 characters max)
float single precision floating value

The "**access**" box is used to define how the parameter can be accessed by the end user. If the "**User defined**" option is set, the parameter is shown as an input field during the I/O board connection. The OEM parameter default value is used as default for the parameter editing. If the "**Hidden**" option is set, the parameter is a constant and does not appear in the I/O board connection box. The OEM parameter default value defines the value of the constant parameter. The "**Read only**" option indicates that the parameter is visible for the user, but cannot be modified. Its default value is used as a constant value.

A.22.5 Functions and blocks written in IEC languages

ISaGRAF handles a library of functions and function blocks written in IEC languages. The available languages to describe such a function or block are **FBD** (Function Block Diagram), **LD** (Ladder Diagram), **ST** (Structured Text) or **IL** (Instruction list). Note that LD and FBD languages can be mixed in the same diagram. **SFC** language (Sequential Function Chart) cannot be used to describe a function or a block in library. The language attached to a library element is selected when the function is created, and cannot be changed later.

Compiling

Functions and blocks defined in the library must be compiled (verified) before they can be used within an ISaGRAF project. Nothing else has to be changed on the Library side concerning functions and blocks. Elements of the library will automatically appear in box selection menu when using the LD/FBD graphic editor within a project.



A function defined in the library can call other functions of the library. However, the ISaGRAF system **does not support recursive function calling**. A function block written in IEC language cannot call other function blocks (neither in IEC nor in "C" language).



Entering source code

The source code of a library function or function block is entered using standard ISaGRAF tools: graphic editor for LD or FBD programs, text editor for ST or IL programs. Refer to the corresponding sections in this manual for more information about these tools. The ISaGRAF Code Generator can be directly called from the graphic or text-editing window, to compile the source code of a library function or block.



Dictionary of local variables

A library function or function block can have local variables, and local defined words. To access the variable declaration, the user must run the commands of the "**Dictionary**" command of the "**File**" menu, in the editor window, while editing the source code of the function.



A library function or function block cannot access a global variable or function block instance. Local variables of a function should be initialised in the function body.

Local variables of a function block written in IEC language are copied (instanced) each time the block is used in a project. Local variables of an instance keep their values from one call to the other.



Defining the interface

Functions or function blocks may have up to **32** parameters (input or output). A function always has one (and only one) return parameter, which must have the same name as the function, in order to conform to ST language writing conventions. The list in the upper left side of the window shows the parameters, in the order of the calling model: first the calling parameters, last the return parameters. The lower part of the window shows the detailed description of the parameter currently selected in the list. Any of the ISaGRAF data types may be used for a parameter. The return parameters must be located after calling parameters in the list. Naming parameters must conform to the following rules:

- the length of the name cannot exceed 16 characters
- the first character must be a letter
- the following characters must be letters, digits or underscore character
- naming is case insensitive

The "**Insert**" command is used to insert a new parameter before the selected parameter. The "**Delete**" command is used to erase the selected parameter. The "**Arrange**" command automatically rearranges (sorts) the parameters, so that the return parameters are put at the end of the list.

A.22.6 "C" Functions and function blocks

The "C" functions and function blocks are **computer functions** called from the automation application, according to the ST language function calling interface. Functions are **synchronous** processes. The ISaGRAF target application is suspended during the function execution. Function blocks associate operations and static hidden data. For example, a "counter" function block represents the counting operation, as well as the counting result. Functions and function blocks may be used to complete the standard automation language capabilities, or to access system resources.



The parameters definition box is used to define the name and the type of each calling or return parameter of the function or function block. The **"Edit"** menu commands are used to define the parameters of the selected function or function block. A function can have up to **31** calling parameters, and always has **one** return parameter. A function block can have up to **32** parameters, with any mix of call and return parameters. Below is the correspondence between ISaGRAF types and "C" types:

BOOLEAN	unsigned long	unsigned 32 bit word: 1=true / 0=false
ANALOG	long	signed integer 32 bit word
REAL	float	single precision floating value
TIMER	unsigned long	unsigned integer 32 bit word (unit is 1 ms)
MESSAGE	char *	character string.

When a message value is passed onto a "C" function or function block, it cannot contain null characters. The string passed to the "C" code is null-terminated. Refer to the ISaGRAF Target User's Guide for further information on how to manage the "C" source code of a function or a function block, and how to integrate a new element in the ISaGRAF target system.

A.22.7 Conversion functions

A conversion function is a "C" function called by the ISaGRAF I/O manager each time the analog variables using this conversion are input to or output from the project.

The function creates the relationship between the **electrical value** of the variable (read on the input sensor or sent to the output device) and its **physical value** (used in the application expressions). The function is therefore divided into two parts: input conversion and output conversion. The ISaGRAF library manager allows the user to control the "C" source code of a conversion function.

A conversion can be used for an **integer** or **real** analog variable. This implies that the conversion function interface is always defined by floating values. The interface is the same for any conversion function. The "C" definition of this interface is made in the **"TACNODEF.H"** definition file.

Refer to the ISaGRAF Target User's Guide for further information on how to manage the "C" source code of a conversion function, and how to integrate a new element in the ISaGRAF target system.

A.23 Using the Archive utility

The ISaGRAF archive utility enables the user to save the ISaGRAF projects and libraries on diskettes or backup directory. The ISaGRAF archive manager is a dialog box that can be called from ISaGRAF Project Management or Library Management windows.



To create and maintain reliable archives, it is suggested that the following guidelines be used:

- Write the name and description of the saved object on the disk sticker
- Do not save projects and libraries on the same diskette
- Do not save different projects on the same diskette

A.23.1 Calling the archive manager

The "Archive" dialog box can be called from the "**Tools / Archive**" menu of the Project Management window, to save or restore either a project, or common data.

The "Archive" dialog box can also be run from the "**Tools / Archive**" command of the ISaGRAF Library Manager, to save or restore elements of the library currently selected in the Library Management window.

▢ **Projects**

A project is always saved in its entire form. All the components of the project (program source files, object code and application executable code) are saved together in the same archive file. Selection of the "**compression**" option reduces the size of the project archive.

▢ **Library elements**

The elements of ISaGRAF libraries can be saved individually. All the components of a library element (technical note, definition, interface, source code...) are saved together in the same archive file.

▢ **Common data**

The "**Tool / Archive / Common data**" command of the Project Management window enables the user to backup or restore the "common range" data existing in the ISaGRAF Workbench. This command does not act on the ISaGRAF libraries. Below is the list of the files that can be copied with this command:

common.eqv common defined words
oem.bat user defined MS-DOS command file

These files are saved one by one on the archive disk, in their original form. The corresponding archive files are never compressed.

A.23.2 Options

The path used for ISaGRAF archives is displayed at the bottom of the dialog box. Press the "Browse" button to browse the disks and select another archive disk and directory.



When the "**Compression**" option is set, all the archive files created during a "**Backup**" procedure are compressed. This option is very useful to reduce the size of a large project archive file, and save it on only one diskette. Archive compression is generally not needed for library components. The ISaGRAF Archive Manager automatically recognises the status of an archive file (compressed or not) when restoring the archive. This implies that the "**compression**" option has no effect for a "**Restore**" procedure.



A.23.3 Backup and restore

The "**Workbench**" list (on the left) shows the objects existing in the ISaGRAF Workbench installed on the hard disk. The "**Archive**" list (on the right) shows the objects saved on the specified archive disk and directory.

Backup

Saving an object on archive is achieved by selecting the object in the list on the **left** (objects of the ISaGRAF workbench) and pressing the "**Backup**" button. More than one object on the list can be selected. The "**Backup**" button is disabled when an element is selected from the list on the **right** (restore mode).

Restore

Copying an object from the archive to the ISaGRAF Workbench is achieved by selecting the object in the list on the **right** (archive objects) and pressing the "**Restore**" button. More than one object on the list can be selected. The "**Restore**" button is disabled when an element is selected from the list on the **left** (backup mode).

A.23.4 Archive files

The ISaGRAF archive manager creates a unique archive file for each saved object. The archive file has the same name as the object. Its file suffix indicates its type. Below are the used suffixes:
.pia..... project

.bia..... I/O board
.iia..... function in IEC language
.aia..... function block in IEC language
.uia..... C function
.fia..... C function block
.cia..... C conversion function
.ria..... I/O configuration
.xia..... I/O equipment

A.24 Printing a complete document

The ISaGRAF Document Generator allows the user to build and print a complete document for the selected project. It can be called by the **"Project / Print"** commands of the Project Management or the Program windows to print a complete document. The Document Generator is also run by the **"Print"** command of all other ISaGRAF editors to print the contents of a single ISaGRAF document. However, the Document Generator provides the same features in both cases.

The commands of the **"Edit"** menu are used to define the elements of the project that must be inserted in the document. Doing this the user builds the "table of contents" for the desired document. Any information about the project (programs, variables, options, I/O connection...) may be inserted in the project document. No information from another project or from ISaGRAF libraries may appear in this document.



The **"File / Print"** command generates the document and send it to the printer, according to the specified table of contents. The "Print" job may take few minutes to build and format the document. It is highly recommended to wait until "Printing Job" is done in the ISaGRAF Document Generator window, before running other commands of the ISaGRAF Workbench. Building the whole document may require a large space on the hard disk. An error message will be displayed if the disk is full. In such a case, the user will have to either free up disk space by removing files, or reduce the size of the print job. When the **"Print"** command is run, a dialog box appears. It allows the user to enter a note describing the actual print command. Those notes are stored in a history file, and will be printed on the first page of any future document (including the present one).

A.24.1 Customising the table of contents

The **"Edit"** menu contains the commands to define the "Table of Contents" of the document. A choice of commands allow the user to use a default table (with all the components of the project), build a specific table (with only some components) or move items in the table and modify it.



The default list

The **"Default list"** command of the **"Edit"** menu defines a standard table of contents for the document, which includes all the components of the project. The standard table consists of:

- Project descriptor
- Hierarchy tree (links between programs)
- Source code for any program
- Diary file for any program
- Common definitions
- Global definitions
- Local definitions for any program
- Global variables
- Local variables for any program

- Application options
- I/O Connection
- Lists of variables
- Conversion tables
- Condensed cross references
- Detailed cross references
- Declaration summary
- Network addresses map
- History of modifications

The table of contents can be saved on disk using the "**File / Save**" command. This command is greyed when document generator is run from an ISaGRAF editor to print a single document.



Cut and paste

Use "**Edit / Cut**" and "**Edit / Paste**" commands to move items in the list, in order to customise the order of the table. The Document Generator allows multiple selection so that a group of items may be cut and pasted.



Clearing the table

Use "**Edit / Clear**" command reset the table of contents, so that it can be totally rebuilt using single item insertion.

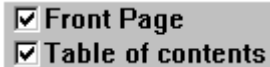


Inserting items in the table

When the "**Edit / Insert**" command is run, the "**Add item**" dialog box appears. It allows the user to insert items (components of the project) into the table of contents. For an item relative to a program, use the "**Program**" combo box to select a program name. Press the "**Add**" button to insert the selected item to the table of contents. The same item can appear only once in the table.

A.24.2 Options

The commands of the "**Options**" menu are used to define and customise the format of the generated document. Other options are directly available from buttons of the Document Generator window:



When the "**Font page**" option is set, a header page is printed at the beginning of the document, containing the project title and the history of printouts. When this option is not set, the first item to be printed starts on the first page.

When the "**Table of contents**" option is set, a table of contents is printed at the end of the generated document.

Both options are initially unchecked when the Document Generator is run from a "**Print**" command of an ISaGRAF editor (program, dictionary...)

SFC charts

The "**Separate SFC levels**" option directs the system to print, for each SFC program, first the level 1 of the SFC (chart and comments), and then the level 2 programming. When this option is not checked, levels 1 and 2 appear together on the same printout.



Page format


The "**Page format**" command of the "**Options**" menu is used to define the main parameters operated by the Document Generator when formatting a page. The following parameters can be specified:

- **Left margin:** (1 or 2 centimeters, or no margin)
- **Page border:** When this option is selected, a border is drawn around any printed page.



Page title template

The "**Page Title**" command of the "**Options**" menu is used to define the contents of the title box printed at the bottom of any page. The standard layout of this box is as follows:

	Text1	ISaGRAF - Project 'PrName'	date
	Text2		page
	Text3	User defined title	

The first line of the main title (with the name of the ISaGRAF project), the current date and the page number are automatically generated by the Document Manager, and cannot be changed.

The three lines of text on the left side of the box (text1, text2, text3) and the second line of the main title are user-defined. The user also can change the logo printed in the box on the left. To use another logo, the user has to specify the pathname of a bitmap image file (**.BMP**). The image can have any dimension. It will be stretched or shrunk, according to the exact dimensions of the printed page. Clicking on the logo area, in the dialog box, shows the new specified image. The image file must be on the disk (at the specified directory and with the specified filename) when the "**Print**" command is run.



Selecting character fonts

The "**Text font**" and "**Title font**" commands of the "**Options**" menu are used to define the fonts of characters used when printing text, and titles for any item of the document. The size and style of characters may also be selected for text and titles. The selection of a font is made with the standard dialog box defined by Windows. Any text (literal programs, names within diagrams...) will be printed with the selected size, style and font of characters. Only titles will be printed with the font selected for titles.

If the fonts of characters are not defined, the standard font of the printer will be used for any text, with the following styles:

- "Normal" style for texts and names within diagrams
- "Bold" style for titles

A.25 Password protection

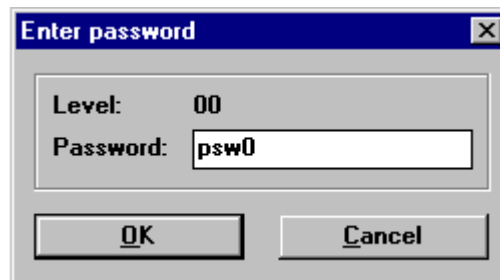
The ISaGRAF Workbench includes a full data protection system, which enables the user to protect with passwords projects and library elements. A library element can be an I/O configuration, an I/O board or complex equipment, a function or function block written in IEC languages, a "C" function, function block or conversion function. A password protection database is dedicated to one project or library element, and cannot be shared between several ones.

▣ **Protection levels**

Within one project or library element, the user can define up to **16** access levels, corresponding to different passwords. Access levels are sorted in a hierarchy system. They are numbered from **0** to **15**. The higher access level is numbered **0**. When a user knows a password, he can access all the items protected by the corresponding access level, plus all the ones protected with lower levels. Each elementary command or data of a project or library element can be separately protected with an access level. For example, the "Make application code" command from the ISaGRAF menus can be protected separately. Elementary data can be a program, a list of options, the technical note of a library element, etc...

▣ **Defining password protection**

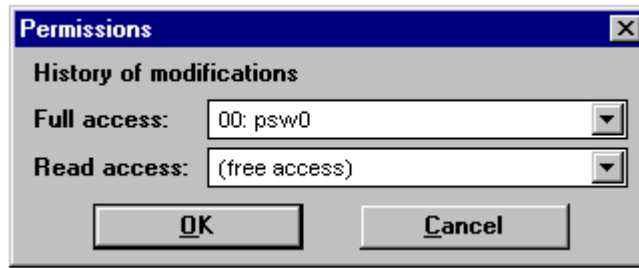
The "**Set password**" command of the ISaGRAF menus is used to define the passwords and access levels for one project or one library element. This command is called from the menus of the ISaGRAF Project Manager (for a project), or the ISaGRAF Library Manager (for a library element). No password is required when first running this command. If passwords are already defined, the user must enter the highest level password he knows, before accessing this command. Upper level passwords and protected items then cannot be modified. The "**Set password**" command enables the user to define the passwords corresponding to the different access levels, and to protect elementary commands or data with the defined levels. Passwords (corresponding to protection levels) are entered by double clicking on a line of the upper list. The following box is used to enter a password.



The image shows a standard Windows-style dialog box titled "Enter password". It has a blue title bar with a close button (X) on the right. The main area is light gray and contains two labels: "Level:" followed by the text "00", and "Password:" followed by a white text input field containing the text "psw0". At the bottom of the dialog, there are two buttons: "OK" and "Cancel".

The list in the lower area shows the different items (data or functions) which can be protected, and current protection level attached to either "read access" or "full

access" permissions. Assigning a protection level to "read" permission enables you to prevent users without sufficient permission even to open or print a document. Double click on a line in the lower list to set permissions for the selected item or data. The following box is open:



Both permissions can be set either to "free access", or to a protection level defined by a password. Full access permission cannot be attached to a level with less priority than the one selected for read access.

Note that for some documents, naturally visible when using ISaGRAF Workbench, such as project descriptor, read access cannot be protected with a password.

☰ **Accessing protected data**

No password or user's name is asked when the Workbench is started. Each time a user wants to have access to a protected data or function, he must enter the required password in a dialog box.

If the user enters the required password (or a password attached to a higher access level), he can continue normally. Each time a password is entered by the user, it is stored in memory, so the user will not have to enter it again later. Stored passwords are held each time an ISaGRAF tool is run from another ISaGRAF tool (for example, the Project Manager runs the Program Manager). Stored passwords are lost when the last remaining ISaGRAF window is closed. Passwords entered during project editing, or by using the Library Manager, or by using the Archive manager cannot be shared. If the user enters a bad password, he cannot run the selected function.

☰ **Links with the archive manager**

When saving an object (project or library element) on archive disk, the data protection item named "**Backup on archive**" is invoked. This corresponds to the data protection system attached to the object in the Workbench (hard disk). No test is performed on the data protection system of the object on the archive disk if it already exists. The "**Backup**" command of the ISaGRAF Archive Manager saves the data protection information with the object on the archive disk.

When restoring an object which already exists in the Workbench (hard disk), the data protection item named "**Overwrite with archive**" is invoked. This corresponds to the data protection system attached to the object in the Workbench (hard disk). No test is performed on the data protection system of the object on the archive disk. If this command is validated, the restored data protection information will then replace the existing one on the hard disk.

▣ **Setting individual protection for variables and I/O channels**

The ISaGRAF workbench provides a complete data protection system based on hierarchised passwords. Variable declaration and I/O connection can be globally protected by a password. Additionally, ISaGRAF enables you to set individual protection to any variable or I/O channel. This assumes that:

- passwords are already defined in the password definition system (use the "**Project / Set password**" command of the Project Management window) so that protection levels are available for individual protection.

- you use protection levels with higher priority for individual protection compared to global variable or I/O protection.

When a variable or an I/O channel has individual protection, a small icon is drawn close to its name in dictionary or I/O connection window.

Use the "**Set protection**" and "**Remove protection**" commands of the "**Edit**" menu in dictionary or I/O connection windows to set or remove an individual protection for selected variable or channel. Both commands ask you to enter a valid password so that a protection level can be attached to the variable or channel. Then, each time you want to change a variable or a connection to a channel having individual protection you must enter a password with sufficient priority level.

Warning: if a variable or channel is protected with a level, and the corresponding password is removed from protection system, and if no higher level password is defined, variable or channel cannot be changed anymore unless a new password with sufficient level is defined.

A.26 Advanced programming techniques

This chapter contains more information about the ISaGRAF Workbench and target system. The user is advised to be familiar with the ISaGRAF tools and methods, before reading this section.

A.26.1 More about ISaGRAF tools

When using the ISaGRAF editing tools, the user can press the **right mouse button** to open a popup menu, which contains the main editing commands. The menu is opened at the current position of the cursor. This is very useful to reduce mouse operations during cut and paste commands.

The ISaGRAF tools support **multiple execution**. Although same tool cannot be opened twice to edit the same document, it is possible to open different windows with the same tool and edit different objects as parallel operations.

Other commands are available to find information about graphic buttons in toolbars. Double click an empty area of a toolbar to display the contents of the toolbar as a popup menu. Stay with the mouse cursor on a graphic button displays the corresponding text command.

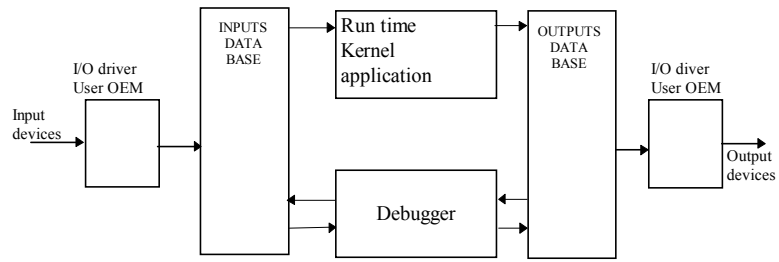
A.26.2 Locked I/Os and virtual I/Os

Defining an I/O board as **virtual** disconnects the processing of the physical I/O channels. When a board is defined as virtual, the ISaGRAF kernel operations are not changed. The only difference is that input sensors are not read and output devices are not updated. In this mode, it is possible to use the ISaGRAF debugger to modify the input values. The **Virtual** attribute applies to a complete board. It is programmed during the I/O board definition, **before** the application code generation. The **virtual** attribute is a **static** feature, and is stored when the application is stopped and restarted.

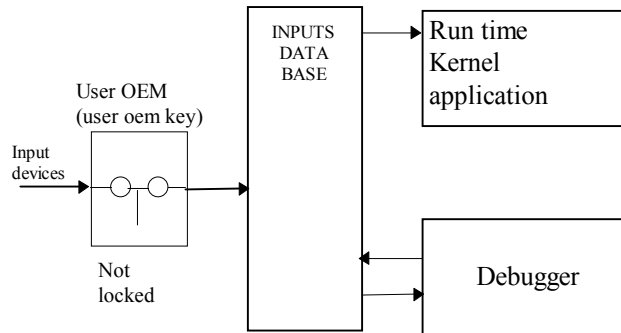
Another possibility is the I/O variable **locking**. It consists of disconnecting one physical device and the corresponding ISaGRAF I/O variable. Variable locking and unlocking is performed through the debugger. Variable locking is a **dynamic** operation, and is not memorised when the application restarts. The **lock** operation applies to only **one** variable (one I/O channel) at a time. This is the summary of main I/O controlling features:

	<i>Virtual Attribute</i>	<i>Lock command</i>
selection tool	I/O board connection	debugger
definition	static	dynamic
selection mode	board	variable
application	validation and tests	maintenance

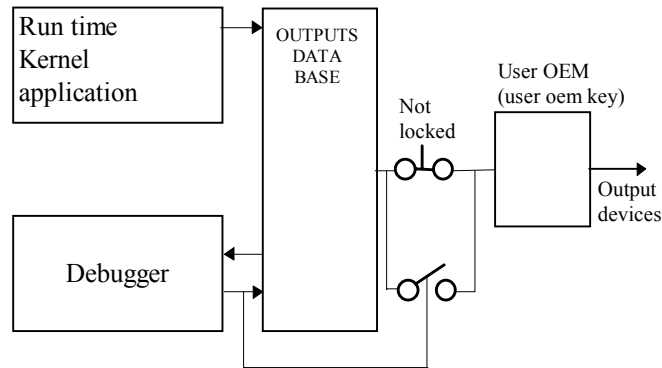
The following chart explains the I/O data flow between the ISaGRAF tasks:



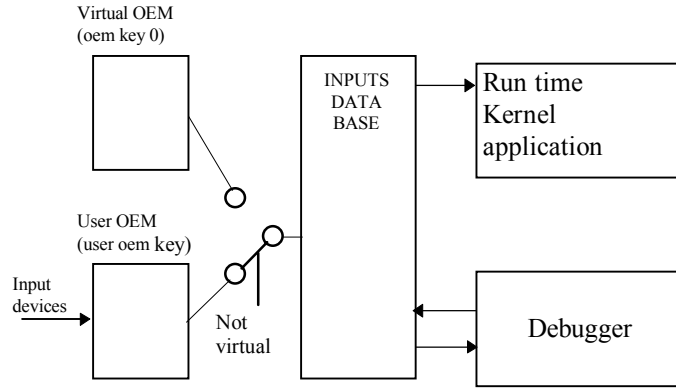
When an input variable is locked, the various accesses to the database are not changed, but the input device is disconnected. Input values can be set with the debugger and processed by the ISaGRAF kernel:



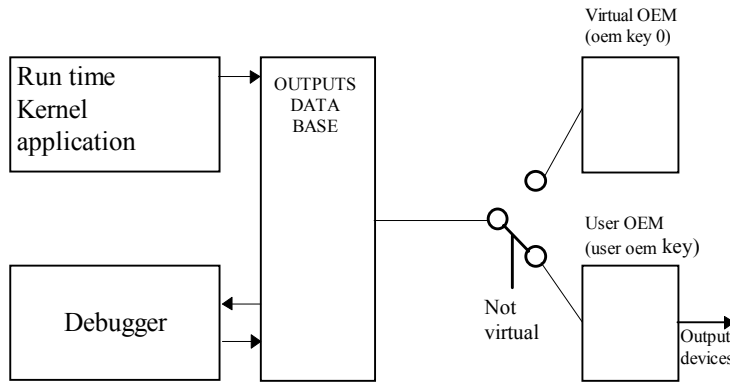
When an output variable is locked, the run-time kernel and the output driver are disconnected. In this case, access is still possible to the output device, via the output driver, with the ISaGRAF debugger:



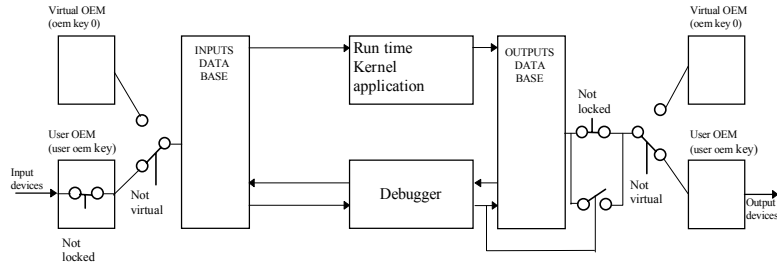
When setting the virtual attribute for an input the input database and the associated input devices are disconnected. A virtual I/O driver replaces the real one.



Setting the virtual attribute follows the same rules for an input board or an output board. For output boards, the ISaGRAF kernel updates the output database. This database and the associated output devices are, however, disconnected. A virtual I/O driver replaces the real one.



To summarise all possibilities:



A.26.3 PC-PLC link validation

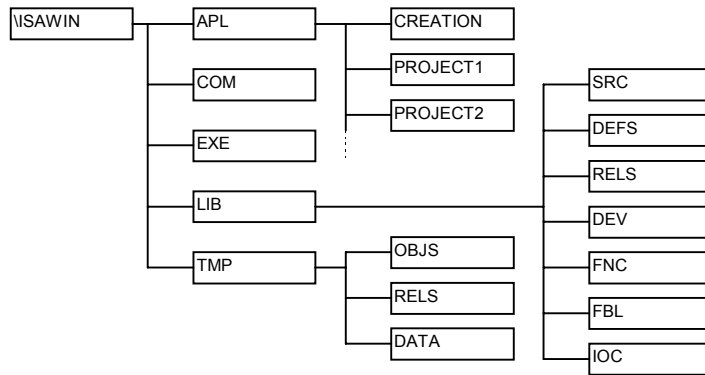
Most of the problems related to poor communication between the ISaGRAF workbench and the target PLC are represented in the debugger window by the "**disconnected**" status message. Before any diagnostic tests are performed, the communication should be validated when **no application is active** in the target PLC. This way the serial communication link can be validated on its own, isolating it from execution related effects.

The "C" language, used for description of the conversion functions and C functions, allows direct access to the target system. A programming error in such a software component may generate system errors or incorrect ISaGRAF system behaviour. Such problems may occur when I/O drivers are developed with the ISaGRAF I/O toolkit. System errors, for example, may be caused if an I/O board is connected on an invalid bus address. The following table gives a synthetic summary of error diagnostics:

status	context	Diagnostic
"disconnected" (before download)		- target is not running - no cable / invalid cable - invalid link parameters - ISaGRAF target badly installed
"disconnected" (after download)	cycle to cycle starting mode	- invalid I/O configuration - system crash
	real time starting mode	- invalid I/O configuration - system crash (due of "C" programming)
"no application"		- application not downloaded - application not started (due of "C" programming) - Intel/Motorola mismatch - Invalid target version

A.26.4 ISaGRAF directories

The ISaGRAF Workbench works on a dedicated disk directory structure. The root directory of this architecture is specified by the user during the installation of ISaGRAF. The default name for the ISaGRAF root directory is **ISAWIN**. This is the standard disk architecture created by the installation program:



These are the standard ISaGRAF sub-directories

DIRECTORY CONTENTS

- APL** root directory for the ISaGRAF projects
each project corresponds to one sub directory
which contains all the data of the project
other directories may exist for other project groups. ISaGRAF
installation program creates "**SMP**" directory where are stored
samples applications.
- COM** "common" range data
Data can be used by any project
- EXE** ISaGRAF programs and help files
- LIB** ISaGRAF libraries:
 - lists of elements
 - parameters or interface for each element
 - technical notes
- LIB\IOC** source code for I/O configurations
- LIB\FNC** source code of functions written in IEC languages
- LIB\FBL** source code of function blocks written in IEC languages
- LIB\SRC** source code for conversions and C functions
- LIB\DEFS** source header for conversions and C functions
- LIB\RELS** Conversions and C functions object code
- LIB\DEV** command files for developing "C" libraries
makefiles, link lists, etc...
- TMP** Temporary files: sub-directories of TMP are reserved for the
ISaGRAF Code Generator and cannot be deleted.

The sub-directories can be moved to other disk locations. When the user has a non-standard architecture, the pathnames of the sub-directories should be declared in the **WS001** section, in the **ISA.ini** initialisation file, in the **EXE** sub-directory of ISaGRAF. Here are the entries of the **WS001** section:

- Isa** root directory for ISaGRAF architecture
- IsaExe** root directory for ISaGRAF programs and help files

IsaApl	root directory for ISaGRAF projects
IsaTmp	directory for temporary files
IsaSrc	directory for library source code
IsaDefs	directory for library source headers

Note that if you change the IsaTmp entry to another directory, you must create the sub-directories OBJS, RELS and DATA in the new directory. The following example uses the entries of the **WS001** section to redefine the standard ISaGRAF disk architecture:

```
;file c:\ISAWIN\EXE\ISA.ini

[WS001]
Isa=c:\isawin
IsaExe=c:\isawin\exe
IsaApl=c:\isawin\apl
IsaTmp=c:\isawin\tmp
IsaSrc=c:\isawin\lib\src
IsaDefs=c:\isawin\lib\defs
```

When you want to add "C" functions or function blocks to the ISaGRAF target, the **ISAWIN\LIB\DEV** directory is used to store development files: command files, makefiles, maps, etc... The **ISAWIN\LIB\RELS** directory is used to store the object files generated during "C" compiling, and the ISaGRAF "C" libraries required for LINK operations.

A.26.5 Application symbols

Each object of an ISaGRAF application is referenced by a name (entered during variable declaration) and an internal **virtual address**, calculated by the code generator. The virtual address of a variable is not its **network address** entered during the declaration of the variable. Virtual addresses are used for communication work, and special "C" applications using the OEM option. When the ISaGRAF code generator is run, it makes an ASCII file with the logical correspondence between names and virtual addresses for all the objects (variable, programs, steps...) of the project. This file can be easily interrogated for information about the ISaGRAF static database from any user's application. The file is named "**APPLI.TST**" and is located in the directory of the ISaGRAF project: "**ISAWIN\APL\prname**" (prname is the name of the project). This section describes the detailed format of the "**APPLI.TST**" file. The main notations used for the following descriptions, is shown below:

VA	virtual address
ATTR	attribute of a variable
USP	"C" function

Possible values for the attributes of a variable are shown below. Such values occur in the "**attributes**" fields:

+X	internal variable
+C	read-only internal variable
+I	input variable

+o output variable

All the numbers, except virtual addresses, are expressed as decimal integers. The virtual addresses (**VA**) are expressed as hexadecimal 4 digit numbers, and are preceded by the character "!". For example:

```
123            this is a decimal number
!A003          this is an hexadecimal virtual address
```

The main structure of the file "**APPLI.TST**" is shown below. The file is structured as a list of **blocks**. A block is a list of **records**. Each record is described on one line of text. Each block begins with a header, put on one line of text.

```
Start block
description blocks
end block
```

The general syntax of one block is shown below:

```
@ <block_name> <arguments>
#record...
#record...
...
```

The structure of the first block, containing the main information about the application, is shown below:

```
@ISA_SYMBOLS,<appli_crc>
#NAME,<appli_name>,<version>
#DATE,<creation_date>
#SIZE,G=<nbprg>,S=<nbstep>,T=<nbtra>,L=0,P=<nbpro>,V=<nbvar>
#COMMENT,cj` international
```

appli_crc application symbols checksum
appli_name name of the application
version ISaGRAF workbench version number
creation_date application generation date
nbprg number of programs
nbstep number of SFC steps
nbtra number of SFC transitions
nbpro number of "C" functions used
nbvar total number of variables

The structure of the last block, which signals the end of the file, is shown below:

```
@END_SYMBOLS
```

The structure of the block used to describe the programs of the application, is shown below:

```
@PROGRAMS,<nbprg>
#<va>,<name>
```

. . .

nbprg..... number of programs defined in this block
va..... virtual address of the program
name..... program name

The structure of the block used to describe the SFC steps of the application is shown below. Note that there is one virtual step defined for each non-SFC program:

```
@STEPS, <nbsteps>
#<va>, <name>, <father>
# . . .
```

nbsteps number of steps defined in this block
va..... virtual address of the step
name..... step name
father virtual address of the father

The structure of the block used to describe the SFC transitions of the application, is shown below:

```
@TRANSITIONS, <nbtrans>
#<va>, <name>, <father>
# . . .
```

nbtrans..... number of transitions defined in this block
va..... virtual address of the transition
name..... transition name
father virtual address of the father

The structure of the block used to describe the boolean variables of the application, is shown below:

```
@BOOLEANS, <nb_boo>
#<va>, <name>, <attr>, <program>, <eq_false>, <eq_true>
# . . .
```

and if variable number exceeds 4095:

```
X#(1.<varno>), <name>, <attr>, <program>, <eq_false>, <eq_true>
```

nb_boo..... number of variables in this block
va..... virtual address of the variable
varno range of the address (within boolean data type)
name..... name of the variable
attr attribute of the variable
program virtual address of the parent program
..... or "**!0000**" for a global variable
eq_false..... string used for false value
eq_true string used for true value

The structure of the block used to describe the analog variables of the application, is shown below:

```
@ANALOGS, <nb_ana>
#<va>, <name>, <attr>, <program>, <format>, <unit>
#...
```

and if variable number exceeds 4095:

```
X# (2.<varno>), <name>, <attr>, <program>, <format>, <unit>
```

nb_ana number of variables in this block
va virtual address of the variable
varno range of the address (within analog data type)
name name of the variable
attr attribute of the variable
program virtual address of the parent program
..... or **"!0000"** for a global variable
format = **"I"** for an integer variable
..... = **"F"** for a real variable
unit unit string

The structure of the block used to describe the timer variables of the application, is shown below:

```
@TIMERS, <nb_tmr>
#<va>, <name>, <attr>, <program>
#...
```

and if variable number exceeds 4095:

```
X# (3.<varno>), <name>, <attr>, <program>
```

nb_tmr number of variables in this block
va virtual address of the variable
varno range of the address (within timer data type)
name name of the variable
attr attribute of the variable (always +X: internal)
program virtual address of the parent program
..... or **"!0000"** for a global variable

The structure of the block used to describe the message variables of the application, is shown below:

```
@MESSAGES, <nb_msg>
#<va>, <name>, <attr>, <program>, <max_len>
#...
```

and if variable number exceeds 4095:

```
X# (4.<varno>), <name>, <attr>, <program>, <max_len>
```

nb_msg number of variables in this block
va virtual address of the variable
varno range of the address (within message data type)
name name of the variable
attr attribute of the variable
program virtual address of the parent program
 or "**!0000**" for a global variable
max_len maximum length (declared capacity)

The structure of the block used to describe the "C" functions used in the application, is shown below:

```
@USP, <nb_usp>
#<va>, <name>
#...
```

nb_usp number of C functions in this block
va virtual address of the C function
name name of the C function

The structure of the block used to describe the "C" function block instances used in the application, is shown below:

```
@FBINSTANCES, <nb_fb>
#<va>, <inst_name>, <fb_name>
#...
```

nb_fb number of instances of a C function blocks in this block
va virtual address of the C function block instance
inst_name name of the C function block instance
fb_name name of the reference C function block

A.26.6 Limits of ISaGRAF "LARGE" (WDL) workbench

There are some limitations for the objects used in the ISaGRAF Workbench. Of course, many other practical limits are due to the configuration of the computer used (available memory and disk space), and the capabilities of the ISaGRAF target system (available memory, available hardware and software resources...). The following numbers absolute limits that cannot be exceeded.

☰ **For a project:**

<i>Object</i>	<i>Maximum</i>	<i>Notes</i>
Programs	255	grouping main, sub and child programs
Levels in the hierarchy	20	

The number of projects installed on the Workbench is only limited by the available space on the hard disk.

☰ **For names:**

<i>Name for:</i>	<i>Maximum</i>	<i>Notes</i>
Project	8 char	
Program	8 char	
Variable	16 char	+ 60 characters for comment
Defined word label	16 char	
Defined equivalence	255 char	+ 60 characters for comment
Conversion table	16 char	
List of variables	16 char	
function / f.block (lib)	8 char	this applies to C functions, C function blocks
function parameter (lib)	16 char	this applies to C functions, C function blocks or functions written in IEC languages
IO board	8 char	
IO configuration	8 char	
Board oem parameter	16 char	
Conversion function	8 char	

≡ **Editing (for one program):**

<i>Object</i>	<i>Maximum</i>	<i>Notes</i>
SFC rows	600	
SFC columns	20	
SFC steps	4095	for the whole project, grouping steps, initial steps, beginning and ending steps
SFC transitions	4095	for the whole application
LD/FBD editing	200 cols 2000 rows	this is the size of the editing area in cell units.
Quick LD editing	no limit	limits are imposed by the PC capacity
IL labels	251	in the same IL program
Text editing	40KBytes	or less according to the system configuration

≡ **For the dictionary (for one project):**

<i>Object</i>	<i>Maximum</i>	<i>Notes</i>
Boolean variables	65535	
Analog variables	65535	grouping integer and real variables
Timers	65535	
Message variables	65535	
Defined words	4095	in the same list (same range)
Defined words	255	used in the same program
Conversion tables	127	used in the application
Points in one table	32	defined in the same conversion table

The limits given for maximum number of boolean, analog or message variables group internal, input and output variables. It also includes all hidden temporary or

variables allocated by compilers. The number of variables edited together (same type, same scope), in the dictionary editor cannot exceed 16000. Depending on PC configuration, the limit can be less than 16000. The application cannot run on an ISaGRAF target version V3.21 or earlier if the total number of variable for one type exceeds 4095. The standard "Modbus" link using network addresses does not cannot be used if number of variables for one type exceeds 4095.

☰ **IO connections:**

<i>Object</i>	<i>Maximum</i>	<i>Notes</i>
IO Boards	256	defined for the same application (boards or complex equipments)

Number of I/O boards including single boards and items of complex equipments cannot exceed 256.

IO channels	128	on the same board
-------------	-----	-------------------

☰ **For libraries:**

<i>Object</i>	<i>Maximum</i>	<i>Notes</i>
Functions (IEC lang.)	255	installed together in the library
Function blocks (IEC lang.)	255	installed together in the library
C functions	255	installed together in the library
C function blocks	255	installed together in the library
function blocks instances	4095	for the same type of function block in the same application
Function input parameters	31	this applies to C functions and functions written in IEC languages
Function block parameters	32	freely distributed between input and output parameters. At least 1 output parameter is required.
Conversion function	128	installed together in the library
IO configurations	255	installed together in the library
IO boards	255	installed together in the library
Complex IO equipt.	255	installed together in the library
Board oem parameters	16	

B. Language reference

B.1 Project architecture

An ISaGRAF project is divided into several programming units called **programs**. The programs of the project are linked together in a tree-like architecture. Programs can be described using any of **SFC**, **FC (Flow Chart)**, **FBD**, **LD**, **ST** or **IL** graphic or literal languages.

B.1.1 Programs

A **program** is a logical programming unit, which describes operations between **variables** of the process. Programs describe either **sequential** or **cyclic** operations. Cyclic programs are executed at each target system cycle. The execution of sequential programs follows the dynamic rules of either the **SFC** language or the **FC** language.

Programs are linked together in a hierarchy tree. Programs placed on the top of the hierarchy are activated by the system. Sub-programs (lower level of the hierarchy) are activated by their father. A program can be described with any of the available graphic or literal following languages:

Sequential Function Chart (SFC) for high level programming
Flow Chart (FC) for high level programming
Function Block Diagram (FBD) for cyclic complex operations
Ladder Diagram (LD) for boolean operations only
Structured Text (ST) for any cyclic operations
Instruction List (IL) for low level operations

The same program cannot mix several languages, except LD and FBD can be combined in one diagram.

B.1.2 Cyclic and sequential operations

The hierarchy of programs is divided into four main **sections** or groups:

Begin	programs executed at the beginning of each target cycle
Sequential	programs following SFC or FC dynamic rules
End	programs executed at the end of each target cycle
Functions	set of non-dedicated sub-programs

Programs of the **'Begin'** or **'End'** sections describe cyclic operations, and are not time dependent. Programs of the **'Sequential'** section describe sequential operations, where the time variable explicitly synchronises basic operations. Main programs of the **'Begin'** section are systematically executed at the beginning of each run time cycle. Main programs of the **'End'** section are systematically executed at the end of each run time cycle. Main programs of the **'Sequential'** section are executed according to either the **SFC** or the **FC** dynamic rules.

Programs of the **"Functions"** section are sub-programs that can be called by any other program in the project. A program of the **"Function"** section can call another program of this section.

Main and child programs of the sequential section must be described with **SFC** or **FC** language. Programs of cyclic sections (**begin** and **end**) cannot be described with **SFC** or **FC** language. Any program of any section may own one or more sub-programs. Any program of the sequential section may own one or more **SFC** or **FC** child programs (according to its own programming language). Sub-programs cannot be described with **SFC** or **FC** language.

Programs of the **Begin** section are typically used to describe preliminary operations on input devices to build high level filtered variables. Such variables are frequently used by the programs of the **Sequential** section. Programs of the **End** section are typically used to describe security operations on the variables operated on by the **Sequential** section, before sending values to output devices.

B.1.3 Child SFC and FC programs

Any **SFC** program of the sequential section may control other **SFC** programs. Such low-level programs are called **child SFC programs**. A **child SFC program** is a parallel program that can be started, killed, frozen or restarted by its parent program. The parent program and child program must both be described with the **SFC** language. A child SFC program may have local variables and defined words.

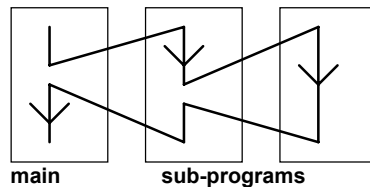
When a parent program starts a child **SFC** program, it puts an SFC **token** (activates) into each initial step of the child program. This command is described with the **GSTART** statement. When a parent program kills a child **SFC** program, it clears all the tokens existing in the **steps** of the child. Such a command is described with the **GKILL** statement.

When a parent program freezes a child **SFC** program, it suspends its execution. The suspended program can then be restarted using the **GRST** statement.

Any **FC** program of the sequential section may control other **FC** sub-programs. An **FC** father program is blocked (waits) during execution of an FC sub-program. It is not possible that simultaneous operations are done in father FC program and one of its FC sub-programs.

B.1.4 Functions and sub-programs

A sub-program or a function execution is driven by its parent program. The execution of the parent program is suspended until the sub-program or the function ends:



Any program of any section may have one or more sub-programs. A sub-program is owned by only one father program. A sub-program may have local variables and defines. Any language but **SFC** or **FC** can be used to describe a sub-program. Programs of the **"Functions"** section are sub-programs that can be called by any other program in the project.

Unlike other sub-programs, they are not dedicated to one father program. A program of the "**Function**" section can call another program of this section. A function can be located in the Library.

Warning: The ISaGRAF system does not support **recursive function calls**. A run time error will occur if a program of the "**Functions**" section is called by itself or by one of its called sub-program.

Warning: A function or sub-program does not "store" the local value of its local variables. A function or sub-program is not instantiated and so can not call function blocks.

The interface of a sub-program must be explicitly defined, with a **type** and a **unique name** for each of its calling or return parameter. In order to support the **ST** language convention, the return parameter must have the same name as the sub-program.

The following table shows how to set the value of the return parameter in the body of a sub-program, in the different languages:

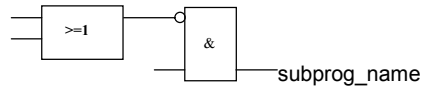
ST: assign the return parameter using its name
(the same name as the sub-program):

```
subprog_name := <expression>;
```

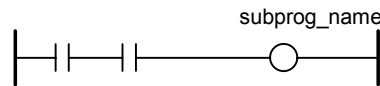
IL: the value of the current result (IL register)
at the end of the sequence is stored in the return parameter:

```
LD 10
ADD 20 (* return parameter value = 30 *)
```

FBD: set the return parameter using its name:

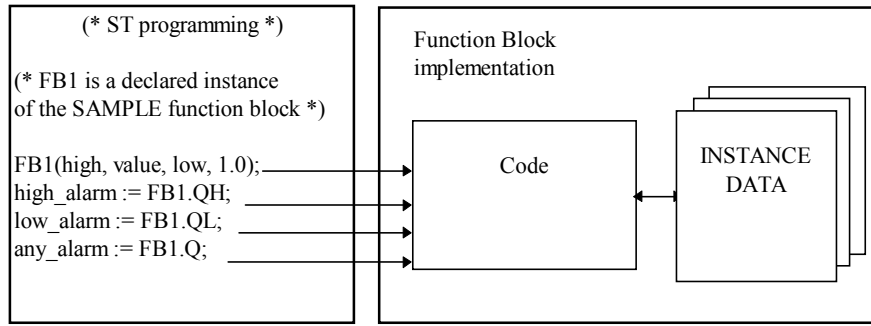


LD: use a coil symbol with the name of the return parameter:



B.1.5 Function blocks

Function blocks can use the languages: LD, FBD, ST or IL. Function blocks are instantiated. It means local variables of a function block are copied for each instance. When calling a block in a program, you actually call the instance of the block: the same code is called but the data used are the one which have been allocated for the instance. Values of the variables of the instance are stored from one cycle to the other.



Warnings:

- A function block written with one of the IEC languages can not call other function blocks: the instantiation mechanism only manages the local variables of the block itself. Here is the list of standard function blocks that you cannot use inside an IEC function block:

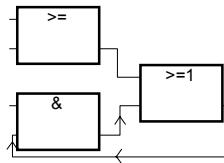
SR, RS, R_Trig, F_Trig, SEMA, CTU, CTD, CTUD, TON, TOF, TP, CMP, StackInt, AVERAGE, HYSTER, LIM_ALRM, INTEGRAL, DERIVATE, BLINK, SIG_GEN

- For the same reason, you can not use Positive or Negative contact or coils, or Set and Reset coils.

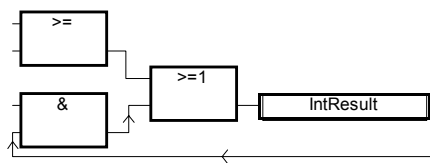
- TSTART and TSTOP functions to start and stop timers cannot be used in a function block for 3.0x targets. It works since the 3.20 target.

- When you need loop in your function block, you must use local variable before doing the loop. See the example below:

This will not work:



This is OK:



B.1.6 Description language

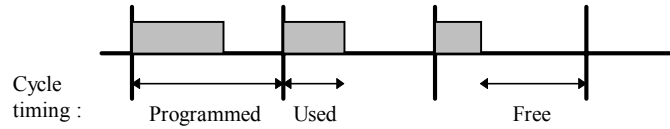
A program can be described with any of the following graphic or literal languages:

- Sequential Function Chart (SFC)** for high level operations
- Flow Chart (FC)** for high level operations
- Function Block Diagram (FBD)** for cyclic complex operations
- Ladder Diagram (LD)** for boolean operations only
- Structured Text (ST)** for any cyclic operations
- Instruction List (IL)** for low level operations

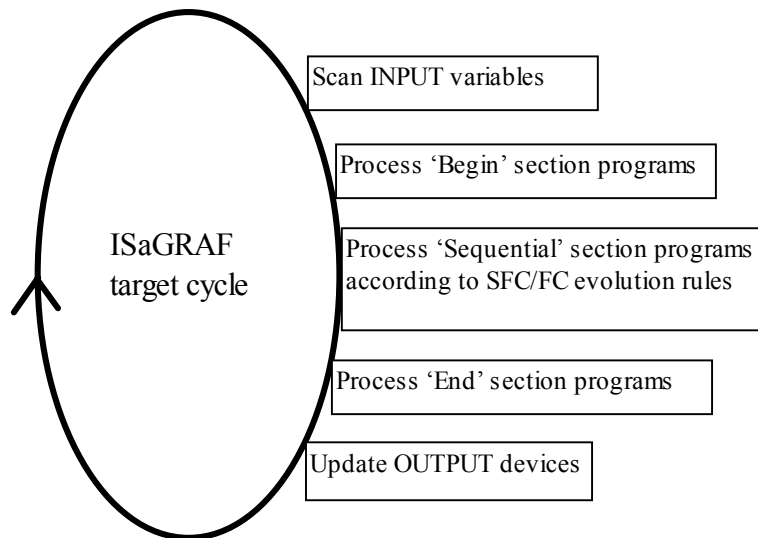
The same program cannot mix several languages. The language used to describe a program is chosen when the program is created, and cannot be changed later on. The exception is that it is possible to combine FBD and LD in a single program.

B.1.7 Execution rules

ISaGRAF is a **synchronous** system. All the operations are triggered by a clock. The basic duration of the clock is called the cycle timing:



Basic operations processed during a target cycle are:



This system makes it possible to:

- guarantee that an input variable keeps the same value within a cycle,
- guarantee that an output device is not updated more than once in a cycle,
- work safely on the same global variable from different programs,
- estimate and control the response time of the complete application.

B.2 Common objects

These are main features and common **objects** of the ISaGRAF programming database. Such objects can be used in any program written with any of the **SFC**, **FC**, **FBD**, **LD**, **ST** or **IL** languages.

B.2.1 Basic types

Any constant, expression or variable used in a program (written in any language) must be characterised by a type. Type coherence must be followed in graphic operations and literal statements. These are the available basic types for programming objects:

BOOLEAN:	logic (true or false) value
ANALOG:	integer or real (floating) continuous value
TIMER:	time value
MESSAGE:	character string

Note: Timers contain values less than one day and cannot be used to store dates.

B.2.2 Constant expressions

Constant expressions are relative to one type. The same notation cannot be used to represent constant expressions of different types.

B.2.2.1 Boolean constant expressions

There are only two boolean constant expressions:

TRUE	is equivalent to the integer value 1
FALSE	is equivalent to the integer value 0

"True" and "False" keywords are case insensitive.

B.2.2.2 Integer analog constant expressions

Integer constant expressions represent signed long integer (32 bit) values: from **-2147483647** to **+2147483647**. Integer analog constants may be expressed with one of the following **bases**. Integer constants must begin with a **prefix** that identifies the bases used:

Base	Prefix	Example
DECIMAL	(none)	-908
HEXADECIMAL	"16#"	16#1A2B3C4D
OCTAL	"8#"	8#1756402
BINARY	"2#"	2#1101_0001_0101_1101

The underscore character ('_') may be used to separate groups of digits. It has no particular significance, and is used to increase constant expression readability.

B.2.2.3 Real analog constant expressions

Real analog constant expressions can be written with either **decimal** or **scientific** representation. The **decimal point** ('.') separates the integer and decimal parts. The decimal point must be used to differentiate a real constant expression from an integer one. The scientific representation uses the 'E' or 'F' letter to separate the **mantissa** part and the **exponent**. Exponent part of a real scientific expression must be a signed integer value from -37 to +37. Below are examples of real analog constant expressions:

3.14159	-1.0E+12
+1.0	1.0F-15
-789.56	+1.0E-37

The expression "123" does not represent a real constant expression. Its correct real representation is "123.0".

B.2.2.4 Timer constant expressions

Timer constant expressions represent time values from **0 second** to **23h59m59s999ms**. The lowest allowed unit is a millisecond. Standard time units used in constant expressions are:

Hour	The "h" letter must follow the number of hours
Minute	The "m" letter must follow the number of minutes
Second	The "s" letter must follow the number of seconds
Millisecond	The "ms" letters must follow the number of milliseconds

The time constant expression must begin with "T#" or "TIME#" prefix. Prefixes and unit letters are case insensitive. Some units may not appear. These are examples of timer constant expressions:

T#1H450MS	1 hour, 450 milliseconds
time#1H3M	1 hour, 3 minutes

The expression "0" does not represent a time value, but an analog constant.

B.2.2.5 Message string constant expressions

String or message constant expressions represent character strings. Characters must be preceded by a quote and followed by an apostrophe. For example:

'THIS IS A MESSAGE'

Warning: The apostrophe "'" character cannot be used within a string constant expression. A string constant expression must be expressed on one line of the program source code. Its length cannot exceed 255 characters, including spaces.

Empty string constant expression is represented by two apostrophes, with no space or tab character between them:

" (* this is an empty string *)

The special character dollar ('\$'), followed by other special characters, can be used in a string constant expression to represent a non-printable character:

Sequence	Meaning	ASCII (hexa)	Example
\$\$	'\$' character	16#24	'I paid \$\$5 for this'
\$'	apostrophe	16#27	'Enter '\$Y\$' for YES'
\$L	line feed	16#0a	'next \$L line'
\$R	carriage return	16#0d	' llo \$R He'
\$N	new line	16#0d0a	'This is a line\$N'
\$P	new page	16#0c	'lastline \$P first line'
\$T	tabulation	16#09	'name\$Tsize\$Tdate'
\$hh (*)	any character	16#hh	'ABCD = \$41\$42\$43\$44'

(*) "**hh**" is the hexadecimal value of the ASCII code for the expressed character.

B.2.3 Variables

Variables can be **LOCAL** to one program, or **GLOBAL**. Local variables can be used by one program only. Global variables can be used in any program of the project. Variable names must conform to the following rules:

name cannot exceed **16** characters

first character must be a **letter**

following characters can be **letters, digits** or the underscore character

B.2.3.1 Reserved keywords

A list of the reserved keywords is shown below. Such identifiers cannot be used to name a program, a variable or a "C" function or function block:

A	ANA, ABS, ACOS, ADD, ANA, AND, AND_MASK, ANDN, ARRAY, ASIN, AT, ATAN,
B	BCD_TO_BOOL, BCD_TO_INT, BCD_TO_REAL, BCD_TO_STRING, BCD_TO_TIME, BOO, BOOL, BOOL_TO_BCD, BOOL_TO_INT, BOOL_TO_REAL, BOOL_TO_STRING, BOOL_TO_TIME, BY, BYTE,
C	CAL, CALC, CALCN, CALN, CALNC, CASE, CONCAT, CONSTANT, COS,
D	DATE, DATE_AND_TIME, DELETE, DINT, DIV, DO, DT, DWORD,
E	ELSE, ELSIF, EN, END_CASE, END_FOR, END_FUNCTION, END_IF, END_PROGRAM, END_REPEAT, END_RESSOURCE, END_STRUCT, END_TYPE, END_VAR, END_WHILE, ENO, EQ, EXIT, EXP, EXPT,
F	FALSE, FEDGE, FIND, FOR, FUNCTION,
G	GE, GFREEZE, GKILL, GRST, GSTART, GSTATUS, GT,

I	IF, INSERT, INT, INT_TO_BCD, INT_TO_BOOL, INT_TO_REAL, INT_TO_STRING, INT_TO_TIME,
J	JMP, JMPC, JMPCN, JMPN, JMPNC,
L	LD, LDN, LE, LEFT, LEN, LIMIT, LINT, LN, LOG, LREAL, LT, LWORD,
M	MAX, MID, MIN, MOD, MOVE, MSG, MUL, MUX,
N	NE, NOT,
O	OF, ON, OPERATE, OR, OR_MASK, ORN,
P	PROGRAM
R	R, REDGE, READ_ONLY, READ_WRITE, REAL, REAL_TO_BCD, REAL_TO_BOOL, REAL_TO_INT, REAL_TO_STRING, REAL_TO_TIME, REDGE, REPEAT, REPLACE, RESSOURCE, RET, RETAIN, RETC, RETCN, RETN, RETNC, RETURN, RIGHT, ROL, ROR,
S	S, SEL, SHL, SHR, SIN, SINT, SQRT, ST, STN, STRING, STRING_TO_BCD, STRING_TO_BOOL, STRING_TO_INT, STRING_TO_REAL, STRING_TO_TIME, STRUCT, SUB, SYS_ERR_READ, SYS_ERR_TEST, SYS_INITALL, SYS_INITANA, SYS_INITBOO, SYS_INITTMR, SYS_RESTALL, SYS_RESTANA, SYS_RESTBOO, SYS_RESTTMR, SYS_SAVALL, SYS_SAVANA, SYS_SAVBOO, SYS_SAVTMR, SYS_TALLOWED, SYS_TCURRENT, SYS_TMAXIMUM, SYS_TOVERFLOW, SYS_TRESET, SYS_TWRITE, SYSTEM,
T	TAN, TASK, THEN, TIME, TIME_OF_DAY, TIME_TO_BCD, TIME_TO_BOOL, TIME_TO_INT, TIME_TO_REAL, TIME_TO_STRING, TMR, TO, TOD, TRUE, TSTART, TSTOP, TYPE,
U	UDINT, UINT, ULINT, UNTIL, USINT,
V	VAR, VAR_ACCESS, VAR_EXTERNAL, VAR_GLOBAL, VAR_IN_OUT, VAR_INPUT, VAR_OUTPUT,
W	WHILE, WITH, WORD,
X	XOR, XOR_MASK, XORN

All keywords beginning with an underscore ('_') character are internal keywords and must not be used in textual instructions.

B.2.3.2 Directly represented variables

ISaGRAF enables the use of **directly represented variables** in the source of the programs to represent a free channel. Free channels are the ones which are not linked to a declared I/O variable. The identifier of a directly represented variable always begins with "%" character.

Below are the naming conventions of a directly represented variable for a channel of a single board. "s" is the slot number of the board. "c" is the number of the channel.

%IXs.c free channel of a boolean input board
%IDs.c free channel of an integer input board
%ISs.c free channel of a message input board
%QXs.c free channel of a boolean output board
%QDs.c free channel of an integer output board
%QSs.c free channel of a message output board

Below are the naming conventions of a directly represented variable for a channel of a complex equipment. "**s**" is the slot number of the equipment. "**b**" is the index of the single board within the complex equipment. "**c**" is the number of the channel.

<code>%IXs.b.c</code>	free channel of a boolean input board
<code>%IDs.b.c</code>	free channel of an integer input board
<code>%ISs.b.c</code>	free channel of a message input board
<code>%QXs.b.c</code>	free channel of a boolean output board
<code>%QDs.b.c</code>	free channel of an integer output board
<code>%QSs.b.c</code>	free channel of a message output board

Below are examples:

<code>%QX1.6</code>	6th channel of the board #1 (boolean output)
<code>%ID2.1.7</code>	7th channel of the board #1 in the equipment #2 (integer input)

A directly represented variable cannot have the "**real**" data type.

B.2.3.3 Boolean variables

Boolean means **logic**. Such variables can take one of the boolean values: **TRUE** or **FALSE**. Boolean variables are typically used in boolean expressions. Boolean variables can have one of the following **attributes**:

Internal:	memory variable updated by the program
Constant:	read-only memory variable with an initial value
Input:	variable connected to an input device (refreshed by the system)
Output:	variable connected to an output device

Warning: When declaring a boolean variable, strings can be defined to replace 'true' and 'false' values during debug. Those strings cannot be used in the programs unless entered as '**defined words**' for the language.

B.2.3.4 Analog variables

Analog means **continuous**. Such variables have signed integer or real (floating) values. Available formats for an analog variable are:

Integer	32 bit signed integer: from -2147483647 to +2147483647
Real	standard IEEE 32 bit floating value (single precision) 1 sign bit + 23 mantissa bits + 8 exponent bits

REAL analog exponent value cannot be less than **-37** or greater than **+37**. Analog variables can have one of the following **attributes**:

Internal	memory variable updated by the program
Constant:	read-only memory variable with an initial value
Input	variable connected to an input device (refreshed by the system)
Output	variable connected to an output device

Note: When a real variable is connected to an I/O device, the corresponding I/O driver operates the equivalent integer value.

Warning: Integer and real analog variables or constant expressions cannot be mixed in the same analog expression.

B.2.3.5 Timer variables

Timer means **clock** or **counter**. Such variables have time values and are typically used in time expressions. A timer value cannot exceed **23h59m59s999ms** and cannot be negative. Timer variables are stored in 32 bit words. The internal representation is a positive number of milliseconds.

Timer variables can have one of the following **attributes**:

Internal memory variable managed by the program, refreshed by ISaGRAF system
Constant: read-only memory variable with an initial value

Warning: Timer variables cannot have the INPUT or OUTPUT attributes.

Timer variables can be automatically refreshed by the ISaGRAF system. When a timer is **active**, its value is automatically increased according to the target system real time clock. The following statements of the **ST** language can be used to control a timer:

TSTART starts automatic refresh of a timer
TSTOP stops automatic refresh of a timer

B.2.3.6 Message string variables

Message or string variables contain character strings. The length of the string can change during process operations. The length of a message variable cannot exceed the capacity (maximum length) specified when the variable is declared. Message capacity is limited to 255 characters. Message variables can have one of the following **attributes**:

Internal memory variable updated by the program
Constant: read-only memory variable with an initial value
Input variable connected to an input device (refreshed by the system)
Output variable connected to an output device

String variables can contain any character of the standard ASCII table (ASCII code from **0** to **255**). The null character can exist in a character string. Some "C" functions of the standard ISaGRAF library will not correctly operate messages which contain null (**0**) characters.

B.2.4 Comments

Comments may be freely inserted in literal languages such as **ST** and **IL**. A comment must begin with the special characters "**(***" and terminate with the characters "***)**". Comments can be inserted anywhere in a **ST** program, and can be written on more than one line.

These are examples of comments:

```
counter := ival; (* assigns the main counter *)
(* this is a comment expressed
on two lines *)
c := counter (* you can put comments anywhere *) + base_value + 1;
```

Interleave comments cannot be used. This means that the "(" characters cannot be used within a comment.

Warning: The IL language only accepts comments as the last component of an instruction line.

B.2.5 Defined words

The ISaGRAF system allows the re-definition of constant expressions, true and false boolean expressions, keywords or complex **ST** expressions. To achieve this, an **identifier** name has to be given to the corresponding expression. For example:

```
YES      is      TRUE
PI       is      3.14159
OK       is      (auto_mode AND NOT (alarm))
```

When such equivalence is defined, its **identifier** can be used anywhere in an **ST** program to replace the attached expression. This is an example of **ST** programming using defines:

```
If OK Then
  angle := PI / 2.0;
  isdone := YES;
End_if;
```

Defined words can be **LOCAL** to one program, **GLOBAL**, or **COMMON**.

Local defined words can be used by only one program.

Global defined words can be used in any program of the project.

Common defined words can be used in any program of any project.

Note that common defined can be stored separately with the Archive manager.

Warning: When the same identifier is defined twice with different **ST** equivalencies, the last defined expression is used. For example:

```
Define:      OPEN      is      FALSE
             OPEN      is      TRUE

means:      OPEN      is      TRUE
```

Naming defined words must conform to following rules:

- name cannot exceed **16** characters
- first character must be a **letter**
- following characters can be **letters**, **digits** or underscore ('_') character

Warning: A defined word can not use a defined word in its definition, for example, you can not have:

P1 is **3.14159**
~~**P12** is **P1*2**~~

write the complete equivalence using constants or variables and operations:

P12 is **6.28318**

B.3 SFC language

Sequential Function Chart (SFC) is a **graphic** language used to describe **sequential operations**. The process is represented as a set of well-defined **steps**, linked by **transitions**. A **boolean condition** is attached to each transition. **Actions** within the steps are detailed by using other languages (**ST**, **IL**, **LD** and **FDB**).

B.3.1 SFC chart main format

An SFC program is a graphic set of **steps** and **transitions**, linked together by **oriented links**. Multiple connection links are used to represent divergences and convergences. Some parts of the complete program may be separated and represented in the main chart by a single symbol, called **macro steps**. The basic **graphic rules** of the SFC are:

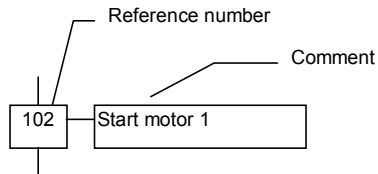
- A step cannot be followed by another step
- A transition cannot be followed by another transition

B.3.2 SFC basic components

The basic components (graphic symbols) of the SFC language are: steps and initial steps, transitions, oriented links, and jumps to a step.

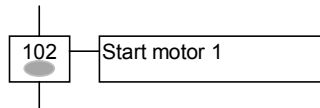
B.3.2.1 Steps and initial steps

A step is represented by a single **square**. Each step is **referenced** by a number, written in the step square symbol. A main description of the step is written in a rectangle linked to the step symbol. This description is a **free comment** (not part of the programming language). The above information is called the **Level 1** of the step:

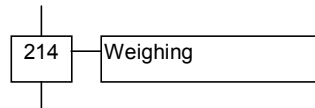


At run time, a **token** indicates that the step is **active**:

Active step:

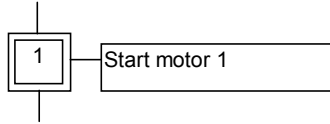


Inactive step:



The **initial situation** of an SFC program is expressed with **initial steps**. An initial step has a **double-bordered** graphic symbol. A token is automatically placed in each initial step when the program is started.

Initial step:



An SFC program must contain **at least one** initial step.

These are the attributes of a step. Such fields may be used in any of the other languages:

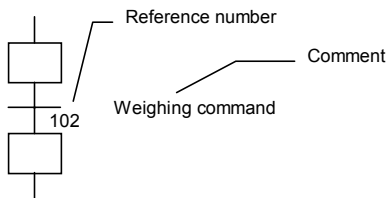
GSnnn.x..... activity of the step (boolean value)

GSnnn.t..... activation duration of the step (time value)

(where **nnn** is the reference number of the step)

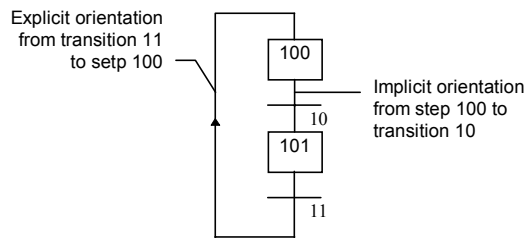
B.3.2.2 Transitions

Transitions are represented by a small horizontal bar that crosses the connection link. Each transition is **referenced** by a number, written next to the transition symbol. A main description of the transition is written on the right side of the transition symbol. This description is a **free comment** (not part of the programming language). The above information is called the **Level 1** of the transition:



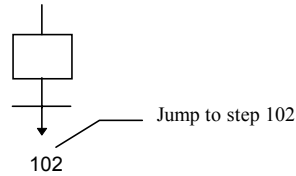
B.3.2.3 Oriented links

Single lines are used to link steps and transitions. These are oriented links. When the orientation is not explicitly given, the link is oriented from the top to the bottom.

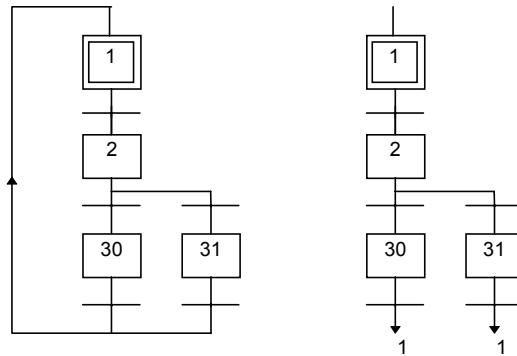


B.3.2.4 Jump to a step

Jump symbols may be used to indicate a connection link from a transition to a step, without having to draw the connection line. The jump symbol must be referenced with the number of the destination step:



A jump symbol cannot be used to represent a link from a step to a transition. Example of jumps - the following charts are equivalent:

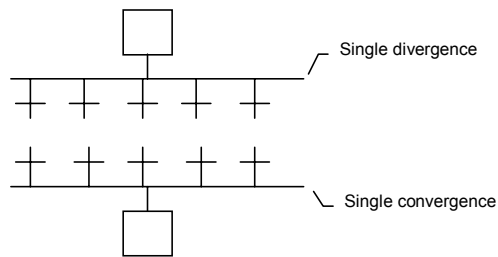


B.3.3 Divergences and convergences

Divergences are **multiple connection links** from one SFC symbol (step or transition) to many other SFC symbols. Convergences are multiple connection links from more than one SFC symbols to one other symbol. Divergences and convergences can be single or double.

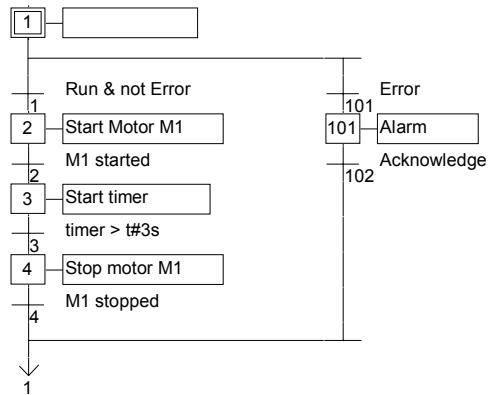
B.3.3.1 single divergences

A single divergence is a multiple link from one step to many transitions. It allows the active token to pass into one of a number of branches. A single convergence is a multiple link from many transitions to the same step. A single convergence is generally used to group the SFC branches which were started on a single divergence. Single divergences and convergences are represented by single horizontal lines.



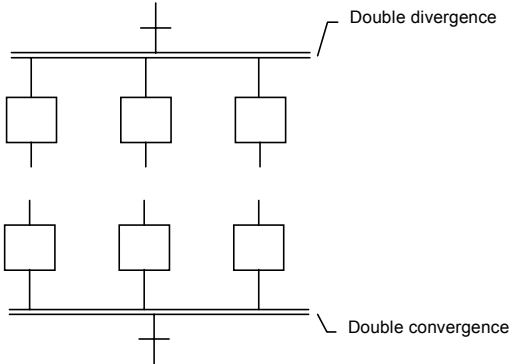
Warning: The conditions attached to the different transitions at the beginning of a single divergence are **not implicitly exclusive**. The exclusivity has to be explicitly detailed in the conditions of the transitions to ensure that only one token progresses in one branch of the divergence at run time. Below is an example of single divergence and convergence:

(* SFC program with single divergence and convergence *)



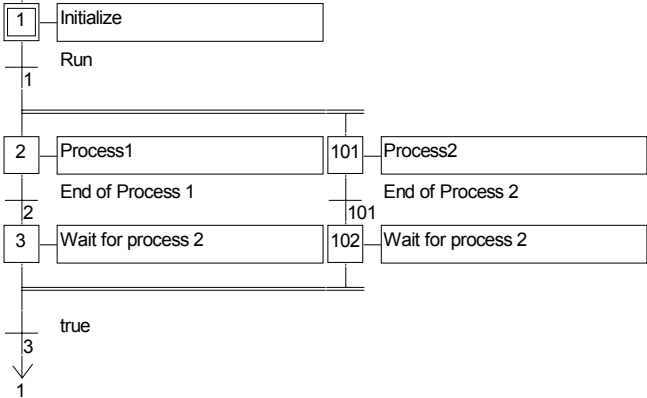
B.3.3.2 Double divergences

A double divergence is a multiple link from one transition to many steps. It corresponds to parallel operations of the process. A double convergence is a multiple link from many steps to the same transition. A double convergence is generally used to group the SFC branches started on a double divergence. Double divergences and convergences are represented by double horizontal lines.



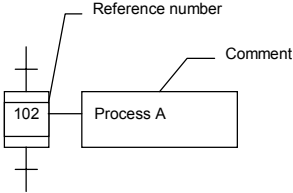
Example of double divergence and convergence:

(* SFC program with double divergence and convergence *)



B.3.4 Macro steps

A macro step is a unique representation of a unique group of steps and transitions. The body of the macro step is described separately, elsewhere in the same SFC program. It appears as a single symbol in the main SFC chart. This is the symbol used for a macro step:



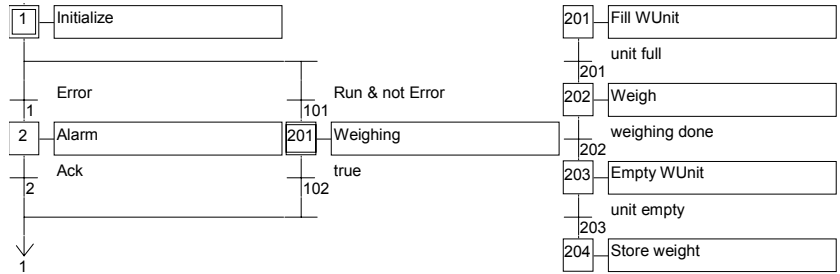
The reference number written in the macro step symbol is the reference number of the first step in the body of the macro step. The macro step body must begin with a **beginning step** and terminate with an **ending step**. The chart must be self-contained. A beginning step has no upper link (no backward transition). An ending step has no lower link (no forward transition). A macro step symbol may be put in the body of another macro step.

Warning: Because macro step is a **unique** set of steps and transitions, the same macro step cannot be used more than once in an SFC program.

Example of macro step:

(* SFC program with macro step *)
 (* Main chart *)

(* Body of the macro step *)



B.3.5 Actions within the steps

The **level 2** of an SFC step is the detailed description of the **actions** executed **during the step activity**. This description is made by using **SFC literal features**, and other languages such as Structured Text (**ST**). The basic types of actions are:

- Boolean actions
- Pulse actions programmed in ST
- Non-stored actions programmed in ST
- SFC actions

Several actions (with same or different types) can be described in the same step. The special features that enable the use of any of the other languages are:

- Calling sub-programs
- Instruction List (IL) language convention

B.3.5.1 Boolean actions

Boolean actions assign a boolean variable with the activity of the step. The boolean variable can be an output or an internal. It is assigned each time the step activity starts or stops. This is the syntax of the basic boolean actions:

<boolean_variable> (N) ; assigns the step activity signal to the variable
<boolean_variable> ; same effect (N attribute is optional)

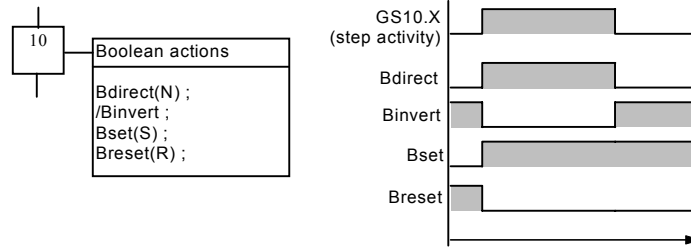
/ <boolean_variable> ; assigns the negation of the step activity signal to the variable

Other features are available to set or reset a boolean variable, when the step becomes active. This is the syntax of set and reset boolean actions:

<boolean_variable> (S) ; sets the variable to TRUE when the step activity signal becomes TRUE

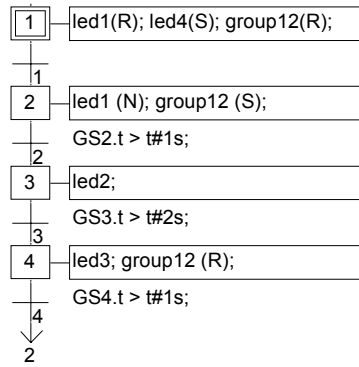
<boolean_variable> (R) ; resets the variable to FALSE when the step activity signal becomes TRUE

The boolean variable must be an OUTPUT or an INTERNAL. The following SFC programming leads to the following behaviour:



Example of boolean actions:

(* SFC program using BOOLEAN actions *)



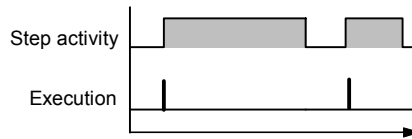
B.3.5.2 Pulse actions

A pulse action is a list of ST or IL instructions, which are executed only **once** at the **activation** of the step. Instructions are written according to the following SFC syntax:

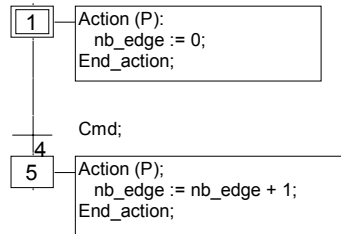
ACTION (P) :

(* ST statements *)
END_ACTION ;

The following shows the results of a pulse action:



Example of pulse action:

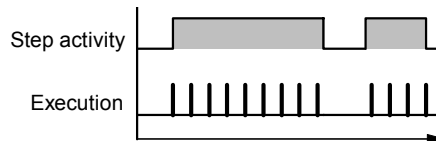


B.3.5.3 Non-stored actions

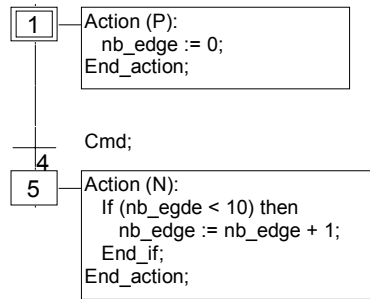
A non-stored (normal) action is a list of ST or IL instructions which are executed **at each cycle** during the whole **active** period of the step. Instructions are written according to the following SFC syntax:

ACTION (N) :
 (* ST statements *)
END_ACTION ;

The following is the results of a non-stored action:



Example of non-stored action:



B.3.5.4 SFC actions

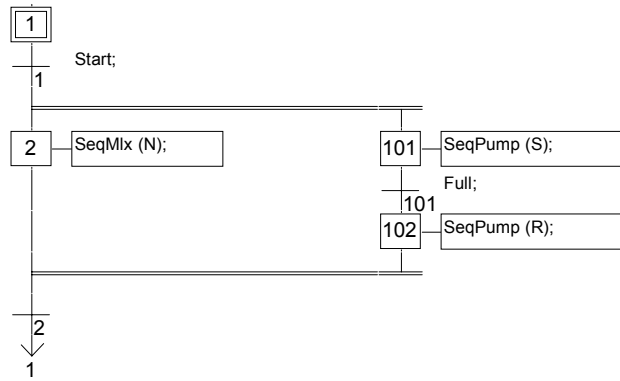
An SFC action is a child SFC sequence, started or killed according to the change of the step activity signal. An SFC action can have the **N** (Non stored), **S** (Set), or **R** (Reset) qualifier. This is the syntax of the basic SFC actions:

<child_prog> (N);	starts the child sequence when the step becomes active, and kills the child sequence when the step becomes inactive same effect (N attribute is optional)
<child_prog> ;	
<child_prog> (S);	starts the child sequence when the step becomes active. Nothing is done when the step becomes inactive
<child_prog> (R);	kills the child sequence when the step becomes active. Nothing is done when the step becomes inactive

The SFC sequence specified as an action must be a **child SFC program** of the program currently being edited. Note that using the **S** (Set) or **R** (Reset) qualifiers for an SFC action has exactly the same effect as the **GSTART** and **GKILL** statements, programmed in an **ST** pulse action.

Below is an example of an SFC action. The main SFC program is named **Father**. It has two SFC children, called **SeqMix** and **SeqPump**. The SFC programming of the father SFC program is:

(* SFC program using SFC actions *)



B.3.5.5 Calling function and function blocks from an action

Sub-programs, functions or function blocks (written in ST, IL, LD or FBD language) or "C" functions and "C" function blocks, can be directly called from an SFC action block, based on the following syntax:

For sub-programs, functions and "C" functions:

```

ACTION (P) :
    result := sub_program ( ) ;
END_ACTION;
  
```

or

```

ACTION (N) :
    result := sub_program ( ) ;
END_ACTION;
  
```

For function blocks in "C" or in ST, IL, LD, FBD:

```

ACTION (P) :
    Fbinst(in1, in2);
    result1 := Fbinst.out1;
    result2 := Fbinst.out2;
END_ACTION;
  
```

or

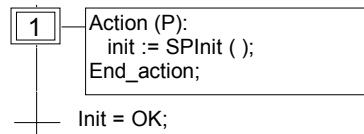
```

ACTION (N) :
    Fbinst(in1, in2);
    result1 := Fbinst.out1;
    result2 := Fbinst.out2;
END_ACTION;
  
```

Detailed syntax can be found in the ST language section.

Example of a sub-program call in action blocks:

(* SFC program with a sub-program call in an action block *)

**B.3.5.6 IL convention**

Instruction List (IL) programming may be directly entered in an SFC action block, based on the following syntax:

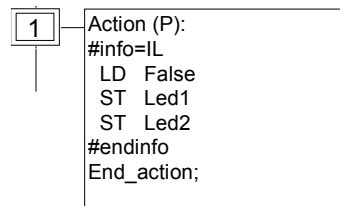
```

ACTION (P) :          (* or N *)
#info=IL
  <instruction>
  <instruction>
  ....
#endinfo
END_ACTION;

```

The special "#info=IL" and "#endinfo" keywords must be entered exactly this way, and **are case sensitive**. Space or tab characters cannot be inserted into, after or before the keywords. Below is an example of an IL program in an action block:

(* SFC program with an IL sequence in an action block *)

**B.3.6 Conditions attached to transitions**

At each transition, a **boolean expression** is attached that conditions the clearing of the transition. The condition is usually expressed with ST language or using the LD language (Quick LD editor). This is the **Level 2** of the transition. Other structures may, however, be used:

- ST language convention
- LD language convention
- IL language convention
- Calling function from a transition

Warning: When no expression is attached to the transition, the default condition is **TRUE**.

B.3.6.1 ST convention

The **Structured Text** (ST) language can be used to describe the **condition** attached to a transition. The complete expression must have **boolean** type and must be terminated by a **semicolon**, according to the following syntax:

< boolean_expression > ;

The expression may be a TRUE or FALSE constant expression, a single input or an internal boolean variable, or a combination of variables that leads to a boolean value. Below is an example of ST programming for transitions:

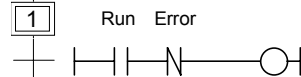
(* SFC program with ST programming for transitions *)



Run & not Error;

B.3.6.2 LD convention

The **Ladder Diagram** (LD) language can be used to describe the **condition** attached to a transition. The diagram is composed of only one rung with one coil. The coil value represents the transition value. Below is an example of LD programming for transitions:



B.3.6.3 IL convention

Instruction List (IL) programming may be directly used to describe an SFC transition, according to the following syntax:

```
#info=IL
  <instruction>
  <instruction>
  ....
#endinfo
```

The value contained by the **current result** (IL register) at the end of the IL sequence causes the resulting of the condition to be attached to the transition:

current result = 0	→	condition is FALSE
current result <> 0	→	condition is TRUE

The special "#info=IL" and "#endinfo" keywords must be entered exactly this way, and **are case sensitive**. Space or tab characters cannot be inserted into, after or before the keywords. Below is an example of IL programming for transitions:

(* SFC program with an IL program for transitions *)

```

1
├── #info=IL
│   LD Run
│   &N Error
└── #endinfo

```

B.3.6.4 Calling functions from a transition

Any sub-program or a function (written in FBD, LD, ST or IL language), or a "C" function can be called to evaluate the condition attached to a transition, according to the following syntax:

```
< sub_program > ( ) ;
```

The value returned by the sub-program or the function must be boolean and yields the resulting condition:

```

return value = FALSE    →    condition is FALSE
return value = TRUE     →    condition is TRUE

```

Example of a sub-program called in a transition:

(* SFC program with sub-program call for transitions *)

```

1
├── EvalCond ( );

```

B.3.7 SFC dynamic rules

The **five** dynamic rules of the SFC language are:

Initial situation

The initial situation is characterised by the **initial steps** which are, by definition, in the active state at the beginning of the operation. **At least one** initial step must be present in each SFC program.

Clearing of a transition

A transition is either **enabled** or **disabled**. It is said to be enabled when all immediately preceding steps linked to its corresponding transition symbol are **active**, otherwise it is disabled. A transition cannot be **cleared** unless:

- it is enabled, and
- the associated transition condition is true.

 **Changing of state of active steps**

The clearing of a transition simultaneously leads to the active state of the immediately following steps and to the inactive state of the immediately preceding steps.

 **Simultaneous clearing of transitions**

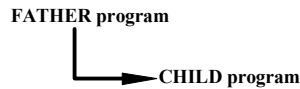
Double lines may be used to indicate transitions which have to be cleared simultaneously. If such transitions are shown separately, the activity state of preceding steps (GSnnn.x) can be used to express their conditions.

 **Simultaneous activation and deactivation of a step**

If, during operation, a step is simultaneously activated and deactivated, priority is given to the activation.

B.3.8 SFC program hierarchy

The ISaGRAF system enables the description of the vertical structure of SFC programs. SFC programs are organised in a **hierarchy tree**. Each SFC program can control (start, kill...) other SFC programs. Such programs are called **children** of the SFC program which controls them. SFC programs are linked together into a main **hierarchy tree**, using a "**father - child**" relation:



The basic rules implied by the hierarchy structure are:

- SFC programs which have no father are called "**main**" SFC programs
- Main SFC programs are activated by the system when the application starts
- A program can have several child programs
- A child of a program cannot have more than one father
- A child program can only be controlled by its father
- A program cannot control the children of one of its own children

The basic actions that a father SFC program can take to control its child program are:

Start	(GSTART) Starts the child program: activates each of its initial steps. Children of this child program are not automatically started.
Kill	(GKILL) Kills the child program by deactivating each of its active steps. All the children of the child program are also killed.
Freeze	(GFREEZE) Suspends the execution of the program (deactivates actions of each of the active steps and suspend transition calculation), and memorises the status of the program steps so the program can be restarted. All the children of the child program are also frozen.
Restart	(GRST) Restarts a frozen SFC program by reactivating all the suspended steps. Children of the program are not automatically restarted.
Get status	(GSTATUS) Gets the current status (active, inactive or frozen) of a child program.

B.4 Flow Chart language

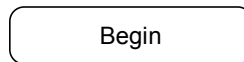
Flow Chart (FC) is a graphic language used to describe **sequential operations**. A Flow Chart diagram is composed of **Actions** and **Tests**. Between Actions and test are **oriented links** representing data flow. Multiple connection links are used to represent divergences and convergences. Actions and Tests can be described with ST, LD or IL languages. Functions and Function blocks of any language (except SFC) can be called from actions and tests. A Flow Chart program can call another Flow Chart program. The called FC program is a **sub-program** of the calling FC program.

B.4.1 FC components

Below are graphic components of the Flow Chart language:

≡ **Beginning of FC chart**

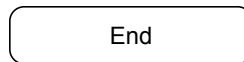
A "**begin**" symbol must appear at the beginning of a Flow Chart program. It is unique and cannot be omitted. It represents the initial state of the chart when it is activated. Below is the drawing of a "begin" symbol:



The "Begin" symbol always has a connection (on the bottom) to the other objects of the chart. A flow chart is not valid if no connection is drawn from "Begin" to another object.

≡ **Ending of FC chart**

An "**end**" symbol must appear at the end of a Flow Chart program. It is unique and cannot be omitted. It is possible that no connection is drawn to the "End" symbol (always looping chart), but "End" symbol is still drawn anyway at the bottom of the chart. It represents the final state of the chart, when its execution has been completed. Below is the drawing of an "end" symbol:



The "End" symbol generally has a connection (on the top) to the other objects of the chart. A flow chart may have no connection to the "End" object (always looping chart). The "End" object is still visible at the bottom of the chart in this case.

≡ **FC flow links**

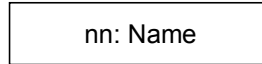
A flow **link** is a line that represents a flow between two points of the diagram. A link is always terminated by an arrow. Below is the drawing of a flow link:



Two links cannot start from the same source connection point.

▣ **FC actions**

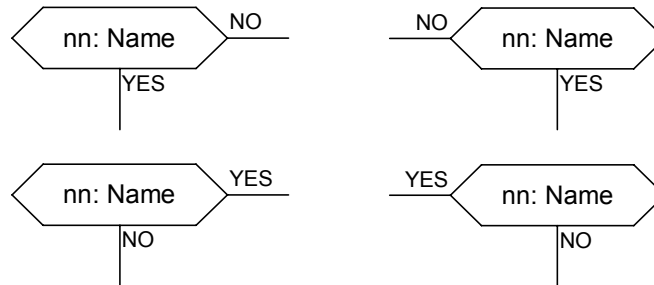
An **action** symbol represents actions to be performed. An action is identified by a number and a name. Below is the drawing of an "action" symbol:



Two different objects of the same chart cannot have the same name or logical number. Programming language for an action can be ST, LD or IL. An action is always connected with links, one arriving to it, one starting from it.

▣ **FC conditions**

A **condition** represents a boolean **test**. A condition is identified by a number and a name. According to the evaluation of attached ST, LD or IL expression, the flow is directed to "YES" or "NO" path. Below are the possible drawings for a condition symbol:



Two different objects of the same chart cannot have the same name or logical number. The programming of a test is either

- an expression in ST, or
- a single rung in LD, with no symbol attached to the unique coil, or
- several instructions in IL. The IL register (or current result) is used to evaluate the condition.

When programmed in ST text, the expression may optionally be followed by a semicolon.

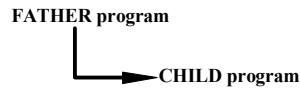
When programmed in LD, the unique coil represents the condition value. A condition equal to:

- 0 or FALSE directs the flow to NO
- 1 or TRUE directs the flow to YES

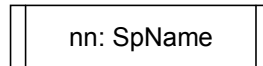
A test is always connected with an arriving link, and both forward connections must be defined.

▣ **FC sub-program**

The system enables the description of the vertical structure of FC programs. FC programs are organised in a **hierarchy tree**. Each FC program can call other FC programs. Such a program is called a **child program** of the FC program which calls them. FC programs which call FC sub-programs are called **father program**. FC programs are linked together into a main hierarchy tree, using a "father - child" relation:



A **sub-program** symbol in a Flow Chart represents a call to a Flow Chart sub-program. Execution of the calling FC program is suspended till the sub-program execution is complete. A Flow Chart sub-program is identified by a number and a name, as other programs, functions or function blocks. Below is the drawing of a "sub-program call" symbol:



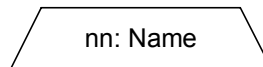
Two different objects of the same chart cannot have the same logical number. The basic rules implied by the FC hierarchy structure are:

- FC programs which have no father are called main FC programs.
- Main FC programs are activated by the system when the application starts
- A program can have several child programs
- A child of a program cannot have more than one father
- A child program can be called only by its father
- A program cannot call the children of one of its own children

The same sub-program may appear several times in the father chart. A Flow Chart sub-program call represents the complete execution of the sub chart. The father chart execution is suspended during the child chart is performed. The sub-program calling blocks must follow the same connection rules as the ones defined for action.

▬ **FC I/O specific action**

An **I/O specific action** symbol represents actions to be performed. As other actions, an I/O specific action is identified by a number and a name. The same semantic is used on standard actions and I/O specific actions. The aim of I/O specific actions is only to make the chart more readable and to give focus on non-portable parts of the chart. Using I/O specific actions is an optional feature. Below is the drawing of an "I/O specific action" symbol:



I/O specific blocks have exactly the same behaviour as standard actions. This covers their properties, ST, LD or IL programming, and connection rules.

▬ **FC connectors**

Connectors are used to represent a link between two points of the diagram without drawing it. A connector is represented as a circle and is connected to the source of the flow. The

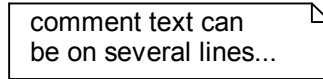
drawing of the connector is completed, on the appropriate side (depending on the direction of the data flow), by the identification of the target point (generally the name of the target symbol). Below is the standard drawing of a connector:



A connector always targets a defined Flow Chart symbol. The destination symbol is identified by its logical number.

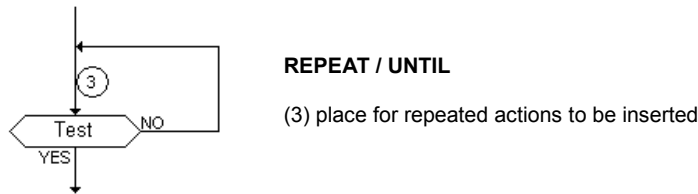
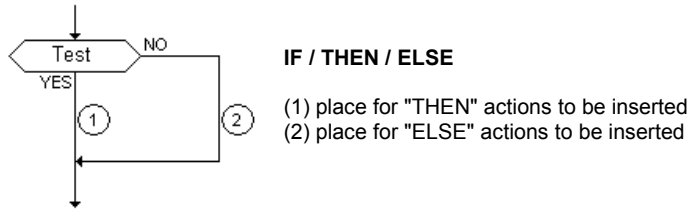
FC comments

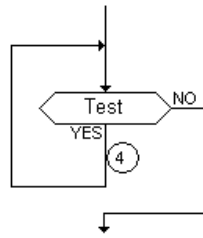
A **comment** block contains text that has no sense for the semantic of the chart. It can be inserted anywhere on an unused space of the Flow Chart document window, and is used to document the program. Below is the drawing of a "comment" symbol:



B.4.2 FC complex structures

This section shows **complex structure** examples that can be defined in a Flow Chart diagram. Such structures are combinations of basic objects linked together.



**WHILE / DO**

(3) place for repeated actions to be inserted

B.4.3 FC dynamic behaviour

The **execution** of a Flow Chart diagram can be explained as follows:

- The Begin symbol takes one target cycle
- The End symbol takes one target cycle and ends the execution of the chart. After this symbol is reached, no more actions of the chart are executed.
- The flow is broken each time an item (action, decision) is encountered that has already been reached in the same cycle. In such a case the flow will continue on the next cycle.

Note: Contrary to SFC, an action is not a stable state. There is no repetition of instructions while the action symbol is highlighted.

B.4.4 FC checking

Apart of attached ST, LD or IL programming, some other **syntactic rules** apply to flow chart itself. Below is the list of main rules:

- All "connection" points of all symbols must be wired. (connection to "End" symbol may be omitted)
- All symbols must be linked together (no isolated part should appear)
- All connectors should have valid destination

Other minor syntax errors can be reported:

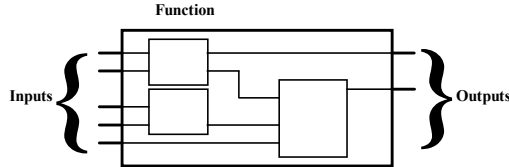
- Empty actions (no programming) are considered as steps during run time scheduling
- Empty tests (no programming) are considered as "always true"

B.5 FBD language

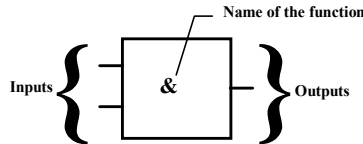
The **Functional Block Diagram** (FBD) is a graphic language. It allows the programmer to build complex procedures by taking existing **functions** from the ISaGRAF library and **wiring** them together in the graphic diagram area.

B.5.1 FBD diagram main format

FBD diagram describes a function between **input variables** and **output variables**. A function is described as a set of **elementary function blocks**. Input and output variables are connected to blocks by **connection lines**. An output of a function block may also be connected to an input of another block.



An entire function operated by an FBD program is built with standard **elementary** function blocks from the ISaGRAF library. Each function block has a fixed number of **input connection points** and a fixed number of **output connection points**. A function block is represented by a single **rectangle**. The inputs are connected on its **left** border. The outputs are connected on its **right** border. An elementary function block performs a single **function** between its inputs and its outputs. The name of the function to be performed by the block is written in its rectangle symbol. Each input or output of a block has a well-defined **type**.



Input variables of an FBD program must be connected to input connection points of function blocks. The type of each variable must be the same as the type expected for the associated input. An input for FBD diagram can be a **constant** expression, any **internal** or **input** variable, or an **output** variable.

Output variables of an FBD program must be connected to output connection points of function blocks. The type of each variable must be the same as the type expected for the associated block output. An Output for FBD diagram can be any **internal** or **output** variable, or the name of the program (for **sub-programs** only). When an output is the name of the currently edited sub-program, it represents the assignment of the return value for the sub-program (returned to the calling program).

Input and output variables, inputs and outputs of the function blocks are wired together with **connection lines**. Single lines may be used to **connect** two logical points of the diagram:

- An input variable and an input of a function block
- An output of a function block and an input of another block
- An output of a function block and an output variable

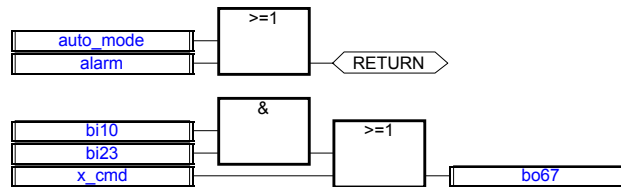
The connection is **oriented**, meaning that the line carries associated data from the left extremity to the right extremity. The left and right extremities of the connection line must be of the **same type**.

Multiple right connection can be used to broadcast an information from its left extremity to each of its right extremities. All the extremities of the connection must be of the same type.

B.5.2 RETURN statement

The "<RETURN>" keyword may occur as a diagram output. It must be connected to a boolean output connection point of a function block. The RETURN statement represents a **conditional end** of the program: if the output of the box connected to the statement has the boolean value **TRUE**, the end (remaining part) of the diagram is not executed.

(* Example of an FBD program using RETURN statement *)



(* ST equivalence: *)

```
If auto_mode OR alarm Then
  Return;
End_if;
bo67 := (bi10 AND bi23) OR x_cmd;
```

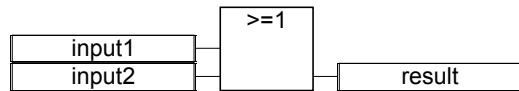
B.5.3 Jumps and labels

Labels and jumps are used to control the execution of the diagram. No other object may be connected on the right of a jump or label symbol. The following notations are used:

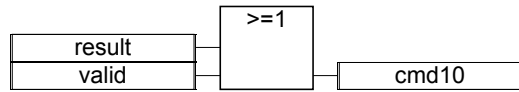
>>LAB jump to a label (label name is "LAB")
 LAB: definition of a label (label name is "LAB")

If the connection line on the **left** of the jump symbol has the boolean state **TRUE**, the execution of the program directly jumps after the corresponding label symbol.

(* Example of an FBD program using labels and jumps *)



NOMODIF:



(* IL Equivalence: *)

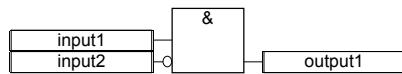
```

ld    manual
and   b1
jmpc  NOMODIF
ld    input1
or    input2
st    result
NOMODIF:
ld    result
or    valid
st    cmd10
    
```

B.5.4 Boolean negation

A single connection line with its right extremity connected to an input of a function block can be terminated by a **boolean negation**. The negation is represented by a small circle. When a boolean negation is used, the left and right extremities of the connection line must have the **BOOLEAN** type.

(* Example of an FBD program using a boolean negation *)



(* ST equivalence: *)

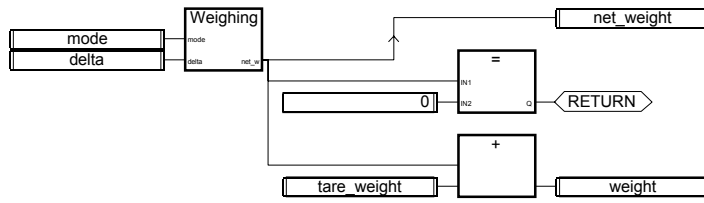
```
output1 := input1 AND NOT (input2);
```

B.5.5 Calling function or function blocks from the FBD

The FBD language enables the calling of sub-programs, functions or function blocks. A sub-program, or function or function block is represented by a function box. The name written in the box is the name of the sub-program or function or function blocks.

In case of a sub-program or a function, the return value is the only output of the function box. A function block can have more than one output.

(* Example of an FBD program using SUB PROGRAM block *)

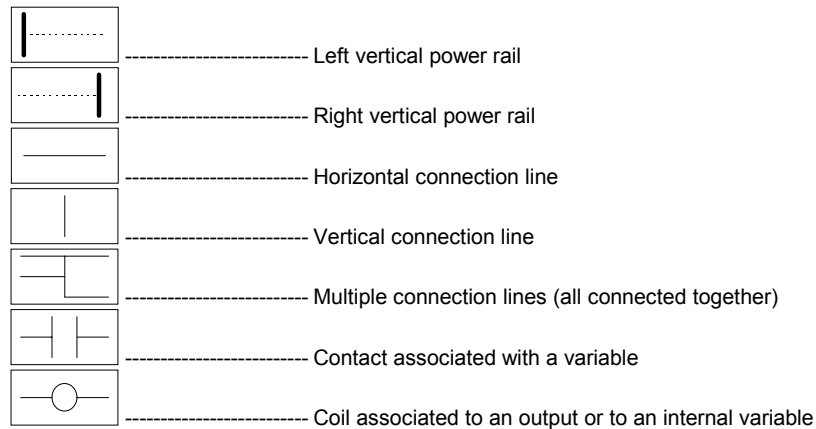


(* ST Equivalence *)

```
net_weight := Weighing (mode, delta); (* call sub-program *)
If (net_weight = 0) Then Return; End_if;
weight := net_weight + tare_weight;
```

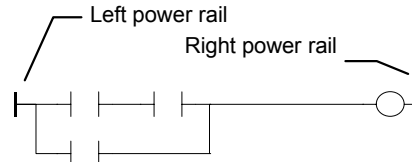
B.6 LD language

Ladder Diagram (LD) is a graphic representation of boolean equations, combining **contacts** (input arguments) with **coils** (output results). The LD language enables the description of tests and modifications of **boolean** data by placing **graphic symbols** into the program chart. LD graphic symbols are organized within the chart exactly as an electric contact diagram. LD diagrams are connected on the left side and on the right side to vertical **power rails**. These are basic graphic components of an LD diagram:

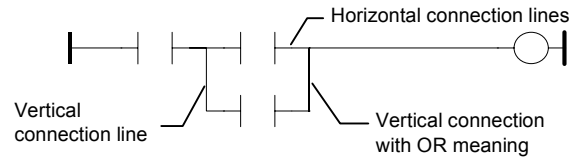


B.6.1 Power rails and connection lines

An LD diagram is limited on the left and right side by vertical lines, named **left power rail** and **right power rail** respectively.



LD diagram graphic symbols are connected to power rails or to other symbols by **connection lines**. Connection lines are horizontal or vertical.



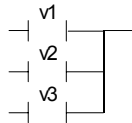
Each line segment has a boolean state **FALSE** or **TRUE**. The boolean state is the same for all the segments directly linked together. Any horizontal line connected to the left **vertical power rail** has the **TRUE** state.

B.6.2 Multiple connection

The boolean state given to a single horizontal connection line is the same on the left and on the right extremities of the line. Combining horizontal and vertical connection lines enables the building of **multiple connections**. The boolean state of the extremities of a multiple connection follows logic rules.

A **multiple connection on the left** combines **more than one** horizontal lines connected on the **left** side of a vertical line, and **one** line connected on its **right** side. The boolean state of the right extremity is the **LOGICAL OR** between all the left extremities.

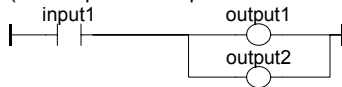
(* Example of multiple LEFT connection *)



(* right extremity state is (v1 OR v2 OR v3) *)

A **multiple connection on the right** combines **one** horizontal line connected on the **left** side of a vertical line, and **more than one** line connected on its **right** side. The boolean state of the left extremity is propagated into each of the right extremities.

(* Example of multiple RIGHT connection *)

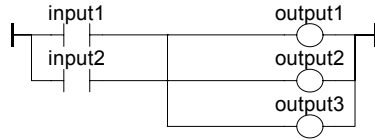


(* ST equivalence: *)

```
output1 := input1;
output2 := input1;
```

A **multiple connection on the left and on the right** combines **more than one** horizontal line connected on the **left** side of a vertical line, and **more than one** line connected on its **right** side. The boolean state of each of the right extremities is the **LOGICAL OR** between all the left extremities

(* Example of multiple LEFT and RIGHT connection *)



(* ST Equivalence: *)
 output1 := input1 OR input2;
 output2 := input1 OR input2;
 output3 := input1 OR input2;

B.6.3 Basic LD contacts and coils

There are several symbols available for input contacts:

- Direct contact
- Inverted contact
- Contacts with edge detection

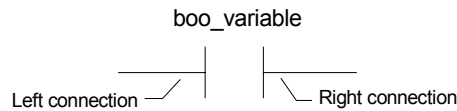
There are several symbols available for output coils:

- Direct coil
- Inverted coil
- SET coil
- RESET coil
- Coils with edge detection

The name of the variable is written above any of these graphic symbols:

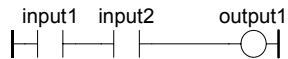
☰ **Direct contact**

A direct contact enables a **boolean operation** between a **connection line** state and a boolean **variable**.



The state of the connection line on the right of the contact is the **LOGICAL AND** between the state of the left connection line and the value of the variable associated with the contact.

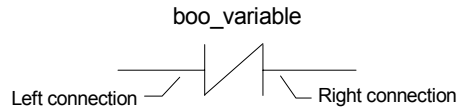
(* Example using DIRECT contacts *)



(* ST Equivalence: *)
 output1 := input1 AND input2;

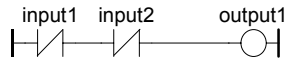
☰ **Inverted contact**

An inverted contact enables a **boolean operation** between a **connection line** state and the boolean negation of a boolean **variable**.



The state of the connection line on the right of the contact is the **LOGICAL AND** between the state of the left connection line and the **boolean negation** of the value of the variable associated with the contact.

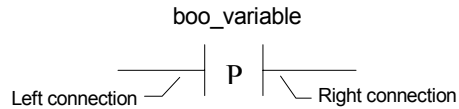
(* Example using INVERTED contacts *)



(* ST Equivalence: *)
output1 := NOT (input1) AND NOT (input2);

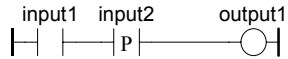
▬ **Contact with rising edge detection**

This contact (positive) enables a **boolean operation** between a **connection line** state and the rising edge of a boolean **variable**.



The state of the connection line on the right of the contact is set to **TRUE** when the state of the connection line on the left is **TRUE**, and the state of the associated variable **rises** from FALSE to TRUE. It is reset to FALSE in all other cases.

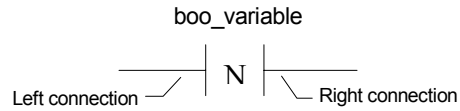
(* Example using RISING EDGE contacts *)



(* ST Equivalence: *)
output1 := input1 AND (input2 AND NOT (input2prev));
(* input2prev is the value of input2 at the previous cycle *)

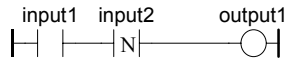
▬ **Contact with falling edge detection**

This contact (negative) enables a **boolean operation** between a **connection line** state and the falling edge of a boolean **variable**.



The state of the connection line on the right of the contact is set to **TRUE** when the state of the connection line on the left is **TRUE**, and the state of the associated variable **falls** from TRUE to FALSE. It is reset to FALSE in all other cases.

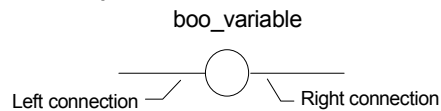
(* Example using FALLING EDGE contacts *)



(* ST Equivalence: *)
 output1 := input1 AND (NOT (input2) AND input2prev);
 (* input2prev is the value of input2 at the previous cycle *)

≡ **Direct coil**

Direct coils enable a **boolean output** of a **connection line** boolean state.

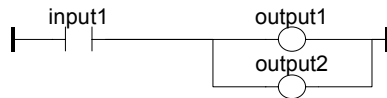


The associated variable is assigned with the boolean **state of the left connection**. The state of the left connection is propagated into the right connection. The right connection may be connected to the right vertical power rail.

The associated boolean variable must be **OUTPUT** or **INTERNAL**.

The associated name can be the name of the program (for **sub-programs** only). This corresponds to the assignment of the return value of the sub-program.

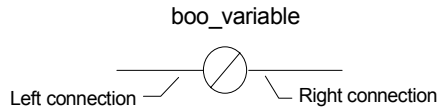
(* Example using DIRECT coils *)



(* ST Equivalence: *)
 output1 := input1;
 output2 := input1;

≡ **Inverted coil**

Inverted coils enable a **boolean output** according to the boolean **negation** of a **connection line** state.

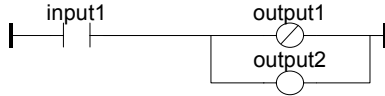


The associated variable is assigned with the boolean **negation** of the **state of the left connection**. The state of the left connection is propagated into the right connection. Right connection may be connected to the right vertical power rail.

The associated boolean variable must be **OUTPUT** or **INTERNAL**.

The associated name can be the name of the program (for **sub-programs** only). This corresponds to the assignment of the return value of the sub-program.

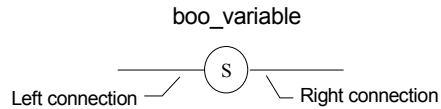
(* Example using INVERTED coils *)



(* ST Equivalence: *)
 output1 := NOT (input1);
 output2 := input1;

≡ **SET coil**

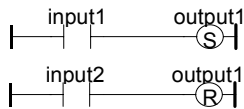
"Set" coils enable a **boolean output** of a **connection line** boolean state.



The associated variable is **SET TO TRUE** when the boolean **state of the left connection** becomes TRUE. The output variable keeps this value until an inverse order is made by a "RESET" coil. The state of the left connection is propagated into the right connection. Right connection may be connected to the right vertical power rail.

The associated boolean variable must be **OUTPUT** or **INTERNAL**.

(* Example using "SET" and "RESET" coils *)

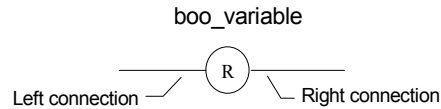


(* ST Equivalence: *)
 IF input1 THEN
 output1 := TRUE;
 END_IF;

```
IF input2 THEN
  output1 := FALSE;
END_IF;
```

⇐ **RESET coil**

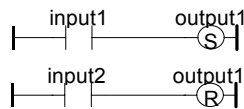
"Reset" coils enable **boolean output** of a **connection line** boolean state.



The associated variable is **RESET TO FALSE** when the boolean **state of the left connection** becomes **TRUE**. The output variable keeps this value until an inverse order is made by a "SET" coil. The state of the left connection is propagated into the right connection. Right connection may be connected to the right vertical power rail.

The associated boolean variable must be **OUTPUT** or **INTERNAL**.

(* Example using "SET" and "RESET" coils *)

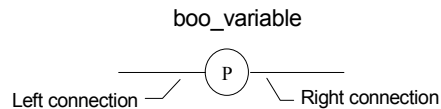


(* ST Equivalence: *)

```
IF input1 THEN
  output1 := TRUE;
END_IF;
IF input2 THEN
  output1 := FALSE;
END_IF;
```

⇐ **Coil with rising edge detection**

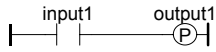
"Positive" coils enable **boolean output** of a **connection line** boolean state. This type of coils are only available using the Quick ladder editor.



The associated variable is set to **TRUE** when the boolean **state of the left connection** rises from FALSE to TRUE. The output variable resets to FALSE in all other cases. The state of the left connection is propagated into the right connection. Right connection may be connected to the right vertical power rail.

The associated boolean variable must be **OUTPUT** or **INTERNAL**.

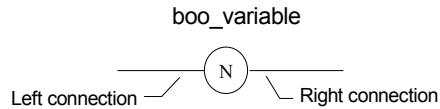
(* Example using a "Positive" coil *)



```
(* ST Equivalence: *)
IF (input1 and NOT(input1prev)) THEN
  output1 := TRUE;
ELSE
  output1 := FALSE;
END_IF;
(* input1prev is the value of input1 at the previous cycle *)
```

⊖ **Coil with falling edge detection**

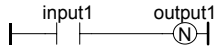
"Negative" coils enable **boolean output** of a **connection line** boolean state. This type of coils are only available using the Quick ladder editor.



The associated variable is set to **TRUE** when the boolean **state of the left connection** falls from TRUE to FALSE. The output variable resets to FALSE in all other cases. The state of the left connection is propagated into the right connection. Right connection may be connected to the right vertical power rail.

The associated boolean variable must be **OUTPUT** or **INTERNAL**.

(* Example using a "Positive" coil *)



```
(* ST Equivalence: *)
IF (NOT(input1) and input1prev) THEN
  output1 := TRUE;
ELSE
  output1 := FALSE;
END_IF;
(* input1prev is the value of input1 at the previous cycle *)
```

B.6.4 RETURN statement

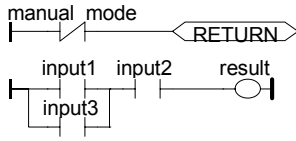
The **RETURN** label can be used as an output to represent a conditional end of the program. No connection can be put on the right of a RETURN symbol.



If the **left connection** line has the **TRUE** boolean state, the program ends without executing the equations entered on the following lines of the diagram.

Note: When the LD program is a sub-program, its name has to be associated with an output coil to set the return value (returned to the calling program).

(* Example using RETURN symbol *)



(* ST Equivalence: *)

If Not (manual_mode) Then RETURN; End_if;
 result := (input1 OR input3) AND input2;

B.6.5 Jumps and labels

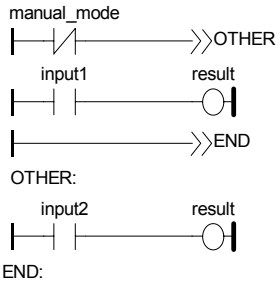
Labels, conditional and unconditional JUMPS symbols, can be used to control the execution of the diagram. No connection can be put on the right of the label and jump symbol. The following notations are used:

>>LABjump to label named "LAB"

LAB: definition of the label named "LAB"

If the **connection on the left** of the jump symbol has the **TRUE** boolean state, the program execution is driven after the label symbol.

(* Example using JUMP and LABEL symbols *)



(* IL Equivalence: *)

ldn	manual_mode
jmpc	other
ld	input1
st	result
jmp	END


```

OTHER:      ld      input2
           st      result
END:        (* end of program *)

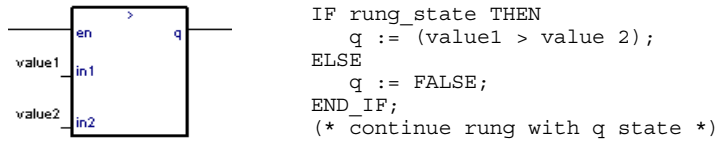
```

B.6.6 Blocks in LD

Using the Quick LD editor, you connect function boxes to boolean lines. A function can actually be an operator, a function block or a function. As all blocks do not have always a boolean input and/or a boolean output, inserting blocks in an LD diagram leads to the addition of new parameters EN, ENO to the block interface. The EN, ENO parameters are not added if you use the FBD/LD editor as you can connect the variable with the required type.

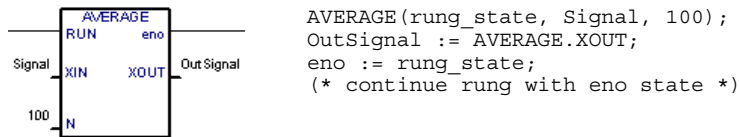
▣ The "EN" input

On some operators, functions or function blocks, the first input does not have boolean data type. As the first input must always be connected to the rung, another input is automatically inserted at the first position, called "EN". The block is executed only if the EN input is TRUE. Below is the example of a comparison operator, and the equivalent code expressed in ST:



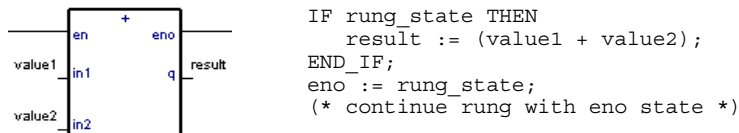
▣ The "ENO" output

On some operators, functions or function blocks, the first output does not have boolean data type. As the first output must always be connected to the rung, another output is automatically inserted at the first position, called "ENO". The ENO output always takes the same state as the first input of the block. Below is an example with AVERAGE function block, and the equivalent code expressed in ST:



▣ The "EN" and "ENO" parameters

On some cases, both EN and ENO are required. Below is an example with an arithmetic operator, and the equivalent code expressed in ST:



B.7 ST language

ST (**Structured Text**) is a high level structured language designed for automation processes. This language is mainly used to implement complex procedures that cannot be easily expressed with graphic languages. ST is the default language for the description of the actions within the steps and conditions attached to the transitions of the **SFC** language.

B.7.1 ST main syntax

An ST program is a list of ST **statements**. Each statement ends with a semi-colon (";") separator. Names used in the source code (variable identifiers, constants, language keywords...) are separated with **inactive separators** (space character, end of line or tab stops) or by **active separators**, which have a well defined significance (for example, the ">" separator indicates a "greater than" comparison. Comments may be freely inserted into the text. A comment must begin with "(" and ends with "*)". Each statement terminates with a semi-colon (";") separator. These are basic types of ST statements:

- **assignment** statement (variable := expression;)
- **sub-program** or **function** call
- **function block** call
- **selection** statements (IF, THEN, ELSE, CASE...)
- **iteration** statements (FOR, WHILE, REPEAT...)
- **control** statements (RETURN, EXIT...)
- special statements for links with other languages such as **SFC**

Inactive separators may be freely entered between active separators, constant expressions and identifiers. ST inactive separators are: **Space** (blank) character, **Tabs** and **End of line** character. Unlike line-formatted languages such as IL, end of lines may be entered anywhere in the program. The rules shown below should be followed when using inactive separators to increase ST program readability:

- Do not write more than one statement on one line
- Use tabs to indent complex statements
- Insert comments to increase readability of lines or paragraphs

B.7.2 Expression and parentheses

ST expressions combine ST **operators** and variable or constant **operands**. For each single expression (combining operands with one ST operator), the **type** of the operands must be the same. This single expression has the same type as its operands, and can be used in a more complex expression. For example :

(boo_var1 AND boo_var2)	has BOO type
not (boo_var1)	has BOO type
(sin (3.14) + 0.72)	has REAL ANALOG type
(t#1s23 + 1.78)	is an invalid expression

Parentheses are used to isolate sub parts of the expression, and to explicitly order the priority of the operations. When no parentheses are given for a complex expression, the operation sequence is implicitly given by the default **priority** between ST operators. For example:

$2 + 3 * 6$	equals $2+18=20$	because multiplication operator has a higher priority
$(2+3) * 6$	equals $5*6=30$	priority is given by parenthesis

Warning: A maximum number of **8** levels of parentheses can be nested within an expression.

B.7.3 Function or function block calls

Standard ST function calls may be used for each of following objects:

- Sub-programs
- Library functions and function blocks written in IEC languages
- "C" functions and function blocks
- Type conversion functions

▬ **Calling sub-programs or functions**

Name: name of the called sub-program
or library function written in IEC language or in "C"

Meaning: calls a ST, IL, LD or FBD sub-program or function or a "C" function and gets its return value

Syntax: **<variable> := <subprog> (<par1>, ... <parN>);**

Operands: The type of return value and calling parameters must follow the interface defined for the sub-program.

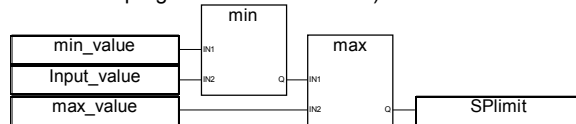
Return value: value returned by the sub-program

Sub-program calls may be used in any expression. They also may be used in an SFC transition.

Example1: Sub-program call

```
(* Main ST program *)
(* gets an analog value and converts it into a limited time value *)
ana_timeprog := SPLimit ( tprog_cmd );
appl_timer := tmr (ana_timeprog * 100);
```

(* Called FBD program named 'SPLimit' *)



Example2: Function call

(* functions used in complex expressions: min, max, right, mlen and left are standard "C" functions *)

```
limited_value := min (16, max (0, input_value) );  
rol_msg := right (message, mlen (message) - 1) + left (message, 1);
```

≡ **Calling function blocks**

Name: name of the function block instance
Meaning: calls a function block from the ISaGRAF library or from the user's library and accesses its return parameters
Syntax: **(* call of the function block *)**
<blockname> (**<p1>**, **<p2>** ...);
(gets its return parameters *)
<result> := **<blockname>**. **<ret_param1>**;
...
<result> := **<blockname>**. **<ret_paramN>**;
Operands: parameters are expressions which match the type of the parameters specified for that function block
Return value: See Syntax to get the return parameters.

Consult the ISaGRAF library to find the meaning and type of each function block parameter. The function block instance (name of the copy) must be declared in the dictionary

Example :

```
(* ST program calling a function block *)  
  
(* declare the instance of the block in the dictionary: *)  
(* trigb1 : block R_TRIG - rising edge detection *)  
  
(* function block activation from ST language *)  
trigb1 (b1);  
(* return parameters access *)  
If (trigb1.Q) Then nb_edge := nb_edge + 1; End_if;
```

B.7.4 ST specific boolean operators

The following boolean operators are specific to the ST language:

```
- REDGE          rising edge detection  
- FEDGE          falling edge detection
```

Other standard boolean operators such as:

```
- NOT            boolean negation  
- AND (&)        logical AND  
- OR            logical OR  
- XOR            logical exclusive OR
```

can be used. Their description is to be found in the section 'Standard operators, function blocks and functions'.

≡ **"REDGE" operator**

Name: **REDGE**
Meaning: evaluates the rising edge of a complete boolean expression
Syntax: **<edge>** := **REDGE** (**<boo_expression>**, **<memo_variable>**);

Operands: first operand is any boolean variable or complex expression
second operand is an internal boolean variable used to store the last state of the expression

Return value: TRUE when the expression changes from FALSE to TRUE
FALSE for all other cases

The rising edge of an expression cannot be detected more than once in the same execution cycle, using the REDGE operator. This operator can be used to describe the condition attached to an SFC transition.

Warning: The "memory" boolean variable used to store the last state of the expression cannot be used as a trigger for edges of different expressions.

When the expression is a boolean variable named "**xxx**", a unique internal variable named "**EDGE_xxx**" should be declared and used it in the REDGE expressions for this variable. This method ensures that the memory variable is not overwritten during other REDGE evaluations.

Example:

```
(* ST program using REDGE operator *)

(* this program counts the rising edges of a boolean input *)
(* Bi120 is an input boolean variable *)
(* Edge_Bi120 is the memory of the Bi120 variable state *)

If REDGE (Bi120, Edge_Bi120) Then
    Counter := Counter + 1;
End_if;
```

Note: this operator is not in the IEC1131-3 norm. You may prefer the use of R_TRIG standard block. It has been kept for compatibility reasons.

= "FEDGE" operator

Name: FEDGE

Meaning: evaluates the falling edge of a boolean expression

Syntax: <edge> := FEDGE (<boo_expression>, <memo_variable>);

Operands: first operand is any boolean variable or complex expression
second operand is an internal boolean variable used to store the last state of the expression

Return value: TRUE when the expression changes from TRUE to FALSE
FALSE for all other cases

The falling edge of an expression cannot be detected more than once in the same execution cycle, using the REDGE operator. The operator can be used to describe the condition attached to an SFC transition.

Warning: The "memory" boolean variable used to store the last state of the expression cannot be used as a trigger for edges of different expressions.

When the expression is a boolean variable named "**xxx**", a unique internal variable named "**EDGE_xxx**" should be declared and used it in the FEDGE expressions for this variable. This method ensures that the memory variable is not overwritten during other FEDGE evaluations.

Example:

```
(* ST program using FEDGE operator *)

(* this program counts the falling edges of a boolean input *)
(* Bi120 is an input boolean variable *)
(* Edge_Bi120 is the memory of the Bi120 variable state *)

If FEDGE (Bi120, Edge_Bi120) Then
    Counter := Counter + 1;
End_if;
```

Note: this operator is not in the IEC1131-3 norm. You may prefer the use of F_TRIG standard block. It has been kept for compatibility reasons.

B.7.5 ST basic statements

The basic statements of the ST language are:

- Assignment
- RETURN statement
- IF-THEN-ELSIF-ELSE structure
- CASE statement
- WHILE iteration statement
- REPEAT iteration statement
- FOR iteration statement
- EXIT statement

≡ **Assignment**

Name:	:=
Meaning:	assigns a variable to an expression
Syntax:	<variable> := <any_expression> ;
Operands:	variable must be internal or output variable and expression must have the same type

The expression can be a call to a sub-program or a function from the ISaGRAF library

Example:

```
(* ST program with assignments *)

(* variable <=<= variable *)
bo23 := bo10;

(* variable <=<= expression *)
bo56 := bx34 OR alm100 & (level >= over_value);
result := (100 * input_value) / scale;

(* assignment with sub-program return value *)
rc := PSelect ( );
```

```
(* assignment with function call *)
limited_value := min (16, max (0, input_value) );
```

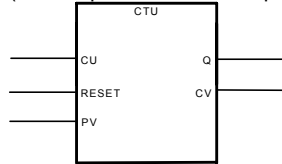
▬ **RETURN statement**

Name: RETURN
Meaning: terminates the execution of the current program
Syntax: RETURN ;
Operands: (none)

In an SFC action block, the RETURN statement indicates the end of the execution of that block only.

Example:

```
(* FBD specification of the program: programmable counter *)
```



```
(* ST implementation of the program, using RETURN statement *)
```

```
If not (CU) then
    Q := false;
    CV := 0;
    RETURN; (* terminates the program *)
end_if;

if R then
    CV := 0;
else
    if (CV < PV) then
        CV := CV + 1;
    end_if;
end_if;
Q := (CV >= PV);
```

▬ **IF-THEN-ELSIF-ELSE statement**

Name: IF ... THEN ... ELSIF ... THEN ... ELSE ... END_IF
Meaning: executes one of two lists of ST statements
selection is made according to the value
of a boolean expression
Syntax: IF <boolean_expression> THEN
<statement> ;
<statement> ;
...
ELSIF <boolean_expression> THEN

```
<statement> ;  
<statement> ;  
...  
ELSE  
<statement> ;  
<statement> ;  
...  
END_IF;
```

The ELSE and ELSIF statements are optional. If the ELSE statement is not written, no instruction is executed when the condition is FALSE.

Example:

```
(* ST program using IF statement *)
```

```
IF manual AND not (alarm) THEN  
    level := manual_level;  
    bx126 := bi12 OR bi45;  
ELSIF over_mode THEN  
    level := max_level;  
ELSE  
    level := (lv16 * 100) / scale;  
END_IF;
```

```
(* IF structure without ELSE *)
```

```
If overflow THEN  
    alarm_level := true;  
END_IF;
```

≡ **CASE statement**

Name: **CASE ... OF ... ELSE ... END_CASE**
Meaning: executes one of several lists of ST statements
selection is made according to an integer expression
Syntax: **CASE <integer_expression> OF**
 <value> : <statements> ;
 <value> , <value> : <statements> ;
 ...
 ELSE
 <statements> ;
 END_CASE;

Case values must be integer constant expressions. Several values, separated by comas, can lead to the same list of statements. The ELSE statement is optional.

Example:

```
(* ST program using CASE statement *)
```

```
CASE error_code OF  
    255: err_msg := 'Division by zero';
```



```

        fatal_error := TRUE;
    1:    err_msg := 'Overflow';
    2, 3: err_msg := 'Bad sign';
ELSE
    err_msg := 'Unknown error';
END_CASE;

```

▣ **WHILE statement**

Name: **WHILE ... DO ... END_WHILE**
Meaning: iteration structure for a group of ST statements
the "continue" condition is evaluated BEFORE any iteration
Syntax: **WHILE <boolean_expression> DO**
 <statement> ;
 <statement> ;
 ...
END_WHILE ;

Warning: Because ISaGRAF is a **synchronous** system, input variables are not refreshed during WHILE iterations. The change of state of an input variable cannot be used to describe the condition of a WHILE statement.

Example:

(* ST program using WHILE statement *)

(* this program uses specific "C" functions to read characters *)
(* on a serial port *)

```

string := ""; (* empty string *)
nbchar := 0;

WHILE ((nbchar < 16) & ComIsReady ( )) DO
    string := string + ComGetChar ( );
    nbchar := nbchar + 1;
END_WHILE;

```

▣ **REPEAT statement**

Name: **REPEAT ... UNTIL ... END_REPEAT**
Meaning: iteration structure for a group of ST statements
the "continue" condition is evaluated AFTER any iteration
Syntax: **REPEAT**
 <statement> ;
 <statement> ;
 ...
UNTIL <boolean_condition>
END_REPEAT ;

Warning: Because ISaGRAF is a **synchronous** system, input variables are not refreshed during REPEAT iterations. The change of state of an input variable cannot be used to describe the ending condition of a REPEAT statement.

Example:

```
(* ST program using REPEAT statement *)
```

```
(* this program uses specific "C" functions to read characters *)  
(* on a serial port *)
```

```
string := ""; (* empty string *)  
nbchar := 0;  
IF ComIsReady ( ) THEN  
  REPEAT  
    string := string + ComGetChar ( );  
    nbchar := nbchar + 1;  
  UNTIL ( (nbchar >= 16) OR NOT (ComIsReady ( )) )  
  END_REPEAT;  
END_IF;
```

▬ FOR statement

Name:	FOR ... TO ... BY ... DO ... END_FOR
Meaning:	executes a limited number of iterations, using an integer analog index variable
Syntax:	FOR <index> := <mini> TO <maxi> BY <step> DO <statement> ; <statement> ; END_FOR;
Operands:	index: internal analog variable increased at any loop mini: initial value for index (before first loop) maxi: maximum allowed value for index step: index increment at each loop

The [BY step] statement is optional. If not specified, the increment step is 1

Warning: Because ISaGRAF is a **synchronous** system, input variables are not refreshed during FOR iterations.

This is the "while" equivalent of a FOR statement:

```
index := mini;  
while (index <= maxi) do  
    <statement> ;  
    <statement> ;  
    index := index + step;  
end_while;
```

Example:

```
(* ST program using FOR statement *)  
(* this program extracts the digit characters of a string *)
```

```
length := mlen (message);
```

```
target := ""; (* empty string *)
FOR index := 1 TO length BY 1 DO
  code := ascii (message, index);
  IF (code >= 48) & (code <= 57) THEN
    target := target + char (code);
  END_IF;
END_FOR;
```

≡ **EXIT statement**

Name: EXIT
Meaning: exit from a FOR, WHILE or REPEAT iteration statement
Syntax: EXIT;
Operands: (none)

The EXIT is commonly used within an IF statement, inside a FOR, WHILE or REPEAT block.

Example:

(* ST program using EXIT statement *)
 (* this program searches for a character in a string *)

```
length := mlen (message);
found := NO;
FOR index := 1 TO length BY 1 DO
  code := ascii (message, index);
  IF (code = searched_char) THEN
    found := YES;
    EXIT;
  END_IF;
END_FOR;
```

B.7.6 ST extensions

The following functions are extensions of the ST language:

- TSTART - TSTOP: timer control

The following statements and functions are available to control the execution of the SFC child programs. They may be used inside ACTION(): ... END_ACTION; blocks in SFC steps.

- GSTART	starts an SFC program
- GKILL	kills an SFC program
- GFREEZE	freezes an SFC program
- GRST	restarts a frozen SFC program
- GSTATUS	gets current status of an SFC program

Warning: These functions are not in the IEC 1131-3 norm.

Easy equivalent can be found for GSTART and GKILL using the following syntax in the SFC step:

```
child_name(S); (* equivalent to GSTART(child_name); *)
child_name(R); (* equivalent to GKILL(child_name); *)
```

The following fields can be used to access the status of an SFC step:

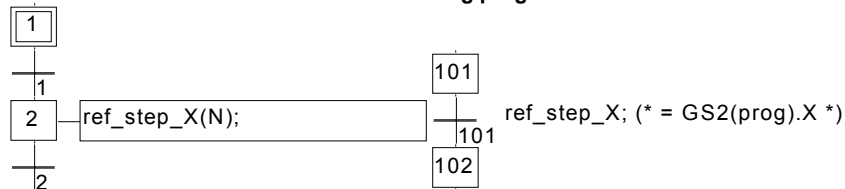
GSnnn.x boolean value that represents the activity of the step
GSnnn.t time elapsed since the last activation of the step
 ("nnn" is the reference number of the SFC step)

It is also possible to test the activity of a step declared in another SFC program, by using the following syntax:

GSnnn(progname).x

Warning: referencing a step of an other program, using this syntax is not in the IEC 1131-3 norm. An easy way to do the same respecting IEC rules, is to declare a global boolean variable in the dictionary which will represent the step activity to be tested (for example ref_step_X). Then you insert in the step, the variable with the N qualifier (ref_step_X(N);). Then in the program which wants to test the activity of the step, you use the variable.

Prog program **the other program which needs step activity of Prog program**

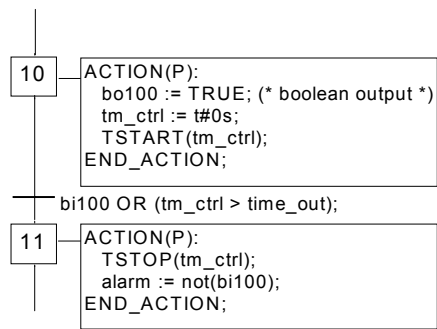


TSTART statement

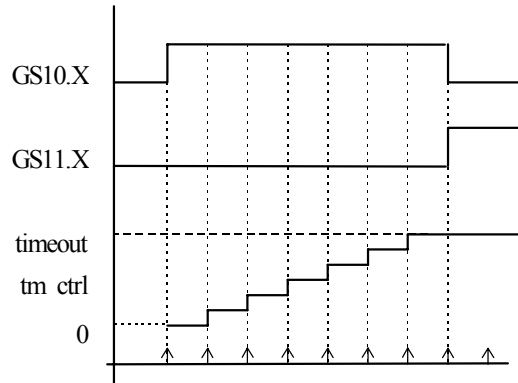
Name: **TSTART**
Meaning: starts the counting of a timer variable
 timer value is not modified by the TSTART command, i.e. the counting starts from the current value of the timer.
Syntax: **TSTART (<timer_variable>);**
Operands: any inactive timer variable
Return value: (none)

Example:

(* SFC program using TSTART and TSTOP statements *)



Time diagram if bi100 is always FALSE:



The timer keeps the same value during one cycle.

≡ **TSTOP statement**

Name: TSTOP
Meaning: stops updating a timer variable
 timer value is not modified by the TSTOP command
Syntax: TSTOP (<timer_variable>);
Operands: any active timer variable
Return value: (none)

Example: See TSTART (the function is described above)

≡ **GSTART statement**

Name: GSTART
Meaning: starts a child SFC program by putting a token
 into each of its initial steps
Syntax: GSTART (<child_program>);
Operands: the specified SFC program must be a child of the one

in which the statement is written

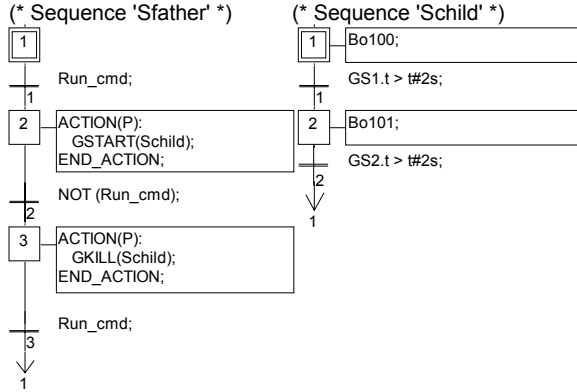
Return value: (none)

Children of the child program are not automatically started by the GSTART statement.

Note: As GSTART is not in the IEC 1131-3 norm, prefer the use of the S qualifier, with the following syntax to start a child SFC:

Child_name(S);

Example: Use of GSTART and GKILL



⊖ **GKILL statement**

Name: GKILL
Meaning: kills a child SFC program by removing the tokens currently existing in its steps
Syntax: GKILL (<child_program>);
Operands: the specified SFC program must be a child of the one in which the statement is written
Return value: (none)

Children of the child program are automatically killed with the specified program.

Note: As GKILL is not in the IEC 1131-3 norm, prefer the use of the R qualifier, with the following syntax to kill a child SFC:

Child_name(R);

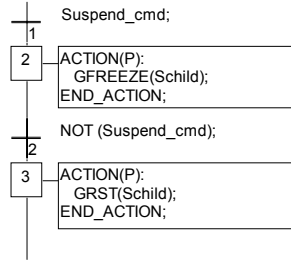
Example: See GSTART (function described above)

⊖ **GFREEZE statement**

Name: GFREEZE
Meaning: Suspends the execution of a child SFC program. Frozen program can be restarted by the GRST statement.
Syntax: GFREEZE (<child_program>);
Operands: the specified SFC program must be a child of the one in which the statement is written
Return value: (none)

Children of the child program are automatically frozen along with the specified program.
Note: GFREEZE is not in the IEC 1131-3 norm.

Example:



▬ **GRST statement**

Name: GRST
Meaning: Restarts a child SFC program frozen by the GFREEZE statement.
Syntax: GRST (<child_program>);
Operands: the specified SFC program must be a child of the one in which the statement is written
Return value: (none)

Children of the child program are automatically restarted by the GRST statement

Note: GRST is not in the IEC 1131-3 norm.

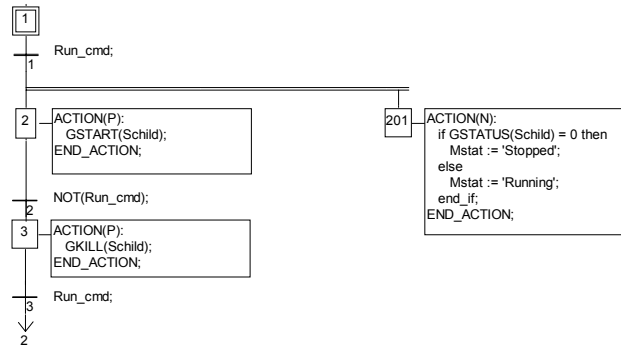
Example: See GFREEZE (function described above)

▬ **GSTATUS statement**

Name: GSTATUS
Meaning: returns the current status of an SFC program
Syntax: <ana_var> := GSTATUS (<child_program>);
Operands: the specified SFC program must be a child of the one in which the statement is written
Return value: 0 = program is inactive (killed)
 1 = program is active (started)
 2 = program is frozen

Note: GSTATUS is not in the IEC 1131-3 norm.

Example:



B.8 IL language

Instruction List, or **IL** is a low level language. Instructions always relate to the **current result** (or **IL register**). The operator indicates the operation that must be made between the current value and the operand. The result of the operation is stored again in the current result.

B.8.1 IL main syntax

An IL program is a list of **instructions**. Each instruction must begin on a new line, and must contain an **operator**, completed with optional **modifiers** and, if necessary, for the specific operation, one or more **operands**, separated with commas (','). A **label** followed by a colon (':') may precede the instruction. If a **comment** is attached to the instruction, it must be the last component of the line. Comments always begin with '**/***' and ends with '***/**'. Empty lines may be entered between instructions. Comments may be put on empty lines. Below are examples of instruction lines:

<i>Label</i>	<i>Operator</i>	<i>Operand</i>	<i>Comments</i>
Start:	LD	IX1	(* push button *)
	ANDN	MX5	(* command is not forbidden *)
	ST	QX2	(* start motor *)

≡ **Labels**

A **label** followed by a colon (':') may precede the instruction. A label can be put on an empty line. Labels are used as operands for some operations such as jumps. Naming labels must conform to the following rules:

- name cannot exceed **16** characters
- first character must be a **letter**
- following characters must be **letters**, **digits** or **'_'** character

The same name cannot be used for more than one label in the same IL program. A label can have the same name as a variable.

≡ **Operator modifiers**

The available operator modifiers are shown below. The modifier character must complete the name of the operator, with no blank characters between them:

N	boolean negation of the operand
(delayed operation
C	conditional operation

The **'N'** modifier indicates a boolean negation of the operand. For example, the instruction **ORN IX12** is interpreted as: **result := result OR NOT (IX12)**.

The parenthesis **'(** modifier indicates that the evaluation of the instruction must be delayed until the closing parenthesis **)'** operator is encountered.

The '**C**' modifier indicates that the attached instruction must be executed only if the current result has the boolean value TRUE (different than 0 for non-boolean values). The '**C**' modifier can be combined with the '**N**' modifier to indicate that the instruction must be executed only if the current result has the boolean value FALSE (or 0 for non-boolean values).

= Delayed operations

Because there is only one IL register (current result), some operations may have to be delayed, so that the execution order or the instructions can be changed. Parentheses are used to indicate delayed operations:

'(' is a modifier indicates the operation to be delayed
)' is an operator executes the delayed operation

The opening parenthesis '(' modifier indicates that the evaluation of the instruction must be delayed until the closing parenthesis ')' operator is encountered. For example, following sequence:

```

AND( IX12
OR  IX35
)

```

is interpreted as:

```

result := result AND ( IX12 OR IX35 )

```

B.8.2 IL operators

The following table summarizes the standard operators of the IL language:

<i>Operator</i>	<i>Modifiers</i>	<i>Operand</i>	<i>Description</i>
LD	N	Variable, constant	Loads operand
ST	N	Variable	Stores current result
S		BOO variable	Sets to TRUE
R		BOO variable	Resets to FALSE
AND	N (BOO	boolean AND
&	N (BOO	boolean AND
OR	N (BOO	boolean OR
XOR	N (BOO	exclusive OR
ADD	(variable, constant	Addition
SUB	(variable, constant	Subtraction
MUL	(variable, constant	Multiplication
DIV	(variable, constant	Division
GT	(variable, constant	Test: >
GE	(variable, constant	Test: >=
EQ	(variable, constant	Test: =
LE	(variable, constant	Test <=
LT	(variable, constant	Test <
NE	(variable, constant	Test <>

CAL	C N	Function block instance name	Calls a function block
JMP	C N	Label	Jumps to label
RET	C N		Returns from sub-program
)			Executes delayed operation

In the next section, only operators which are specific to the IL language are described, other standard operators can be found in the section "standard operators, function blocks and functions".

LD operator

Operation loads a value in the current result
Allowed modifiers N
Operand constant expression
internal, input or output variable

Example:

```
(* EXAMPLES OF LD OPERATIONS *)
LDex:      LD      false      (* result := FALSE boolean constant *)
           LD      true       (* result := TRUE boolean constant *)
           LD      123        (* result := integer constant *)
           LD      123.1      (* result := real constant *)
           LD      t#3ms     (* result := time constant *)
           LD      boo_var1  (* result := boolean variable *)
           LD      ana_var1  (* result := analog variable *)
           LD      tmr_var1  (* result := timer variable *)
           LDN     boo_var2  (* result := NOT ( boolean variable ) *)
```

ST operator

Operation stores the current result in a variable
the current result is not modified by this operation
Allowed modifiers N
Operand internal or output variable

Example:

```
(* EXAMPLES OF ST OPERATIONS *)
STboo:     LD      false
           ST      boo_var1 (* boo_var1 := FALSE *)
           STN    boo_var2 (* boo_var2 := TRUE *)
STana:     LD      123
           ST      ana_var1 (* ana_var1 := 123 *)
STtmr:     LD      t#12s
           ST      tmr_var1 (* tmr_var1 := t#12s *)
```

S operator

Operation: stores the boolean value TRUE in a boolean variable, if the current result has the boolean value TRUE. No operation is processed if current result is FALSE. The current result is not modified by this operation

Allowed modifiers: (none)
Operand: output or internal boolean variable

Example:

```
(* EXAMPLES OF S OPERATIONS *)
SETex:    LD      true      (* current result := TRUE *)
          S      boo_var1  (* boo_var1 := TRUE *)
          (* current result is not modified *)
          LD      false     (* current result := FALSE *)
          S      boo_var1  (* nothing done - boo_var1 unchanged *)
```

≡ **R operator**

Operation stores the boolean value FALSE in a boolean variable, if the current result has the boolean value TRUE. No operation is processed if current result is FALSE. The current result is not modified by this operation

Allowed modifiers (none)

Operand output or internal boolean variable

Example:

```
(* EXAMPLES OF R OPERATIONS *)
RESETex:  LD      true      (* current result := TRUE *)
          R      boo_var1  (* boo_var1 := FALSE *)
          (* current result is not modified *)
          ST      boo_var2  (* boo_var2 := TRUE *)
          LD      false     (* current result := FALSE *)
          R      boo_var1  (* nothing done - boo_var1 unchanged *)
```

≡ **JMP operator**

Operation jumps to the specified label

Allowed modifiers C N

Operand label defined in the same IL program

Example:

(* the following example tests the value of an analog selector (0 or 1 or 2)
(* to set one from 3 output booleans. Test "is equal to 0" is made with
(* the JMPC operator *)

```
JMPex:    LD      selector  (* selector is 0 or 1 or 2 *)
          BOO      (* conversion to boolean *)
          JMPC     test1    (* if selector = 0 then *)
          LD      true
          ST      bo0      (* bo0 := true *)
          JMP      JMPend  (* end of the program *)
test1:    LD      selector
          SUB      1        (* decrease selector: is now 0 or 1 *)
          BOO      (* conversion to boolean *)
```

```

                JMPC   test2    (* if selector = 0 then *)
                LD     true
                ST     bo1      (* bo1 := true *)
test2:          JMP    JMPend   (* end of the program *)
                LD     true      (* last possibility *)
                ST     bo2      (* bo2 := true *)
JMPend:
                (* end of the IL program *)

```

⌵ **RET operator**

Operation ends the current instruction list. If the IL sequence is a sub-program, the current result is returned to the calling program

Allowed modifiers C N

Operand (none)

Example:

(* the following example tests the value of an analog selector (0 or 1 or 2)
 (* to set one from 3 output booleans. Test "is equal to 0" is made with
 (* the JMPC operator

```

JMPex:         LD     selector  (* selector is 0 or 1 or 2 *)
                BOO                    (* conversion to boolean *)
                JMPC   test1    (* if selector = 0 then *)
                LD     true
                ST     bo0      (* bo0 := true *)
                RET                    (* end - return 0 *)
                (* decrease selector *)

test1:         LD     selector
                SUB     1          (* selector is now 0 or 1 *)
                BOO                    (* conversion to boolean *)
                JMPC   test2    (* if selector = 0 then *)
                LD     true
                ST     bo1      (* bo1 := true *)
                LD     1          (* load real selector value *)
                RET                    (* end - return 1 *)
                (* last possibility *)

test2:         RETNC              (* returns if the selector has *)
                (* an invalid value *)

                LD     true
                ST     bo2      (* bo2 := true *)
                LD     2          (* load real selector value *)
                (* end - return 2 *)

```

⌵ **"") operator**

Operation executes a delayed operation. The delayed operation was notified by '('

Allowed modifiers (none)

Operand (none)

Example:

(* The following program interleaves delayed operations: *)
 (* res := a1 + (a2 * (a3 - a4) * a5) + a6; *)

```
Delayed:  LD      a1      (* result := a1; *)
          ADD(   a2      (* delayed ADD - result := a2; *)
          MUL(   a3      (* delayed MUL - result := a3; *)
          SUB    a4      (* result := a3 - a4; *)
          )      (* execute delayed MUL - result := a2 * (a3-a4); *)
          MUL    a5      (* result := a2 * (a3 - a4) * a5; *)
          )      (* execute delayed ADD *)
          ADD    a6      (* result := a1 + (a2 * (a3 - a4) * a5); *)
          ST     res     (* result := a1 + (a2 * (a3 - a4) * a5) + a6; *)
                          (* store current result in variable res *)
```

▬ **Calling sub-programs or functions**

A sub-program or a function (written in any of the IL, ST, LD, FBD or "C" language) is called from the IL language, using its name as an operator.

Operation executes a sub-program or a function - the value returned by the sub-program or function is stored into the IL current result

Allowed modifiers (none)

Operand The first calling parameter must be stored in the current result before the call. The following ones are expressed in the operand field, separated by comas.

Example:

(* Calling program : converts an analog value into a time value *)

```
Main:    LD      bi0
          SUBPRO  bi1,bi2 (* call sub-program to get analog value *)
          ST     result  (* result := value returned by sub-program *)
          GT     vmax    (* test value overflow *)
          RETC   (* return if overflow *)
          LD     result
          MUL    1000    (* converts seconds in milliseconds *)
          TMR   (* converts to a timer *)
          ST     tmval   (* stores converted value in a timer *)
```

(* Called sub-program named 'SUBPRO' : evaluates the analog value *)

(* given as a binary value on three boolean inputs: in0, in1, in2 are the three boolean input parameters of the sub-program *)

```
LD      in2
ANA     (* result = ana (in2); *)
MUL    2 (* result := 2*ana (in2); *)
ST     temporary (* temporary := result *)
LD     in1
ANA
ADD    temporary (* result := 2*ana (in2) + ana (in1); *)
MUL    2 (* result := 4*ana (in2) + 2*ana (in1); *)
ST     temporary (* temporary := result *)
```

```

LD      in0
ANA
ADD     temporary (* result := 4*ana (in2) + 2*ana (in1)+ana (in0); *)
ST      SUBPRO (* return current result to calling program *)

```

▬ **Calling function blocks: CAL operator**

Operation calls a function block

Allowed modifiers C N

Operand Name of the function block instance.
The input parameters of the blocks must be assigned before the call using LD/ST operations sequence.
Output parameters are known if used.

Example1:

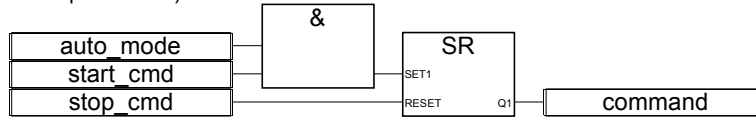
(* Calling function block SR : SR1 is an instance of SR *)

```

LD      auto_mode
AND     start_cmd
ST      SR1.set1
LD      stop_cmd
ST      SR1.reset
CAL     SR1
LD      SR1.Q1
ST      command

```

(* FBD equivalent : *)



Example 2

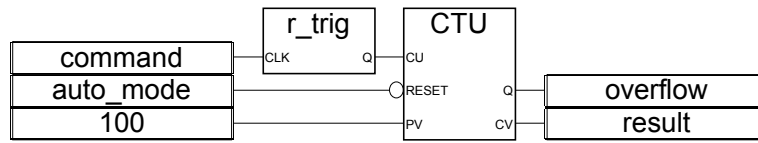
(*We suppose R_TRIG1 is an instance of R_TRIG block and CTU1 is an instance of CTU block*)

```

LD      command
ST      R_TRIG1.clk
CAL     R_TRIG1
LD      R_TRIG1.Q
ST      CTU1.cu
LDN     auto_mode
ST      CTU1.reset
LD      100
ST      CTU1.pv
CAL     CTU1
LD      CTU1.Q
ST      overflow
LD      CTU1.cv
ST      result

```

(* FBD equivalent: *)



B.9 Standard operators, function blocks and functions

B.9.1 Standard operators

The following are standard operators of the IEC languages.

Data manipulation	Assignment, Analog negation
Boolean operations	Boolean AND
	Boolean OR
	Boolean Exclusive OR
Arithmetic operations	Addition
	Subtraction
	Multiplication
	Division
Logic operations	Analog bit to bit AND mask
	Analog bit to bit OR mask
	Analog bit to bit Exclusive OR mask
	Bit to bit negation
Comparison tests	Less than
	Less or equal to
	Greater than
	Greater or equal to
	Is equal to
	Is not equal to
Data conversion	Convert to Boolean
	Convert to Integer Analog
	Convert to Real Analog
	Convert to Timer
	Convert to Message
Other	Message concatenation
	System access
	Operate I/O channel

1 gain



Arguments:

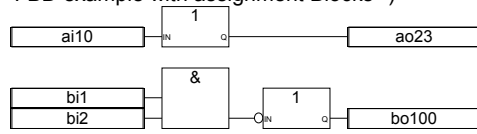
IN	any type
Q	any type

Description:

assignment of one variable into another one

This block is very useful to directly link a diagram input and a diagram output. It can also be used (with a boolean negation line) to invert the state of a line connected to a diagram output.

(* FBD example with assignment Blocks *)



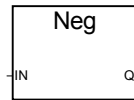
(* ST equivalence: *)

ao23 := ai10;
bo100 := NOT (bi1 AND bi2);

(* IL equivalence: *)

LD ai10
ST ao23
LD bi1
AND bi2
STN bo100

NEG



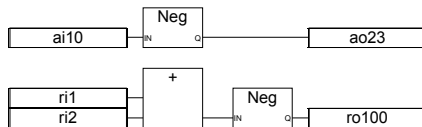
Arguments:

IN	INT-REAL	input and output must have same format
Q	INT-REAL	

Description:

Assignment of the negation of a variable.

(* FBD example with negation Blocks *)



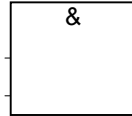
(* ST equivalence: *)

ao23 := - (ai10);
ro100 := - (ri1 + ri2);

(* IL equivalence: *)

LD ai10
MUL -1
ST ao23

LD	ri1
ADD	ri2
MUL	-1.0
ST	ro100

& AND

Note: For this operator, the number of its inputs can be extended to more than two.

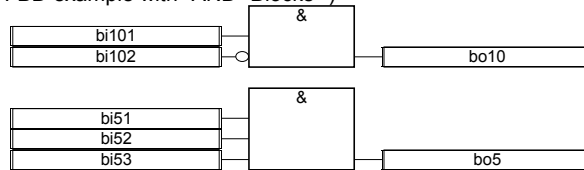
Arguments:

(inputs)	BOOLEAN	
output	BOOLEAN	boolean AND of the input terms

Description:

Boolean AND between two or more terms.

(* FBD example with "AND" Blocks *)



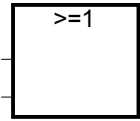
(* ST equivalence: *)

bo10 := bi101 AND NOT (bi102);
bo5 := (bi51 AND bi52) AND bi53;

(* IL equivalence *)

LD	bi101	(* current result := bi101 *)
ANDN	bi102	(* current result := bi101 AND not(bi102) *)
ST	bo10	(* bo10 := current result *)
LD	bi51	(* current result := bi51;
&	bi52	(* current result := bi51 AND bi52 *)
&	bi53	(* current result := (bi51 AND bi52) AND bi53 *)
ST	bo5	(* bo5 := current result *)

>=1 OR

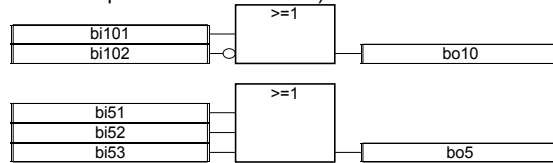


Note: For this operator, the number of its inputs can be extended to more than two.

Arguments:
 (inputs) BOOLEAN
 output BOOLEAN boolean OR of the input terms

Description:
 Boolean OR of two or more terms.

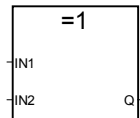
(* FBD example with "OR" Blocks *)



(* ST equivalence: *)
 bo10 := bi101 OR NOT (bi102);
 bo5 := (bi51 OR bi52) OR bi53;

(* IL equivalence: *)
 LD bi101
 ORN bi102
 ST bo10
 LD bi51
 OR bi52
 OR bi53
 ST bo5

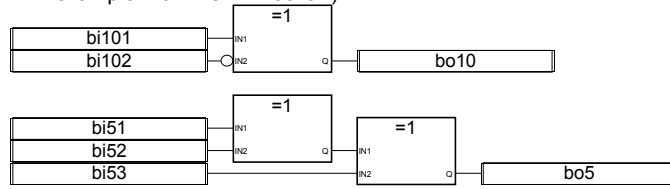
=1 XOR



Arguments:
IN1 BOOLEAN
IN2 BOOLEAN
Q BOOLEAN boolean exclusive OR of the 2 input terms

Description:
 Boolean exclusive OR between two terms.

(* FBD example with "XOR" Blocks *)



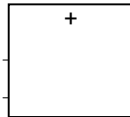
(* ST equivalence: *)

bo10 := bi101 XOR NOT (bi102);
bo5 := (bi51 XOR bi52) XOR bi53;

(* IL equivalence: *)

```
LD      bi101
XORN   bi102
ST      bo10
LD      bi51
XOR    bi52
XOR    bi53
ST      bo5
```

+



Note: For this operator, the number of its inputs can be extended to more than two.

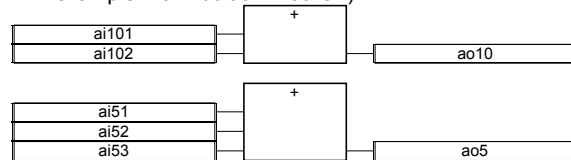
Arguments:

(inputs)	INT-REAL	can be INTEGER or REAL (all inputs must have the same format)
output	INT-REAL	signed addition of the input terms

Description:

Addition of two or more analog variables.

(* FBD example with Addition Blocks *)



(* ST equivalence: *)

ao10 := ai101 + ai102;

ao5 := (ai51 + ai52) + ai53;

(* IL equivalence: *)

```
LD      ai101
ADD     ai102
ST      ao10
LD      ai51
ADD     ai52
ADD     ai53
ST      ao5
```

-



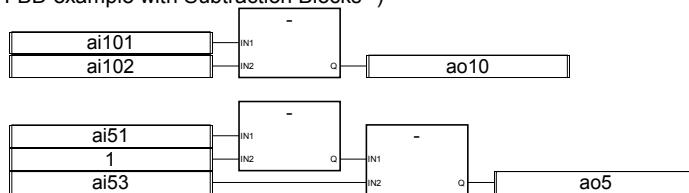
Arguments:

IN1	INT-REAL	can be INTEGER or REAL
IN2	INT-REAL	(IN1 and IN2 must have the same format)
Q	INT-REAL	subtraction (first - second)

Description:

Subtraction of two analog variables (first - second).

(* FBD example with Subtraction Blocks *)



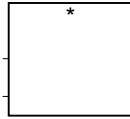
(* ST equivalence: *)

```
ao10 := ai101 - ai102;
ao5 := (ai51 - 1) - ai53;
```

(* IL equivalence: *)

```
LD      ai101
SUB     ai102
ST      ao10
LD      ai51
SUB     1
SUB     ai53
ST      ao5
```

*



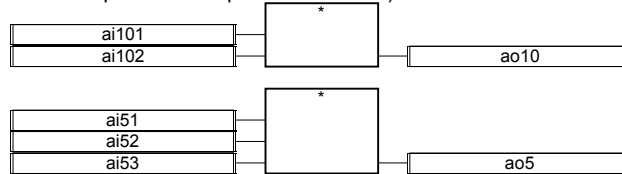
Note: For this operator, the number of its inputs can be extended to more than two.

Arguments:

(inputs)	INT-REAL	can be INTEGER or REAL (all inputs must have the same format)
output	INT-REAL	signed multiplication of the input terms

Description:
Multiplication of two or more analog variables.

(* FBD example with Multiplication blocks *)

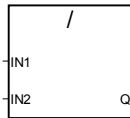


(* ST equivalence *)
 $ao10 := ai101 * ai102;$
 $ao5 := (ai51 * ai52) * ai53;$

(* IL equivalence: *)

```
LD      ai101
MUL     ai102
ST      ao10
LD      ai51
MUL     ai52
MUL     ai53
ST      ao5
```

/



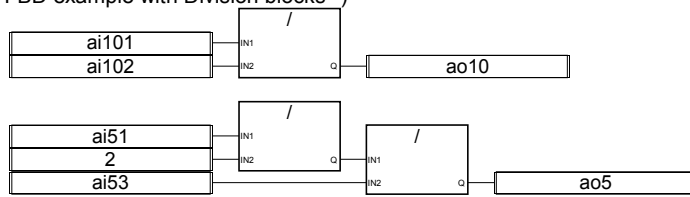
Arguments:

IN1	INT-REAL	can be INTEGER or REAL (operand)
IN2	INT-REAL	non-zero analog value (divisor) (IN1 and IN2 must have the same format)
Q	INT-REAL	signed integer or real division of IN1 by IN2

Description:

Division of two analog variables (the first divided by the second).

(* FBD example with Division blocks *)



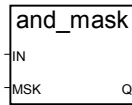
(* ST Equivalence: *)

ao10 := ai101 / ai102;
ao5 := (ai5 / 2) / ai53;

(* IL equivalence: *)

```
LD      ai101
DIV     ai102
ST      ao10
LD      ai51
DIV     2
DIV     ai53
ST      ao5
```

AND_MASK



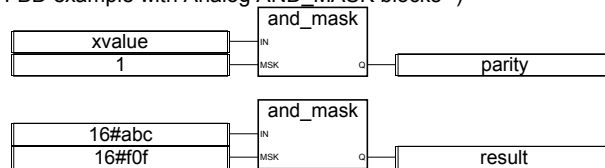
Arguments:

IN	INT	must have integer format
MSK	INT	must have integer format
Q	INT	bit to bit logical AND between IN and MSK

Description:

Integer analog AND bit to bit mask.

(* FBD example with Analog AND_MASK blocks *)

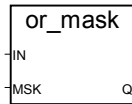


(* ST Equivalence: *)


```
parity := AND_MASK (xvalue, 1); (* 1 if xvalue is odd *)
result := AND_MASK (16#abc, 16#f0f); (* equals 16#a0c *)
```

```
(* IL equivalence: *)
LD      xvalue
AND_MASK 1
ST      parity
LD      16#abc
AND_MASK 16#f0f
ST      result
```

OR_MASK



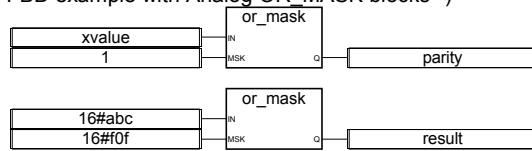
Arguments:

IN	INT	must have integer format
MSK	INT	must have integer format
Q	INT	bit to bit logical OR between IN and MSK

Description:

Integer analog OR bit to bit mask.

(* FBD example with Analog OR_MASK blocks *)

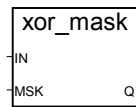


(* ST Equivalence: *)

```
is_odd := OR_MASK (xvalue, 1); (* makes value always odd *)
result := OR_MASK (16#abc, 16#f0f); (* equals 16#fbf *)
```

```
(* IL equivalence: *)
LD      xvalue
OR_MASK 1
ST      is_odd
LD      16#abc
OR_MASK 16#f0f
ST      result
```

XOR_MASK



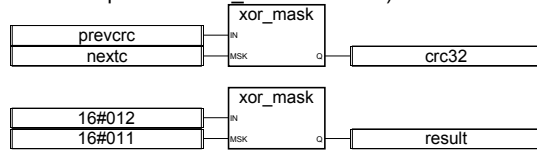
Arguments:

IN	INT	must have integer format
MSK	INT	must have integer format
Q	INT	bit to bit logical Exclusive OR between IN and MSK

Description:

Integer analog exclusive OR bit to bit mask

(* FBD example with XOR_MASK blocks *)



(* ST Equivalence: *)

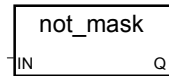
`crc32 := XOR_MASK (prevcrc, nextc);`
`result := XOR_MASK (16#012, 16#011); (* equals 16#003 *)`

(* IL equivalence: *)

```

LD      prevcrc
XOR_MASK nextc
ST      crc32
LD      16#012
XOR_MASK 16#011
ST      result
    
```

NOT_MASK



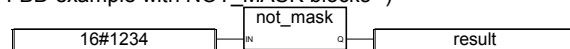
Arguments :

IN	INT	must have integer format
Q	INT	bit to bit negation on 32 bits of IN

Description:

Integer analog bit to bit negation mask

(* FBD example with NOT_MASK blocks *)

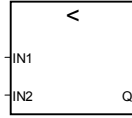


(*ST equivalence: *)

```
result := NOT_MASK (16#1234);
(* result is 16#FFFF_EDCB *)
```

```
(* IL equivalence: *)
LD      16#1234
NOT_MASK
ST      result
```

<



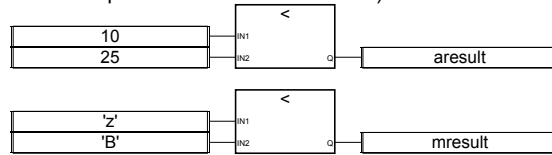
Arguments:

IN1	INT-REAL- TMR-MSG	
IN2	INT-REAL- TMR-MSG	both inputs must have the same type
Q	BOOLEAN	TRUE if IN1 < IN2

Description:

Test if one value is LESS THAN another one (on analog, timer or messages)

(* FBD example with "Less than" blocks *)



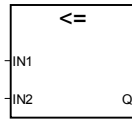
(* ST Equivalence: *)

```
areult := (10 < 25); (* areult is TRUE *)
mresult := ('z' < 'B'); (* mresult is FALSE *)
```

(* IL equivalence: *)

```
LD      10
LT      25
ST      areult
LD      'z'
LT      'B'
ST      mresult
```

<=



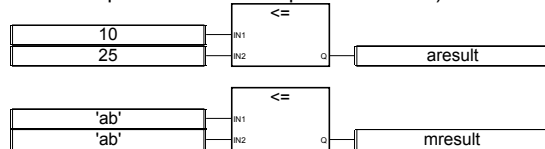
Arguments:

IN1 INT-REAL-MSG
IN2 INT-REAL-MSG both inputs must have the same type
Q BOOLEAN TRUE if IN1 <= IN2

Description:

Test if one value is LESS THAN or EQUAL TO another one (on analog, or messages)

(* FBD example with "Less or equal to" blocks *)



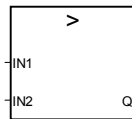
(* ST Equivalence: *)

areresult := (10 <= 25); (* areresult is TRUE *)
mresult := ('ab' <= 'ab'); (* mresult is TRUE *)

(* IL equivalence: *)

LD 10
LE 25
ST areresult
LD 'ab'
LE 'ab'
ST mresult

>



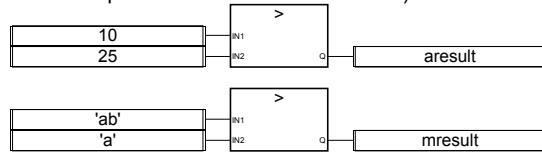
Arguments:

IN1 INT-REAL-TMR-MSG
IN2 INT-REAL-TMR-MSG both inputs must have the same type
Q BOOLEAN TRUE if IN1 > IN2

Description:

Test if one value is GREATER THAN another one (on analog, timer or messages)

(* FBD example with "Greater than" blocks *)



(* ST Equivalence: *)

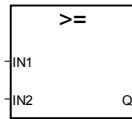
result := (10 > 25); (* areult is FALSE *)

mresult := ('ab' > 'a'); (* mresult is TRUE *)

(* IL equivalence: *)

```
LD      10
GT      25
ST      areult
LD      'ab'
GT      'a'
ST      mresult
```

>=



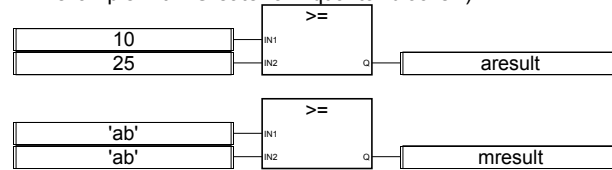
Arguments:

IN1	INT-REAL-MSG
IN2	INT-REAL-MSG both inputs must have the same type
Q	BOOLEAN TRUE if IN1 >= IN2

Description:

Test if one value is GREATER THAN or EQUAL TO another one (on analog, or messages)

(* FBD example with "Greater or Equal to" blocks *)



(* ST Equivalence: *)

result := (10 >= 25); (* areult is FALSE *)

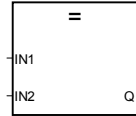
mresult := ('ab' >= 'ab'); (* mresult is TRUE *)

(* IL equivalence: *)

```
LD      10
```

GE 25
 ST aresult
 LD 'ab'
 GE 'ab'
 ST mresult

=



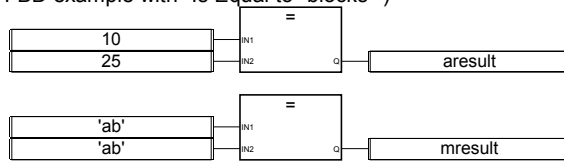
Arguments:

IN1 INT-REAL-MSG
IN2 INT-REAL-MSG both inputs must have the same type
Q BOOLEAN TRUE if IN1 = IN2

Description:

Test if one value is EQUAL TO another one (on analog, or messages)

(* FBD example with "Is Equal to" blocks *)



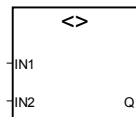
(* ST Equivalence: *)

aresult := (10 = 25); (* aresult is FALSE *)
 mresult := ('ab' = 'ab'); (* mresult is TRUE *)

(* IL equivalence: *)

LD 10
 EQ 25
 ST aresult
 LD 'ab'
 EQ 'ab'
 ST mresult

<>



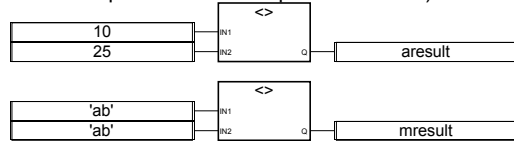
Arguments:

IN1 INT-REAL-MSG
IN2 INT-REAL-MSG both inputs must have the same type
Q BOOLEAN TRUE if first <> second

Description:

Test if one value is NOT EQUAL TO another one (on analog, or messages)

(* FBD example with "Is Not Equal to" blocks *)



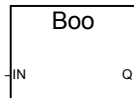
(* ST Equivalence: *)

areresult := (10 <> 25); (* areresult is TRUE *)
 mresult := ('ab' <> 'ab'); (* mresult is FALSE *)

(* IL equivalence: *)

LD 10
 NE 25
 ST areresult
 LD 'ab'
 NE 'ab'
 ST mresult

BOO



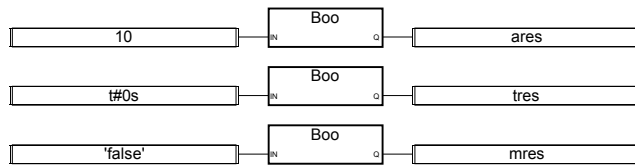
Arguments:

IN ANY any non-boolean value
Q BOO TRUE for non-zero numerical value
 FALSE for zero numerical value
 TRUE for 'TRUE' message
 FALSE for 'FALSE' message

Description:

Convert any variable to a boolean one

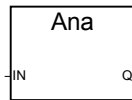
(* FBD example with "Convert to Boolean" blocks *)



(* ST Equivalence: *)
ares := BOO (10); (* ares is TRUE *)
tres := BOO (#0s); (* tres is FALSE *)
mres := BOO ('false'); (* mres is FALSE *)

(* IL equivalence: *)
LD 10
BOO
ST ares
LD #0s
BOO
ST tres
LD 'false'
BOO
ST mres

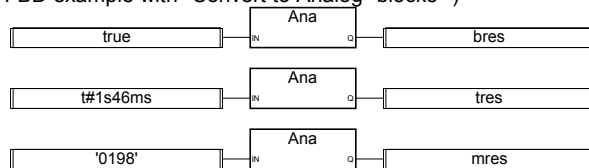
ANA



Arguments:
IN ANY any non-integer analog value
Q INT 0 if IN is FALSE / 1 if IN is TRUE
number of milliseconds for a timer
integer part for real analog
decimal number represented by a string

Description:
Convert any variable to an integer one

(* FBD example with "Convert to Analog" blocks *)

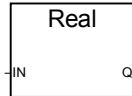


(* ST Equivalence: *)
bres := ANA (true); (* bres is 1 *)


```
tres := ANA (t#1s46ms);          (* tres is 1046 *)
mres := ANA ('0198');           (* mres is 198 *)
```

```
(* IL equivalence: *)
LD      true
ST      bres
ANAL    t#1s46ms
ANAL    tres
LD      '0198'
ANAL    mres
ST      mres
```

REAL



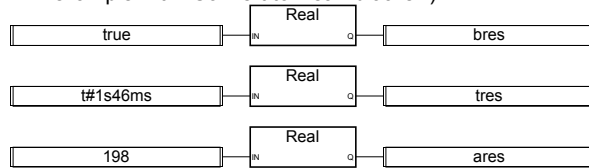
Arguments:

IN	BOO-INT-	
	TMR	any non-real analog value (no message)
Q	REAL	0.0 if IN is FALSE / 1.0 if IN is TRUE number of milliseconds for a timer equivalent number for integer analog

Description:

Convert any variable to a real one

(* FBD example with "Convert to Real" blocks *)

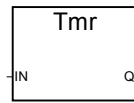


```
(* ST Equivalence: *)
bres := REAL (true);           (* bres is 1.0 *)
tres := REAL (t#1s46ms);      (* tres is 1046.0 *)
ares := REAL (198);           (* ares is 198.0 *)
```

```
(* IL equivalence: *)
LD      true
REAL
ST      bres
LD      t#1s46ms
REAL
ST      tres
```

LD 198
 REAL
 ST ares

TMR



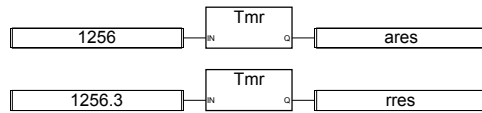
Arguments:

IN	INT-REAL	any non-timer value IN (or integer part of IN if it is real) is the number of milliseconds
Q	TIMER	time value represented by IN

Description:

Convert any analog variable to a timer one

(* FBD example with "Convert to Timer" blocks *)



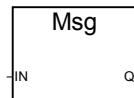
(* ST Equivalence: *)

ares := TMR (1256);	(* ares := t#1s256ms *)
rres := TMR (1256.3);	(* rres := t#1s256ms *)

(* IL equivalence: *)

```
LD 1256
TMR
ST ares
LD 1256.3
TMR
ST rres
```

MSG



Arguments:

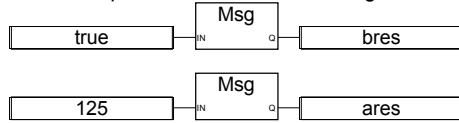
IN	BOO- INT-REA	any non-message value
Q	MSG	'false' or 'true' if IN is a boolean

decimal representation if IN is an analog

Description:

Convert any variable to a message one

(* FBD example with "Convert to Message" blocks *)



(* ST Equivalence: *)

bres := MSG (true); (* bres is 'TRUE' *)

ares := MSG (125); (* ares is '125' *)

(* IL equivalence: *)

LD true

MSG

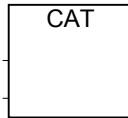
ST bres

LD 125

MSG

ST ares

CAT



Note: For this operator, the number of its inputs can be extended to more than two.

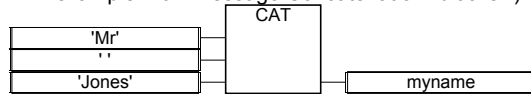
Arguments:

(inputs)	MSG	(addition of all message length must not exceed output message capacity)
output	MSG	concatenation of the input messages

Description:

Concat several messages into one

(* FBD example with "Message Concatenation" blocks *)



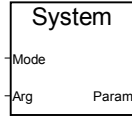
(* ST Equivalence: use the + operator *)

myname := ('Mr' + ' ') + 'Jones';

(* means: myname := 'Mr Jones' *)

```
(* IL equivalence: *)
LD      'Mr'
ADD     ''
ADD     'Jones'
ST      myname
```

SYSTEM



Arguments:

Mode	INT	identifies the system parameter and the access mode
Arg	INT-TMR	new value for a "write" access
Param	INT	value of the accessed parameter

Description:

Access to the system parameters

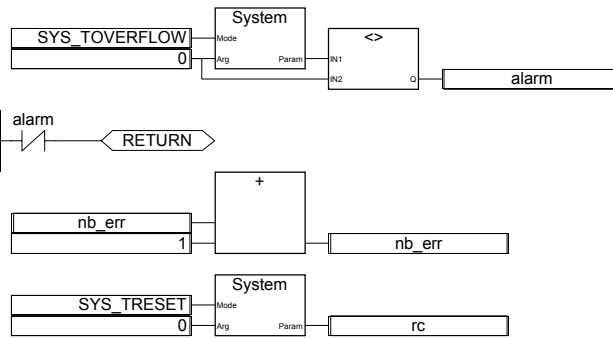
The following is the list of available commands (pre-defined keywords) for the SYSTEM function:

command	meaning
SYS_TALLOWED	read allowed cycle timing
SYS_TCURRENT	read current cycle timing
SYS_TMAXIMUM	read maximum cycle timing
SYS_TOVERFLOW	read cycle timing overflows
SYS_TRESET	reset timing counters
SYS_TWRITE	change cycle timing
SYS_ERR_TEST	check for run time errors
SYS_ERR_READ	read oldest run time error

These are expected arguments for pre-defined functions of the SYSTEM function:

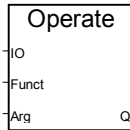
command	argument	return value
SYS_TALLOWED	0	allowed cycle timing
SYS_TCURRENT	0	current cycle timing
SYS_TMAXIMUM	0	maximum detected timing
SYS_TOVERFLOW	0	number of timing overflows
SYS_TRESET	0	0
SYS_TWRITE	new allowed cycle timing	written time
SYS_ERR_TEST	0	0 if no error detected
SYS_ERR_READ	0	oldest error code

(* FBD example with "System" blocks *)



```
(* ST Equivalence: *)
alarm := (SYSTEM (SYS_TOVERFLOW, 0) <> 0);
If (alarm) Then
    nb_err := nb_err + 1;
    rc := SYSTEM (SYS_TRESET, 0);
End_If;
```

OPERATE



Arguments:

IO	ANY	input or output variable
Funct	INT	action to be operated
Arg	INT	argument for I/O action
Q	INT	return check

Description:
Access to an I/O channel

The meaning of OPERATE arguments depends on the I/O interface implementation. Refer to your hardware manual or corresponding I/O board technical note to learn more about OPERATE capabilities.

B.9.2 Standard function blocks

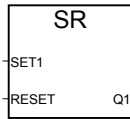
These are standard function blocks supported by the ISaGRAF system. Such function blocks are pre-defined and do not have to be declared in the library.

Booleans.....	SR	Set dominant bistable
	RS	Reset dominant bistable
	R_Trig	Rising edge detection

	F_Trig	Falling edge detection
	SEMA	Semaphore
Counting	CTU	Up counter
	CTD	Down counter
	CTUD	Up-down counter
Timers	TON	On-delay timing
	TOF	Off-delay timing
	TP	Pulse timing
Integer analogs	CMP	Full comparison function block
	StackInt	Stack of integer analogs
Real analogs	AVERAGE	Running average over N samples
	HYSTER	Boolean hysteresis on difference of reals
	LIM_ALARM	High/low limit alarm with hysteresis
	INTEGRAL	Integration over time
	DERIVATE	Differentiation according to time
Signal generation	BLINK	Blinking boolean signal
	SIG_GEN	Signal generator

Note: When new "C" function blocks are created, they can be called from the FBD language.

SR



Arguments:

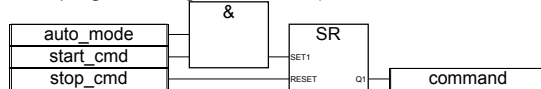
SET1	BOO	if TRUE, sets Q1 to TRUE (dominant)
RESET	BOO	if TRUE, resets Q1 to FALSE
Q1	BOO	boolean memory state

Description:

Set dominant bistable: See True table below:

Set1	Reset	Q1	result Q1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(* FBD program using "SR" block *)



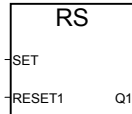
(* ST Equivalence: We suppose SR1 is an instance of SR block *)

```
SR1((auto_mode & start_cmd), stop_cmd);
command := SR1.Q1;
```

(* IL Equivalence: *)

```
LD      auto_mode
AND     start_cmd
ST      SR1.set1
LD      stop_cmd
ST      SR1.reset
CAL     SR1
LD      SR1.Q1
ST      command
```

RS



Arguments:

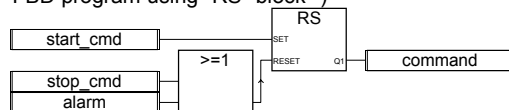
SET	BOO	if TRUE, sets Q1 to TRUE
RESET1	BOO	if TRUE, resets Q1 to FALSE (dominant)
Q1	BOO	boolean memory state

Description:

Reset dominant bistable: See True table below:

Set	Reset1	Q1	result Q1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

(* FBD program using "RS" block *)



(* ST Equivalence: We suppose RS1 is an instance of RS block *)

```
RS1(start_cmd, (stop_cmd OR alarm));
command := RS1.Q1;
```

(* IL Equivalence: *)

```
LD      start_cmd
ST      RS1.set
```

LD stop_cmd
 OR alarm
 ST RS1.reset1
 CAL RS1
 LD RS1.Q1
 ST command

R_TRIG



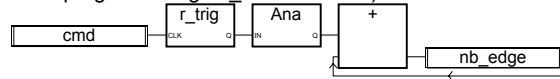
Arguments:

CLK	BOO	any boolean variable
Q	BOO	TRUE when CLK rises from FALSE to TRUE FALSE if all other cases

Description:

Detects a rising edge of a boolean variable

(* FBD program using "R_TRIG" block *)



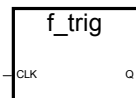
(* ST Equivalence: We suppose R_TRIG1 is an instance of R_TRIG block *)

```
R_TRIG1(cmd);
nb_edge := ANA(R_TRIG1.Q) + nb_edge;
```

(* IL Equivalence: *)

```
LD cmd
ST R_TRIG1.clk
CAL R_TRIG1
LD R_TRIG1.Q
ANA
ADD nb_edge
ST nb_edge
```

F_TRIG



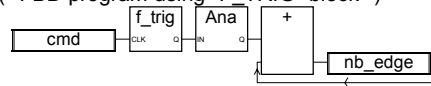
Arguments:

CLK	BOO	any boolean variable
Q	BOO	TRUE when CLK changes from TRUE to FALSE FALSE if all other cases

Description:

Detects a falling edge of a boolean variable

(* FBD program using "F_TRIG" block *)



(* ST Equivalence: We suppose F_TRIG1 is an instance of F_TRIG block *)

F_TRIG1(cmd);

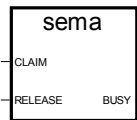
nb_edge := ANA(F_TRIG1.Q) + nb_edge;

(* IL Equivalence: *)

```

LD      cmd
ST      F_TRIG1.clk
CAL     F_TRIG1
LD      F_TRIG1.Q
ANA
ADD     nb_edge
ST      nb_edge
  
```

SEMA



Arguments:

CLAIM	BOOLEAN	"test and set" command
RELEASE	BOOLEAN	releases the semaphore
BUSY	BOOLEAN	state of the semaphore

Description:

(* "x" is a boolean variable initialized to FALSE *)

busy := x;

If claim Then

 x := True;

Else

 If release Then

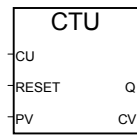
 busy := False;

 x := False;

 End_if;

End_if;

CTU



Arguments:

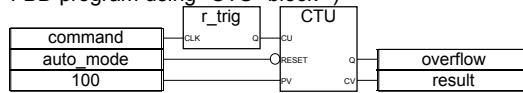
CU	BOO	counting input (counting when CU is TRUE)
RESET	BOO	reset command (dominant)
PV	INT	programmed maximum value
Q	BOO	overflow: TRUE when CV = PV
CV	INT	counter result

Warning: The CTU block does not detect the rising or falling edges of the counting input (CU). It must be associated with an "R_TRIG" or "F_TRIG" block to create a pulse counter.

Description:

Count (integer) from 0 up to a given value 1 by 1

(* FBD program using "CTU" block *)



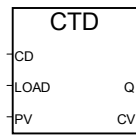
(* ST Equivalence: We suppose R_TRIG1 is an instance of R_TRIG block and CTU1 is an instance of CTU block*)

```
CTU1(R_TRIG1(command),NOT(auto_mode),100);
overflow := CTU1.Q;
result := CTU1.CV;
```

(* IL Equivalence: *)

```
LD      command
ST      R_TRIG1.clk
CAL     R_TRIG1
LD      R_TRIG1.Q
ST      CTU1.cu
LDN     auto_mode
ST      CTU1.reset
LD      100
ST      CTU1.pv
CAL     CTU1
LD      CTU1.Q
ST      overflow
LD      CTU1.cv
ST      result
```

CTD



Arguments:

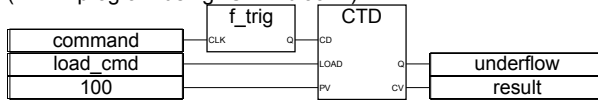
CD	BOO	counting input (down-counting when CD is TRUE)
LOAD	BOO	load command (dominant) (CV = PV when LOAD is TRUE)
PV	INT	programmed initial value
Q	BOO	underflow: TRUE when CV = 0
CV	INT	counter result

Warning: The CTD block does not detect the rising or falling edges of the counting input (CD). It must be associated with an "R_TRIG" or "F_TRIG" block to create a pulse counter.

Description:

Count (integer) from a given value down to 0 1 by 1

(* FBD program using "CTD" block *)



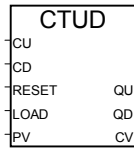
(* ST Equivalence: We suppose F_TRIG1 is an instance of F_TRIG block and CTD1 is an instance of CTD block*)

```
CTD1(F_TRIG1(command),load_cmd,100);
underflow := CTD1.Q;
result := CTD1.CV;
```

(* IL Equivalence: *)

```
LD      command
ST      F_TRIG1.clk
CAL     F_TRIG1
LD      F_TRIG1.Q
ST      CTD1.cd
LD      load_cmd
ST      CTD1.load
LD      100
ST      CTD1.pv
CAL     CTD1
LD      CTD1.Q
ST      underflow
LD      CTD1.cv
ST      result
```

CTUD



Arguments:

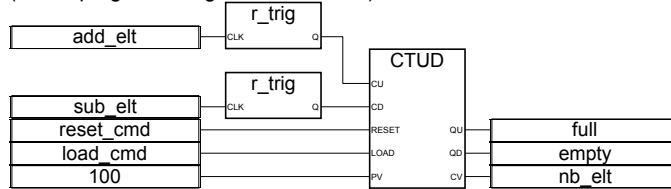
CU	BOO	up-counting (when CU is TRUE)
CD	BOO	down-counting (when CD is TRUE)
RESET	BOO	reset command (dominant) (CV = 0 when RESET is TRUE)
LOAD	BOO	load command (CV = PV when LOAD is TRUE)
PV	INT	programmed maximum value
QU	BOO	overflow: TRUE when CV = PV
QD	BOO	underflow: TRUE when CV = 0
CV	INT	counter result

Warning: The CTUD block does not detect the rising or falling edges of the counting inputs (CU and CD). It must be associated with an "R_TRIG" or "F_TRIG" block to create a pulse counter.

Description:

Count (integer) from 0 up to a given value 1 by 1
or from a given value down to 0 1 by 1

(* FBD program using "CTUD" block *)



(* ST Equivalence: We suppose R_TRIG1 and R_TRIG2 are two instances of R_TRIG block and CTUD1 is an instance of CTUD block*)

```
CTUD1(R_TRIG1(add_elt), R_TRIG2(sub_elt), reset_cmd, load_cmd,100);
full := CTUD1.QU;
empty := CTUD1.QD;
nb_elt := CTUD1.CV;
```

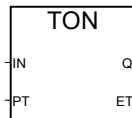
(* IL Equivalence: *)

```
LD      add_elt
ST      R_TRIG1.clk
CAL     R_TRIG1
LD      R_TRIG1.Q
ST      CTUD1.cu
LD      sub_elt
ST      R_TRIG2.clk
CAL     R_TRIG2
```

```

LD      R_TRIG2.Q
ST      CTUD1.cd
LD      reset_cmd
ST      CTUD1.reset
LD      load_cmd
ST      CTUD1.load
LD      100
ST      CTUD1.pv
CAL     CTUD1
LD      CTUD1.QU
ST      full
LD      CTUD1.QD
ST      empty
LD      CTUD1.CV
ST      nb_elt

```

TON

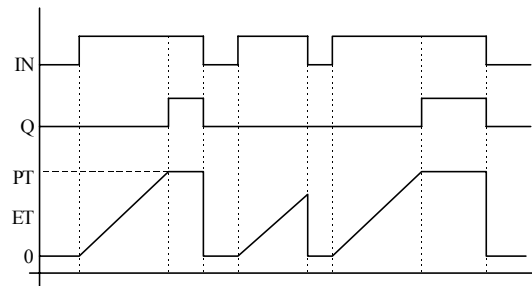
Arguments: {XE "TON"} {XE "On-delay timing"}

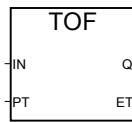
IN	BOO	If Rising edge, starts increasing internal timer If Falling Edge, stops and resets internal timer
PT	TMR	maximum programmed time
Q	BOO	If TRUE, programmed time is elapsed
ET	TMR	current elapsed time

Description:

Increase an internal timer up to a given value.

Timing diagram:

**TOF**



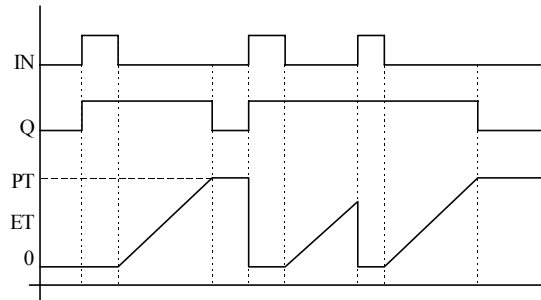
Arguments:

IN	BOO	If Falling edge, starts increasing internal timer If Rising edge, stops and resets internal timer
PT	TMR	maximum programmed time
Q	BOO	If TRUE: total time is not elapsed
ET	TMR	current elapsed time

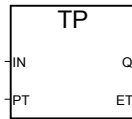
Description:

Increase an internal timer up to a given value.

Timing diagram:



TP



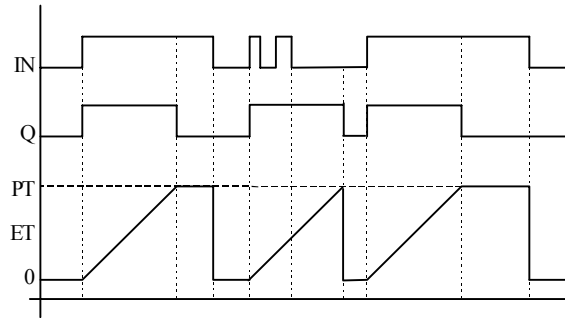
Arguments:

IN	BOO	If Rising edge, starts increasing internal timer (if not already increasing) If FALSE and only if timer is elapsed, resets the internal timer Any change on IN during counting has no effect.
PT	TMR	maximum programmed time
Q	BOO	If TRUE: timer is counting
ET	TMR	current elapsed time

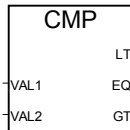
Description:

Increase an internal timer up to a given value.

Timing diagram:



CMP



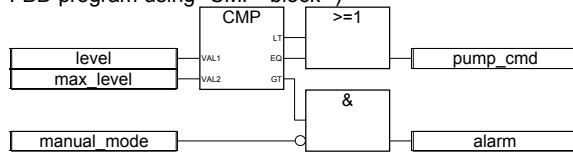
Arguments:

VAL1	INT	any signed integer analog value
VAL2	INT	any signed integer analog value
LT	BOO	TRUE if val1 is Less Than val2
EQ	BOO	TRUE if val1 is Equal to val2
GT	BOO	TRUE if val1 is Greater Than val2

Description:

Compare two values: tell if they are equal, or if the first is less or greater than the second one.

(* FBD program using "CMP" block *)



(* ST Equivalence: We suppose CMP1 is an instance of CMP block *)

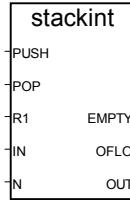
```
CMP1(level, max_level);
pump_cmd := CMP1.LT OR CMP1.EQ;
alarm := CMP1.GT AND NOT(manual_mode);
```

(* IL Equivalence: *)

```
LD      level
ST      CMP1.val1
LD      max_level
```

ST	CMP1.val2
CAL	CMP1
LD	CMP1.LT
OR	CMP1.EQ
ST	pump_cmd
LD	CMP1.GT
ANDN	manual_mode
ST	alarm

STACKINT



Arguments:

PUSH	BOO	push command (on rising edge only) add the IN value on the top of the stack
POP	BOO	pop command (on rising edge only) delete in the stack the last value pushed (top of the stack)
R1	BOO	resets the stack to its empty state
IN	INT	pushed value
N	INT	application defined stack size
EMPTY	BOO	TRUE if the stack is empty
OFLO	BOO	overflow: TRUE if the stack is full
OUT	INT	value at the top of the stack

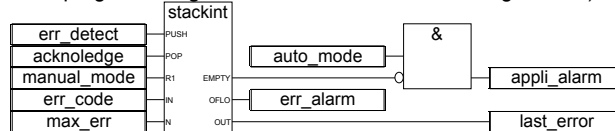
Description:

Manage a stack of integer values.

The "STACKINT" function block includes a rising edge detection for both PUSH and POP commands. The maximum size of the stack is **128**. The application defined stack size **N** cannot be less than 1 or greater than 128.

Note that OFLO value is valid only after a reset (R1 has been set to TRUE at least once and back to FALSE).

(* FBD program using "STACKINT" block: error management *)



(* ST Equivalence: We suppose STACKINT1 is an instance of STACKINT block *)


```

STACKINT1(err_detect, acknowledge, manual_mode, err_code, max_err);
appli_alarm := auto_mode AND NOT(STACKINT1.EMPTY);
err_alarm := STACKINT1.OFLO;
last_error := STACKINT1.OUT;

```

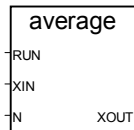
(* IL Equivalence: *)

```

LD      err_detect
ST      STACKINT1.push
LD      acknowledge
ST      STACKINT1.pop
LD      manual_mode
ST      STACKINT1.r1
LD      err_code
ST      STACKINT1.IN
LD      max_err
ST      STACKINT1.N
CAL     STACKINT1
LD      auto_mode
ANDN   STACKINT1.empty
ST     appli_alarm
LD     STACKINT1.OFLO
ST     err_alarm
LD     STACKINT1.OUT
ST     last_error

```

AVERAGE



Arguments:

RUN	BOO	TRUE=run / FALSE=reset
XIN	REAL	any real analog variable
N	INT	application defined number of samples
XOUT	REAL	running average of XIN value

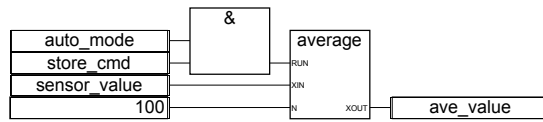
Description:

Stores a value at each cycle and calculates the average value of all already stored values. Only the N last values are stored.

The number of samples **N** cannot exceed **128**.

If the "**RUN**" command is **FALSE** (reset mode), the output value is equal to the input value. When the maximum N of stored values is reached, the first stored value is erased by the last one.

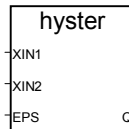
(* FBD program using "AVERAGE" block: *)



(* ST Equivalence: AVERAGE1 instance of AVERAGE block *)
 AVERAGE1((auto_mode & store_cmd), sensor_value, 100);
 ave_value := AVERAGE1.XOUT;

(* IL Equivalence: *)
 LD auto_mode
 AND store_cmd
 ST AVERAGE1.run
 LD sensor_value
 ST AVERAGE1.xin
 LD 100
 ST AVERAGE1.N
 CAL AVERAGE1
 LD AVERAGE1.XOUT
 ST ave_value

HYSTER

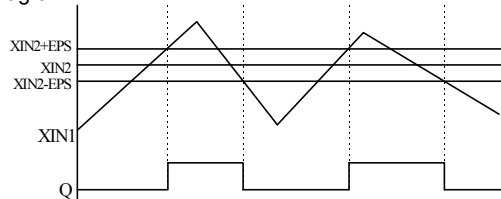


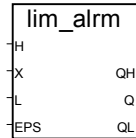
Arguments:

XIN1	REAL	any real analog value
XIN2	REAL	to test if XIN1 has overpassed XIN2+EPS
EPS	REAL	hysteresis value (must be greater than zero)
Q	BOO	TRUE if XIN1 has overpassed XIN2+EPS and is not yet below XIN2-EPS

Description:
 Hysteresis on a real value for a high limit.

Example of timing diagram:



LIM_ALARM

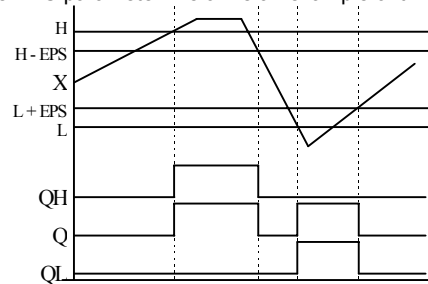
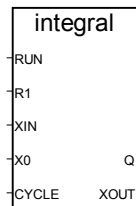
Arguments:

H	REAL	high limit value
X	REAL	input: any real analog value
L	REAL	low limit value
EPS	REAL	hysteresis value (must be greater than zero)
QH	BOO	"high" alarm: TRUE if X above high limit H
Q	BOO	alarm output: TRUE if X out of limits
QL	BOO	"low" alarm: TRUE if X below low limit L

Description:

Hysteresis on a real value for high and low limits.

An hysteresis is applied on high and low limits. The hysteresis delta used for either high or low limit is one half of the EPS parameter. Below is an example of timing diagram:

**INTEGRAL**

Arguments:

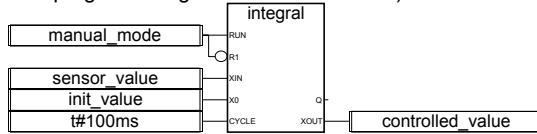
RUN	BOO	mode: TRUE=integrate / FALSE=hold
R1	BOO	overriding reset
XIN	REAL	input: any real analog value

X0	REAL	initial value
CYCLE	TMR	sampling period
Q	BOO	Not R1
XOUT	REAL	integrated output

Description:
Integration of a real value.

If the "**CYCLE**" parameter value is less than the cycle timing of the ISaGRAF application, the sampling period is the cycle timing of the application.

(* FBD program using "INTEGRAL" block: *)

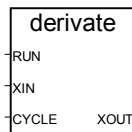


(* ST Equivalence: INTEGRAL1 instance of INTEGRAL block *)
INTEGRAL1(manual_mode, NOT(manual_mode), sensor_value, init_value, t#100ms);
controlled_value := INTEGRAL1.XOUT;

(* IL Equivalence: *)

```
LD      manual_mode
ST      INTEGRAL1.run
STN     INTEGRAL1.R1
LD      sensor_value
ST      INTEGRAL1.XIN
LD      init_value
ST      INTEGRAL1.X0
LD      t#100ms
ST      INTEGRAL1.CYCLE
CAL     INTEGRAL1
LD      INTEGRAL1.XOUT
ST      controlled_value
```

DERIVATE



Arguments:

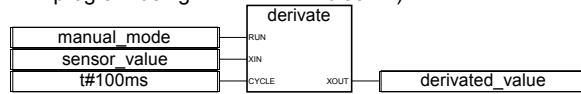
RUN	BOO	mode: TRUE=normal / FALSE=reset
XIN	REAL	input: any real analog value
CYCLE	TMR	sampling period
XOUT	REAL	differentiated output

Description:

Differentiation of a real value.

If the "**CYCLE**" parameter value is less than the cycle timing of the ISaGRAF application, the sampling period is the cycle timing of the application.

(* FBD program using "DERIVATE" block: *)



(* ST Equivalence: DERIVATE1 instance of DERIVATE block *)

```

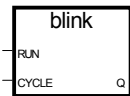
DERIVATE1(manual_mode, sensor_value, t#100ms);
derivated_value := DERIVATE1.XOUT;
  
```

(* IL Equivalence: *)

```

LD      manual_mode
ST      DERIVATE1.run
LD      sensor_value
ST      DERIVATE1.XIN
LD      t#100ms
ST      DERIVATE1.CYCLE
CAL     DERIVATE1
LD      DERIVATE1.XOUT
ST      derivated_value
  
```

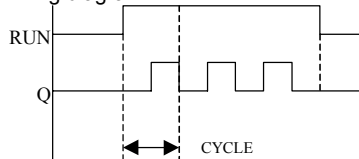
BLINK

**Arguments:**

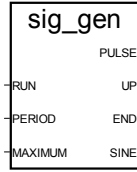
RUN	BOO	mode: TRUE=blinking / FALSE=reset the output to false
CYCLE	TMR	blinking period
Q	BOO	output blinking signal

Description:

Generates a blinking signal.

Timing diagram:

SIG_GEN



Arguments:

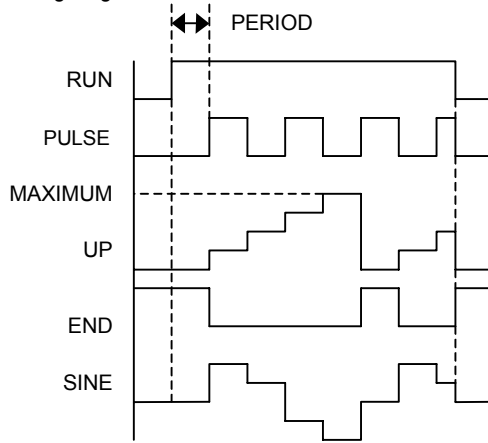
RUN	BOO	mode: TRUE=running / FALSE=reset to false
PERIOD	TMR	duration of one sample
MAXIMUM	INT	maximum counting value
PULSE	BOO	inverted after each sample
UP	INT	up-counter, increased on each sample
END	BOO	TRUE when up-counting ends
SINE	REAL	sine signal (period = counting duration)

Description:

Generates various signal: blink on a boolean, a integer counter-up, and real sine wave.

When counting reaches maximum value, it restarts from 0 (zero). So END keeps the TRUE value only during 1 PERIOD.

Timing diagram:

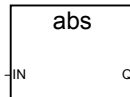


B.9.3 Standard functions

These are standard functions supported by the ISaGRAF system. Such functions are pre-defined and do not have to be declared in the library.

MathABS Absolute value

	EXPT, POW	Exponent, Power calculation
	LOG	Logarithm
	SQRT	Square root
	TRUNC	Truncate decimal part
Trigonometric.....	ACOS, ASIN,	Arc cosine, Arc sine,
	ATAN	Arc tangent
	COS, SIN,	Cosine, Sine,
	TAN	Tangent
Register control.....	ROL, ROR	Rotate Left, Rotate Right
	SHL, SHR	Shift Left, Shift Right
Data manipulation.....	MIN, MAX,	Minimum, Maximum,
	LIMIT	Limit
	MOD	Modulo
	MUX4, MUX8	Multiplexer (4 or 8 entries),
	SEL	Binary selector
	ODD	Odd parity
	RAND	Random value
Data conversion.....	ASCII	Character → ASCII code
	CHAR	ASCII code → Character
String management	MLEN	Get string length
	DELETE	Delete sub-string,
	INSERT	Insert string
	FIND,	Find sub-string,
	REPLACE	Replace sub-string
	LEFT, MID	Extract left, middle
	RIGHT	or right of a string
	DAY_TIME	Time of day
Array manipulation	ARCREATE	Create array of integer values
	ARREAD	Read /
	ARWRITE	Write array element
Binary file management	F_ROPEN	Open a binary file in Read mode
	F_WOPEN	Open a binary file in Write mode
	F_CLOSE	Close a binary file
	F_EOF	Test the end of a binary file
	FA_READ	Read an analog value in a binary file
	FA_WRITE	Write an analog value to a binary file
	FM_READ	Read a message string in a binary file
	FM_WRITE	Write a message string to a binary file

ABS

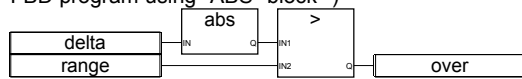
Arguments:

IN	REAL	any signed real analog value
Q	REAL	absolute value (always positive)

Description:

Gives the absolute (positive) value of a real value.

(* FBD program using "ABS" block *)



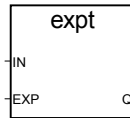
(* ST Equivalence: *)

over := (ABS (delta) > range);

(* IL Equivalence: *)

LD delta
ABS
GT range
ST over

EXPT



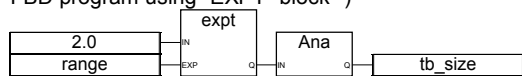
Arguments:

IN	REAL	any signed real analog value
EXP	INT	integer analog exponent
Q	REAL	(IN ^{EXP})

Description:

Gives the real result of the operation: (base^{exponent}) 'base' being the first argument and 'exponent' the second one.

(* FBD program using "EXPT" block *)



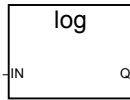
(* ST Equivalence: *)

tb_size := ANA (EXPT (2.0, range));

(* IL Equivalence: *)

LD 2.0
EXPT range
ANA
ST tb_size

LOG



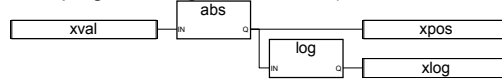
Arguments:

IN	REAL	must be greater than zero
Q	REAL	logarithm (base 10) of the input value

Description:

Calculates the logarithm (base 10) of a real value.

(* FBD program using "LOG" block *)



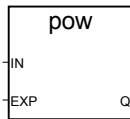
(* ST Equivalence: *)

```
xpos := ABS (xval);
xlog := LOG (xpos);
```

(* IL Equivalence: *)

```
LD      xval
ABS
ST      xpos
LOG
ST      xlog
```

POW



Arguments:

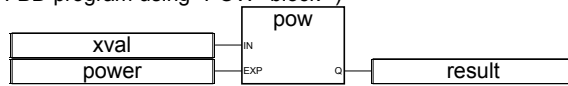
IN	REAL	real analog number to be raised
EXP	REAL	power (exponent)
Q	REAL	(IN^{EXP})

1.0 if IN is not 0.0 and EXP is 0.0
0.0 if IN is 0.0 and EXP is negative
0.0 if both IN and EXP are 0.0
0.0 if IN is negative and Y does not correspond to an integer

Description:

Gives the real result of the operation: (base^{exponent}) 'base' being the first argument and 'exponent' the second one. The exponent is a real value.

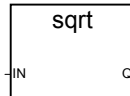
(* FBD program using "POW" block *)



(* ST Equivalence: *)
 result := POW (xval, power);

(* IL Equivalence: *)
 LD xval
 POW power
 ST result

SQRT



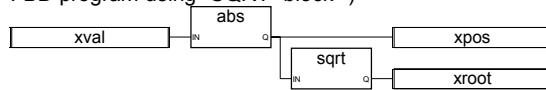
Arguments:

IN	REAL	must be greater than or equal to zero
Q	REAL	square root of the input value

Description:

Calculates the square root of a real value.

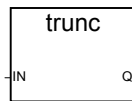
(* FBD program using "SQRT" block *)



(* ST Equivalence: *)
 xpos := ABS (xval);
 xroot := SQRT (xpos);

(* IL Equivalence: *)
 LD xval
 ABS
 ST xpos
 SQRT
 ST xroot

TRUNC



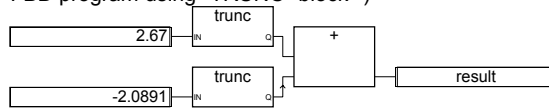
Arguments:

IN	REAL	any REAL analog value
Q	REAL	if IN>0, biggest integer less or equal to the input if IN<0, least integer greater or equal to the input

Description:

Truncates a real value to have just the integer part

(* FBD program using "TRUNC" block *)



(* ST Equivalence: *)

result := TRUNC (+2.67) + TRUNC (-2.0891);

(* means: result := 2.0 + (-2.0) := 0.0; *)

(* IL Equivalence: *)

LD 2.67

TRUNC

ST temporary (* temporary result of first TRUNC *)

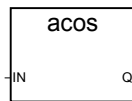
LD -2.0891

TRUNC

ADD temporary

ST result

ACOS



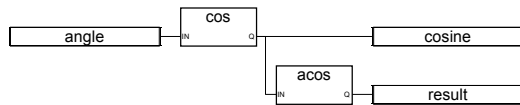
Arguments:

IN	REAL	must be in set [-1.0 .. +1.0]
Q	REAL	arc-cosine of the input value (in set [0.0 .. PI]) = 0.0 for invalid input

Description:

Calculates the Arc cosine of a real value.

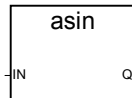
(* FBD program using "COS" and "ACOS" blocks *)



(* ST Equivalence: *)
 cosine := COS (angle);
 result := ACOS (cosine); (* result is equal to angle *)

(* IL Equivalence: *)
 LD angle
 COS
 ST cosine
 ACOS
 ST result

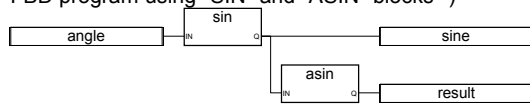
ASIN



Arguments: {XE "ASIN"} {XE "Arc sine"}
IN REAL must be in set [-1.0 .. +1.0]
Q REAL arc-sine of the input value (in set [-PI/2 .. +PI/2])
 = 0.0 for invalid input

Description:
 Calculates the Arc sine of a real value.

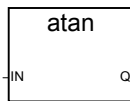
(* FBD program using "SIN" and "ASIN" blocks *)



(* ST Equivalence: *)
 sine := SIN (angle);
 result := ASIN (sine); (* result is equal to angle *)

(* IL Equivalence: *)
 LD angle
 SIN
 ST sine
 ASIN
 ST result

ATAN



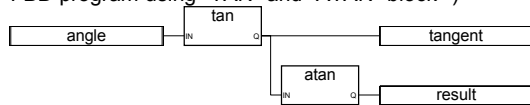
Arguments:

IN	REAL	any real analog value
Q	REAL	arc-tangent of the input value (in set $[-\pi/2 .. +\pi/2]$) = 0.0 for invalid input

Description:

Calculates the Arc tangent of a real value.

(* FBD program using "TAN" and "ATAN" block *)



(* ST Equivalence: *)

tangent := TAN (angle);

result := ATAN (tangent); (* result is equal to angle*)

(* IL Equivalence: *)

LD angle

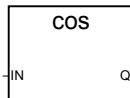
TAN

ST tangent

ATAN

ST result

COS



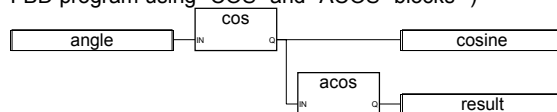
Arguments:

IN	REAL	any REAL analog value
Q	REAL	cosine of the input value (in set $[-1.0 .. +1.0]$)

Description:

Calculates the Cosine of a real value.

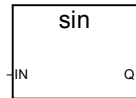
(* FBD program using "COS" and "ACOS" blocks *)



(* ST Equivalence: *)
 cosine := COS (angle);
 result := ACOS (cosine); (* result is equal to angle *)

(* IL Equivalence: *)
 LD angle
 COS
 ST cosine
 ACOS
 ST result

SIN



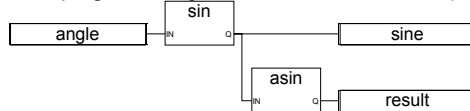
Arguments:

IN	REAL	any REAL analog value
Q	REAL	sine of the input value (in set [-1.0 .. +1.0])

Description:

Calculates the Sine of a real value.

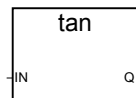
(* FBD program using "SIN" and "ASIN" blocks *)



(* ST Equivalence: *)
 sine := SIN (angle);
 result := ASIN (sine); (* result is equal to angle *)

(* IL Equivalence: *)
 LD angle
 SIN
 ST sine
 ASIN
 ST result

TAN



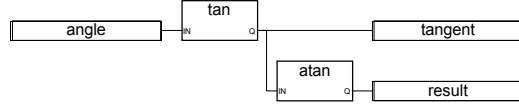
Arguments:

IN	REAL	cannot be equal to PI/2 modulo PI
Q	REAL	tangent of the input value = 1E+38 for invalid input

Description:

Calculates the Tangent of a real value.

(* FBD program using "TAN" and "ATAN" block *)



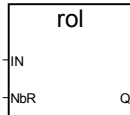
(* ST Equivalence: *)

```
tangent := TAN (angle);
result := ATAN (tangent); (* result is equal to angle*)
```

(* IL Equivalence: *)

```
LD      angle
TAN
ST      tangent
ATAN
ST      result
```

ROL

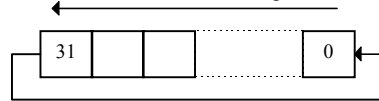


Arguments:

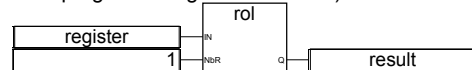
IN	INT	any integer analog value
NbR	INT	number of 1 bit rotations (in set [1..31])
Q	INT	left rotated value no effect if NbR <= 0

Description:

Make the bits of an integer rotate to the left. Rotation is made on 32 bits:



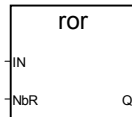
(* FBD program using "ROL" block *)



```
(* ST Equivalence: *)
result := ROL (register, 1);
(* register = 2#0100_1101_0011_0101 *)
(* result = 2#1001_1010_0110_1010 *)
```

```
(* IL Equivalence: *)
LD      register
ROL     1
ST      result
```

ROR

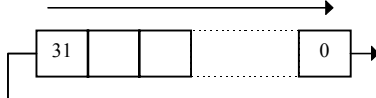


Arguments:

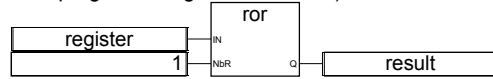
IN	INT	any integer analog value
Nbr	INT	number of 1 bit rotations (in set [1..31])
Q	INT	right rotated value
		no effect if Nbr <= 0

Description:

Make the bits of an integer rotate to the right. Rotation is made on 32 bits:



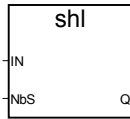
(* FBD program using "ROR" block *)



```
(* ST Equivalence: *)
result := ROR (register, 1);
(* register = 2#0100_1101_0011_0101 *)
(* result = 2#1010_0110_1001_1010 *)
```

```
(* IL Equivalence: *)
LD      register
ROR     1
ST      result
```

SHL

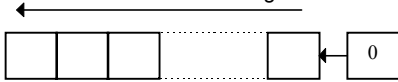


Arguments:

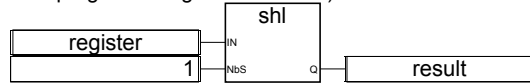
IN	INT	any integer analog value
NbS	INT	number of 1 bit shifts (in set [1..31])
Q	INT	left shifted value no effect if NbS <= 0 0 is used to replace lowest bit

Description:

Make the bits of an integer shift to the left. Shift is made on 32 bits:



(* FBD program using "SHL" block *)



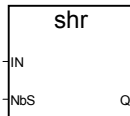
(* ST Equivalence: *)

```
result := SHL (register,1);
(* register = 2#0100_1101_0011_0101 *)
(* result   = 2#1001_1010_0110_1010 *)
```

(* IL Equivalence: *)

```
LD     register
SHL   1
ST     result
```

SHR

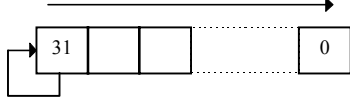


Arguments:

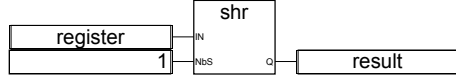
IN	INT	any integer analog value
NbS	INT	number of 1 bit shifts (in set [1..31])
Q	INT	right shifted value no effect if NbS <= 0 highest bit is copied at each shift

Description:

Make the bits of an integer shift to the right. Shift is made on 32 bits:



(* FBD program using "SHR" block *)



(* ST Equivalence: *)

result := SHR (register,1);

(* register = 2#1100_1101_0011_0101 *)

(* result = 2#1110_0110_1001_1010 *)

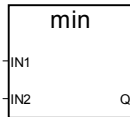
(* IL Equivalence: *)

LD register

SHR 1

ST result

MIN



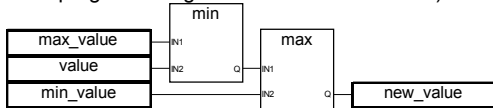
Arguments:

IN1	INT	any signed integer analog value
IN2	INT	(cannot be REAL)
Q	INT	minimum of both input values

Description:

Gives the minimum of two integer values.

(* FBD program using "MIN" and "MAX" block *)



(* ST Equivalence: *)

new_value := MAX (MIN (max_value, value), min_value);

(* bounds the value to the [min_value..max_value] set *)

(* IL Equivalence: *)

LD max_value

MIN value

MAX min_value

ST new_value

MAX

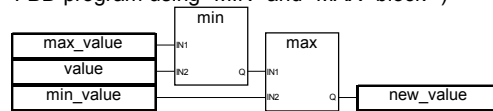
Arguments:

IN1	INT	any signed integer analog value
IN2	INT	(cannot be REAL)
Q	INT	maximum of both input values

Description:

Gives the maximum of two integer values.

(* FBD program using "MIN" and "MAX" block *)



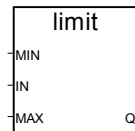
(* ST Equivalence: *)

new_value := MAX (MIN (max_value, value), min_value);

(* bounds the value to the [min_value..max_value] set *)

(* IL Equivalence: *)

LD	max_value
MIN	value
MAX	min_value
ST	new_value

LIMIT

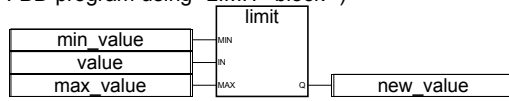
Arguments:

MIN	INT	minimum allowed value
IN	INT	any signed integer analog value
MAX	INT	maximum allowed value
Q	INT	input value bounded to allowed range

Description:

Limits an integer value into a given interval. Whether it keeps its value if it is between minimum and maximum, or it is changed to maximum if it is above, or it is changed to minimum if it is below.

(* FBD program using "LIMIT" block *)



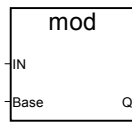
(* ST Equivalence: *)

new_value := LIMIT (min_value, value, max_value);
 (* bounds the value to the [min_value..max_value] set *)

(* IL Equivalence: *)

LD min_value
 LIMIT value, max_value
 ST new_value

MOD



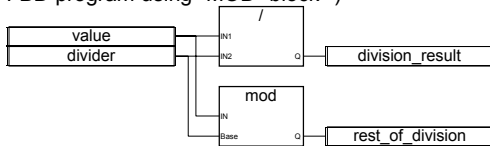
Arguments:

IN	INT	any signed INTEGER analog value
Base	INT	must be greater than zero
Q	INT	modulo calculation (input MOD base) returns -1 if Base <= 0

Description:

Calculates the modulo of an integer value.

(* FBD program using "MOD" block *)



(* ST Equivalence: *)

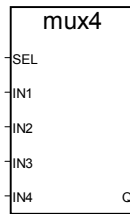
division_result := (value / divider); (* integer division *)
 rest_of_division := MOD (value, division_result); (* rest of the division *)

(* IL Equivalence: *)

LD value
 DIV divider
 ST division_result

LD value
 MOD divider
 ST rest_of_division

MUX4



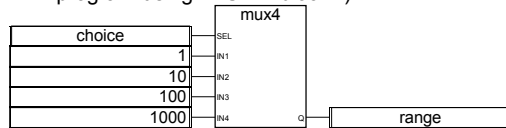
Arguments:

SEL	INT	selector integer value (must be in set [0..3])
IN1..IN4	INT	any integer analog values
Q	INT	= value1 if SEL = 0 = value2 if SEL = 1 = value3 if SEL = 2 = value4 if SEL = 3 = 0 for all other values of the selector

Description:

Multiplexer with 4 entries: selects a value between 4 integer values.

(* FBD program using "MUX4" block *)



(* ST Equivalence: *)

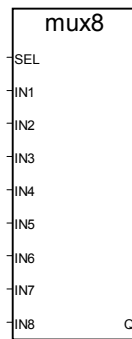
range := MUX4 (choice, 1, 10, 100, 1000);

(* select from 4 predefined ranges, for example, if choice is 1, range will be 10 *)

(* IL Equivalence: *)

LD choice
 MUX4 1,10,100,1000
 ST range

MUX8



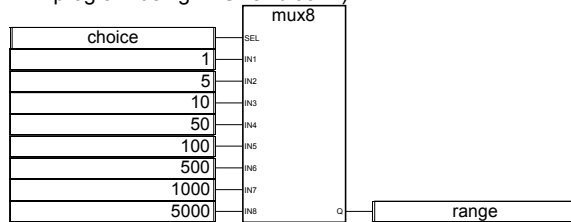
Arguments:

SEL	INT	selector integer value (must be in set [0..7])
IN1..IN8	INT	any integer analog values
Q	INT	= value1 if selector = 0
		= value2 if selector = 1
		...
		= value8 if selector = 7
		= 0 for all other values of the selector

Description:

Multiplexer with 8 entries: selects a value between 8 integer values.

(* FBD program using "MUX8" block *)



(* ST Equivalence: *)

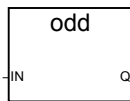
range := MUX8 (choice, 1, 5, 10, 50, 100, 500, 1000, 5000);

(* select from 8 predefined ranges, for example, if choice is 3, range will be 50 *)

(* IL Equivalence: *)

LD	choice
MUX8	1,5,10,50,100,500,1000,5000
ST	range

ODD



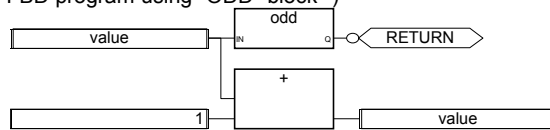
Arguments:

IN	INT	any signed integer analog value
Q	BOO	TRUE if input value is odd FALSE if input value is even

Description:

Tests the parity of an integer: result is odd or even.

(* FBD program using "ODD" block *)



(* ST Equivalence: *)

If Not (ODD (value)) Then Return; End_if;

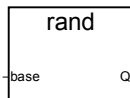
value := value + 1;

(* makes value always even *)

(* IL Equivalence: *)

```
LD      value
ODD
RETNC
LD      value
ADD    1
ST      value
```

RAND



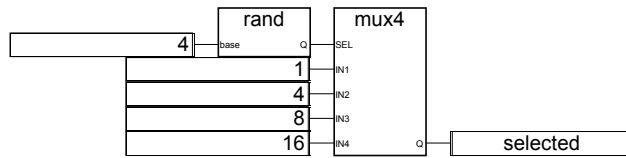
Arguments:

base	INT	defines the allowed set of number
Q	INT	random value in set [0..base-1]

Description:

Gives a random integer value in a given range.

(* FBD program using "RAND" block *)



(* ST Equivalence: *)
 selected := MUX4 (RAND (4), 1, 4, 8, 16);
 (*
 random selection of 1 of 4 pre-defined values
 the value issued of RAND call is in set [0..3],
 so 'selected' issued from MUX4, will get 'randomly' the value
 1 if 0 is issued from RAND,
 or 4 if 1 is issued from RAND,
 or 8 if 2 is issued from RAND,
 or 16 if 3 is issued from RAND,
 *)

(* IL Equivalence: *)
 LD 4
 RAND
 MUX4 1,4,8,16
 ST selected

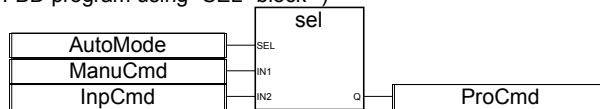
SEL



Arguments:
SEL BOO indicates the chosen value
IN1, IN2 INT any integer analog values
Q INT = value1 if SEL is FALSE
 = value2 if SEL is TRUE

Description:
 Binary selector: selects a value between 2 integer values.

(* FBD program using "SEL" block *)



(* ST Equivalence: *)
 ProCmd := SEL (AutoMode, ManuCmd, InpCmd);

(* process command selection *)

(* IL Equivalence: *)

LD AutoMode
SEL ManuCmd,InpCmd
ST ProCmd

ASCII



Arguments:

IN	MSG	any non-empty string
Pos	INT	position of the selected character in set [1.. len] (len is the length of the IN message)
Code	INT	code of the selected character (in set [0 .. 255]) returns 0 is Pos is out of the string

Description:

Gives the ASCII code of one character in a message string.

(* FBD program using "ASCII" block *)



(* ST Equivalence: *)

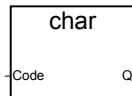
FirstChr := ASCII (message, 1);

(* FirstChr is the Ascii code of the first character of the string *)

(* IL Equivalence: *)

LD message
ASCII 1
ST FirstChr

CHAR



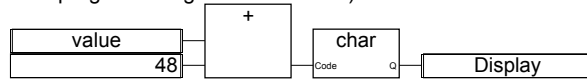
Arguments:

Code	INT	code in set [0 .. 255]
Q	MSG	one character string the character has the ASCII code given in input Code (ASCII code is used modulo 256)

Description:

Gives a one character message string from a given ASCII code.

(* FBD program using "CHAR" block *)



(* ST Equivalence: *)

Display := CHAR (value + 48);

(* value is in set [0..9] *)

(* 48 is the ascii code of '0' *)

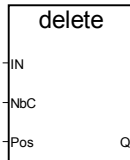
(* result is one character string from '0' to '9' *)

(* IL Equivalence: *)

```

LD      value
ADD     48
CHAR
ST      Display
  
```

DELETE



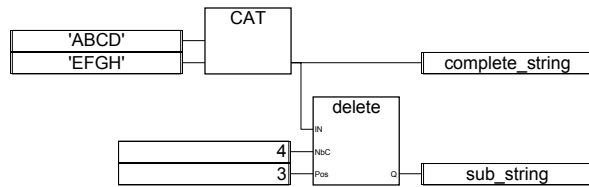
Arguments:

IN	MSG	any non-empty message
NbC	INT	number of characters to be deleted
Pos	INT	position of the first deleted character (first character of the string has position 1)
Q	MSG	modified string empty string if Pos < 1 initial string if Pos > IN string length initial string if NbC <= 0

Description:

Deletes a part of a message string.

(* FBD program using "DELETE" block *)



(* ST Equivalence: *)
 complete_string := 'ABCD' + 'EFGH'; (* complete_string is 'ABCDEFGH' *)
 sub_string := DELETE (complete_string, 4, 3); (* sub_string is 'ABGH' *)

(* IL Equivalence: *)
 LD 'ABCD'
 ADD 'EFGH'
 ST complete_string
 DELETE 4,3
 ST sub_string

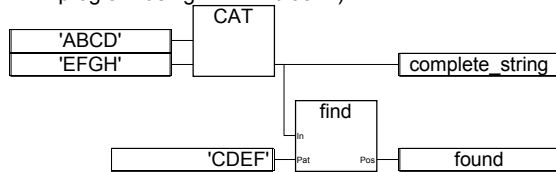
FIND



Arguments:
In MSG any message string
Pat MSG any non-empty string (Pattern)
Pos INT = 0 if sub string Pat not found
 = position of the first character of the first occurrence of the sub-string Pat (first position is 1)
 this function is **case sensitive**

Description:
 Finds a sub-string in a message string. Gives the position in the string of the sub-string.

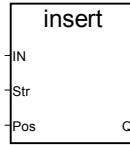
(* FBD program using "FIND" block *)



(* ST Equivalence: *)
 complete_string := 'ABCD' + 'EFGH'; (* complete_string is 'ABCDEFGH' *)
 found := FIND (complete_string, 'CDEF'); (* found is 3 *)

```
(* IL Equivalence: *)
LD      'ABCD'
ADD     'EFGH'
ST      complete_string
FIND   'CDEF'
ST      found
```

INSERT



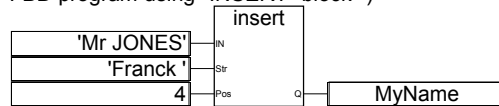
Arguments:

IN	MSG	initial string
Str	MSG	string to be inserted
Pos	INT	position of the insertion the insertion is done before the position (first valid position is 1)
Q	MSG	modified string empty string if Pos <= 0 concatenation of both strings if Pos is greater than the length of the IN string

Description:

Inserts a sub-string in a message string at a given position.

(* FBD program using "INSERT" block *)



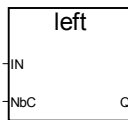
(* ST Equivalence: *)

```
MyName := INSERT ('Mr JONES', 'Frank ', 4);
(* MyName is 'Mr Frank JONES' *)
```

(* IL Equivalence: *)

```
LD      'Mr JONES'
INSERT  'Frank ',4
ST      MyName
```

LEFT



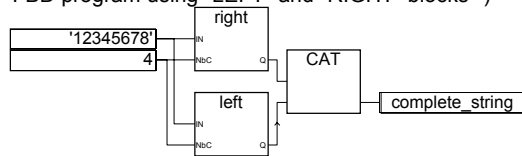
Arguments:

IN	MSG	any non-empty string
NbC	INT	Number of characters to be extracted cannot be greater than the length of the IN string
Q	MSG	left part of the IN string (its length = NbC) empty string if NbC <= 0 complete IN string if NbC >= IN string length

Description:

Extracts the left part of a message string. The number of characters to be extracted is given.

(* FBD program using "LEFT" and "RIGHT" blocks *)



(* ST Equivalence: *)

complete_string := RIGHT ('12345678', 4) + LEFT ('12345678', 4);

(* complete_string is '56781234'

the value issued from RIGHT call is '5678'

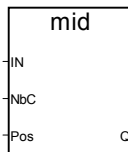
the value issued from LEFT call is '1234'

*)

(* IL Equivalence: First done is call to LEFT *)

```
LD      '12345678'
LEFT    4
ST      sub_string (* intermediate result *)
LD      '12345678'
RIGHT   4
ADD     sub_string
ST      complete_string
```

MID



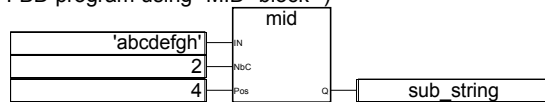
Arguments:

IN	MSG	any non-empty string
NbC	INT	number of characters to be extracted cannot be greater than the length of the IN string
Pos	INT	position of the sub-string the sub-string first character will be the one pointed to by Pos (first valid position is 1)
Q	MSG	middle part of the string (its length = NbC) empty string if parameters are not valid

Description:

Extracts a part of a message string. The number of characters to be extracted and the position of the first character are given.

(* FBD program using "MID" block *)



(* ST Equivalence: *)

sub_string := MID ('abcdefgh', 2, 4);

(* sub_string is 'de' *)

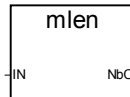
(* IL Equivalence: *)

LD 'abcdefgh'

MID 2,4

ST sub_string

MLEN



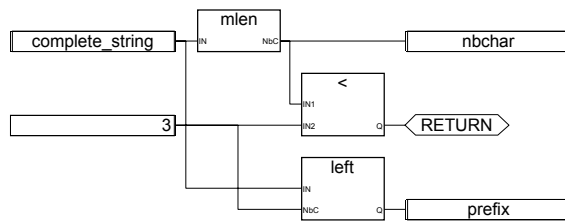
Arguments:

IN	MSG	any string message
NbC	INT	number of characters in the IN string

Description:

Calculates the length of a message string.

(* FBD program using "MLEN" block *)



(* ST Equivalence: *)

nbchar := MLEN (complete_string);

If (nbchar < 3) Then Return; End_if;

prefix := LEFT (complete_string, 3);

(* this program extracts the 3 characters on the left of the string and put the result in the prefix message variable

nothing is done if the string length is less than 3 characters *)

(* IL Equivalence: *)

LD complete_string

MLEN

ST nbchar

LT 3

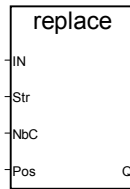
RETC

LD complete_string

LEFT 3

ST prefix

REPLACE



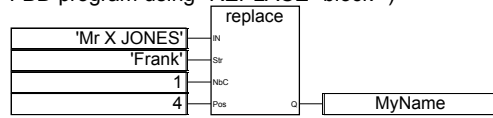
Arguments:

IN	MSG	any string
Str	MSG	string to be inserted (to replace NbC chars)
NbC	INT	number of characters to be deleted
Pos	INT	position of the first modified character (first valid position is 1)
Q	MSG	modified string: - NbC characters are deleted at position Pos - then substring Str is inserted at this position returns empty string if Pos <= 0 returns strings concatenation (IN+Str) if Pos is greater than the length of the IN string returns initial string IN if NbC <= 0

Description:

Replaces a part of a message string by a new set of characters.

(* FBD program using "REPLACE" block *)



(* ST Equivalence: *)

MyName := REPLACE ('Mr X JONES, 'Frank', 1, 4);

(* MyName is 'Mr Frank JONES' *)

(* IL Equivalence: *)

LD 'Mr X JONES'

REPLACE 'Frank',1,4

ST MyName

RIGHT



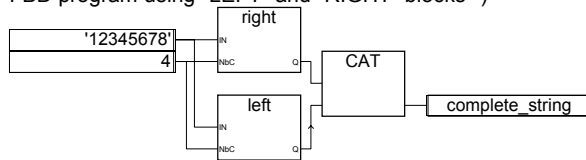
Arguments:

IN	MSG	any non-empty string
NbC	INT	cannot be greater than the length of the IN string
Q	MSG	right part of the string (length = NbC) empty string if NbC <= 0 complete string if NbC >= string length

Description:

Extracts the right part of a message string. The number of characters to be extracted is given.

(* FBD program using "LEFT" and "RIGHT" blocks *)



(* ST Equivalence: *)

complete_string := RIGHT ('12345678', 4) + LEFT ('12345678', 4);

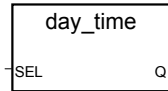
(* complete_string is '56781234' *)

the value issued from RIGHT call is '5678'
 the value issued from LEFT call is '1234'
 *)

(* IL Equivalence: First done is call to LEFT *)

```
LD      '12345678'
LEFT    4
ST      sub_string (* intermediate result *)
LD      '12345678'
RIGHT   4
ADD     sub_string
ST      complete_string
```

DAY_TIME



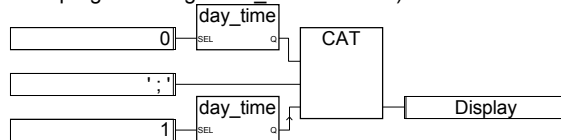
Arguments:

SEL	INT	output selection 0= get current date 1= get current time 2= get day of week
Q	MSG	time/date expressed on a character string 'YYYY/MM/DD' if SEL = 0 'HH:MM:SS' if SEL = 1 day name if SEL = 2 (ex: 'Monday')

Description:

Gives date or time of the day as a message string.

(* FBD program using "DAY_TIME" block *)



(* ST Equivalence: *)

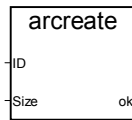
```
Display := Day_Time (0) + ' ; ' + Day_Time (1);
(* Display text format is: 'YYYY/MM/DD ; HH:MM:SS' *)
```

(* IL Equivalence: First done is call to day_time(1) *)

```
LD      1
DAY_TIME
ST      hour_str (* intermediate result *)
LD      0
DAY_TIME
```

```
ADD      ' ; '
ADD      hour_str
ST       Display
```

ARCREATE



Arguments:

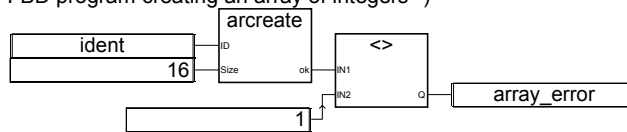
ID	INT	identifier of the array (must be in set [0..15])
Size	INT	number of elements in the array
ok	INT	execution status :
		1 = if array has been successfully created
		2 = invalid array identifier or array already created
		3 = invalid size
		4 = not enough memory

Description:

Creation of an array of integers.

Warning: There are at most **16** arrays in an application. Arrays contain **integer analog** values. As dynamic memory allocation is performed, this function may cause a system error if the array size is too close to the size of the available memory.

(* FBD program creating an array of integers *)



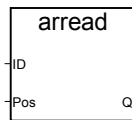
(* ST Equivalence: *)

```
array_error := (ARCREATE (ident, 16) <> 1);
```

(* IL Equivalence: *)

```
LD      ident
ARCREATE 16
NE      1
ST      array_error
```

ARREAD



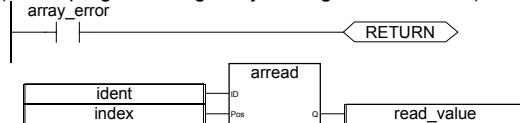
Arguments:

ID	INT	identifier of the array (must be in set [0..15])
Pos	INT	position of the element in the array must be in set [0 .. size-1]
value	INT	value of the element read 0 if the arguments are not valid

Description:

Reads an element in an array of integers.

(* FBD program using array management blocks *)



(* ST Equivalence: *)

```
If (array_error) Then Return; End_if;
read_value := ARREAD (ident, index);
(* array_error comes from the ARCREATE call *)
```

(* IL Equivalence: *)

```
LD      array_error
RETC
LD      ident
ARREAD  index
ST      read_value
```

ARWRITE



Arguments:

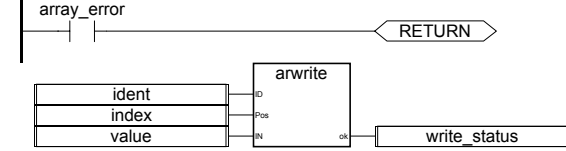
ID	INT	identifier of the array (must be in set [0..15])
Pos	INT	position of the element in the array must be in set [0 .. size-1]
IN	INT	new value for the element
ok	INT	execution status 1 = writing has succeeded

2 = invalid array identifier
 3 = invalid index

Description:

Stores (writes) a value in an array of integers.

(* FBD program using array management blocks *)



(* ST Equivalence: *)

```
If (array_error) Then Return; End_if;
write_status := ARWRITE (Ident, Index, value);
(* array_error comes from the ARCREATE call *)
```

(* IL Equivalence: *)

```
LD      array_error
RETC
LD      ident
ARWRITE index,value
ST      write_status
```

F_ROPEN



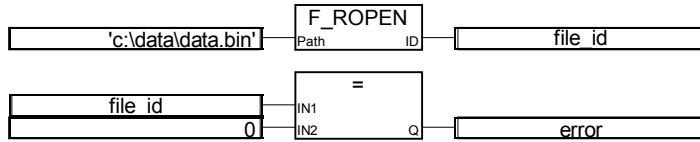
Arguments:

Path	MSG	file name It may include the access path to the file using the \ or / symbol to specify a directory. To ease application portability, / or \ is equivalent.
ID	INT	file number 0 if an error occurs: file does not exists.

Description:

Opens a binary file in read mode. To be used with FX_READ and F_CLOSE.
 This function is not included in the ISaGRAF simulator.

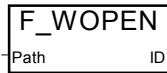
(* FBD program using file management blocks *)



(* ST Equivalence: *)
file_id := F_ROPEN('c:\data\data.bin');
error := (file_id=0);

(* IL Equivalence: *)
LD 'c:\data\data.bin'
F_ROPEN
ST file_id
EQ 0
ST error

F_WOPEN

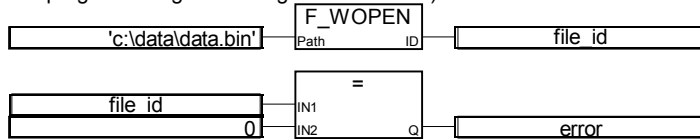


Arguments:

Path	MSG	file name It may include the access path to the file using the \ or / symbol to specify a directory. To ease application portability, / or \ is equivalent.
ID	INT	file number 0 if an error occurs. If the file already exists, it is overwritten.

Description:
Opens a binary file in write mode. To be used with FX_WRITE and F_CLOSE.
This function is not included in the ISaGRAF simulator.

(* FBD program using file management blocks *)

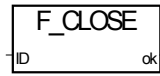


(* ST Equivalence: *)
file_id := F_WOPEN('c:\data\data.bin');
error := (file_id=0);

(* IL Equivalence: *)
LD 'c:\data\data.bin'
F_WOPEN

ST file_id
 EQ 0
 ST error

F_CLOSE



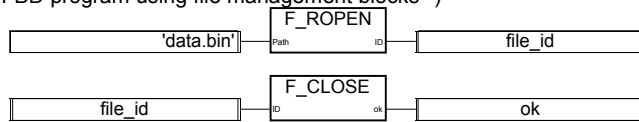
Arguments:

ID	INT	file number: returned by F_ROPEN or F_WOPEN.
ok	BOO	return status TRUE if file close is OK FALSE if an error occurred

Description:

Closes a binary file open with functions F_ROPEN or F_WOPEN.
 This function is not included in the ISaGRAF simulator.

(* FBD program using file management blocks *)



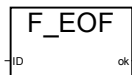
(* ST Equivalence: *)

```
file_id := F_ROPEN('data.bin');
ok := F_CLOSE(file_id);
```

(* IL Equivalence: *)

```
LD            'data.bin'
F_ROPEN
ST            file_id
F_CLOSE            (* file_id is already in the current IL result *)
ST            ok
```

F_EOF



Arguments:

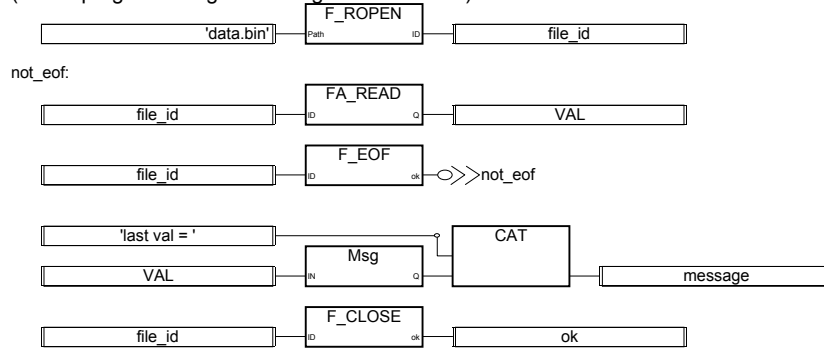
ID	INT	file number: returned by F_ROPEN or F_WOPEN.
ok	BOO	end of file indicator

TRUE if end of file has been reached at the last read or write procedure call.
 With FM_READ, the last message read from a file may not be correct, if the last character is not a string terminator.

Description:

Tests if end of file has been reached.
 This function is not included in the ISaGRAF simulator.

(* FBD program using file management blocks *)



(* ST Equivalence: *)

```

file_id := F_ROPEN('data.bin');
WHILE not(F_EOF(file_id))
  VAL := FA_READ(file_id);
END_WHILE;
MESSAGE := 'last val = ' + msg(VAL);
ok := F_CLOSE(file_id);
  
```

(* IL Equivalence: *)

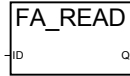
```

          LD      'data.bin'
          F_ROPEN
          ST      file_id
          LD      file_id
          F_EOF
          JMPNC  END_OF_FILE
NOT_EOF: LD      file_id
          FA_READ
          ST      VAL
          LD      file_id
          F_EOF
          JMPNC  NOT_EOF (* if not eof, go on reading *)
END_OF_FILE:LD      VAL
          MSG
          ST      val_msg (* conversion of VAL into a message *)
          LD      'last val = '
  
```

```

ADD      val_msg
ST       MESSAGE
LD       file_id
F_CLOSE
ST       ok
    
```

FA_READ



Arguments:

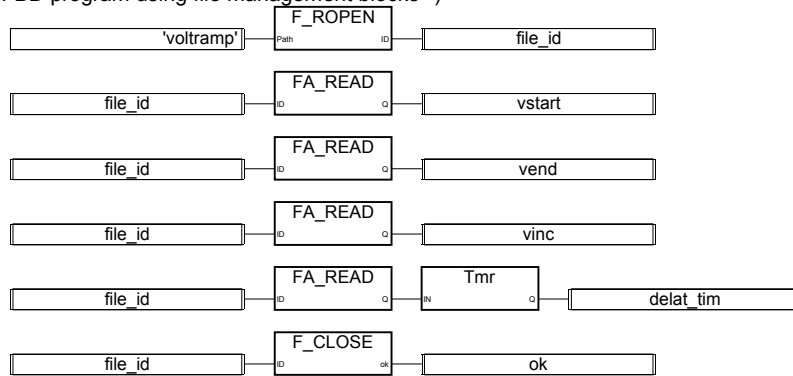
```

ID      INT      file number: returned by F_ROPEN.
Q       INT      integer analog value read from file
    
```

Description:

Reads ANALOG variables from a binary file. To be used with F_ROPEN and F_CLOSE.
 This procedure makes a sequential access to the file, from the previous position.
 The first call after F_ROPEN reads the first 4 bytes of the file,
 each call pushes the reading pointer.
 To check if the end of file is reached, use F_EOF.
 This function is not included in the ISaGRAF simulator.

(* FBD program using file management blocks *)



(* ST Equivalence: *)

```

file_id := F_ROPEN('voltramp.bin');
vstart := FA_READ(file_id);
vend := FA_READ(file_id);
vinc := FA_READ(file_id);
delta_tim := tmr(FA_READ(file_id));
ok := F_CLOSE(file_id);
    
```

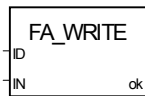
(* IL Equivalence: *)


```

LD      'voltramp.bin'
F_ROPEN
ST      file_id
FA_READ      (* read vstart *)
ST      vstart
LD      file_id
FA_READ      (* read vend *)
ST      vend
LD      file_id
FA_READ      (* read vinc *)
ST      vinc
LD      file_id
FA_READ      (* read delta_tim *)
TMR      (* conversion into a timer *)
ST      delta_tim
LD      file_id
F_CLOSE
ST      ok

```

FA_WRITE



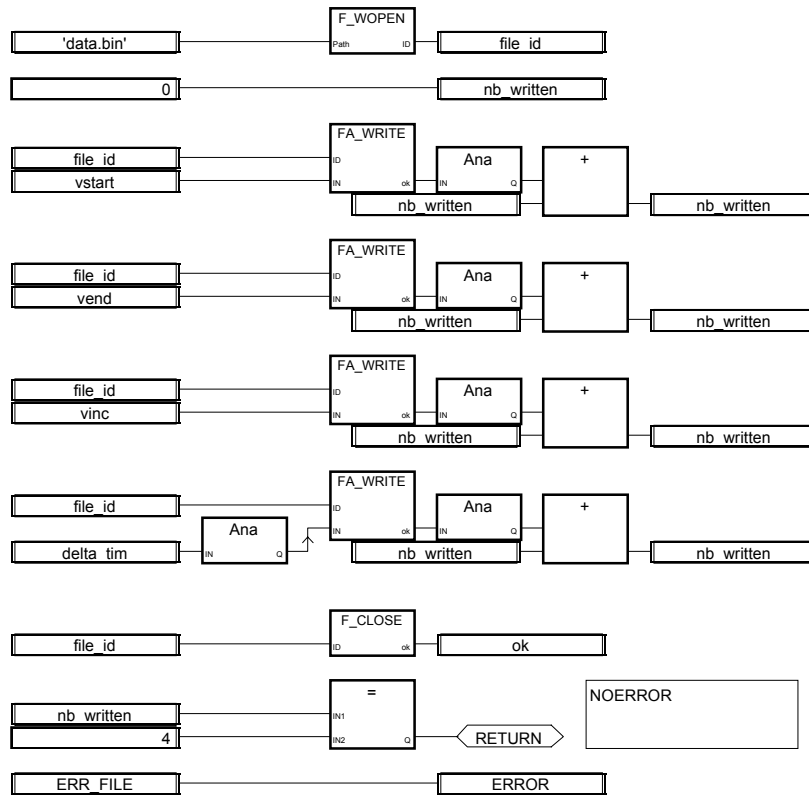
Arguments:

ID	INT	file number: returned by F_WOPEN.
IN	INT	integer analog value To be written in the file
OK	BOO	execution status: TRUE if ok

Description:

Writes ANALOG variables to a binary file.
 This procedure makes a sequential access to the file, from the previous position.
 The first call after F_WOPEN writes the first 4 bytes of the file,
 each call pushes the writing pointer.
 This function is not included in the ISaGRAF simulator.

(* FBD program *)



(* ST Equivalence: *)

```
file_id := F_WOPEN('voltramp.bin');
nb_written := 0;
nb_written := nb_written + ana(FA_WRITE(file_id,vstart));
nb_written := nb_written + ana(FA_WRITE(file_id,vend));
nb_written := nb_written + ana(FA_WRITE(file_id,vinc));
nb_written := nb_written + ana(FA_WRITE(file_id,ana(delta_tim)));
ok := F_CLOSE(file_id);
IF ( nb_written <> 4) THEN
    ERROR := ERR_FILE;
END_IF;
```

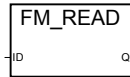
(* IL Equivalence: *)

```
LD      'voltramp.bin'
F_WRITE
ST      file_id
LD      0
```

```

ST      nb_written
LD      file_id      (* write vstart *)
FA_WRITE vstart
ANA
ADD      nb_written
ST      nb_written
LD      file_id      (* write vend *)
FA_WRITE vend
ANA
ADD      nb_written
ST      nb_written
LD      file_id      (* write vinc *)
FA_WRITE vinc
ANA
ADD      nb_written
LD      (* write delta_tim *)
ANA      (* convert it to an integer *)
ST      ana_delta_tim
LD      file_id
FA_WRITE ana_delta_tim
ANA
ADD      nb_written
ST      nb_written
F_CLOSE
ST      ok
LD      nb_written
EQ      4
RETC    (* return if equal 4 *)
LD      ERR_FILE    (* else error *)
ST      ERROR

```

FM_READ

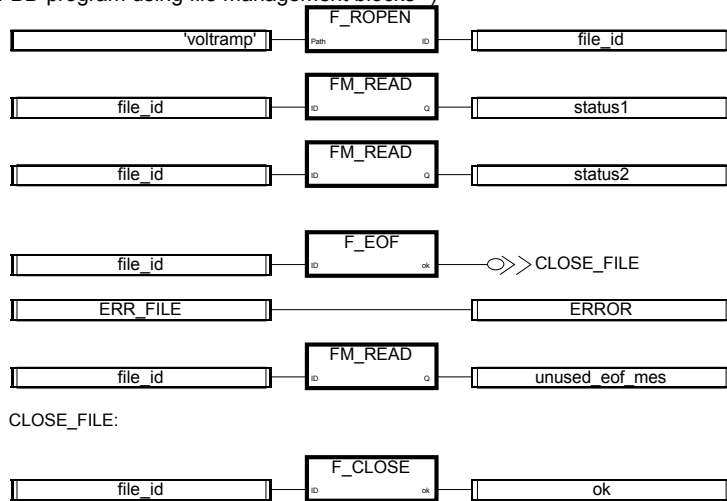
Arguments:

ID	INT	file number: returned by F_ROPEN.
Q	MSG	message value read from file

Description:

Reads MESSAGE variables from a binary file.
 To be used with F_ROPEN and F_CLOSE.
 This procedure makes a sequential access to the file, from the previous position.
 The first call after F_ROPEN reads the first string of the file,
 each call pushes the reading pointer.
 A string is terminated by null (0), end of line ('\n') or return ('\r');
 To check if the end of file is reached, use F_EOF.
 This function is not included in the ISaGRAF simulator.

(* FBD program using file management blocks *)



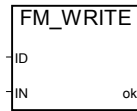
(* ST Equivalence: *)

```
file_id := F_ROPEN('voltramp.bin');
status1 := FM_READ(file_id);
status2 := FM_READ(file_id);
IF (F_EOF(file_id)) THEN
    ERROR := ERR_FILE;
    unused_eof_mes := FM_READ(file_id);
END_IF;
ok := F_CLOSE(file_id);
```

(* IL Equivalence: *)

```
LD 'voltramp.bin'
F_ROPEN
ST file_id
FM_READ (* read status1 *)
ST status1
LD file_id
FM_READ (* read status2 *)
ST status2
LD file_id
F_EOF
JMPNC CLOSE_FILE (* if end of file jump not done *)
LD ERR_FILE
ST ERROR
LD file_id
FM_READ (* read unused_eof_mes *)
ST unused_eof_mes
CLOSE_FILE LD file_id
F_CLOSE
```

ST ok

FM_WRITE

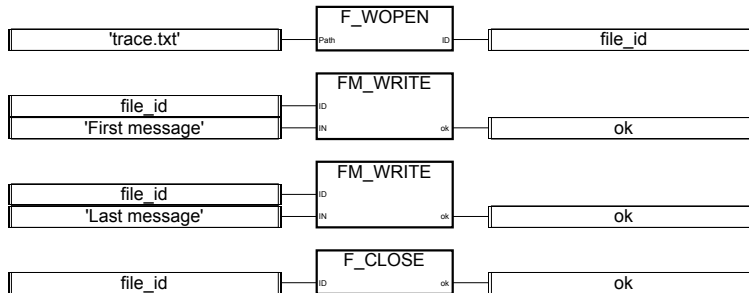
Arguments:

ID	INT	file number: returned by F_WOPEN.
IN	MSG	message value to be written in the file
ok	BOO	execution status TRUE if succeeded

Description:

Writes MESSAGE variables to a binary file.
 To be used with F_WOPEN and F_CLOSE.
 A message is written in the file as a null terminated string.
 This procedure makes a sequential access to the file, from the previous position.
 The first call after F_WOPEN writes the first string to the file,
 each call pushes the writing pointer.
 This function is not included in the ISaGRAF simulator.

(* FBD program using file management blocks *)



(* ST Equivalence: *)

```
file_id := F_WOPEN('trace.txt');
ok := FM_WRITE(file_id,'First message');
ok := FM_WRITE(file_id,'Last message');
ok := F_CLOSE(file_id);
```

(* IL Equivalence: *)

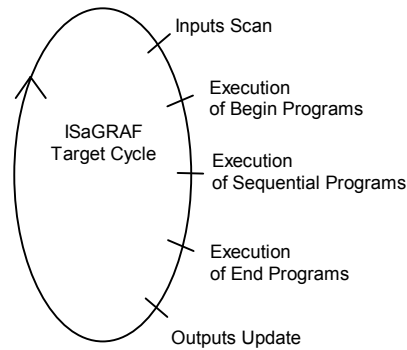
```
LD        'trace.txt'
F_WOPEN
ST        file_id
FM_WRITE 'First message'    (* write first msg *)
```

```
ST      ok
LD      file_id
FM_WRITE'Last message'    (* write second msg *)
ST      ok
LD      file_id
F_CLOSE
ST      ok
```

C. Target User's guide

C.1 Introduction

The ISaGRAF target is a real time software running an ISaGRAF application on your industrial computer system or board according to the following well known scheme:



The target cycle consists in scanning the physical inputs of the process to drive, processing application data according to the ISaGRAF workbench¹ application programs and then performing physical outputs update.

- First part of this section explains how getting started with a specific system target. Respectively DOS, OS-9, VxWorks and NT target. For each one you will find first how to run the ISaGRAF target. Afterwards you will get information on specific features such as: target start up at power up, error management, general behavior, ...
- Second part deals with user's C functions, function blocks and conversion functions implementation method to enhance the ISaGRAF target.
- Third part provides information on Modbus and the ISaGRAF implementation. It describes the frames format of the different function codes.
- Fourth part gives some tools for managing power fail and target restart.

¹ This manual assumes the user is familiar with the ISaGRAF workbench

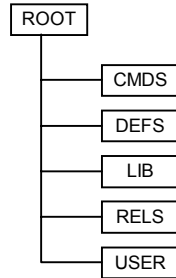
C.2 Installation

The installation requires about 1 Mbyte of free space on your disk. The install.bat delivered with the disk installs all needed files for a specified platform on your PC.

Example: `a:\install a: c:\path`

will install files from disk drive a: to c: on *path* directory.

The following directory architecture is used:



the ROOT directory contains some tools and readme files

the CMDS directory contains executable files

the DEFS directory contains header definition files

the LIB directory contains libraries

the RELS directory contains relocatable (object) files

the USER directory contains user's 'C' procedures for C functions, function blocks and conversion functions (source and header files)

Then just get started with the installed platform.

C.3 Getting started with ISaGRAF DOS target

C.3.1 Running ISaGRAF: ISA.EXE

In the MS-DOS implementation, the target runs as a single program: ISA.EXE. To get started you can simply run the help command `isa -?` from CMDS directory. In such a system, operations can be critical. It is for instance recommended not to overload the communication link to guarantee good performance. The target program does not prevent the running of interrupt driven routines.

▬ **Communication link and configuration: -t Option**

The ISaGRAF target uses a serial link for debugger communication. The name of the port is specified with the `-t` option. As the communication interface is designed to be compatible with any machine, ports COM1, COM2 or COM3 can be used, depending on the BIOS version.

No Default value: If this option is not used, no communication with the target is possible. In such a case, error number 7 may be displayed.

Communication using an Ethernet link is not available with DOS ISaGRAF target. Ask your supplier for special implementation.

The communication parameters have to be set before starting ISaGRAF, so that the user is totally free to use the parameters needed. When using the workbench debugger, make sure the workbench communication parameters (see user's guide: Managing programs) match with the target ones.

Example:

```
MODE COM1:9600,N,8,1
```

Sets up communication parameters to the following values:

- baud rate is 9600
- no parity check
- 8 bits of data
- 1 stop bit

Note that on some BIOS versions, the default workbench setting with 19200 baud is not authorized.

CJ provides the ISAMOD.EXE utility to set the workbench parameters:

```
ISAMOD COM1
```

Is equivalent to `MODE COM1:19200,N,8,1`

▬ **Slave number: -s Option**

This option specifies the target slave number. It can be from 1 to 255 except number 13 (\$0D). This slave number is used through the communication link protocol. It is mainly designed to distinguish slaves from each other when more than one target are connected together. When using the workbench debugger, make sure the workbench slave parameter (see user's guide: Managing programs) matches with the target one.

Default value: The default slave number is 1 (same as the workbench one)

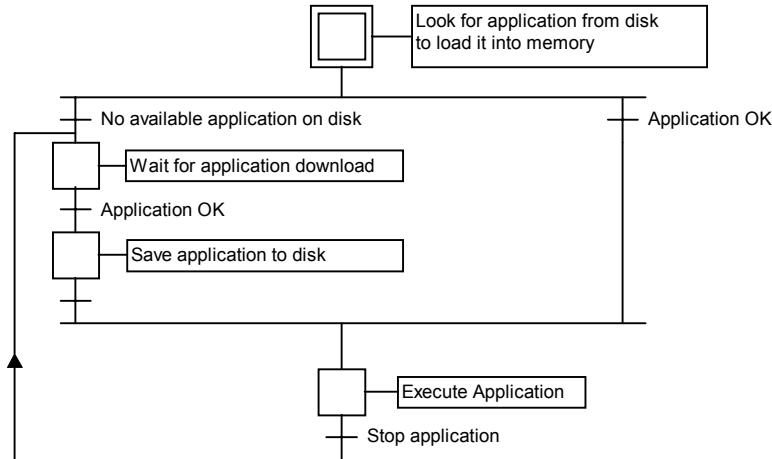
▬ **Examples:**

isamod COM1 Configure COM1 to 19200 baud, no parity, 8 data bits, 1 stop bit.
isa -t=COM1 Starts the ISaGRAF target with default slave number (1) and with COM1 as the communication port.
isa -s=3 -t=COM1 Starts the ISaGRAF target with slave number 3 and with COM1 as the communication port.

C.3.2 Specific features

▬ **ISaGRAF start up**

When the target is started, the following algorithm is executed.



- **Definitions**

The application code is the binary data base generated and downloaded by the workbench and then, executed by the target. It may be completed by the symbol table.

The application symbol table is an ASCII data base generated and downloaded by the workbench. This table makes the link between symbol objects and internal target objects. It is not required on target except for user's specific symbols management. For more information on symbol table see user's guide: Advanced programming techniques.

- **Application backup**

When a new application is downloaded from the workbench debugger into the target, the application code is saved on the target current directory with the file name:

ISAx1 ISaGRAF application code backup file (where x is the slave number)

Furthermore if the application symbol table has been downloaded before, it is also saved on the target current directory with the file name:

ISAx6 ISaGRAF application symbols backup file (where x is the slave number)

When the ISaGRAF target is started, these application code and application symbols files are searched on the current directory and loaded into memory.

If no symbols file is available, then the target starts running the application code, with no symbols loaded.

If no application code is available, then the target is waiting for an application to be downloaded

In order to start the target with a specific application at power up, without using the debugger link, these files can be directly copied to the target current directory disk from the same disk if the workbench is on the same PC, or using a floppy disk. If there is no disk on your target machine you may use a virtual disk.

If the ISaGRAF workbench is installed on the standard \ISAWIN directory:

the application code file of the project MYPROJ is:

\ISAWIN\APL\MYPROJ\appli.x8m

the application symbols file of the project MYPROJ is:

\ISAWIN\APL\MYPROJ\appli.tst

Example:

From the directory where isa.exe is installed, if the following command is entered:

```
copy \ISAWIN\APL\MYPROJ\appli.x8m isa11
```

Then isa.exe will find and execute 'myproj' application.

All these commands can be grouped for instance into a batch file and then started from the workbench tool menu (see user's guide: Managing programs).

≡ **Error management and output messages**

The ISaGRAF target software integrates an error detection management. You will find the warning error list and their description in appendix.

Error detection is processed as follows:

- An error is composed of an error and argument number sent to the ISaGRAF error routine
- If the error detection flag is set in the workbench Make options, the error is processed. If not, the information is lost and the error management ends.

When processed:

- Error number (decimal value) and argument (hexadecimal value) are displayed on the default stdout output
- Error number and argument are pushed into a ring FIFO error buffer in order to be retrieved at a later time. The error buffer size is set in the workbench Make options. When the buffer is full, at each new incoming error, the oldest one is lost.
- Errors can be pulled either from the debugger or from the running application using the SYSTEM call (see user's guide).

When the debugger detects an error, a message describing the error is displayed in the error window. Depending on the context of the application (running or not) the debugger may

display the name of the object (variable or program) where the error comes from, or the argument error (decimal value) into brackets [x] which has a different meaning for each error.

A welcome message and error values are displayed on the default stdout output when the target starts and when an error is detected. If the display is not wanted on the standard output channel, a redirection command can be used such as:

```
isa -t=COM1 -s=1 >NUL
```

≡ **System clock**

As the ISaGRAF target is designed to run on any system, the time reference used for both cycle synchronization and timer variables refresh is the standard tick which is about 55 milliseconds.

Thus, it is not possible to have an accuracy on timer variables better than 55ms. For the same reason, a specified cycle duration less or equal to 55 ms and different from zero will generate an cycle duration overflow error (error 62) and no triggered cycles.

The advantage of not modifying the system tick is that any of the resident applications, or C functions and function blocks integrated into the application will never be disturbed by the ISaGRAF execution.

Ask your supplier for a special implementation if your application requires more accuracy.

≡ **Exit key**

While testing an application in non-industrial conditions on a desktop PC, the user may wish to stop ISaGRAF: this is done by pressing a complex combination of keys to prevent unexpected stops. This key sequence is:

shift + ctrl + alt

Of course, if the industrial application should not be stopped when a key is hit, something should be provided to disable these combinations.

One dangerous side effect of these fast exits, is that the IO board interface is not closed. Thus the clean way for stopping your ISaGRAF target is:

- stop the application from the debugger (this will close the IO boards)
- stop ISaGRAF target from the keyboard

≡ **Application size**

As the ISaGRAF MS-DOS target is designed for Intel real mode, the maximum size of a data structure is 64K. Thus, the application code downloaded by the workbench should not exceed this limit. In some very rare cases, internal structure allocated by ISaGRAF may also exceed this limit and crash your application after download. Furthermore the whole available memory is limited to the 640K of conventional memory.

Ask your supplier for a special implementation if your application require more memory capacity.

C.4 Getting started with ISaGRAF OS9 target

First of all you need to transfer files (at least executable files from CMDS directory) to your OS-9 target using any file transfer tool.

Then to get started you can simply run the help commands from your OS-9 system CMDS directory:

```
isa -?  
isaker -?  
isatst -?  
Isanet -?
```

C.4.1 Running the ISaGRAF single task: isa

The ISaGRAF target can be run as single task. But in such a configuration operations can be critical. It is for instance recommended not to overload the communication link to guarantee good performance. On the OS-9 multitasking system, different ISaGRAF single task targets can be run on the same CPU as long as their slave number and communication port are different.

This single task implementation has mainly been designed for poor hardware platform such as low cost boards or MS-DOS PC's or to make a first prototype when porting on a new platform. Therefore the multitasking ISaGRAF target implementation should be preferred.

The ISaGRAF single task target does not prevent the running of background processes or interrupt driven routines.

▬ **Communication link and configuration: -t Option**

The ISaGRAF single task target uses a serial link for debugger communication. The name of the descriptor is specified with the -t option.

No Default value: If this option is not used, no communication with the target is possible. In such a case, error number 7 may be displayed.

Communication using an Ethernet link is not available with the single task implementation.

The serial link device is opened in binary data transfer mode (no control characters, no XON/XOFF). Other communication parameters have to be set before starting ISaGRAF, so that the user is totally free to use the parameters needed. When using the workbench debugger, make sure the workbench communication parameters (see user's guide: Managing programs) match with the target ones.

Example:

```
xmode /t0 baud=19200
```

Sets up communication baud rate to 19200 baud on /t0 device

▬ **Slave number: -s Option**

This option specifies the target slave number. It can be from 1 to 255 except number 13 (\$0D). This slave number is used through the communication link protocol. It is needed to distinguish slaves from each other when more than one target are running. When using the workbench debugger, make sure the workbench slave parameter (see user's guide: Managing programs) matches with an existing target.

Default value: The default slave number is 1 (same as the workbench one)

≡ **Examples:**

- isa -t=/t0** Starts an ISaGRAF single task target with default slave number (1) and with /t0 as the communication port.
- isa -s=3 -t=/t1** Starts an ISaGRAF single task target with slave number 3 and with /t1 as the communication port.
- isa -t=/t0 &**
- isa -s=3 -t=/t1** Starts two ISaGRAF single task targets. One with default slave number (1) and with /t0 as the communication port. The other with slave number 3 and with /t1 as the communication port.

C.4.2 Running the ISaGRAF multitasks: isaker, isatst, isanet

To improve the response time of the ISaGRAF target kernel and of the communication link, the target is split into two tasks separating communication job (isatst or isanet communication task) from application execution (isaker kernel task).

Such architecture is more flexible. It allows the user to run more than one communication task linked with the same kernel task or to run up to 4 kernels with the same communication task. This makes easy some integration such as a process visualization link and the workbench debugger link on the same application or a single link up to 4 different applications through the same physical port.

The kernel and communication tasks are independent and can be separately forked. The only requirement is that the kernel task(s) has to be started first so that it initializes its system environment and the communication task(s) can link it.

The ISaGRAF multitask target does not prevent the running of background processes or interrupt driven routines

C.4.2.1 Running the kernel task: isaker

≡ **Slave number: -s Option**

This option specifies the target kernel slave number. It can be from 1 to 255 except number 13 (\$0D). This slave number is used through the communication link protocol and by the communication task(s) linked to the kernel. It is needed to distinguish slaves from each other when more than one target are running.

Default value: The default slave number is 1 (same as the workbench one)

C.4.2.2 Running the serial communication task: isatst

≡ **Communication link and configuration: -t Option**

The target communication task isatst uses a serial link for debugger communication. The name of the descriptor is specified with the -t option.

No Default value: If this option is not used, no communication with the target is possible. In such a case, error number 7 may be displayed.

Communication using an Ethernet link is not available with isatst task implementation.

The serial link device is opened in binary data transfer mode (no control characters, no XON/XOFF). Other communication parameters have to be set before starting ISaGRAF, so that the user is totally free to use the parameters needed. When using the workbench debugger, make sure the workbench communication parameters (see user's guide: Managing programs) match with the target ones

Example:

```
xmode /t0 baud=19200
```

Sets up communication baud rate to 19200 baud on /t0 device

═ ***Slave number: -s Option***

This option specifies the target kernel slave number(s) the communication task is linked to. It can be from 1 to 255 except number 13 (\$0D). This option can be repeated up to 4 times to link up to 4 different kernel slaves. This slave number is used through the communication link protocol. It is needed to distinguish slaves from each other when more than one target are running. When using the workbench debugger, make sure the workbench slave parameter (see user's guide: Managing programs) matches with an existing target (kernel and communication tasks).

Default value: The default slave number is 1 (same as the workbench one)

═ ***Communication task logical number: -c Option***

This option specifies the communication task logical number. It is used to manage more than one communication task at a time. It can be from 1 to 255 and must be different for each communication task.

Default value: The last -s specified option is used. The default value ensure compatibility with previous (3.0) ISaGRAF versions.

C.4.2.3 Running the Ethernet communication task: isanet

═ ***Communication link and configuration: -t Option***

The target communication task isanet uses a standard Ethernet link for debugger communication. The port number is specified with the -t option.

No Default value: If this option is not used, no communication with the target is possible. In such a case, error number 7 may be displayed.

When using the workbench debugger, make sure the workbench communication parameters (see user's guide: Managing programs) match with the target ones

For ISaGRAF, the OS-9 target is the server and the debugger is the client which connects the specified port number.

Before starting your first debug session on Ethernet, you should make sure your OS-9 Ethernet device is well configured. You may for instance send a ping to the OS-9 system

▬ **Slave number: -s Option**

This option specifies the target kernel slave number(s) the communication task is linked to. It can be from 1 to 255 except number 13 (\$0D). This option can be repeated up to 4 times to link up to 4 different kernel slaves. This slave number is used through the communication link protocol. It is needed to distinguish slaves from each other when more than one target are running. When using the workbench debugger, make sure the workbench slave parameter (see user's guide: Managing programs) matches with an existing target (kernel and communication tasks).

Default value: The default slave number is 1 (same as the workbench one)

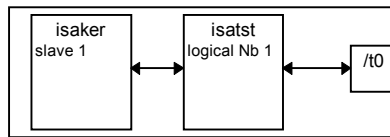
▬ **Communication task logical number: -c Option**

This option specifies the communication task logical number. It is used to manage more than one communication task at a time. It can be from 1 to 255 and must be different for each communication task.

Default value: The last -s specified option is used. The default value ensure compatibility with previous (3.0) ISaGRAF versions.

C.4.2.4 Examples:

**isaker &
isatst -t=/t0**

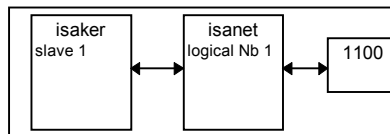


Starts:

An ISaGRAF kernel task with default slave number (1).

An ISaGRAF serial communication task, on /t0 com Port, linked to default slave number (1), and with default logical number (last specified slave number = default = 1).

**isaker &
isanet -t=1100**



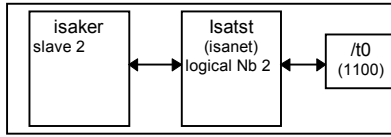
Starts:

An ISaGRAF kernel task with default slave number (1).

An ISaGRAF Ethernet communication task, on Port number 1100, linked to default slave number (1), and with default logical number (last specified slave number = default = 1).

isaker -s=2 &

isatst -t=/t0 -s=2 (respectively isanet -t=1100 -s=2)



Starts:

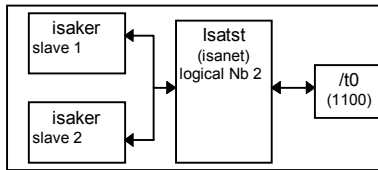
An ISaGRAF kernel task with slave number 2.

An ISaGRAF serial (Ethernet) communication task, on /t0 com Port (Port number 1100), linked to slave number 2, and with default logical number (last specified slave number = 2).

isaker -s=1 &

isaker -s=2 &

isatst -t=/t0 -s=1 -s=2 (respectively isanet -t=1100 -s=1 -s=2)



Starts:

An ISaGRAF kernel task with slave number 1.

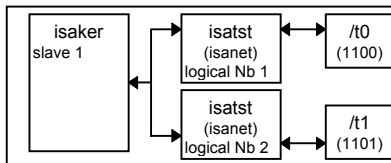
An ISaGRAF kernel task with slave number 2.

An ISaGRAF serial (Ethernet) communication task, on /t0 com Port (Port number 1100), linked to slaves number 1 and 2, and with default logical number (last specified slave number = 2).

isaker -s=1 &

isatst -t=/t0 -s=1 -c=1 & (respectively isanet -t=1100 -s=1 -c=1 &)

isatst -t=/t1 -s=1 -c=2 (respectively isanet -t=1101 -s=1 -c=2)



Starts:

An ISaGRAF kernel task with slave number 1.

An ISaGRAF serial (Ethernet) communication task, on /t0 com Port (Port number 1100), linked to slaves number 1, and with logical number 1.

An ISaGRAF serial (Ethernet) communication task, on /t1 com Port (Port number 1101), linked to slaves number 1, and with logical number 2.

Note:

Serial and Ethernet communication tasks can be mixed.

C.4.3 Specific features

Communication link

As OS-9 Serial Character Manager is very flexible, almost any bi-directional physical device supported by Microware may be used:

Example:

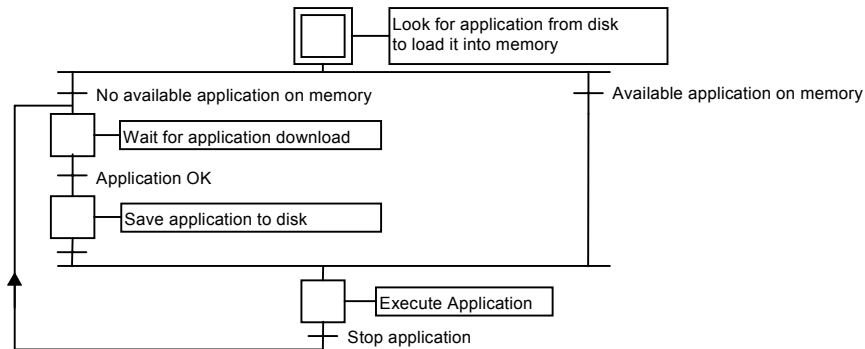
The serial link can be a network path to a physical port located on an another CPU.

Then the -t option would be used for example as following: -t=/nr/MASTER/t0

Where the communication link is deported on a CPU called MASTER on a ramnet network. The physical port used is /t0.

ISaGRAF start up

When the target is started, the following algorithm is executed.



Definitions

The application code is the binary data base generated and downloaded by the workbench and then, executed by the target. It may be completed by the symbol table.

The application symbol table is an ASCII data base generated and downloaded by the workbench. This table makes the link between symbol objects and internal target objects. It is not required on target except for user's specific symbols management. For more information on symbol table see user's guide: Advanced programming techniques.

- **ISaGRAF OS-9 objects and Multi-application**

Every ISaGRAF public object name begins with '**ISAxn**' where **x** is the kernel slave number and **n** a space number with a specific meaning, except for **ISAy3** where **y** is the communication task logical number in the multitask implementation.

Different applications (kernels and communication tasks) can run at the same time on a CPU, as far as they have respectively different slave numbers and different communication task logical numbers. Nevertheless while running different applications, the user must take care of some application objects shared access such as I/O boards. For instance different applications (kernels) may use distinct physical boards unless some kind of I/O server or semaphore is implemented through the I/O driver.

OS-9 object names:

Disk Files:

ISAx1	ISaGRAF application code backup file
ISAx6	ISaGRAF application symbol backup file

Memory Modules:

ISAx0	ISaGRAF kernel system data
ISAx1	ISaGRAF application code
ISAx2	ISaGRAF kernel real time data base
ISAy3	ISaGRAF communication data exchange buffer
ISAx4	ISaGRAF on line modification 1 application code
ISAx5	ISaGRAF on line modification 2 application code
ISAx6	ISaGRAF application symbol

Therefore the user must take care to not use the same object names.

- **Application backup**

When a new application is downloaded from the workbench debugger into the target, the application code is saved on the target current directory with the file name:

ISAx1 ISaGRAF application code backup file (where x is the slave number)

Furthermore if the application symbol table has been downloaded before, it is also saved on the target current directory with the file name:

ISAx6 ISaGRAF application symbols backup file (where x is the slave number)

When the ISaGRAF target is started, these application code and application symbols files are searched on the current directory and loaded into memory as data modules with same names.

Then, if no symbols table is available on memory, the target starts running the application code, with no symbols loaded.

If no application code is available on memory, then the target is waiting for an application to be downloaded

In order to start the target with a specific application at power up, without using the debugger link:

- A first way may consists in directly copying these files to the target current directory disk from the PC host where the workbench is installed, using any file transfer tool. You can use the workbench tool menu (see user's guide: Managing programs) to ease these manipulations.

- A second way may consists in storing the application code (and if necessary the application symbol table) in a non volatile memory (like a PROM or EPROM), from files from the PC host where the workbench is installed, with your own tools.

Then at system power up, if required (for example because of faster access or breakpoint management), you may load the application code (and if necessary the application symbol table) from the PROM to the RAM as **ISAx1** (and if necessary **ISAx6**) memory data module(s), with your own tools.

WARNING:

The breakpoint management of the ISaGRAF debugger cannot run correctly if the application code module is not accessible for writing. This is not a problem, as your application has normally been fully tested before.

On the PC host, if the ISaGRAF workbench is installed on the standard \ISAWIN directory: the application code file of the project MYPROJ is:

\ISAWIN\APL\MYPROJ\appli.x6m (corresponding to isax1 on the target).

the application symbols file of the project MYPROJ is:

\ISAWIN\APL\MYPROJ\appli.tst (corresponding to isax6 on the target).

≡ **Error management and output messages**

The ISaGRAF target software integrates an error detection management. You will find the warning error list and their description in appendix.

Error detection is processed as follows:

- An error is composed of an error and argument number sent to the ISaGRAF error routine
- If the error detection flag is set in the workbench Make options, the error is processed. If not, the information is lost and the error management ends.

When processed:

- Error number (decimal value) and argument (hexadecimal value) are displayed on the default stdout output
- Error number and argument are pushed into a ring FIFO error buffer in order to be retrieved at a later time. The error buffer size is set in the workbench Make options. When the buffer is full, at each new incoming error, the oldest one is lost.
- Errors can be pulled either from the debugger or from the running application using the SYSTEM call (see user's guide).

When the debugger detects an error, a message describing the error is displayed in the error window. Depending on the context of the application (running or not) the debugger may display the name of the object (variable or program) where the error comes from, or the argument error (decimal value) into brackets [x] which has a different meaning for each error.

A welcome message and error values are displayed on the default stdout output when the target starts and when an error is detected. If the display is not wanted on the standard output channel, a redirection command can be used such as:

```
prog_name [options] >>>nil
```

≡ **Cycle duration, task behaviors, and task priorities**

- At the end of an ISaGRAF cycle, just before starting a new one, the following algorithm is performed:
 - If a cycle timing is specified (from the workbench: see user's guide: Managing programs) then the CPU is relinquished for the remained time period (specified cycle time - current application one). If this remained time period is negative an overflow is generated and the CPU is relinquished for 1 tick to force the scheduling

If no cycle timing is specified, or if the remained time is less or equal than 1 tick or equal to zero, then the CPU is relinquished for 1 tick to force the scheduling

The target timing accuracy correspond to the OS-9 system tick one.

A specified cycle timing is commonly used to trig cycles or to yield the CPU to other tasks running on the OS-9 system.

- The communication task is in sleep status while there are no incoming data through the communication link. When needed, this task gets information on the running application through a question/answer protocol with the kernel task. The communication task asks for a question to the kernel. At the end of the cycle (to have a synchronous application image), the kernel gives the answer to the communication task.

The ISaGRAF tasks do no modify the priority they have been given. The user is free to adjust these priorities according to ISaGRAF task behaviors described above and its whole application requirements.

For instance, to make sure that ISaGRAF is not preempted by a low priority task, the OS-9 task management parameters such as **MIN_AGE** and **MAX_AGE**, can be modified.

Terminal mode

The target serial communication protocol recognizes a sequence of 3 carriage return characters (\$0D) and then starts an OS-9 shell task, if it is available, on the serial link device. This allows to get an OS-9 shell prompt on any terminal, using the ISaGRAF target serial link.

Example:

From the host PC:

- Close the ISaGRAF debugger.
- Start a Windows Terminal session (accessories group) with the right communication parameters
- Hit 3 carriage return

You are now logged on an OS-9 Shell

- Type **logout** to exit the terminal mode.

WARNING:

The terminal mode session must always be left in a clean way using logout and nothing else, otherwise next connection with the workbench will be unsuccessful.

C.5 Getting started with ISaGRAF VxWorks target

To run the ISaGRAF target(s), a few commands need to be executed on the VxWorks system, in order to set the configuration environment and finally spawn the ISaGRAF target(s). All these commands may be started from a script file. They are described in next chapters.

C.5.1 The system resource manager: isassr.o

This module is always needed, in any configuration of the ISaGRAF target, and must be the first of the ISaGRAF target loaded modules. It enables the system resource managing of multi targets running.

C.5.2 Common features to isa.o, isakerse.o and isakeret.o

To run ISaGRAF, one of these modules may be loaded.

isa.o: enable to start ISaGRAF single task targets (serial communication link only).
 isakerse.o: enable to start ISaGRAF multitask targets (serial communication link only).
 isakeret.o: enable to start ISaGRAF multitask targets (serial or/and Ethernet communication link)

These modules are detailed in next chapters

▣ **Serial communication link configuration**

The ISaGRAF target basically uses a serial link for debugger communication. When opened, no configuration are performed on the specified serial link device by the ISaGRAF target. So that, the user is totally free to use the parameters needed. Nevertheless a binary data transfer mode (RAW mode) is required. In that way the *ISAMOD ()* subroutine is provided.

```
uchar ISAMOD
(
  char *desc,      /* Serial device name */
  uint32 baudrate /* Baud rate          */
)
```

Description:

Configure specified serial link device for binary data transfer with specified baud rate

return value:

0 if successful, BAD_RET if errors occur

When using the workbench debugger, make sure the workbench communication parameters (see user's guide: Managing programs) match with the target ones.

▣ **System clock rate**

The global variable CLKRATE (uint32) needs to be initialized to the VxWorks system clock rate. In that way you can use:

```
CLKRATE = sysClkRateGet ()
```

The default value of CLKRATE is 60Hz.

C.5.3 Running the ISaGRAF single task: isa.o

The ISaGRAF target can be run as single task. But in such a configuration operations can be critical. It is for instance recommended not to overload the communication link to guarantee good performance. On the VxWorks multitasking system, different ISaGRAF single task targets can be run on the same CPU as long as their slave number and communication port are different.

This single task implementation has mainly been designed for poor hardware platform such as low cost boards or MS-DOS PC's or to make a first prototype when porting on a new platform. Therefore the multitasking ISaGRAF target implementation should be preferred.

The ISaGRAF single task target does not prevent the running of background processes or interrupt driven routines.

▬ **Slave(s) registration**

An ISaGRAF target is characterized by its slave number. Its value can be from 1 to 255 except number 13 (\$0D). This slave number is used through the communication link protocol. It is needed to distinguish slaves from each other when more than one target are running. Therefore, before starting the ISaGRAF target(s) task, you need to register it (them). In that way the *isa_register_slave()* subroutine is provided.

```
uchar isa_register_slave
(
    uchar slave /* slave number */
)
```

Description:

Add a new slave registration to the multi targets management system

return value:

0 if succeeded, BAD_RET if errors occur

▬ **Application backup file storage unit**

The global variable TSK_FUNIT (char *) can be initialized to a string containing the path unit for application file backup. The ISaGRAF target simply uses the standard file management routines fopen, fread, fwrite, fclose for application file backup.

The default value is an empty string ("") to specify that there are no storage unit.

Example:

```
TSK_FUNIT = "host name:/C:/ISaGRAF/target/apl/"
```

Specify ISaGRAF\target\apl\, on root of C: unit, on the *host_name* PC, as application file backup directory. Be careful not to forget the last slash, otherwise the backup is done on ISaGRAF\target\ directory with apl prefixed file names.

If needed, this variable can be set to different path units, for each target, before each spawn. You will find more detailed information on application backup files, in the specific features; application backup chapter.

▬ **End of Cycle control**

The TSK_NBTCKSCHED (uint 32) variable can be set to a value specifying a delay in tick, used by the ISaGRAF target at the end of the cycle.

The default value is 0 (same priority task scheduling).

If needed, this variable can be set to different value, for each target to start, before each spawn.

You will find more detailed information, in the specific features; Cycle duration, task behaviors and task priorities chapter.

≡ **ISaGRAF target spawning**

Once the configuration environment has been set, the last step consists in spawning the ISaGRAF target(s): isa_main.

```
uchar isa_main
(
  uchar slave,    /* Slave number    */
  char *com       /* Serial device name */
)
```

Description:

Starts an ISaGRAF target task.

return value:

return a value different from zero if errors occur.

The slave number is the same as discussed in the slave registration chapter.

More than one target can be started as long as their slave number and communication port are different.

When using the workbench debugger, make sure the workbench slave parameter (see user's guide: Managing programs) matches with an existing target.

≡ **Example**

This example shows how to start an ISaGRAF single task target with slave number 1 and with /tyCo/1 device for the serial link.

The current host directory is the one where the target is installed.

```
load isassr.o module
```

```
ld < RELS/isassr.o
```

```
load isa.o module
```

```
ld < CMDS/isa.o
```

```
serial communication configuration
```

```
ISAMOD ("/tyCo/1", 19200)
```

```
System clock rate
```

```
CLKRATE = sysClkRateGet ()
```

```
slave registration
```

```
isa_register_slave (1)
```

```
File storage unit (could be skipped because default set)
```

```
TSK_FUNIT = ""
```

```
End of cycle control (could be skipped because default set)
```

TSK_NBTCKSCHED = 0

ISaGRAF target spawning
sp (isa_main, 1, "/tyCo/1")

C.5.4 Running the ISaGRAF multitasks: isakerse.o and isakeret.o

To improve the response time of the ISaGRAF target kernel and of the communication link, the target is split into two tasks separating communication job (communication task) from application execution (kernel task).

Such architecture is more flexible. It allows the user to run more than one communication task linked with the same kernel task or to run up to 4 kernels with the same communication task. This makes easy some integration such as a process visualization link and the workbench debugger link on the same application or a single link up to 4 different applications through the same physical port.

The kernel and communication tasks are independent and can be separately spawned. The only requirement is that the kernel task(s) has to be started first so that it initializes its system environment and the communication task(s) can link it.

The ISaGRAF multitask target does not prevent the running of background processes or interrupt driven routines.

Two modules are proposed depending on communication hardware capabilities:

- Kernel and serial link: isakerse.o

This module enable to start the kernel task(s) and the serial communication task(s).

- Kernel and serial or/and Ethernet link: isakeret.o

This module enable to start the kernel task(s) and the serial or/and Ethernet communication task(s).

The way of starting ISaGRAF is the same for isakerse.o and isakeret.o modules, except that for isakeret.o, you can either specify a serial device name, or a port number for Ethernet link, as communication device name parameter when starting the ISaGRAF communication task(s): `tst_main_ex` (see below).

For ISaGRAF, the VxWorks target is the server and the debugger is the client which connects the specified port number.

▬ Kernel(s) registration

An ISaGRAF kernel is characterized by its slave number. Its value can be from 1 to 255 except number 13 (\$0D). This slave number is used through the communication link protocol and by the communication task(s) linked to the kernel. It is needed to distinguish slaves from each other when more than one target are running. Therefore, before starting the ISaGRAF kernel task(s), you need to registered it (them). In that way the `isa_register_slave()` subroutine is provided.

```
uchar isa_register_slave
(
    uchar slave /* slave number */
)
```

Description:

Add a new kernel slave registration to the multi targets management system

return value:

0 if succeeded, BAD_RET if errors occur

═ **Communication task(s) registration**

An ISaGRAF communication task is characterized by its logical number. It is used to manage more than one communication task at a time. It can be from 1 to 255 and must be different for each communication task. Therefore, before starting the ISaGRAF communication task(s), you need to registered it (them). In that way the *isa_register_com()* subroutine is provided.

```
uchar isa_register_com
(
    uchar com_id    /* com. task identifier */
)
```

Description:

Add a new communication task registration to the multi targets management system

return value:

0 if succeeded, BAD_RET if errors occur

═ **Application backup file storage unit**

The global variable TSK_FUNIT (char *) can be initialized to a string containing the path unit for application file backup. The ISaGRAF target simply uses the standard file management routines fopen, fread, fwrite, fclose for application file backup.

The default value is an empty string ("") to specify that there are no storage unit.

Example:

```
TSK_FUNIT = "host name:/C:/ISaGRAF/target/apl/"
```

Specify ISaGRAF\target\apl\, on root of C: unit, on the *host_name* PC, as application file backup directory. Be careful not to forget the last slash, otherwise the backup is done on ISaGRAF\target\ directory with apl prefixed file names.

If needed, this variable can be set to different path units, for each target to start, before each kernel spawn.

You will find more detailed information on application backup files, in the specific features ; application backup chapter.

═ **End of Cycle control**

The TSK_NBTCKSCHED (uint 32) variable can be set to a value specifying a delay in tick used by the ISaGRAF target at the end of the cycle.

The default value is 0 (same priority task scheduling).

If needed, this variable can be set to different value, for each kernel, before each kernel spawn.

You will find more detailed information, in the specific features; Cycle duration, task behaviors and task priorities chapter.

═ ISaGRAF kernel spawning

Once the configuration environment has been set, one of the last steps consists in spawning the ISaGRAF kernel(s): isa_main.

```
uchar isa_main
(
  uchar slave,      /* Slave number      */
  char *com         /* NOT USED Empty string is OK */
)
```

Description:

Starts an ISaGRAF kernel task

return value:

return a value different from zero if errors occur.

The slave number is the same as discussed in the slave registration chapter. More than one kernels can be started as long as their slave number are different.

═ ISaGRAF communication task spawning

Once the configuration environment has been set, one of the last steps consists in spawning the ISaGRAF communication task(s): tst_main_ex.

```
uchar tst_main_ex
(
  char *com,        /* Communication device name */
  uchar *slave,     /* Location of a 4 Bytes field specifying kernel(s) slave
                    to link to */
  uchar com_id      /* communication task identifier */
)
```

Description:

Starts an ISaGRAF communication task

return value:

return a value different from zero if errors occur.

The 4 Bytes field specifies the kernel slave(s), the communication task is linked to. If less than 4 kernel slaves are needed, the field must be completed with zero. Once the task has started, this field is not needed any more.

The communication device name corresponds to the serial device name to be used for the communication link.

More than one communication tasks can be started as long as their task identifier are different.

When using the workbench debugger, make sure the workbench communication link parameters (see user's guide: Managing programs) match with an existing target (kernel and communication tasks).

═ Example:

This example shows how to start:

An ISaGRAF kernel task with slave number 1.

An ISaGRAF communication task identified with number 1, linked to the kernel slave 1 and with /tyCo/1 device for the serial link.

An ISaGRAF communication task identified with number 2, linked to the kernel slave 1 and with 1100 port number for the Ethernet communication link.

The current host directory is the one where the target is installed.

load isassr.o module

ld < RELS/isassr.o

load isakeret.o module (You may load isakerse.o when no Ethernet communication link is needed)

ld < CMDS/isakeret.o

serial communication configuration

ISAMOD ("/tyCo/1", 19200)

System clock rate

CLKRATE = sysClkRateGet ()

slave registration

isa_register_slave (1)

communication registration

isa_register_com (1)

isa_register_com (2)

File storage unit (could be skipped because default set)

TSK_FUNIT = ""

End of cycle control (could be skipped because default set)

TSK_NBTCKSCHEM = 0

ISaGRAF kernel spawning

sp (isa_main, 1, "")

Communication task, slaves link

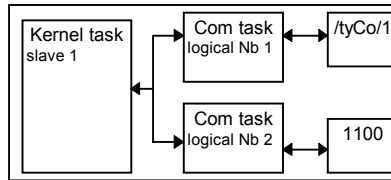
SlavesLink = 0x01000000

ISaGRAF communication tasks spawning

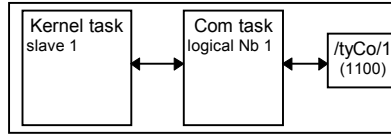
sp (tst_main_ex, "/tyCo/1", &SlavesLink, 1)

sp (tst_main_ex, "1100", &SlavesLink, 2)

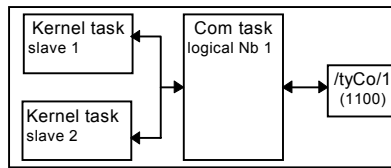
This startup correspond to the following figure



You also have the choice of the following basic configurations.



The most basic configuration consists in a kernel task associated to a communication task on a serial (Ethernet) link.

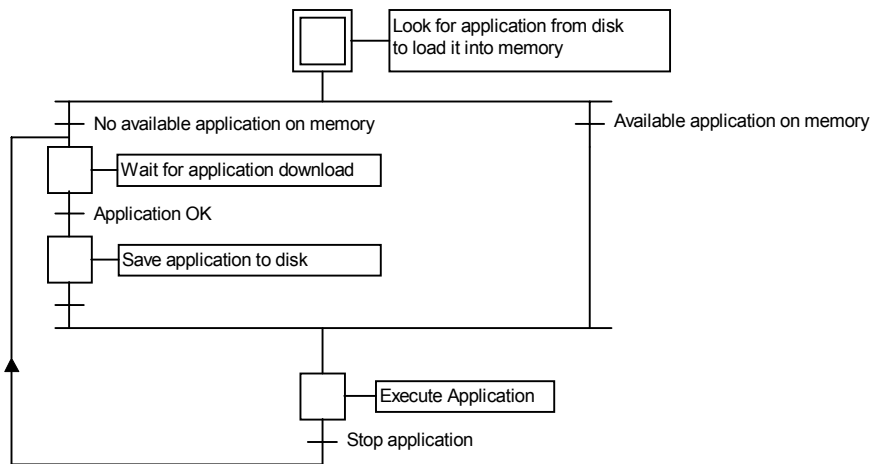


Another configuration consists in 2 kernel associated to a communication task on a serial (Ethernet) link. In this case, SlavesLink = 0x01020000.

C.5.5 Specific features

⇒ **ISaGRAF start up**

When the target is started, the following algorithm is executed.



- **Definitions**

The application code is the binary data base generated and downloaded by the workbench and then, executed by the target. It may be completed by the symbol table.

The application symbol table is an ASCII data base generated and downloaded by the workbench. This table makes the link between symbol objects and internal target objects. It is not required on target except for user's specific symbols management. For more information on symbol table see user's guide: Advanced programming techniques.

The path to the disk file unit is specified at the ISaGRAF target startup using the global variable TSK_FUNIT (default value = "" to specify there are no disk file unit)

- **ISaGRAF Multi-applications**

Different applications (kernels and communication tasks) can run at the same time on a CPU, as far as they have respectively different slave numbers and different communication task logical numbers. Nevertheless while running different application, the user must take care of some application objects shared access such as I/O boards. For instance different application (kernels) may use distinct physical boards unless some kind of I/O server or semaphore is implemented through the I/O driver.

- **Application backup**

When a new application is downloaded from the workbench debugger into the target, the application code is saved (the target uses standard file management routines fopen,...) with the file name:

*path***ISAx1** ISaGRAF application code backup file (where x is the slave number)

Furthermore if the application symbol table has been downloaded before, it is also saved on the target current directory with the file name:

*path***ISAx6** ISaGRAF application symbols backup file (where x is the slave number)

The *path* is specified at the ISaGRAF target startup using the global variable TSK_FUNIT. An empty string ("") will specify there are no disk file unit (default value).

When the ISaGRAF target is started, these application code and application symbols files are searched on the current directory and loaded into memory.

Then, if no symbols table is available on memory, the target starts running the application code, with no symbols loaded.

If no application code is available on memory, then the target is waiting for an application to be downloaded

In order to start the target with a specific application at power up, without using the debugger link:

- A first way may consists in directly copying these files to the application backup storage unit from the PC host where the workbench is installed, using any file transfer tool. You can use the workbench "Tools" menu (see user's guide: Managing programs) to ease these manipulations.
- A second way may consists in storing the application code (and if necessary the application symbol table) in a non volatile memory (like a PROM or EPROM), from files from the PC host where the workbench is installed, with your own tools.

Then at system power up, if required (for example because of faster access or breakpoint management), you may load the application code (and if necessary the application symbol table) from the PROM to the RAM, with your own tools.

Then at ISaGRAF startup (just before tasks spawning) you must specify the address(es) where the application code (and if necessary the application symbol table) is located in memory. In that way you need to initialize the SSR global variable as following:

```
SSR[x][1].space = address location of application code
```

And if necessary:

```
SSR[x][6].space = address location of application symbol table
```

In that way you may write a short procedure. The SSR global variable is declared as an `str_ssr` structure type which is defined in `tasy0ssr.h` file.

WARNING:

The breakpoint management of the ISaGRAF debugger cannot run correctly if the application code is not accessible for writing. This is not a problem, as your application has normally been fully tested before.

On the PC host, if the ISaGRAF workbench is installed on the standard \ISAWIN directory: the application code file of the project MYPROJ is:

```
\\ISAWIN\APL\MYPROJ\appli.x6m (corresponding to isax1 on the target).
```

the application symbols file of the project MYPROJ is:

```
\\ISAWIN\APL\MYPROJ\appli.tst (corresponding to isax6 on the target).
```

▣ Error management and output messages

The ISaGRAF target software integrates an error detection management. You will find the warning error list and their description in appendix.

Error detection is processed as follows:

- An error is composed of an error and argument number sent to the ISaGRAF error routine

- If the error detection flag is set in the workbench Make options, the error is processed. If not, the information is lost and the error management ends.

When processed:

- Error number (decimal value) and argument (hexadecimal value) are displayed on the default stdout output
- Error number and argument are pushed into a ring FIFO error buffer in order to be retrieved at a later time. The error buffer size is set in the workbench Make options. When the buffer is full, at each new incoming error, the oldest one is lost.
- Errors can be pulled either from the debugger or from the running application using the SYSTEM call (see user's guide).

When the debugger detects an error, a message describing the error is displayed in the error window. Depending on the context of the application (running or not) the debugger may display the name of the object (variable or program) where the error comes from, or the argument error (decimal value) into brackets [x] which has a different meaning for each error.

On the target, when an error is detected, error values are displayed on the default stdout output. Thus the display can be directed using VxWorks routines such as

`ioGlobalStdSet()`

or `ioTaskStdSet()`

In last case, not that either the kernel or the communication tasks can generate errors

▬ **Cycle duration, task behaviors, and task priorities**

- At the end of an ISaGRAF cycle, just before starting a new one, the following algorithm is performed:

If a cycle timing is specified (from the workbench: see user's guide: Managing programs) then the CPU is relinquished for the remained time period (specified cycle time - current application one). If this remained time period is negative an overflow is generated and the CPU is relinquished for TSK_NBTCKSCHED (variable set at ISaGRAF startup) tick(s) to force the scheduling.

If no cycle timing is specified, or if the remained time is less than 1 tick or equal to zero, then the CPU is relinquished for TSK_NBTCKSCHED tick(s) to force the scheduling.

The target timing accuracy correspond to the VxWorks system tick one.

a specified cycle timing is commonly used to trig cycles or to yield the CPU to other tasks running on the VxWorks system.

- The communication task is in sleep status while there are no incoming data through the communication link. When needed, this task gets information on the running application through a question/answer protocol with the kernel task. The communication task asks for a question to the kernel. At the end of the cycle (to have a synchronous application image), the kernel gives the answer to the communication task.

The ISaGRAF tasks do no modify the priority they have been given. The user is free to adjust these priorities according to ISaGRAF task behaviors described above and its whole application requirements.

C.6 Getting started with ISaGRAF NT target

C.6.1 Running ISaGRAF

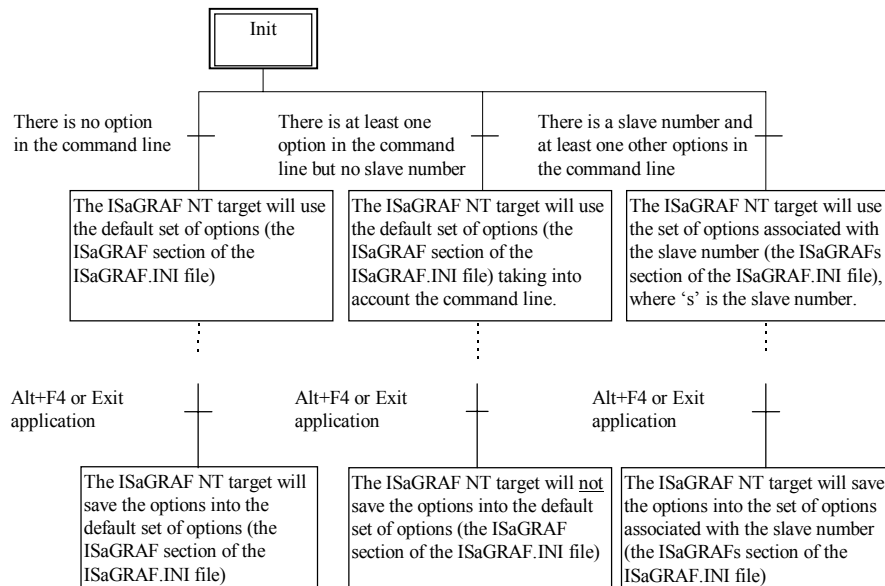
In the NT implementation, the target runs as a single program: WISAKER.EXE, which can be launched several times. This allows to have as many as ISaGRAF NT target you want as each instance has a separate slave number.

The target program does not prevent the running of interrupt driven routines.

The WISAKER software is designed to run under Windows NT 3.51 or later.

C.6.2 General information on options

Options are saved and retrieved according to the following diagram:



Note that ISAGRAF.INI file is saved in the current working directory.

▣ **Slave number: -s Option**

This option specifies the target slave number. It can be from 1 to 255 except number 13 (\$0D). This slave number is used through the communication link protocol. It is mainly designed to distinguish slaves from each other when more than one target are connected to the same host workbench or when more than one target run on the same PC. When using the workbench debugger, make sure the workbench slave setting (see user's guide: Managing programs) matches the target one.

Default value: The default slave number is 1 or the one in the ISaGRAF.INI file.

Example:

WISAKER.EXE -s=2

User interface: This window is display from the "Options/Slave" command of the main window of ISaGRAF NT target.



Using the mouse or the arrows (Up and Down) it is possible to change the value of this option. In order to use it, the ISaGRAF NT target should be restarted.

▣ **Communication link and configuration: -t Option**

The ISaGRAF target can use a serial link or an Ethernet link for debugger communication. The name of the port is specified with the -t option. As the communication interface is designed to be compatible with any machine, ports COM1, COM2, COM3 or COM4 can be used for serial communication, and port numbers starting from 1100 can be used for Ethernet communication.

Default value: The default communication port is the 1100 for Ethernet and COM1 for serial communication or the one in the ISaGRAF.INI file.

TO BE NOTED: The default communication link is the Ethernet.

Examples:

WISAKER -t=COM2

WISAKER -t=1101

Serial configuration:

Some options can only be used if the -t=COMx option is specified.

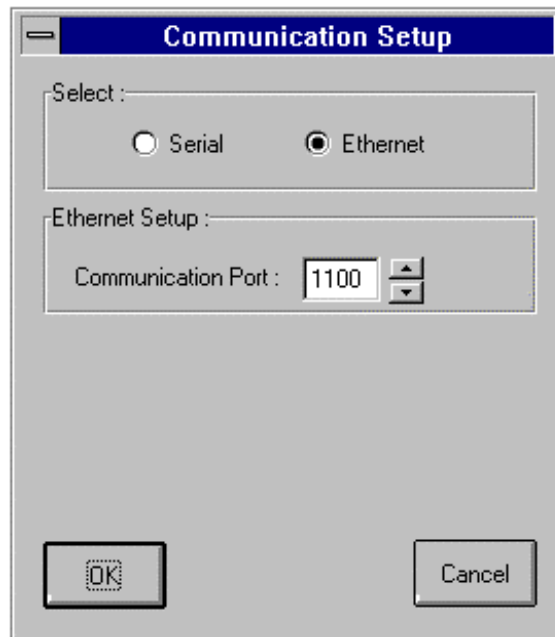
Those are configuration options for the serial link:

Option	Values	Meaning
baud	600	Baud rate
	1200	
	2400	
	4800	
	9600	
	19200	
parity	n	No parity
	e	Even
	o	Odd
data	7 or 8	Number of bits
stop	1 or 2	Length of the stop bit
flow	h	Hardware control
	n	No control

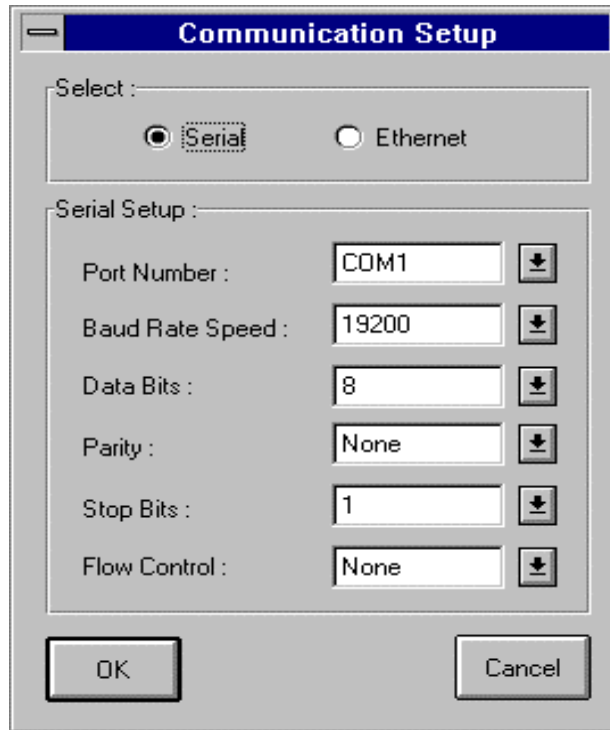
The defaults values are 19200, no parity, 8 data bits, 1 stop, no flow control

Example:
WISAKER -t=COM1 baud=1200 data=8 parity=n stop=2

User interface: This window is displayed from the "Options/Communication" command of the main window of ISaGRAF NT target.



It is possible to choose the serial communication or the Ethernet communication. The Ethernet communication gives the possibility to modify the port number. This port number should be the same as in the workbench PC-PLC Link specification.



By choosing the serial communication, the configuration will appear. This configuration should be the same as in the workbench PC-PLC Link specification.

☰ **Graphic simulation of virtual boards: -x Option**

If this option is set, the boards declared virtual, in the I/O connection editor (See Part A), will be simulated.

Possible values are 0 or 1, 0 means no simulation, and 1 means simulation on.

Default value: The default value is 0 or the one in the ISaGRAF.INI file.

Example:

WISAKER -x=1 will simulate virtual boards,

User interface: The menu item will be checked or unchecked reflecting the state of the option. Simulated boards appear in a graphic panel.

☰ **Priority of the ISaGRAF NT target: -p Option**

As the target is running under NT, it is very useful to specify a priority level. It is, for instance, possible to have a time critical ISaGRAF application running within a target with the higher priority and one or more targets running in background with lower priorities.

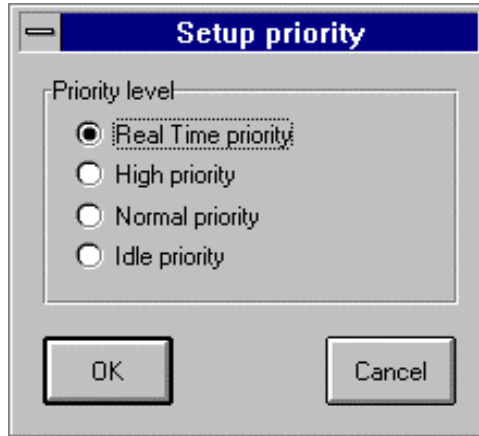
Possible values are 0, 1, 2 or 3. 0 is the highest priority, and 3 is the lowest priority.

Examples:

WISAKER -p=0

WISAKER -p=1

User interface: This window is displayed from the "Options/Priority" command of the main window of ISaGRAF NT target.



The highest priority is the real time and the lowest is the idle priority.

0: Real time priority

1: High priority

2: Normal priority

3: Idle priority

⇒ **Examples:**

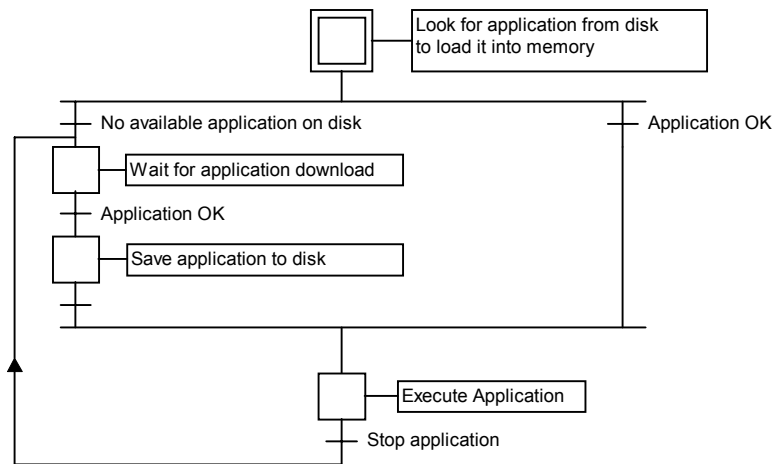
wisaker -t=COM1 Starts the ISaGRAF target with default slave number (1) and with COM1 as the communication port.

wisaker -s=3 -t=COM1 Starts the ISaGRAF target with slave number 3 and with COM1 as the communication port.

C.6.3 Specific features

⇒ **ISaGRAF start up**

When the target is started, the following algorithm is executed.



- **Definitions**

The application code is the binary data base generated and downloaded by the workbench and then, executed by the target. It may be completed by the symbol table.

The application symbol table is an ASCII data base generated and downloaded by the workbench. This table makes the link between symbol objects and internal target objects. It is NOT required on target except for user's specific symbols management as for instance the DDE feature or I/Os simulation with symbol names feature. For more information on symbol table see user's guide: Advanced programming techniques.

- **ISaGRAF Multi-applications**

Different applications can run at the same time on a CPU, as far as they have respectively different slave numbers and different communication task logical numbers. Nevertheless while running different application, the user must take care of some application objects shared access such as I/O boards. For instance different application may use distinct physical boards unless some kind of I/O server or semaphore is implemented through the I/O driver.

- **Application backup**

When a new application is downloaded from the workbench debugger into the target, the application code is saved on the target current directory with the file name:

ISAx1 ISaGRAF application code backup file (where x is the slave number)

Furthermore if the application symbol table has been downloaded before, it is also saved on the target current directory with the file name:

ISAx6 ISaGRAF application symbols backup file (where x is the slave number)

When the ISaGRAF target is started, these application code and application symbols files are searched on the current directory and loaded into memory.

If no symbols file is available, then the target starts running the application code, with no symbols loaded.

If no application code is available, then the target is waiting for an application to be downloaded.

In order to start the target with a specific application at power up, without using the debugger link, these files can be directly copied to the target current directory disk from the same disk if the workbench is on the same PC, or using a floppy disk.

If the ISaGRAF workbench is installed on the standard \ISAWIN directory:

the application code file of the project MYPROJ is:

\ISAWIN\APL\MYPROJ\appli.x8m

the application symbols file of the project MYPROJ is:

\ISAWIN\APL\MYPROJ\appli.tst

Example:

From the directory where WISAKER.EXE is installed, if the following command is entered:

copy \ISAWIN\APL\MYPROJ\appli.x8m isa11

Then WISAKER.EXE will find and execute 'myproj' application.

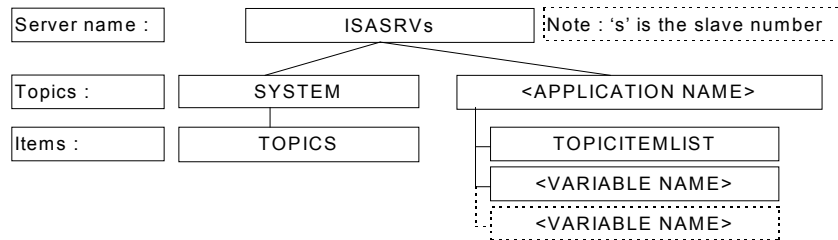
All these commands can be grouped for instance into a batch file and then started from the workbench tool menu (see user's guide: Managing programs)

DDE specification

The ISaGRAF NT target is a DDE server (Dynamic Data Exchange). Any software that can be a client, can be connected with the target to exchange variables. For example, MSEXCEL can animate graphics with values coming from ISaGRAF target via DDE.

The DDE feature requires on the target the application symbols table.

DDE subjects are defined as follows:



- « ISASRVs » is the name of the DDE server, 's' is the slave number.
- « SYSTEM » is a standard topic which gives access to the « TOPICS » item,
- « TOPICS » gives the list of the topics currently defined: system and the name of the application which is running into the ISaGRAF NT target.
- « APPLICATION NAME » is the name of the application.
- « TOPICITEMLIST » is the list of items available under the current topic, this gives the list of the variables which can be accessed via DDE.
- « VARIABLE NAME » is the name of a variable.

DDE advise loop rate for ISaGRAF NT target: -d Option

The DDE client generally polls the variables each time it needs them. This can take a large amount of time if there are a lot of variables. There is another mode which is called advise

mode (advise loop), in which the server itself will only send modified variables. So that communications are minimized and efficient. In this mode the server periodically looks at the variables marked as advised variables to know which should be sent. This period is called the DDE advise loop rate.

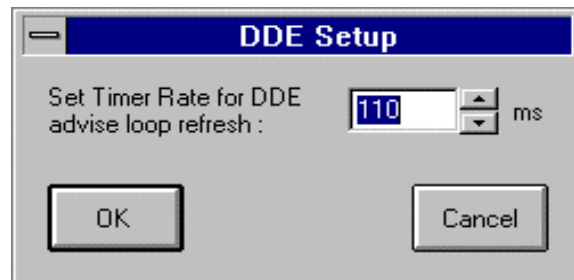
With this option, it is possible to specify the rate (in ms) for the DDE advise loop.

Default value: The default value is 1000 ms or the one in the ISaGRAF.INI file

Example:

WISAKER -d=100

User interface: This window is displayed from the "Options/DDE" command of the main window of ISaGRAF NT target.



Error management and output messages

The ISaGRAF target software integrates an error detection management. You will find the warning error list and their description in appendix.

Error detection is processed as follows:

- An error is composed of an error and argument number sent to the ISaGRAF error routine
- If the error detection flag is set in the workbench Make options, the error is processed. If not, the information is lost and the error management ends.

When processed:

- Error number (decimal value) and argument (hexadecimal value) are displayed on the output (window of the WISAKER.EXE)
- Error number and argument are pushed into a ring FIFO error buffer in order to be retrieved at a later time. The error buffer size is set in the workbench Make options. When the buffer is full, at each new incoming error, the oldest one is lost.
- Errors can be pulled either from the debugger or from the running application using the SYSTEM call (see user's guide).

When the debugger detects an error, a message describing the error is displayed in the error window. Depending on the context of the application (running or not) the debugger may display the name of the object (variable or program) where the error comes from, or the argument error (decimal value) into brackets [x] which has a different meaning for each error.

A welcome message is displayed on the output when the target starts. It is composed of the slave number, the communication configuration and the DDE server name.

≡ **System clock**

As the ISaGRAF NT target is designed to run on any system, the time reference used for both cycle synchronization and timer variables refresh is the standard tick which is 10 milliseconds.

Thus, it is not possible to have an accuracy on timer variables better than 10ms. For the same reason, a specified cycle duration less or equal to 10 ms and different from zero will generate an cycle duration overflow error (error 62). See the following chapter for more information.

Ask your supplier for a special implementation, if your application requires more accuracy.

≡ **Cycle duration and target behavior**

At the end of an ISaGRAF cycle, just before starting a new one, the following algorithm is performed:

If a cycle timing is specified (from the workbench: see user's guide: Managing programs) then the CPU is relinquished for the remained time period (specified cycle time - current application one). If this remained time period is negative an overflow is generated and the CPU is relinquished for 1 tick to force the scheduling

If no cycle timing is specified, or if the remained time is less or equal than 1 tick or equal to zero, then the CPU is relinquished for 1 tick to force the scheduling

The target timing accuracy correspond to the Windows NT system tick one.

A specified cycle timing is commonly used to trig cycles or to yield the CPU to other processes running on the Windows NT system.

≡ **Exit key**

While testing an application in non-industrial conditions on a desktop PC, the user may wish to stop ISaGRAF: this is done by pressing a combination of keys to prevent unexpected stops. This key sequence is:

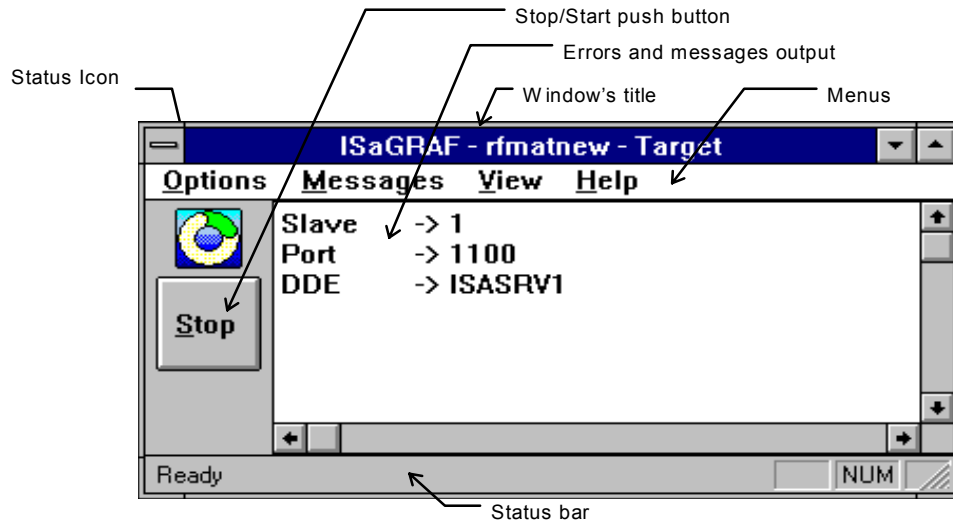
alt + F4

One dangerous side effect of this fast exit, is that the IO board interface is not closed. Thus the clean way for stopping your ISaGRAF target is:

- stop the application from the debugger or from the Stop/Start push button (this will close the IO boards)
- stop ISaGRAF target from the system menu.

C.6.4 User interface

This is the user interface of the ISaGRAF NT target:



There are the main items:

- a window title
- a menu bar
- a running status icon
- a Start/Stop push button
- an errors and messages output
- and a status bar.

The window title contains « ISaGRAF - name_of_appli - target », where name_of_appli is the name of the running application. It contains only « ISaGRAF - - Target » when there is no running application.

☰ **ISaGRAF NT target menu bar:**

The menu bar has four menus:

- Options
- Messages
- View
- Help

- **"Options" menu**

(see also first section on NT:General information on options)

The **"Options"** menu gives access to the running options. It proposes the following options:

Slave gives access to the slave number modification. The modified option will be activated only after next start of the target. This feature isn't available if the target has been started with at least one option in the command line.

Communication gives access to the communication configuration. The modified option will be activated only after next start of the target. This feature isn't available if the target has been started with at least one option different from -s option.

DDE gives access to the DDE advise loop rate modification. The modified option will be activated only after next start of the target. This option isn't available if the target has been started with at least one option different from -s option.

Simulate I/O is checked or unchecked reflecting the state of the option. The modified option will be activated only after next Stop/Start of the application.

Priority gives access to the priority modification. The modified option is activated immediately.

Default Options retrieves the default running options for the following only:

- Communication
- DDE
- coordinates of the window on the screen

The modified options will be activated only after next start of the target. This feature isn't available if the target has been started with at least one option different from -s option.

- **"Messages" menu**

The **"Messages"** menu is the management of the output. It contains the two following commands:

Acknowledge stops the red blink in case of errors or messages.





Clear totally erases the output.

☰ ***ISaGRAF NT target icon:***

The icon reflects the states of the target:

- application is running, then the icon turns
- no application (or application stopped), then the icon is stopped
- errors or messages are present in the output window. The center of the icon blinks red. To stop the blink, it possible to choose the « Acknowledge » item of the « Messages » menu or the « Clear » item of the same menu (beware that this item will completely erase the output window). You will find more information on errors, in the error management and output messages chapter.

The different states are sum up in the following table:

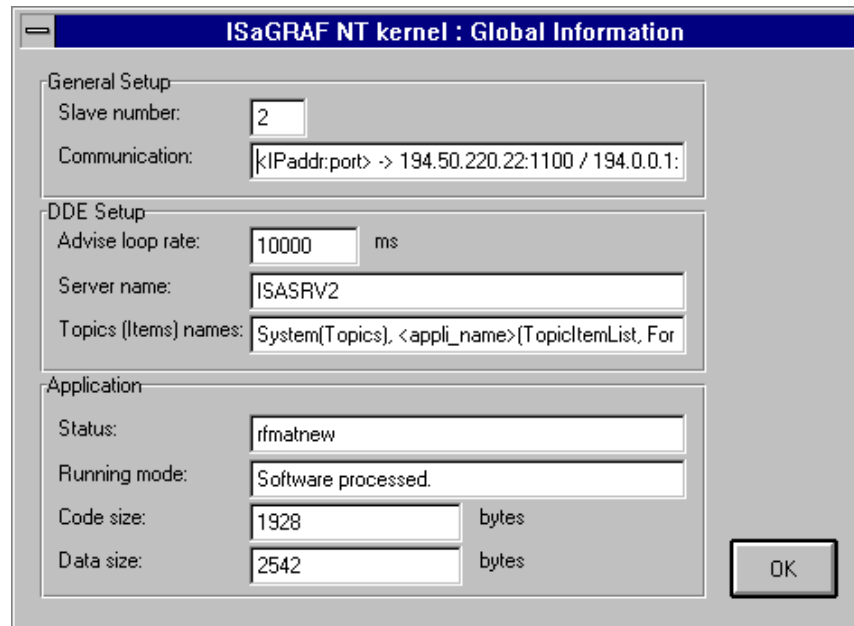
	no error	errors or messages (the center is red)
running application		
no application		

⊞ **ISaGRAF NT target Start/Stop push button:**

The Start/Stop push button is strictly identical to the start/stop function of the debugger. The text in the push button will reflect the running state of the application. If the application is running, the text will be « Stop », if the application is stopped (or if there is no application), the text will be « Start » (please note that if there is no application, and the start action is requested, the push button will toggle into the Stop mode and then it will come back to the Start mode).

⊞ **ISaGRAF NT target, general information**

With the "View / Information" command, the following dialog box gives general information on the target configuration and on the running application:



ISaGRAF NT kernel : Global Information

General Setup

Slave number:

Communication:

DDE Setup

Advise loop rate: ms

Server name:

Topics (Items) names:

Application

Status:

Running mode:

Code size: bytes

Data size: bytes

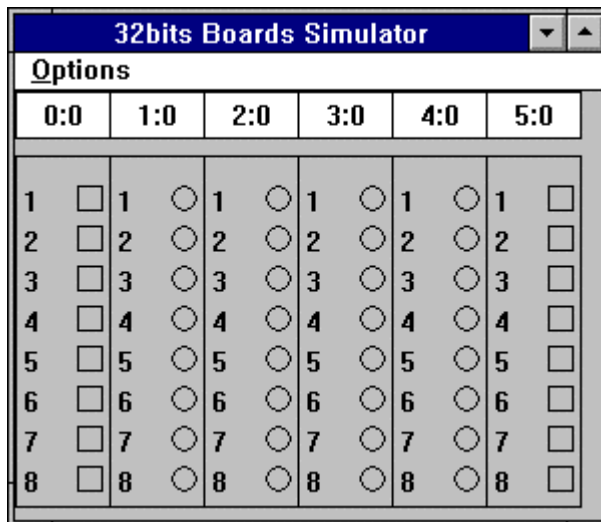
OK

There are three topics:

- a) General setup:
 - The slave number
 - the communication configuration (If the communication link is the Ethernet one, in addition to the port number, the list of available IP address on the current NT system is displayed)
- b) DDE setup
 - the advise loop rate
 - the DDE server name
 - the DDE topics and items name. This is a general information, this doesn't reflect the real values. In fact the fields between < > should be replaced by the real values.
- c) Application
 - The application status which is its name when there is a running application, and is the string 'No application' when there is no running application
 - The running mode of application, which indicates if the application is running through the software processor. It contains in this case the string: « Software processed ». Or if the application had been compiled with a C compiler. It contains in this case the string: « C compiled ». If there is no running application, it contains the string: « No application ».
 - The code size in bytes. If the running mode is « C compiled », this field is zero.
 - The data size in bytes. This is the sum of the runtime internal data and the variables database.

☰ **ISaGRAF NT target simulation of virtual boards:**

When the option « Simulate I/O » is selected, at the next application start the following window will appear:



Depending on your I/O connection configuration, there will be more or less boards (and different) and more or less variables (and different). The numbers « s:b » at the top of each board represent the slot identifier (s) and the board identifier (b). The count starts at zero, and it isn't possible to modify it.

The '32bits Boards Simulator' window works with the Start/Stop application state. So if there is a running application which has virtual boards (or uses simulator boards) and the « Simulate I/O » flag is checked, this window will appear. On the contrary as soon as the Stop push button is depressed, it will be closed. This window works along with the I/O calls.

The "Options" menu proposes two items:

Variable names will show the names of the variables if and only if the symbols table has been downloaded prior to the tic code.

Hexadecimal values will show each integer in hexadecimal format instead of default decimal format

The variable names will look as follows:

32bits Boards Simulator					
Options					
0:0	1:0	2:0	3:0	4:0	5:0
1 <input type="checkbox"/> Row0	1 <input type="radio"/> LED00	1 <input type="radio"/> LED10	1 <input type="radio"/> LED20	1 <input type="radio"/> LED30	1 <input type="checkbox"/> COL0
2 <input type="checkbox"/> Row1	2 <input type="radio"/> LED01	2 <input type="radio"/> LED11	2 <input type="radio"/> LED21	2 <input type="radio"/> LED31	2 <input type="checkbox"/> COL1
3 <input type="checkbox"/> Row2	3 <input type="radio"/> LED02	3 <input type="radio"/> LED12	3 <input type="radio"/> LED22	3 <input type="radio"/> LED32	3 <input type="checkbox"/> COL2
4 <input type="checkbox"/> Row3	4 <input type="radio"/> LED03	4 <input type="radio"/> LED13	4 <input type="radio"/> LED23	4 <input type="radio"/> LED33	4 <input type="checkbox"/> COL3
5 <input type="checkbox"/>	5 <input type="radio"/>	5 <input type="radio"/>	5 <input type="radio"/>	5 <input type="radio"/>	5 <input type="checkbox"/>
6 <input type="checkbox"/>	6 <input type="radio"/>	6 <input type="radio"/>	6 <input type="radio"/>	6 <input type="radio"/>	6 <input type="checkbox"/>
7 <input type="checkbox"/>	7 <input type="radio"/>	7 <input type="radio"/>	7 <input type="radio"/>	7 <input type="radio"/>	7 <input type="checkbox"/>
8 <input type="checkbox"/>	8 <input type="radio"/>	8 <input type="radio"/>	8 <input type="radio"/>	8 <input type="radio"/>	8 <input type="checkbox"/>

C.7 "C" programming

C.7.1 Overview

This manual is aimed at the user already having experience in ISaGRAF concepts and Workbench tools. After developing pure automation applications using **conversion functions**, **"C" functions** and **function blocks** from the CJ International standard libraries, it is possible to develop "user defined" conversion functions, "C" functions and function blocks. This allows the user to enhance the ISaGRAF target PLC by creating new libraries, and to get the maximum out of the workstation flexibility and hardware platform.

With a "C" development system, and with some previous experience in "C" programming, this manual will enable the user to customize his ISaGRAF target PLC for the best possible control. Such developments improve the target PLC performance as well as the comfort and quality of development with the ISaGRAF Workbench for the automation programmer.

Information contained in this document is not dedicated to one special target system. Some features, however, (such as multitasking capabilities) cannot be applied to some monotasking systems.

▬ **Standard ISaGRAF workbench features**

The ISaGRAF Workbench offers many functions to manage the "C" component libraries on the automation development side. For the automation programming, a "C" conversion, function or function block is a "**black box**", completely defined by its interface.

The ISaGRAF Library Manager is used to add components to the existing libraries and define the interface between the "C" implementation and the use of these components in the **ST/FBD** programming. The ISaGRAF Library Manager also provides an automatic generation of the frame of the "C" source code for conversions, functions and function blocks, and includes tools for editing such "C" source files. Refer to the **ISaGRAF User's Guide** for further information about the functions of the Library Manager.

▬ **"C" language development**

The ISaGRAF Workbench does not include any "C" compiler or cross compiler tool. The user must own a "C" compiler, dedicated to the ISaGRAF target system, to integrate his "C" components to the ISaGRAF kernel.

When using a cross compiler, the ISaGRAF Workbench offers the user entry points for running a user defined MS-DOS command file (.bat), in a DOS window. The cross compiler used must run in a DOS emulation window. If not, Windows must be closed before running the compilers and linkers in a pure MS-DOS context.

▬ **Technical notes**

The ISaGRAF Library Manager allows the user to write a text description for each of the library components. This **technical note** is the user's guide of the "C" component developed,

and is for the benefit of the automation programmer, to describe the corresponding conversions, functions or function blocks in ISaGRAF applications.

The conversion, "C" function or function block must be precisely defined in the technical note, so that the automation programmer can really use it as a packaged ISaGRAF function. For a "C" function, the technical note must describe:

- the detailed function processed by the function
- the complete description of its calling parameters
- the meaning of its return value
- the detailed typing of its calling parameters and return value
- the application context

For a "C" function block, the technical note must describe:

- the detailed function processed by the block activation function
- the complete description of its calling parameters
- the meaning of its return parameters
- the detailed typing of its calling and return parameters
- the application context

For a conversion function, the technical note must describe:

- the exact meaning of the conversion when used with an input variable
- the exact meaning of the conversion when used with an output variable
- the limits of the values the conversion can process

Technical notes may also contain information about:

- the complete identification of the conversion, function or function block
- any information about its maintenance and updates
- the supported target system
- the special multitasking features
- the required system services, memory, drivers...

C.7.2 "C" Conversion functions

The ISaGRAF Workbench includes a **linear conversion** utility to carry out simple analog I/O conversion at run time on the ISaGRAF target PLC. This utility does not require any "C" development, as it is limited to strictly increasing or decreasing continuous functions. Refer to the ISaGRAF User's Guide for a complete description of these tools.

Conversion functions enable the user to apply any complex conversion, with specific operations described in the "C" language. Basically, a conversion function is defined for **both inputs and outputs**. Even if one direction is not used, implementation and tests have to be made before integrating the conversion to the ISaGRAF kernel, to prevent any system crash due to a wrong call.

Conversion functions are written in "C" language, compiled and linked with the ISaGRAF kernel. The increased kernel must be installed on the ISaGRAF target PLC before using new conversion functions in ISaGRAF projects. New conversion functions cannot be integrated in

the ISaGRAF Simulator. The ISaGRAF applications have to be simulated **before** inserting the non standard conversion functions.

The "C" source code of the standard conversions written by CJ International are installed with the ISaGRAF Workbench. They can be used as examples for creating new functions. It is recommended **not to modify** the standard functions so they can be used in any ISaGRAF application. The standard conversions delivered with the ISaGRAF Workbench are supported by the ISaGRAF simulator.

Warning: Conversion functions are **synchronous** operations, activated at run time by the ISaGRAF I/O manager, during the input or output phases of the application cycle. Time spent for the execution of a conversion function is included in the ISaGRAF application **cycle timing**. The user has to ensure that no "wait operation" is programmed in a conversion function, so that the ISaGRAF cycle processing is not unnecessarily extended.

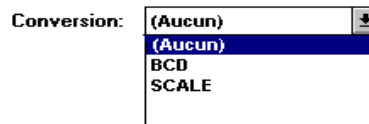
≡ **Adding a function to the ISaGRAF library**

The ISaGRAF Library Manager must be used to add a new conversion function to the ISaGRAF library, on the Workbench side. The "**New**" command of the "**Files**" menu is used, when the conversion function library is selected. No parameter has to be defined on the Workbench, because conversion functions use a standard predefined interface.

When a new conversion function has been created, its **technical note** must be written. The frame of the "C" source code for the new conversion function is automatically generated by the ISaGRAF Library Manager.

≡ **Using a conversion in an ISaGRAF project**

Defined conversion functions can be used to filter values of any input or output analog variable of the selected project. To attach a conversion function to a variable, the variables declaration editor is run, an input or output analog variable selected and then its parameters edited. The "**conversion**" field of the analog declaration dialog box is used to setup the conversion function attached to an analog I/O variable:



Both conversion functions and tables appear in the list. This implies that the same name cannot be used for a function and a table. A variable cannot be attached to a conversion function that has yet to be defined or integrated into the ISaGRAF kernel.

≡ **Standard "C" interface**

The interface of a conversion function always has the same format. Calling and return parameters are passed through a structure. This structure is defined in the "**TACN0DEF.h**" file:

```
/*  
Name: tacn0def.h
```

```

Target conversions definition file
*/

#define DIR_INPUT 0          /* direction = input conversion */
#define DIR_OUTPUT 1        /* direction = output conversion */

typedef int32 T_ANA;        /* integer ANA type */
typedef float T_REAL;      /* real ANA type */

typedef struct {            /* conversion structure */
    uint16 number;          /* conversion number (reserved) */
    uint16 direction;      /* conversion direction */
    T_REAL *before;        /* value before conversion */
    T_REAL *after;         /* value after conversion */
} str_cnv;

#define ARG_BEFORE (*(arg->before))
#define ARG_AFTER (*(arg->after))
#define DIRECTION (arg->direction)

/* eof */

```

The "**str_cnv**" structure completely describes the interface. The only parameter of a "C" conversion function is a pointer to such a structure. The "**number**" field is the logical number of the conversion function (location in the ISaGRAF library) and does not have to be used in the programming.

The "**direction**" field indicates whether the conversion must be applied on an input variable or an output variable. It contains the **DIR_INPUT** value for an input conversion, or the **DIR_OUTPUT** value for an output conversion.

The "**before**" field points to the value before the conversion. This field has a different meaning for an input or an output conversion. It represents the electrical value (read on the input device) for an input conversion, when the **direction** field takes the **DIR_INPUT** value. It represents the physical value (used in the programmed equations) for an output conversion, when the **direction** field takes the **DIR_OUTPUT** value.

The "**after**" field points to the value after the conversion. This field has a different meaning for an input or an output conversion. It represents the physical value (used in the programmed equations) for an input conversion, when the **direction** field takes the **DIR_INPUT** value. It represents the electrical value (sent to the output device) for an output conversion, when the **direction** field takes the **DIR_OUTPUT** value.

The programmer can use the "**ARG_BEFORE**" and "**ARG_AFTER**" definitions to directly access the **before** and **after** field of the structure passed to the "C" conversion function.

Processed values are **single precision floating values**. The result is converted to a long integer when the conversion is applied to an integer analog variable. This means that the same conversion can be used for both real or integer analog I/O variables.

Source code

Because the conversion function can be used for both input and output analog variables, the "C" source code of the function is divided into two main parts: the input conversion, and the output conversion. The **direction** field of the interface structure is used to select the conversion to be applied. The ISaGRAF Library Manager automatically generates the complete frame of the function, when the conversion function has been created. This includes the main selecting "IF" structure. Below is the standard frame of a conversion function:

```
/*
conversion function
name: sample
*/

#include <tasy0def.h>
#include <tacn0def.h>

void CNV_sample (str_cnv *arg)
{
    if (DIRECTION == DIR_INPUT) { /*INPUT CONV*/
        }
    else { /*OUTPUT CONV*/
        }
    }
}

/* The following function shows the link with the ISaGRAF I/O manager, using the
name of the conversion. This function is completely generated by the ISaGRAF
Library Manager. */

UFP cnvdef_sample (char *name)
{
    sys_strepy (name, "SAMPLE"); /* gives the name of the conversion */
    return (CNV_sample); /* returns the implementation function */
}
```

The best way to complete the specific part of the function is to write two separate local functions for input conversion and output conversion. These functions will be called by the main algorithm, as shown in comments in the previous example, in the main **IF** structure.

The "**TASY0DEF.H**" include file from the ISaGRAF kernel is required for system dependent definitions. It also contains the definition of the **UFP** type, which represents a pointer to a void function, and is used for the declaration function.

▬ **Links between projects and "C" implementation**

The logical link between the implementation of a conversion function and the use of the conversion in an ISaGRAF project is made with the name of the conversion. A "declaration" function is added to the "C" source code of the conversion function. This function is called only one time when the application starts, and indicates to the ISaGRAF I/O manager the conversion name which corresponds to the function to be implemented. This is the standard format of such a declaration function:

```
UFP cnvdef_XXX (char *name)
{
    strcpy (name, "XXX");    /* gives the name of the conversion */
    return (CNV_XXX);      /* returns the implementation function */
}
/* (XXX is the name of the conversion) */
```

The name of the function, used for the **strcpy** statement must be written in **uppercase**. It must be written in lowercase in the name of the conversion implementation function and in the name of declaration function.

Using the "**CNV_**" and "**cnvdef_**" prefixes for implementation function and definition function enables the user to name a conversion with a reserved keyword of the "C" language, or the name of an existing function of the "C" ISaGRAF libraries.

Other statements can be added to the declaration function to realize any specific initialization operation relative to this conversion. The ISaGRAF system ensures the user that this function is called **only one time** when the application starts.

The declaration function is called for any integrated conversion function, even if it is not used in the ISaGRAF application. The ISaGRAF kernel fails in a fatal error if a conversion used in the application is not integrated to the kernel.

Before linking new functions with the kernel, the user must write another "C" source file, named "**GRCN0LIB.C**", and insert it (with the retained conversion functions) in the list of files for the linker. The "**GRCN0LIB.C**" only contains an array of declaration functions. This array is read during application initializations, to make a dynamic link with the conversion functions written in "C". This is an example of such a file:

```
/* File "GRCN0LIB.c" - Example with conversions of standard library */

#include <tasy0def.h>          /* required for types definition */

extern UFP cnvdef_scale (char *name);    /* decl. function for SCALE conv */
extern UFP cnvdef_bcd (char *name);     /* decl. function for BCD conv */
```

```
UFP_LIST CNVDEF[ ] = {          /* array of declaration functions for */
    /* integrated conversion functions */
    cnvdef_scale,
    cnvdef_bcd,

    NULL };

/* end of file */
```

The **CNVDEF** array must be terminated by a NULL pointer. Some clashes may occur when this condition is not met. Unresolved references will occur when linking the new ISaGRAF kernel if the **CNVDEF** array is not defined.

By writing this file, a new kernel can be built, including all the existing conversions. A kernel can also be built customized for one project, by inserting in the **CNVDEF** array only the conversions used in the project. The "**GRCNOLIB.C**" file is automatically generated by the ISaGRAF Code Generator, when the code of an application is built. The file is placed in the ISaGRAF project directory, and groups only the conversions used in the project.

▬ **Limits**

The ISaGRAF library may contain up to **128** conversion functions. Any type of operation may be processed in a conversion function. It should be noted that the functions are called in the ISaGRAF cycle in a **synchronous** way, so that the execution of the function has direct effect on the cycle timing.

C.7.3 "C" Functions

"C" functions are used to increase standard capabilities of **ST** and **FBD** languages. They can be used to realize any specific calculations, system calls, communications, or to install a set of services for dialog between an ISaGRAF application and other tasks. Functions are written in "C" language, compiled and linked with the ISaGRAF kernel. The increased kernel must be installed on the ISaGRAF target PLC before using new functions in ISaGRAF projects.

New functions cannot be integrated in the ISaGRAF Simulator. The ISaGRAF applications have to be simulated **before** using the non standard functions.

Warning: Functions are **synchronous** operations, activated at run time by the ISaGRAF kernel, during the application cycle. Time spent for the execution of a function is included in the ISaGRAF application **cycle timing**. The user has to ensure that no "wait operation" is programmed in a function, so that the ISaGRAF cycle processing is not unnecessarily extended.

▬ **Adding a function to the ISaGRAF library**

The ISaGRAF Library Manager must be used to add a new "C" function to the ISaGRAF library, on the Workbench side. The "**New**" command of the "**Files**" menu is used, when the

"C" functions library is selected. When a new function has been created, its **technical note** must be written. The frame of the "C" source code for the new function is automatically generated by the ISaGRAF Library Manager.

The **"Parameters"** command of the **"Edit"** menu is used to define the call and return parameters of the new function.

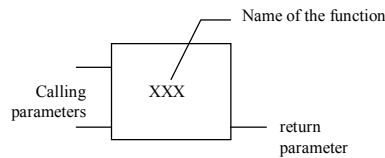
≡ **Using a "C" function in an ISaGRAF project**

Any integrated "C" function can be used as a standard function in the programs of an ISaGRAF project. "C" functions can be called from **ST** and **FBD** languages, and from special statements of the **SFC** language.

Calling a "C" function from the **ST** language follows the function call conventions of the language. The call parameters of the function are written after the name of the function, between parentheses, and separated by commas. The expression represents the value returned by the function. A "C" function call may be inserted into any assignment statement or complex expression. This is an example of a "C" function call in an assignment statement:

result := ProcName (par1, par2, ... parN);

An **FBD** program can call any "C" function. A function is used as a standard function box. Its call parameters are connected to the left hand side of the function box. The return parameter is connected to the right hand side of the box. Here is the standard aspect of such a function box:



A "C" function can be called from any **SFC** action block, or in any boolean condition attached to a transition.

≡ **Defining the interface of a "C" function**

The **"Parameters"** command of the **"Edit"** menu is used to define the call and return parameters of a new function. A function can have up to **31** call parameters, and always has **one** return parameter.

The list in the upper side of the window shows the parameters of the "C" function, according to the order of the function calling prototype: first the calling parameters, lastly the return parameter. The lower part of the window shows the detailed description of the parameter currently selected in the list:

- the name of the parameter
- the direction (call/return) of the parameter
- the type of the parameter

Any of the ISaGRAF data types may be used for a parameter: Boolean, Integer analog, Real analog, Timer or Message. Integer and real analogs must be distinguished.

Below is the correspondence between ISaGRAF types and "C" types:

BOOLEAN	unsigned long	unsigned 32 bit word: 1=true / 0=false
ANALOG	long	signed integer 32 bit word
REAL	float	single precision floating value
TIMER	unsigned long	unsigned integer 32 bit word (unit is 1 millisecond)
MESSAGE	char *	character string.

When a message value is passed onto a "C" function, it cannot contain null characters. The string passed to the "C" code is null-terminated. Do not forget that the return parameter must be the last one in the list. The rules shown below must be followed while naming parameters:

- the length of the name cannot exceed 16 characters
- the first character must be a letter
- the following characters must be letters, digits or underscore character
- naming is case insensitive

The same name cannot be used for more than one parameter of the function. A call parameter cannot have the same name as the return parameter. The same name **can** be used for parameters of different functions. The default name for return parameter is "**Q**". This name can be freely modified. The name of a parameter is used to identify the parameter in the "C" source code.

The "**Insert**" command is used to insert a new parameter before the selected parameter. The "**Delete**" command is used to erase the selected parameter. The "**Arrange**" command automatically rearranges (sorts) the parameters, so that the return parameter is put at the end of the list. Pressing the "**OK**" button stores the definition of the function interface and closes the dialog box. Pressing the "**Cancel**" button closes the dialog box, without changing the definition of the function interface.

≡ **Function "C" interface**

The interface of a function depends on the definition of its parameters. Calling and return parameters are passed through a structure. This structure is defined in the "**GRUS0nnn.H**" file, where "**nnn**" is the logical number of the function in the ISaGRAF library. This is an example of the "C" interface, for the "**SIN**" function (sines calculation):

```
/* File: GRUS0255.h - function "sample" */

typedef long          T_BOO;
typedef long          T_ANA;
typedef float         T_REAL;
typedef long          T_TMR;
typedef char          *T_MSG;

typedef struct {
    /* CALL */          T_REAL _param1;
    /* RETURN */       T_REAL _param2;
} str_arg;
```



```
#define PARAM1          (arg->_param1)
#define PARAM2          (arg->_param2)

/* end of file */
```

The relationship between ISaGRAF types and "C" types is shown below. The ISaGRAF types are defined as "C" types in the definition file of the function.

boolean	T_BOO	long (32 bits)
Integer analog	T_ANA	long
Real analog	T_REAL	float (32 bits - single precision)
timer	T_TMR	long
message	T_MSG	char * (32 bits - char pointer)

Each field of the "**str_arg**" structure corresponds to one parameter of the function. The return parameter is the last in the structure. The calling parameters appear in the structure with the same order than the one established for the function definition. An uppercase identifier is defined to directly have access to one parameter of the structure passed to the "C" implementation of the function. Names of the identifiers are the ones entered during the definition of the function with the ISaGRAF Library Manager.

The "C" definition file is updated each time the interface of the function is changed by using the ISaGRAF Library Manager. This ensures a complete match between the implementation of the function and its use in the programs of the ISaGRAF applications.

Source code

Below is the standard frame of a "C" function implementation:

```
/* Example of user function - Number is "255" - Name is "SAMPLE" */

#include "tasy0def.h"          /* ISaGRAF kernel common definitions */
#include "grus0255.h"         /* interface definition for function 255 */

void USP_sample (str_arg *arg)
{
    /* body of the function */
}

/* The following function is used for the initialization of the function and the
declaration of its implementation. It realizes the link with the ISaGRAF kernel,
using the name of the function. This function is completely generated by the
ISaGRAF Library Manager. */

UFP uspdf_sample (char *name)
{
```

```
        strcpy (name, "SAMPLE"); /* gives the name of the function */
        return (USP_sample);     /* returns the implementation function */
    }

/* end of file */
```

The "**TASY0DEF.H**" include file from the ISaGRAF kernel is required for system dependent definitions. It also contains the definition of the **UFP** type, which represents a pointer to a void function, and is used for the declaration function.

≡ Links between projects and "C" implementation

The logical link between the implementation of a "C" function and its use in the programs of an ISaGRAF project is made with the name of the function. A "declaration" function is added to the "C" source code of the function. This function is called only once when the application starts, and indicates to the ISaGRAF kernel the "C" function name which corresponds to the implemented function. This is the standard format of such a declaration function:

```
UFP uspdef_xxx (char *name)
{
    strcpy (name, "XXX");      /* gives the name of the function */
    return (USP_xxx);         /* returns the implementation function */
}
/* (xxx is the name of the function) */
```

The name of the "C" function, used for **strcpy** statement must be written in **uppercase**. It must be written in lowercase in the name of the implementation function and in the name of the declaration function. Using the "**USP_**" and "**uspdef_**" prefixes for implementation function and definition function enables the user to name a function with a reserved keyword of the "C" language, or the name of an existing function of the "C" ISaGRAF libraries.

Other statements can be added to the declaration function to create any specific initialization operation relative to this function. The ISaGRAF system ensures the user that this function is called **only once** when the application starts. The declaration function is called for any integrated "C" function, even if it is not used in the programs of the ISaGRAF application. The ISaGRAF kernel fails in a fatal error if a "C" function used in the application is not integrated to the kernel.

Before linking new functions with the kernel, the user must write another "C" source file, named "**GRUS0LIB.C**", and insert it (with the retained functions) in the list of files for the link. The "**GRUS0LIB.C**" only contains an array of declaration functions. This array is read during application initialization, to establish a dynamic link with the functions written in "C". This is an example of such a file:

```
/* File "GRUS0LIB.c" - Example using trigonometric functions */

#include <tasy0def.h>          /* required for types definition */
```

```

extern UFP uspdef_fc1 (char *name);      /* declaration functions */
extern UFP uspdef_fc2 (char *name);
extern UFP uspdef_fc3 (char *name);
extern UFP uspdef_fc4 (char *name);

UFP_LIST USPDEF[ ] = {                  /* array of declaration functions */
    /* for integrated functions */
    uspdef_fc1,
    uspdef_fc2,
    uspdef_fc3,
    uspdef_fc4,

    NULL };

/* end of file */

```

The **USPDEF** array must be terminated by a NULL pointer. Some clashes may occur when this condition is not met. Unresolved references will occur when linking the new ISaGRAF kernel if the **USPDEF** array is not defined. By writing this file, a new kernel can be built, including all the existing functions. A kernel dedicated to one project can also be built, by inserting in the **USPDEF** array only the functions used in the project. The "**GRUSOLIB.C**" file is automatically generated by the ISaGRAF Code Generator when the code of an application is built. The file is placed in the ISaGRAF project directory, and groups only the functions used in the project.

▬ **Limits**

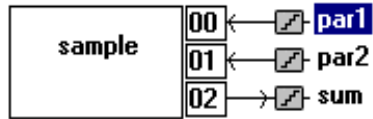
The ISaGRAF library may contain up to **255** "C" functions. Any type of operation may be processed in a function. It should be remembered that the functions are called in the ISaGRAF cycle **synchronously**, so that the execution of the function has a direct effect on the cycle timing.

▬ **Complete example**

Below is the complete programming of a "**sample**" function, which just performs an addition. Below is the technical note of the function:

name:	SAMPLE
description:	just performs an integer analog addition
creation date:	1st July 1992
author:	CJ International
call:	par1, par2: integer operands
return:	integer sum
prototype:	sum := sample (par1, par2);

Below is the interface of the function:



Below is the "C" source header of the function:

```

/* File: GRUS0255.h - user C function definitions - Name: sample */

/* definition of standard ISaGRAF data types */

typedef long T_BOO;
typedef long T_ANA;
typedef float T_REAL;
typedef long T_TMR;
typedef char *T_MSG;

/* definition of the calling and return parameter structure */

typedef struct {
    T_ANA _par1;          /* calling parameter #1 */
    T_ANA _par2;          /* calling parameter #2 */
    T_ANA _sum;           /* return parameter */
} str_arg;

/* identifiers used to access call and return parameters */

#define PAR1              (arg->_par1)
#define PAR2              (arg->_par2)
#define SUM                (arg->_sum)

/* end of file */
  
```

Below is the "C" source code of the function. Only the lines printed with bold characters were manually entered by the C programmer.

```

/* File: GRUS0255.c - user C function - Name: SAMPLE */

#include "tasy0def.h"          /* required for types definition */
#include "grus0255.h"         /* C function source header */
  
```

```

/* C main service: calculates the addition */

void USP_sample (str_arg *arg)
{
    SUM = PAR1 + PAR2;
}

/* declaration service required for dynamic link with ISaGRAF kernel */

UFP uspdev_sample (char *name)
{
    strcpy (name, "SAMPLE");
    return (USP_sample);
}
/* end of file */

```

C.7.4 "C" FUNCTION BLOCKS

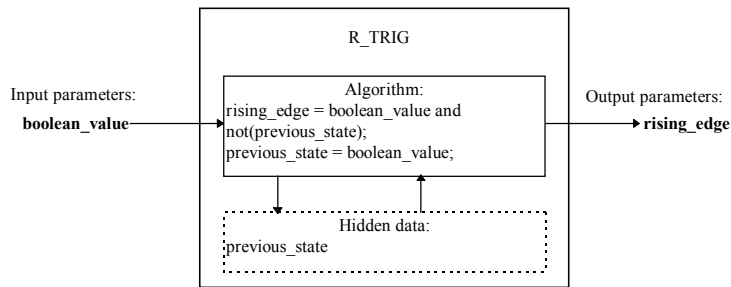
"C" function blocks associate operations and static data. They complete the set of "C" functions, by allowing the processing of static objects. They are commonly used to increase standard capabilities of **ST** and **FBD** languages. Unlike functions, which process values, function blocks can process static data. This means that a function block algorithm can manage the variations of data over time.

Function blocks are written in "C" language, compiled and linked with the ISaGRAF kernel. The increased kernel must be installed on the ISaGRAF target PLC before using new function blocks in ISaGRAF projects. New function blocks cannot be integrated in the ISaGRAF Simulator. The ISaGRAF applications have to be simulated **before** using the non standard functions.

Warning: Function block calls are **synchronous** operations, activated at run time by the ISaGRAF kernel, during the application cycle. Time spent for the execution of a function block activation or read service is included in the ISaGRAF application **cycle timing**. The user has to ensure that no "wait operation" is programmed in a function block, so that the ISaGRAF cycle processing time does not exceed the max time allowed.

▬ **Declaring function block instances**

A function block is an object which combines operations and static data. Below is the example of the "**R_TRIG**" function block which detects the rising edge of a boolean expression. Here is the functional description of the block:



The hidden static variable "**previous_state**" is needed for the calculation of the edge. This variable must be different for each use of the function block "**TRIG**" in the application. The instances of the function blocks used in the ST language must be declared in the dictionary. Because a function block has internal hidden data, each copy (instance) of a function block must be identified by a unique name. Naming the type of block is made by using the library manager. Naming the instances is made by using the dictionary editor.

Function blocks used in FBD language do not have to be declared, because the ISaGRAF FBD editor automatically declares the instances of the used blocks. Function block instances automatically declared by the FBD editor are always **LOCAL** to the edited program.

➤ **Adding a function block to the ISaGRAF library**

The ISaGRAF Library Manager must be used to add a new "C" function block to the ISaGRAF library in the Workbench. The "**New**" command of the "**Files**" menu is used, when the "C" function blocks library is selected. When a new function block has been created, its **technical note** must be written. The frame of the "C" source code for the new function block is automatically generated by the ISaGRAF Library Manager. The "**Parameters**" command of the "**Edit**" menu is used to define the calling and return parameters of the new function block.

➤ **Using a "C" function block in an ISaGRAF project**

Any integrated "C" function block can be used in the programs of an ISaGRAF project. "C" function blocks can be called from **ST** and **FBD** languages.

Calling a "C" function block from the **ST** language follows the function block calling conventions of the language. The calling parameters of the block are written after the name of the function, between parentheses, and separated by commas. The return parameters are accessed one by one. Each return parameter is represented by a name, combining the name of the block instance, and the name of the parameters. The components of the name are separated by a dot. For example, the name:

FBINSTNAME.paname

is used to represent the return parameter named "**paname**", of the function block instance named "**FBINSTNAME**".

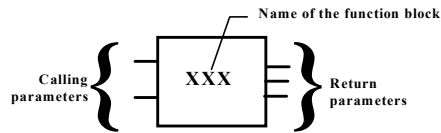
The instances of the function blocks used in the ST language must be declared in the dictionary. Each copy (instance) of a function block must be identified by a unique name. Below is an example of instance declaration in the ISaGRAF dictionary:

instance:	TRIG1	type:	R_TRIG
	TRIG2		R_TRIG

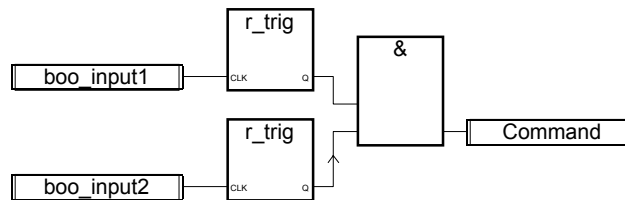
And below is an example using these declared instances in an ST program:

```
TRIG1 (boo_input1);
TRIG2 (boo_input2);
Command := (TRIG1.Q & TRIG2.Q);
```

An **FBD** program can call any "C" function block. A function block is used as a standard function box. Its calling parameters are connected to the left hand side of the function box. Its return parameters are connected to the right hand side of the box. A standard format for a function box appears as follows:



Function blocks used in FBD language do not have to be declared, because the ISaGRAF FBD editor automatically declares the instances of the used blocks. Function block instances automatically declared by the FBD editor are always **LOCAL** to the edited program. Below is the previous example, programmed in FBD language:



Defining the interface of a "C" function block

The "**Parameters**" command of the "**Edit**" menu is used to define the calling and return parameters of a new function block. A function block can have up to **32** parameters, freely arranged as calling or return parameters. Unlike a "C" function, a function block may have several return parameters.

The list in the upper side of the window shows the parameters of the "C" function block, based on the order of the function calling prototype: first the calling parameters, then return parameters. The lower part of the window shows the detailed description of the parameter currently selected in the list:

- the name of the parameter
- the direction (call/return) of the parameter
- the type of the parameter

Any of the ISaGRAF data types may be used for a parameter: Boolean, Integer analog, Real analog, Timer or Message. Integer and real analogs must be distinguished. Below is the relationship between ISaGRAF types and "C" types:

BOOLEAN	unsigned long	unsigned 32 bit word: 1=true / 0=false
ANALOG	long	signed integer 32 bit word
REAL	float	single precision floating value
TIMER	unsigned long	unsigned integer 32 bit word (unit is 1 millisecond)
MESSAGE	char *	character string.

When a message value is passed onto a "C" function, it cannot contain null characters. The string passed to the "C" code is null-terminated. Do not forget that return parameters must be the last ones in the list. The rules shown below must be followed while naming parameters:

- the length of the name cannot exceed 16 characters
- the first character must be a letter
- the following characters must be letters, digits or '_' character
- naming is case insensitive

The same name cannot be used for more than one parameter of the function block. A calling parameter cannot have the same name as a return parameter. The same name **can** be used for parameters of different function blocks. The name of a parameter is used to identify the parameter in the "C" source code.

The **"Insert"** command is used to insert a new parameter before the selected parameter. The **"Delete"** command is used to erase the selected parameter. The **"Arrange"** command automatically rearranges (sorts) the parameters, so that the return parameters are put at the end of the list. Pressing the **"OK"** button stores the definition of the function block interface and closes the dialog box. Pressing the **"Cancel"** button closes the dialog box, without changing the definition of the function block.

Function block "C" interface

The interface of a function block depends on the definition of its parameters. Calling parameters are passed through a structure. This structure is defined in the **"GRFB0nnn.H"** file, where **"nnn"** is the logical number of the function block in the ISaGRAF library. Return parameters are represented by logical numbers, which are also defined in the **"GRFB0nnn.h"** file. This is an example of the "C" interface, for the **"LIM_ALARM"** function block (alarm on limits):

```
/* function block interface - name: sample */
```

```
/* standard ISaGRAF data types */
```

```
typedef          long    T_BOO;
typedef          long    T_ANA;
typedef          float   T_REAL;
typedef          long    T_TMR;
typedef          char    *T_MSG;
```



```

/* structure of calling parameters */

typedef struct {
    /* CALL */          T_BOO _par1;
    /* CALL */          T_BOO _par2;
} str_arg;

/* access to fields of str_arg structure */

#define                PAR1  (arg->_par1)
#define                PAR2  (arg->_par2)

/* return parameter logical numbers */

#define                FBLPNO_Q1  0
#define                FBLPNO_Q2  1

/* end of file */

```

The relationship between ISaGRAF types and "C" types is shown below. The ISaGRAF types are defined as "C" types in the definition file of the function.

boolean	T_BOO	long (32 bits)
analog	T_ANA	long
real	T_REAL	float (32 bits - single precision)
timer	T_TMR	long
message	T_MSG	char * (32 bits - char pointer)

Each field of the "**str_arg**" structure corresponds to one calling parameter of the function block. The parameters appear in the structure in the same order than the one established for the function block definition. An uppercase identifier is defined to directly have access to one parameter of the structure passed to the "C" implementation of the function block activation service. Names of the identifiers are the ones entered during the definition of the function block with the ISaGRAF Library Manager.

The order used for return parameters numbering is the one established for the function block definition. The logical number of the first return parameter is always **0**.

Defined identifiers should be used instead of numerical value to represent the return parameters in the "C" source programming. This ensures that the source file can be easily re-compiled after a modification of the interface definition.

The "C" definition file is updated each time the interface of the function block is changed by using the ISaGRAF Library Manager. This ensures a complete coherence between the implementation of the function block and its use in the programs of the ISaGRAF applications.

Source code

The "C" language implementation of a function block is divided into three different entry points:

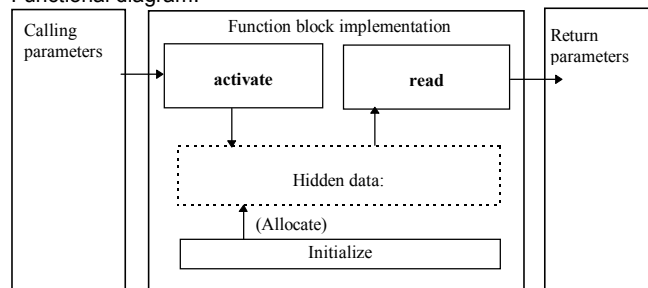
- initialization service
- activation service - processing of the calling parameters
- return parameters read service

The same code is used for each instance of a same function block, and is not duplicated. A static data structure is associated to each instance. Such data cannot be accessed directly by the ISaGRAF programming, and contain the function block instance "hidden variables".

The "activation service" is called once for each instance of each used block, on each target cycle. It processes the calling parameters, and updates the associated data. It represents the "main algorithm" of the function block.

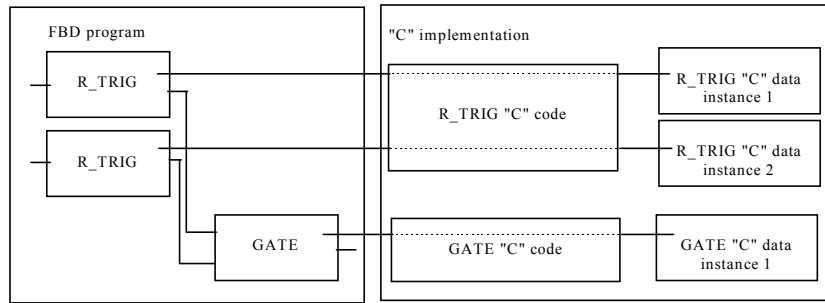
The "read service" is called by the ISaGRAF kernel to read the current value of one return parameter for one instance. No special calculation has to be performed in such a service. It only operates transfer between hidden data and the ISaGRAF application.

Functional diagram:



- **Function block static data**

A function block associates operations and static data. A data structure is associated to each instance of a same function block. Each time a function block is used in ST or FBD programming, it corresponds to one instance, and one data structure. The following example shows the correspondence between "C" data structures and the function block instances used in an FBD program:



The memory needed for data structure of each instance is allocated by the ISaGRAF system, when the application starts. A pointer to the associated instance data structure is passed to the "activate" and "read" services.

The ISaGRAF Library Manager automatically generates the frame of the "C" source code for data structure type definition. The type of the data structure is always called "**str_data**". The programmer should not change this name, to ensure compatibility with service headers. The hidden data generally groups internal variables with an image of the return parameters. The function block "read" service is only used to access the return parameter, and should not be used to perform other operations.

- **The initialization service**

The "initialize" service of a function block is called by the ISaGRAF kernel when the application starts. It allows the "C" programmer to ask the system to allocate memory for an instance. Below is the standard programming of the initialization service:

```
uint16 FBINIT_xxx (uint16 hinstance)
/* "xxx" is the name of the f. block */
{
    return (sizeof (str_data));
}
```

The "**hinstance**" argument is the logical number of the instance. It is reserved for ISaGRAF internal operations, and should not be used in the programming of the service. The initialization service returns the number of memory bytes required for the data of **one** instance. The amount of required memory (return value) cannot exceed **64** Kbytes. No other operation should be performed in this service. The "C" source code of this service is automatically generated by the ISaGRAF Library Manager when the function block is created.

- **The activation service**

The "activation" service is called on each target cycle, for each function block instance used in the application. This service processes the calling parameters and runs the main function block algorithm, in order to update the hidden static data and the value of return parameters. Below is the standard frame of the activation service:

```
void FBACT_xxx (
```

```
uint16 hinstance,          /* "xxx" is the name of the function block */
                           /* logical number of the instance */
str_data *data,           /* data: pointer to the instance data structure */
str_arg *arg              /* pointer to the calling parameters structure */
)
{
}
```

The "**hinstance**" argument is the logical number of the instance. It is reserved for ISaGRAF internal operations, and should not be used in the programming of the service. The "**data**" argument is a far pointer to the data structure associated to the instance. The "**arg**" argument is a far pointer to the structure which contains the value of the calling parameters. The programmer should use the identifiers defined in the function block "C" header to have access to the fields of the "**arg**" structure.

The "activation" algorithm processes the calling parameters (stored in "**arg**" structure), and updates the fields of the "**data**" structure. The following

example shows the "activation" service of the **TRIG** (rising edge detection) function block:

```
/* definitions stored in the function block "C" header */

typedef struct {           /* calling parameters */
    T_BOO _clk;           /* trigger input */
} str_arg;

#define CLK    (arg->_clk)

/* function block instance data structure */

typedef struct {
    T_BOO prev_state;     /* previous state of the trigger input */
    T_BOO edge_detect;    /* edge value: image of return param */
} str_data;

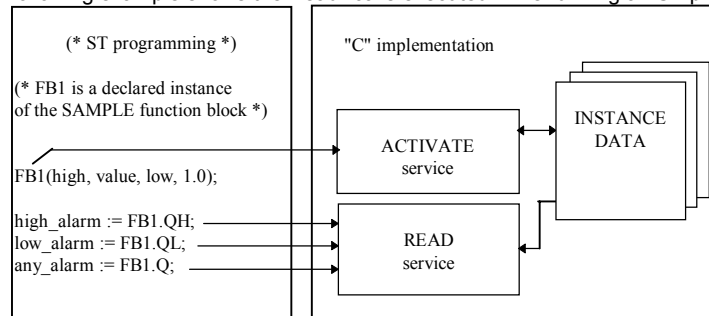
/* activation service */

void FBACT_trig (uint16 hinstance, str_data *data, str_arg *arg)
{
    data->edge_detect = (T_BOO)(CLK && !data->prev_state);
    data->prev_state = CLK; /* calling parameter */
}
```

The "C" source code frame of this service is automatically generated by the ISaGRAF Library Manager when the function block is created.

- **Reading the return parameters**

The "read" service is called each time a return parameter of a function block instance is referenced in an ST or FBD program. It is used to get the value of **one** return parameter. The following example shows the "read" calls executed while running an ST program:



Because the "read" service can be called more than once in the same cycle, for the same return parameter or the same function block instance, no special calculation has to be performed in such a service. It only operates transfer between hidden data and the ISaGRAF application. Below is the standard frame of the read service:

```
/* cast operation used to copy the value of a return parameter */
```

```
#define BOO_VALUE      ((T_BOO *)value)
#define ANA_VALUE      ((T_ANA *)value)
#define REAL_VALUE     ((T_REAL *)value)
#define TMR_VALUE      ((T_TMR *)value)
#define MSG_VALUE      ((T_MSG *)value)
```

```
/* return parameters read service: called for each return parameter */
```

```
void FBREAD_XXX (          /* "xxx" is the name of the function block */
uint16 hinstance,         /* logical number of the instance */
str_data *data,          /* pointer to the instance data structure */
uint16 parno,            /* logical number of read parameter */
void *value)             /* buffer where to copy the value of the param
*/
{
    switch (parno) {
        case FBLPNO_XX: /* ... */ break;
        case FBLPNO_YY: /* ... */ break;
```

```
        /* .... */  
    }  
}
```

The "**hinstance**" argument is the logical number of the instance. It is reserved for ISaGRAF internal operations, and should not be used in the programming of the service. The "**data**" argument is a far pointer to the data structure associated to the instance.

The "**parno**" argument is the logical number of the return parameter which value is required. Use the identifiers defined in the function block "C" header to identify return parameters. Such identifiers begin with the "**FBLPNO_**" prefix. The "**value**" argument is a far pointer to the buffer where to copy the current value of the accessed return parameter. The type of data pointed to by this argument depends on the ISaGRAF type of the return parameter. The following table gives the relationship between ISaGRAF types and buffer "C" data type:

boolean	long	32 bit unsigned word (0=false / 1=true)
analog	long	32 bit signed word
real	float	32 bit single precision floating value
timer	long	32 bit unsigned word (unit is 1ms)
message	char *	array of characters

The following macros are used to have access to the copy buffer, according to the type of the accessed return parameter:

```
#define      BOO_VALUE      ((T_BOO *)value)  
#define      ANA_VALUE      ((T_ANA *)value)  
#define      REAL_VALUE     ((T_REAL *)value)  
#define      TMR_VALUE      ((T_TMR *)value)  
#define      MSG_VALUE      ((T_MSG *)value)
```

These are commonly used programmed operations to copy the value or the parameter to the ISaGRAF buffer:

```
/* for a boolean parameter: */  
*BOO_VALUE = parameter_value;  
/* for an integer analog parameter: */  
*ANA_VALUE = parameter_value;  
/* for a real integer parameter: */  
*REAL_VALUE = parameter_value;  
/* for a time parameter: */  
*TMR_VALUE = parameter_value;  
/* for a string parameter: */  
strcpy (*MSG_VALUE, parameter_value);
```

The "C" source code frame of this service is automatically generated by the ISaGRAF Library Manager when the function block is created.

- **Example of "C" source file**

Below is the standard frame of a "C" function block implementation:

```

/* function block (xxx is the name of the function block) */

#include <tasy0def.h>
#include <grfb0nnn.h> /* nnn is the number of the f.block in library */

/* structure of hidden data for each instance of the block */
typedef struct {
    /* fields definition */
} str_data;

/* initialization service: returns the size of needed hidden data */
word FBINIT_xxx (uint16 hinstance)
{
    return (sizeof (str_data));
}

/* activation service: processes the calling parameters */
void FBACT_xxx (uint16 hinstance, str_data *data, str_arg *arg)
{
    /* ... */
}

/* cast operation used to copy the value of a return parameter */
#define BOO_VALUE ((T_BOO *)value)
#define ANA_VALUE ((T_ANA *)value)
#define REAL_VALUE ((T_REAL *)value)
#define TMR_VALUE ((T_TMR *)value)
#define MSG_VALUE ((T_MSG *)value)

/* return parameters read service: called for each return parameter */
void FBREAD_xxx (uint16 hinstance, str_data *data, uint16 parno, void *value)
{
    switch(parno)
    {
        case FBLPNO_XX: *???_VALUE = ...; break;
        case FBLPNO_YY: *???_VALUE = ...; break;
        ....
    }
}

```

```
}

/* The following function is used for the initialization of the function block and the
declaration of its implementation. It realizes the link with the ISaGRAF kernel,
using the name of the function block. This service is completely generated by the
ISaGRAF Library Manager. */

ABP fbldf_xxx (char *name, IBP *initproc, RBP *readproc)
{
    strcpy (name, "XXX");
    *initproc = (IBP)FBINIT_xxx;
    *readproc = (RBP)FBREAD_xxx;
    return ((ABP)FBACT_xxx);
}

/* end of file */
```

The "**TASY0DEF.H**" include file from the ISaGRAF kernel is required for system dependent definitions. It also contains the definition of data types representing far pointers to the implemented services.

▬ **Links between projects and "C" implementation**

The logical link between the implementation of a "C" function block and its use in the programs of an ISaGRAF project is accomplished by using the name of the function. A "declaration" service is added to the "C" source code of the function block. This service is called only once when the application starts, and indicates to the ISaGRAF kernel the "C" function block name which corresponds to the implemented services. This is the standard format of such a declaration service:

```
ABP fbldf_xxx (char *name, IBP *initproc, RBP *readproc)
{
    strcpy (name, "XXX");           /* name of the f.block */
    *initproc = (IBP)FBINIT_xxx;   /* initialization service */
    *readproc = (RBP)FBREAD_xxx;   /* read service */
    return ((ABP)FBACT_xxx);       /* activation service */
}
/* xxx is the name of the function block */
```

The name of the function block, used for **strcpy** statement must be written in **uppercase**. Lowercase must be used for the name of the implemented services and in the name of the declaration service.

Using the "**FBACT_**", "**FBINIT_**", "**FBREAD_**" and "**fbldf_**" prefixes for implemented services and definition service enables the user to name a function block with a reserved

keyword of the "C" language, or the name of an existing function of the "C" ISaGRAF libraries. No other statement should be added to the declaration service.

The declaration service is called for any integrated "C" function block, even if it is not used in the programs of the ISaGRAF application. The ISaGRAF kernel will detect a fatal error if a "C" function block used in the application is not integrated to the kernel.

Before linking new function blocks with the kernel, the user must write another "C" source file, named "**GRFB0LIB.C**", and insert it (with the retained function blocks) in the list of files for the link. The "**GRFB0LIB.C**" only contains an array of declaration services. This array is read during application initializations, to create a dynamic link with the "C" written function blocks. This is an example of such a file:

```
/* File: grfb0lib.c - implemented function blocks */

#include <tasy0def.h>

extern ABP fbldef_fb1(char *name, IBP *init, RBP *read);
extern ABP fbldef_fb2(char *name, IBP *init, RBP *read);

FBL_LIST FBLDEF[ ] = {
    fbldef_fb1,
    fbldef_fb2,

    NULL };

/* end of file */
```

The **FBLDEF** array must be terminated by a NULL pointer. Some clashes may occur when this condition is not met. Unresolved references will occur when linking the new ISaGRAF kernel if the **FBLDEF** array is not defined.

By writing this file, a new kernel can be built, including all the existing function blocks. A kernel dedicated to one project can also be built, by inserting in the **FBLDEF** array only the function blocks used in the project. The "**GRFB0LIB.C**" file is automatically generated by the ISaGRAF Code Generator, when the code of an application is built. The file is placed in the ISaGRAF project directory, and groups only the function blocks used in the project.

▬ **Limits**

The ISaGRAF library may contain up to **255** "C" function blocks. Any type of operation may be processed in a function. Each type of function block may be copied (instanced) up to **255** times in the same project.

It should be remembered that the function block services are called in the ISaGRAF cycle, **synchronously**, so that the execution of the function block has a direct effect on the cycle timing.

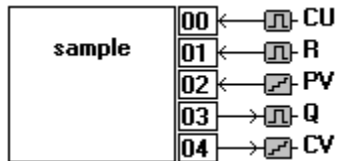
Complete example

Below is the complete programming of a "sample" function block, which is an up-counter.

Below is the technical note of the function block:

name:	SAMPLE
description:	Up counter
creation date:	01 February 1994
author:	CJ international
call:	CU : counting input R : reset command PV : maximum programmed value
return:	Q : max detection CV : counting result
prototype:	SAMPLE (count, reset_command, maximum_value); max_detect := SAMPLE.Q; count_result := SAMPLE.CV;

Below is the interface of the function block:



Below is the "C" source header of the function block:

```

/* function block interface - name: SAMPLE */

/* definition of standard ISaGRAF data types */

typedef long T_BOO;
typedef long T_ANA;
typedef float T_REAL;
typedef long T_TMR;
typedef char *T_MSG;

/* definition of the calling parameters structure */

typedef struct {
    T_BOO _cu;

```

```

    T_BOO _r;
    T_ANA _pv;
} str_arg;

/* identifiers used to access the calling parameters */

#define CU                (arg->_cu)
#define R                 (arg->_r)
#define PV                (arg->_pv)

```

```
/* return parameters logical numbering */
```

```

#define FBLPNO_Q          0
#define FBLPNO_CV        1

```

```
/* end of file */
```

Below is the "C" source code of the function block. Only the lines printed with bold characters were manually entered by the C programmer.

```

/* function block - name: SAMPLE */

#include <tasy0def.h>          /* required for data types definition */
#include <grfb0255.h>        /* function block C source header */

/* definition of the structure which contains the data for one instance */

typedef struct {
    T_BOO overflow;        /* true:counting value >= programmed value */
    T_ANA value;          /* counting current value */
} str_data;

/* initialization service: requires memory for instance data */

word FBINIT_sample (uint16 hinstance)
{
    return (sizeof (str_data));
}

/* activation service: up-counting algorithm */

```

```
void FBACT_sample (uint16 hinstance, str_data *data, str_arg *arg)
{
    if (R) data->value = 0;
    else if (CU && data->value < PV) (data->value)++;
    data->overflow = (data->value >= PV) ? (T_BOO)1 : (T_BOO)0;
}

/* cast operation required to copy parameters to ISaGRAF buffer */

#define BOO_VALUE ((T_BOO *)value)
#define ANA_VALUE ((T_ANA *)value)
#define REAL_VALUE ((T_REAL *)value)
#define TMR_VALUE ((T_TMR *)value)
#define MSG_VALUE ((T_MSG *)value)

/* read service: get the value of one return parameter */

void FBREAD_sample (uint16 hinstance, str_data *data, uint16 parno, void
*value)
{
    switch (parno) {
        case FBLPNO_Q : *BOO_VALUE = data->overflow; break;
        case FBLPNO_CV : *ANA_VALUE = data->value; break;
    }
}

/* declaration service used for dynamic link with the ISaGRAF kernel */

ABP fbldef_sample (char *name, IBP *initproc, RBP *readproc)
{
    strcpy (name, "SAMPLE");
    *initproc = (IBP)FBINIT_sample;
    *readproc = (RBP)FBREAD_sample;
    return ((ABP)FBACT_sample);
}

/* end of file */
```

C.7.5 Compiling and linking techniques

The ISaGRAF Workbench does not include any "C" compiler or linker. However this chapter explains the main techniques which can be applied to easily use the files created by the ISaGRAF Library Manager, and pass them to other tools such as compilers and linkers.

▣ "C" source files

The "C" source files of conversions, functions and function blocks are put by the ISaGRAF Library Manager into the **ISAWIN\LIB\DEFS** and **ISAWIN\LIB\SRC** directories. The name of a source file is built with the number of the corresponding conversion, function or function block in the ISaGRAF library. These are the used filenames:

<code>\isawin\lib\defs\TACN0DEF.H</code>	definition file for any conversion functions
<code>\isawin\lib\src\GRCN0nnn.H</code>	source file of a conversion function
<code>\isawin\lib\defs\GRUS0nnn.H</code>	definition file of a function
<code>\isawin\lib\src\GRUS0nnn.C</code>	source file of a function
<code>\isawin\lib\defs\GRFB0nnn.H</code>	definition file of a function block
<code>\isawin\lib\src\GRFB0nnn.C</code>	source file of a function block

(**nnn** is the number of the conversion, function or function block)

Warning: When renaming or copying library elements, its text and programming lines are not updated by the ISaGRAF Library Manager, according to new element name and logical number. They must be manually updated in the "C" source file.

The file **ISAWIN\LIB\USPNUMS** gives the relationship between names and logical numbers for the "C" functions existing in the ISaGRAF library. This is, as an example of such a file:

```

1    funct_A
10   funct_B
16   funct_C

```

The file **ISAWIN\LIB\FBLNUMS** gives the relationship between names and logical numbers for the "C" function blocks existing in the ISaGRAF library. This is as an example, of such a file:

```

0    fbl_A
1    fbl_B
2    fbl_C

```

The file **ISAWIN\LIB\CNVNUMS** gives the relationship between names and logical numbers for the conversion functions existing in the ISaGRAF library. This is, as an example, the content of this file for the conversions of the standard library:

```

0    SCALE
1    BCD

```

These files are automatically updated by the ISaGRAF Library Manager each time a conversion, function or function block is created, renamed, copied or deleted. The ISaGRAF Code Generator automatically generates the following files when an application is built:

\\isawin\apl\ppp\GRCN0LIB.C	Declaration as an array of all the conversion functions used in the project.
\\isawin\apl\ppp\GRUS0LIB.C	Declaration as an array of all the functions used in the project.
\\isawin\apl\ppp\GRFB0LIB.C	Declaration as an array of all the function blocks used in the project.

(**ppp** is the name of the ISaGRAF project)

These files can be used during link operations to build a new ISaGRAF kernel dedicated to the project, which contains only the conversions, functions and function blocks used in the project.

═ Downloading source files to a native system

The "C" source and definition files created by the ISaGRAF Library Manager may be downloaded to the target ISaGRAF system, if it supports a native compiling tool. To do that, the standard **TERMINAL** tool delivered with Windows can be used.

When source files are managed on the target system, definition files have to be updated with a new download operation each time a function interface is modified with the ISaGRAF Library Manager.

Commands lines to download files can be grouped for instance into a batch file and then started from the workbench tool menu (see user's guide: Managing programs)

═ Using a cross compiler

Source files can also be managed directly on your PC, if the target is a PC, or a cross compiler is available, running on the PC and generating code for the target system.

In this case, the user can run the ISaGRAF Library Manager to complete and modify the sources of conversions, functions or function blocks.

Commands lines for running the compiler and the linker can be grouped for instance into a batch file and then started from the workbench tool menu (see user's guide: Managing programs)

When conversions, functions and function blocks are compiled on the PC, the user simply has to download the new generated ISaGRAF kernel (linked with new components) to the target system before running applications. If the target is another PC, the new generated ISaGRAF kernel can be loaded into the target machine by using a diskette or through a network.

═ Linking with the ISaGRAF kernel libraries

Warning:

The following are general information which may not exactly correspond to your target system.

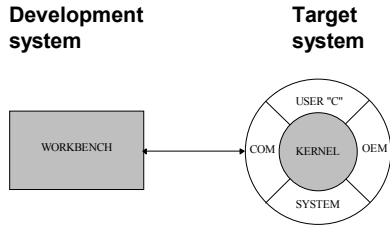
In any case you may consult the readme and .TXT files delivered on the target disk.

The ISaGRAF target diskette contains many utility files to compile and link the conversions, functions and function blocks with the ISaGRAF kernel libraries.

Two implementations exists:

- single task ISaGRAF: all functions are performed in the same program
- multitask ISaGRAF: a separate task (or thread) is dedicated to communication

In either case, the "C" components are grouped in the same libraries: for the "C" programmer, no difference is made for single task or multitask. For a single task version, the user "C" libraries are linked to the single task (generally called **isa**), whereas for the multitask version the libraries are linked to the kernel task (generally called **isaker**).

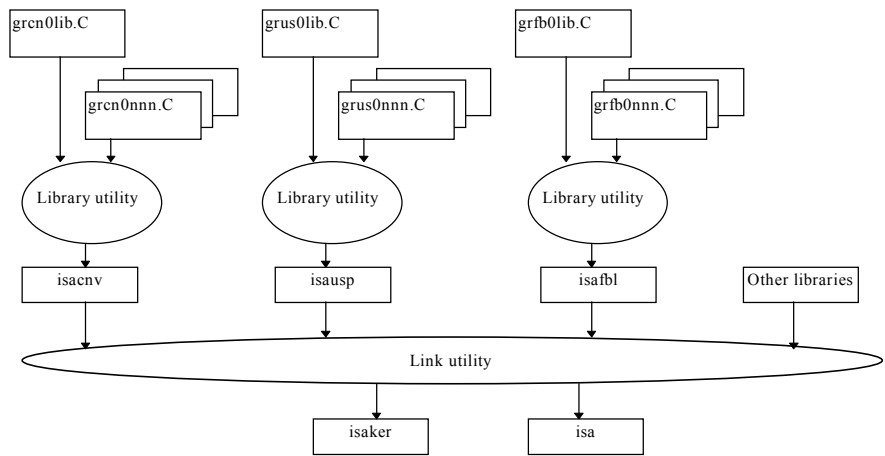


The inner part of ISaGRAF software is independent of the hardware. It executes the IEC languages and has its own variable data base.

The first step, when making the link with the kernel, is to build libraries of all the conversions, functions and function blocks needed for the specific project:

Library	content
ISAUSP	- GRUSOLIB object file (array of declared functions) - object file of each integrated function
ISAFBL	- GRFBOLIB object file (array of declared function blocks) - object file of each integrated function block
ISACNV	- GRCNOLIB object file (array of declared conversions) - object file of each integrated conversion function

Then the programmer has to link these new libraries with other object files and libraries of the ISaGRAF kernel. The different phases of a user "C" development integration are outlined in the following diagram:



This is the exact list of object modules and libraries which have to be joined during the link:

<u>To build isaker:</u>	
Object Module:	tast0mai
Object Module:	tats0com
Kernel library:	isaker
Kernel library:	isaoem
User library:	isausp user defined functions
User library:	isafbl user defined function blocks
User library:	isacnv user defined conversion functions
Kernel library:	isasys
System libraries:	(refer to your "C" compiler manual)

<u>To build isa:</u>	
Object Module:	tast0mai
Object Module:	tast0com
Kernel library:	isaker
Kernel library:	isatst
Kernel library:	isaoem
User library:	isausp user defined functions
User library:	isafbl user defined function blocks
User library:	isacnv user defined conversion functions

Kernel library:	isasys
System libraries:	(refer to your "C" compiler manual)

The programmer may have to follow the exact order of object modules and libraries shown in the preceding figures. Object modules and libraries have standard extensions (".lib", ".obj", ".l", ".r"...) according to the target system.

▣ **Required compiling and linking options**

Convenient options can be selected during compiling and linking. They depend on the type of operations processed in conversions, functions and function blocks. Some operations require other system libraries (math, graphics...) during link.

All the "C" source files of the ISaGRAF Kernel have been compiled with the **LARGE** memory model. The programmer must use the same model for compiling conversions, functions and function blocks.

A special constant has to be defined for compiling "C" library components. It indicates the type of target system and processors, so that the source of conversions, functions and function blocks can be system independent. Below are the names of these constant values:

DOS.....for DOS based systems (INTEL processor)
ISAWNTfor Windows-NT based systems (INTEL processor)
OS9for OS9 system (MOTOROLA processor)
VxWorksfor VxWorks system (MOTOROLA processor)

The utility command files (to compile and link) delivered with the ISaGRAF target software show how to define the convenient constant value in the compiler command line.

▣ **Supported compiler**

The following compilers are supported for the development of conversions, functions and function blocks, and the link with the ISaGRAF Kernel:

Microsoft MSC 7.00 compiler	for DOS based targets
Microsoft MSVC 4.00 compiler	for Windows-NT based targets
Microware ULTRA-C compiler	for OS-9 targets
Tornado 1.0; GNU Toolkit 2.6	for VxWorks targets

Contact CJ International for using other compilers.

▣ **Summary**

Below is the summary of the operations which have to be performed when developing a new conversion, function or function block.

- ⇒1. With the ISaGRAF Library Manager, create the new element: give it a name and a comment text. The frame of the "C" source file is automatically generated.

- ⇒2. With the ISaGRAF Library Manager, describe the interface (call and return parameters), if the element is a function or a function block. The "C" source header file is automatically generated.
- ⇒3. With the ISaGRAF Library Manager, enter the text of the detailed technical note (user's guide) of the element.
- ⇒4. With the ISaGRAF Library Manager, complete the "C" source file, by entering the "C" programming of the conversion, function or function block algorithm. The source code of the element is now complete. Note that another editor may be used.
- ⇒5. Select the "**Show logical number**" option of the Library Manager to know what logical number is attached to the new element. This number is used in the pathnames of the corresponding ".C" and ".H" source files.
- ⇒6. Copy / Download the .C and .H files to your target (if native compiler) or to the corresponding environment (if cross compiler) where the ISaGRAF target libraries and tasks have been installed.
- ⇒7. Run the "C" compiler on the new source file, and correct syntax errors if any.
- ⇒8. Insert the name of the new element declaration service in the "**GR??0LIB.C**" source file which defines the array of inserted elements.
- ⇒9. Run the "C" compiler to compile the "**GR??0LIB.C**" file.
- ⇒10. Insert the name of the object module in the list of object files used to build the corresponding library.
- ⇒11. Run the "C" library builder. Run the "C" linker to build the new kernel.
- ⇒12. Install the new created kernel on your target machine.
- ⇒13. Write a sample ISaGRAF application which tests the activation and the interface of the new element.

C.8 Modbus link

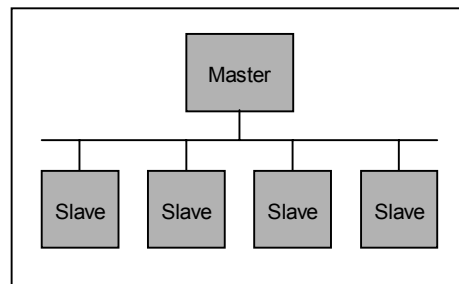
Once the application is completely developed and tested, you may connect it to a process visualization system.

ISaGRAF is an open system offering a large variety of networking possibilities. The simplest industrial network is the MODBUS/MODICON standard protocol, which is available on almost every process visualization system and which only requires a serial link (RS232, RS485, Current Loop).

ISaGRAF communication debugger protocol is MODBUS compatible to enable variable read/write access from a Modbus master.

C.8.1 MODBUS network and protocol

A Modbus network is composed of one master station only (usually a process visualization system) and one or more slave stations (usually PLCs).



The master sends one request at a time to one slave (using its slave number) and waits for the slave to answer before sending the next question. Other non concerned slaves do not answer.

Each frame contains a slave number, a request number and corresponding data, and a 16 bit checksum (CRC).

If no answer is received after a time-out duration, the question can be repeated a certain number of times before the master declares the slave 'disconnected'. The time-out value and the number of retries have to be adjusted on the master station to fit the slave requirements (depending on the application, etc...).

If an error occurs in a request processing, the slave may issue an error message instead of sending the expected answer frame.

Modbus is a Modicon protocol but not an international standard, there are many different implementations of 'Modbus' compatible protocols, with many possible differences, such as:

- List of implemented function codes
- Address mapping
- RTU (binary codes) or ASCII protocol
- etc...

C.8.2 ISaGRAF implementation

▣ **Application Variables access**

The ISaGRAF communication link recognizes five Modbus function codes:

1	read N bits
3	read N words
5	write 1 bit
6	write 1 word
16	write N words

ISaGRAF application variables can be accessed through their 'network address', if, of course, they have been defined in the workbench dictionary. These variables can be:

- Boolean or Analog variables
- inputs, outputs or internal variables
- local or global variables.

To write a Boolean variable, either function 5, 6 or 16 may be used. A TRUE value for writing is any non zero value.

To read a Boolean variable, either function 1 or 3 may be used. With the function 1, values are retrieved in a bit field, with function 3, they are retrieved in Bytes (a TRUE value correspond to 0xFFFF).

To write an analog variable, either function 6 or 16 may be used. The value is a 16 bits integer ranging from -32768 up to +32767 (ISaGRAF target variable are 32 bits).

To read an Analog variable, function 3 should be used. The read value is a 16 bits integer ranging from -32768 up to +32767. On the target side, Analog variables are 32 bits, therefore a value, on the target, over the 16 bits range (positive or negative) will be read with the maximum 16 bits range value (positive or negative).

Real variables cannot be accessed with a Modbus request.

Warning:

The ISaGRAF implementation does not manage the error codes such as 'unknown modbus address'.

Notations:

slv slave number
nbw number of words

nbb number of bytes
 nbi number of bits
 addH network address (High Byte)
 addL network address (Low Byte)
 vH value (High Byte)
 vL value (Low Byte)
 V Byte Value
 bfd Bit field (nbb Bytes)
 crcH checksum (High Byte)
 crcL checksum (Low byte)

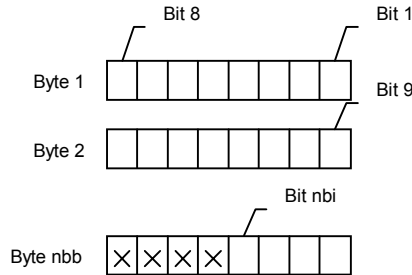
FUNCTION 1: read N bits

Read nbi bits (Booleans) starting from network address addH/addL

Question	slv	01	addH	addL	00	nbi	crch	crcL
Answer	slv	01	nbb	bfd	...		crch	crcL

Byte 1 Byte nbb

bfd is a bit field of nbb Bytes with the following format:



Bit 1 correspond to the value of the variable at addH/addL network address.
 Bit nbi correspond to the value of the variable at addH/addL + nbi -1 network address.
 X means undefined value.

FUNCTION 3: read N words

Read nbw words starting from network address addH/addL

Question	slv	03	addH	addL	00	nbw	crch	crcL
Answer	slv	03	nbb	vH	vL	...	crch	crcL

nbb corresponds to the number of vH, vL bytes

FUNCTION 5: write 1 bit

Write a bit (Boolean) at network address addH/addL

Question	slv	05	addH	addL	vH	00	crcH	crcL
----------	-----	----	------	------	----	----	------	------

Answer	slv	05	addH	addL	vH	00	crcH	crcL
--------	-----	----	------	------	----	----	------	------

FUNCTION 6: write 1 word

Write a word at network address addH/addL

Question	slv	06	addH	addL	vH	vL	crcH	crcL
----------	-----	----	------	------	----	----	------	------

Answer	slv	06	addH	addL	vH	vL	crcH	crcL
--------	-----	----	------	------	----	----	------	------

FUNCTION 16: write N words

Write nbw words starting from network address addH/addL (nbb = 2nbw)

Question	slv	10	addH	addL	00	nbw	nbb	vH	vL	...	crcH	crcL
----------	-----	----	------	------	----	-----	-----	----	----	-----	------	------

Answer	slv	10	addH	addL	00	nbw	crcH	crcL
--------	-----	----	------	------	----	-----	------	------

Examples:

– Function 1: read 15 bits starting from network address 0x1020, on slave 1

Question	01	01	10	20	00	0F	79	04
----------	----	----	----	----	----	----	----	----

Answer	01	01	02	00	12	39	F1
--------	----	----	----	----	----	----	----

The value read is 0x0012, which gives 00000000 00010010 as a bit field. In this example, variables 0x1029 and 0x102C are TRUE, all others are FALSE.

– Function 16: write 2 words at address 0x2100 on slave 1, values written are 0x1234 and 0x5678.

Question	01	10	21	00	00	02	04	12	34	56	78	1C	CA
----------	----	----	----	----	----	----	----	----	----	----	----	----	----

Answer	01	10	21	00	00	02	4B	F4
--------	----	----	----	----	----	----	----	----

≡ **File transfer**

Compared to modern field buses, Modbus protocol offers very poor services if it is not extended by specific manufacturers functions codes.

In our situation, running ISaGRAF on a powerful and flexible computer base, there are two restrictions to the Modbus protocol:

- It is only possible to access ISaGRAF variables
- It is difficult to execute fast transfer of a large amount of data

These are the reasons why ISaGRAF offers a set of file transfer 'Modbus like' requests, or a 'remote file management' protocol. These features have been implemented to enable:

- Binary or ASCII file download
- Binary or ASCII file upload
- Dynamic data exchange through virtual or physical shared file

Thus, with the ISaGRAF communication link, any application « independent from ISaGRAF » can easily communicate with a remote target.

The protocol is based on the following concepts:

- The file on the ISaGRAF target side is called **remote file**
- The file on the master computer is called **local file**
- Each byte in a file is accessed with a 32 bit **base address** plus a 16 bit **byte address**

There are requests to select the remote file name, to select the base address, to read or write data of the remote file using the 16 bits byte address.

FUNCTION 17: write data

nbb correspond to the number of vH, vL bytes

Question	slv	11	addH	addL	00	nbb	nbb	vH	vL	...	crch	crcl
----------	-----	----	------	------	----	-----	-----	----	----	-----	------	------

Answer	slv	11	addH	addL	00	nbb	crch	crcl
--------	-----	----	------	------	----	-----	------	------

The meaning of this request differs according to the address range addH/addL:

- **0xF000: Initialize remote file name**
nbb correspond to the number of characters for the file name, specified in the vH vL fields (in that case High and Low is not meaningful) and **including \0** for end of string.
If the file does not exist, it is created with writable + readable + executable attributes.
- **0xF002: Change base address to the specified value**
nbb should be equal to 4. The first vH/vL byte correspond to the High word of the specified value. Any 32 bit value is accepted.
All future read or write requests will use this base address. When this request is not used the default base address value is zero.
- **0xF004: Delete file**
nbb should be equal to zero.
The file will be deleted if it exists and if it is possible.
- **Greater than 0xF004: Reserved**
- **Less than 0xF000: Write bytes**
The specified address where to write bytes is specified in addH/addL. It must be less than F000. The specified bytes (nbb bytes specified in vH vL fields where High and Low may

be no more meaningful) are written, in the order given (from left to right), to the previously selected remote file name. The start address written to, is the specified address added to the previously selected base address. If the resulting addresses access exceed the current file size, the file is extended. You cannot reduce the file size.

FUNCTION 18: read data

Question	slv	12	addH	addL	00	nbb	crch	crcl
----------	-----	----	------	------	----	-----	------	------

Answer	slv	12	nbb	V	V	...	crch	crcl
--------	-----	----	-----	---	---	-----	------	------

The specified address where to read bytes is specified in addH/addL. It must be less than F000. Read the specified (nbb) number of bytes, from the previously selected remote file name, starting from specified address (addH/addL with any 16 bits value) added to the previously selected base address.

The Values are retrieved (V fields from left to right) in the order they are read in the file.

Example:

Select remote file name: 'target.fil'.

Question	01	11	F0	00	00	0B	0B	74	...	00	25	9F
----------	----	----	----	----	----	----	----	----	-----	----	----	----

Answer	01	11	F0	00	00	0B	8F	0E
--------	----	----	----	----	----	----	----	----

Select base address: 0x10000.

Question	01	11	F0	02	00	04	04	00	01	00	00	76	11
----------	----	----	----	----	----	----	----	----	----	----	----	----	----

Answer	01	11	F0	02	00	04	6E	CA
--------	----	----	----	----	----	----	----	----

Write 4 bytes: absolute address 0x107D0, values 01,02,03,04.

Question	01	11	07	D0	00	04	04	01	02	03	04	28	6F
----------	----	----	----	----	----	----	----	----	----	----	----	----	----

Answer	01	11	07	D0	00	04	FC	87
--------	----	----	----	----	----	----	----	----

Read 4 bytes: absolute address 0x107D0.

Question	01	12	07	D0	00	04	B8	87
----------	----	----	----	----	----	----	----	----

Answer	01	12	04	01	02	03	04	58	7D
--------	----	----	----	----	----	----	----	----	----

C.9 Power fail management

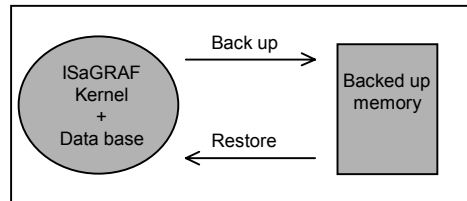
C.9.1 Basics

Managing a power failure is something very critical in an application, for three reasons:

- It depends on the process specifications
- It depends on the hardware capabilities
- It depends on the programming methods

Thus, the ISaGRAF answer to the power failure management is not a complete and absolute universal method, but a set of principles, methods and tools that have to be combined in a specific way for each application, or at least hardware.

To enable a process control system to restart correctly after a power failure, 3 problems must be solved:



- Making a data backup
- Detecting that a power failure has occurred when starting
- Restoring the backed up data

The second problem cannot have a standard software solution, but the system supplier may provide the necessary tools to have access to the hardware status from the ISaGRAF application or from a C program.

Furthermore, the important thing is to decide what data have to be stored and retrieved. Let's define 2 kinds of data:

- Application variables:
 - Such as process variables like number of items processed, date of the power failure, values of application parameters, etc. ...
 - Such as program variables like counters, timers, intermediate values and flags.
- Program state:
 - Such as reference of active steps, status of each C program, etc. ...

These 2 cases are studied in the following chapters to see how ISaGRAF can answer.

C.9.2 Application variables backup

▣ **Retained variables**

The workbench variable editor offers the possibility to select the 'retain' attribute for each internal variable (non IO variable).

At the end of each target cycle, values of retained variables are copied to the specified memory location. This memory location is generally a battery backed up RAM.

At start-up, if at least one variable has the "retained" attribute, ISaGRAF looks for the retained variables:

- If the same application has run before, ISaGRAF recognizes the stored values and assigns them to every 'retained' variables.
- If the previous application was a different one, or if no application has run before, ISaGRAF recognizes that the 'retained' values are not valid, and resets all the 'retained' variables to null.

The specification of the memory area used to store the different types of variables is specified in the workbench, in the **Make** menu: **Application run time option ; retained variables**.

The specified string must have the following format:

boo_add , boo_size , ana_add , ana_size , tmr_add , tmr_size , msg_add , msg_size

with:

boo_add: Hexadecimal address used to store boolean variables. Need to be always different from zero.

boo_size: Hexadecimal size, in bytes, available at this address. One byte per boolean variable to store is required.

ana_add: Hexadecimal address used to store analog variables. Need to be always different from zero.

ana_size: Hexadecimal size, in bytes, available at this address. Minimum of four bytes always required plus four bytes per analog variable to store.

tmr_add: Hexadecimal address used to store timer variables. Need to be always different from zero.

tmr_size: Hexadecimal size, in bytes, available at this address. Five bytes per timer variable to store is required.

msg_add: Hexadecimal address used to store message variables. Need to be always different from zero.

msg_size: Hexadecimal size, in bytes, available at this address. 256 bytes per message variable to store is required.

Requirements:

- All fields of all types need to be specified even if you may not need to make a backup of all types of variables. In such a case you need to specify a size of zero (except for analogs for which you need to specify a size of four bytes) and any address different from zero, for the non required type(s) of variables.

Example:

Let's suppose that we need to make a backup of:

- 20 Booleans variables
- 0 Analog variable
- 0 Timer variable
- 3 Message variables

The backed up memory is located at Hexadecimal address 0xA2F200.

Let's suppose that:

Booleans will be stored at address 0xA2F200 with the exact required size of 20 bytes.

Analog minimum required size of 4 bytes will be stored at address 0xA2F214.

Timers dummy address will be 0xA2F200 and specified with a size of zero.

Messages will be stored at address 0x A2F218 with the exact required size of 256*3 bytes.

Then the workbench entered string should be:

A2F200,14,A2F214,4,A2F200,0,A2F218,300
--

SYSTEM function call

If most of the application variables need to be stored, then the facilities of the SYSTEM function should be used to deal with a complete set of variables (for more information on the SYSTEM function see the user's guide). Note that here, the backup and the restoration are managed by the programmer at application level.

First of all you need to define the memory backup location for a specified type of variable or all types of variables:

```
<new_address> := SYSTEM(SYS_INITxxx,<address>);
```

where:

- <address> is the memory backup address location (16# value for Hexadecimal format). It must be an even address, otherwise the operation fails.
- SYS_INITxxx can be:
 - * SYS_INITBOO to define memory backup location for all boolean variables.
 - * SYS_INITANA to define memory backup location for all analog variables.
 - * SYS_INITTMR to define memory backup location for all timer variables.
 - * SYS_INITALL to define memory backup location for all boolean, analog and timer variables.
- <new_address> gets the next free address, that is to say <address> + backed up variables size (in bytes) according to SYS_INITxxx. This enable to check the required memory backup size. If the operation has failed <new_address> gets zero.

Then you may ask for a backup. This procedure can be called at any time in the application, the backup will be done at the end of the current cycle and once only. If the hardware delivers a boolean input, or a C function to inform the user when the power fail arrives, and allows at least one ISaGRAF cycle delay before crash down, the backup might only be made when the power fail is detected:

```
<error> :=SYSTEM(SYS_SAVxxx,0);
```

where:

- SYS_SAVxxx can be:
 - * SYS_SAVBOO to ask for all boolean variables backup.
 - * SYS_SAVANA to ask for all analog variables backup.
 - * SYS_SAVTMR to ask for all timer variables backup.
 - * SYS_SAVALL to ask for all boolean, analog and timer variables backup.
- <error> gets an error status different from zero when operation has failed (SYS_INITxxx has not been called).

Finally you may want to restore variables. This procedure can be called at any time in the application, the restoration will be done at the end of the current cycle and once only. To ensure the data backed up are valid, an analog variable should be set to a constant value used as a signature:

<error> := SYSTEM(SYS_RESTxxx,0);

where:

- SYS_RESTxxx can be:
 - * SYS_RESTBOO to restore all boolean variables.
 - * SYS_RESTANA to restore all analog variables.
 - * SYS_RESTTMR to restore all timer variables.
 - * SYS_RESTALL to restore all boolean, analog and timer variables.
- <error> gets an error status different from zero when operation has failed (SYS_INITxxx has not been done).

The following is a sum up of commands of the SYSTEM function to manage backup variables:

command		Meaning
pre-defined keyword	Value	
SYS_INITBOO	16#20	init boolean back up
SYS_SAVBOO	16#21	save booleans
SYS_RESTBOO	16#22	restore booleans
SYS_INITANA	16#24	init analog back up
SYS_SAVANA	16#25	save analogs
SYS_RESTANA	16#26	restore analogs
SYS_INITTMR	16#28	init timer back up
SYS_SAVTMR	16#29	save timers
SYS_RESTTMR	16#2A	restore timers
SYS_INITALL	16#2C	init all types back up
SYS_SAVALL	16#2D	save all types
SYS_RESTALL	16#2E	restore all types

command (pre-defined keyword)	Argument	Return value
SYS_INITxxx	memory address	next free address
SYS_SAVxxx	0	zero if OK
SYS_RESTxxx	0	zero if OK

▬ **Customized implementation**

Finally, using C functions or function blocks, you may develop specific user's procedures to have access to a battery backed up memory, to store and restore variables at any moment in the application.

Examples:

1) Procedure dedicated to an application:

backup, restore_temp, restore_date, restore_cpt would be C user's procedures.

backup(temperature, date, cnt); store 3 critical data

temperature := **restore_temp**(); restore temperature

date := **restore_date**(); restore date

cnt := **restore_cnt**(); restore counter

2) General purpose procedures:

backup_init, backup, backup_link, restore would be C user's procedures.

save_id := **backup_init**(address, size); allocate a memory backed up array.

backup(save_id, cpt1, 3); save cpt1 as the 3rd element.

rest_id := **backup_link**(address, size) link backed up memory.

cpt1 := **restore**(rest_id, 3); restore backed up value of cpt1.

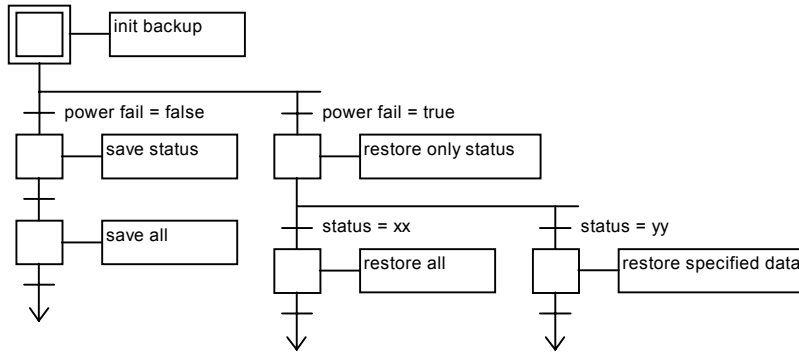
C.9.3 Program state backup

It could be possible to store every state of every application program, but it seems dangerous to restore every program in the state it was at the last backup, for at least 3 reasons:

- Some processes require specific operations before restarting
- Dealing with every status of a complete application is tedious
- Some external resources such as C programs, peripherals, etc. cannot be automatically restarted

The best solution seems to backup analog or boolean variables to describe the status of the process when the programmer thinks the restart stages will be able to use these information. Then it should be possible from an uncompleted but intelligent 'image' of the process to start, kill or freeze SFC programs and to initialize variables to put the application in the adequate state. But no automatic start up procedure can be provided by ISaGRAF.

Example:



C.10 Appendix: Error list and description

Error list:

Code	Message	Type
1	cannot allocate memory for run time data base	system
2	incorrect application data base (Motorola/Intel)	application
3	cannot allocate communication mailbox	system
4	cannot link kernel data base	system
5	time-out sending question to kernel	system
6	time-out waiting answer from kernel	system
7	cannot init communication	system
8	cannot allocate memory for retained variables	application
9	application stopped	application
10	too many simultaneous N or P actions	application
11	too many simultaneous setting actions	application
12	too many simultaneous resetting actions	application
13	unknown TIC instruction	application
16	cannot answer read data request	system
17	cannot answer write data request	system
18	cannot answer debugger session request	system
19	cannot answer modbus request	system
20	cannot answer debugger application request	system
21	cannot answer debugger	system
23	unknown request code	system
24	Ethernet communication error	system
25	communication synchro error	system
28	cannot allocate memory for application	system
29	cannot allocate memory for application update	system
30	unknown OEM key code	application
31	cannot init boolean input board	application
32	cannot init analog input board	application
33	cannot init message input board	application
34	cannot init boolean output board	application
35	cannot init analog output board	application
36	cannot init message output board	application
37	cannot input boolean board	application
38	cannot input analog board	application
39	cannot input message board	application
40	cannot output boolean output variable	application
41	cannot output analog output variable	application
42	cannot output message output variable	application
43	cannot operate boolean variable	application
44	cannot operate analog variable	application
45	cannot operate message variable	application
46	cannot open board	application

47	cannot close board	application
50	cannot overwrite boolean output variable	program
51	cannot overwrite analog output variable	program
52	cannot overwrite message output variable	program
61	unknown system request code	program
62	sampling period overflow	program
63	user function not implemented	application
64	integer divided by zero	program
65	conversion function not implemented	application
66	function block not implemented	application
67	standard function not implemented	application
68	real divided by zero	program
69	invalid operate parameters	application
72	application symbols cannot be modified	application
73	cannot update: different set of boolean variables	application
74	cannot update: different set of analog variables	application
75	cannot update: different set of timer variables	application
76	cannot update: different set of message variables	application
77	cannot update: cannot find new application	application
> 100	specific OEM error code, ask your supplier for more details	

The 3 error types correspond to the different sources of trouble:

– **System errors:**

Such problems are probably due to target software or hardware, not to application setting or to program execution.

Try a hard reset (power off) of your target, and try to run other applications.

These errors should be reported to your ISaGRAF support.

– **Application errors:**

Such problems are due to application parameters, size or content.

These errors should disappear when loading a known and previously validated application. If the problem still appears, it becomes a system error as listed above.

– **Program errors:**

Such problems are due to a particular sequence of program.

These kind of error should disappear when the application is started in cycle by cycle mode, or when the critical program is stopped.

Errors description:

1. cannot allocate memory for run time data base	system
---	---------------

No memory available. Check the hardware.

2. incorrect application data base (Motorola/Intel)	application
--	--------------------

The application file, downloaded or backed up is not correct. This error appears if the application is generated for INTEL and downloaded on MOTOROLA (and reverse) or if the file has been altered.

3. cannot allocate communication mailbox	system
---	---------------

This error is produced by the communication task if it cannot allocate space 3 for inter task communication.

4. cannot link kernel data base	system
--	---------------

This error is produced by the communication task if it cannot find a kernel running with the slave number specified in its command line.

5. time-out sending question to kernel	system
---	---------------

The communication task cannot send a request to the kernel. The kernel is probably not running or busy.

6. time-out waiting answer from kernel	system
---	---------------

The communication task cannot receive an answer from the kernel. The kernel is probably not running or busy.

7. cannot init communication	system
-------------------------------------	---------------

This warning is produced when the communication layer cannot initialize the physical link. This warning is also displayed if no communication path is specified. This does not prevent the target from running correctly, but it cannot communicate.

8. cannot allocate memory for retained variables	application
---	--------------------

ISaGRAF cannot manage retained variables. There may be two reasons for such a problem:

- the string passed as a parameter to the host target is not syntactically correct
- the size of memory specified for each block is not sufficient

You have to verify the syntax of your 'retain variable' parameter, and you can try with a reduced number of retained variables.

9. application stopped	application
-------------------------------	--------------------

This warning is produced every time the application is stopped from the debugger.

10. too many simultaneous N or P actions	application
---	--------------------

This error occurs if one of the target cycles has to execute too many non stored, pulse actions or cyclic blocks. It is possible to locate the trouble in CC mode. The maximum number of simultaneous actions is 2 + 4 per SFC program.

11. too many simultaneous setting actions	<i>application</i>
--	---------------------------

This error occurs if one of the target cycles has to execute too many setting actions (executed when a step becomes active). Proceed as mentioned above.

12. too many simultaneous resetting actions	<i>application</i>
--	---------------------------

This error occurs if one of the target cycles has to execute too many resetting actions (executed when a step is de-activated). Proceed as mentioned above.

13. unknown TIC instruction	<i>application</i>
------------------------------------	---------------------------

The kernel has detected something wrong in the application code (called Target Independent Code), in a program. There are two possible explanations:

- an external program is probably writing into application code. Try to locate the crash in CC mode and make sure no IO interface has wrong parameters.
- your target has a reduced set of instructions, and your application uses a non authorized instruction or variable type.

16. cannot answer read data request	<i>system</i>
--	----------------------

A communication error is detected answering specific ISaGRAF Modbus request function code 18 (file read). Check connection and system configuration on both target and master sides.

17. cannot answer write data request	<i>system</i>
---	----------------------

A communication error is detected answering specific ISaGRAF Modbus request function code 17 (file write). Check connection and system configuration on both target and master sides.

18. cannot answer debugger session request	<i>system</i>
---	----------------------

A communication error is detected answering a debugger request. Check connection and system configuration on both target and master sides.

19. cannot answer modbus request	<i>system</i>
---	----------------------

A communication error is detected answering a Modbus request. Check connection and system configuration on both target and master sides.

20. cannot answer debugger application request	<i>system</i>
---	----------------------

A communication error is detected answering a debugger request. Check connection and system configuration on both target and master sides.

21. cannot answer debugger	<i>system</i>
-----------------------------------	----------------------

A communication error is detected answering a debugger request. Check connection and system configuration on both target and master sides.

23. unknown request code	system
---------------------------------	---------------

A debugger request makes no sense.

24. Ethernet communication error	system
---	---------------

This appears each time the connection is closed when the debugger is closed: the system is working OK. Otherwise it means that an Ethernet communication error is detected. Check connection and system configuration on both target and master sides.

A second field is given, it can be:

- 1: error while sending or receiving
- 2: error while creating the socket
- 3: error while binding or listening the socket
- 4: error while accepting a new client

25. communication synchro error	system
--	---------------

Bad synchronization between the communication task on the target and the master. Check connection and system configuration (communication parameters) on both target and master sides.

28. cannot allocate memory for application	system
---	---------------

No memory available. Check the hardware, according to the size of the application.

29. cannot allocate memory for application update	system
--	---------------

No memory available. Check the hardware, according to the size of the application.

30. unknown OEM key code	application
---------------------------------	--------------------

The application is using a board which manufacturer code is not recognized by the target. Check the I/O connection in the workbench and use 'VIRTUAL' attribute to locate the incorrect board. Your workbench library may not correspond to your target version.

31. cannot init boolean input board	application
--	--------------------

A boolean input board init has failed. Check the I/O connection in the workbench and the parameters of your boolean input boards.

32. cannot init analog input board	application
---	--------------------

An analog input board init has failed. Check the I/O connection in the workbench and the parameters of your analog input boards.

33. cannot init message input board	<i>application</i>
--	---------------------------

A message input board init has failed. Check the I/O connection in the workbench and the parameters of your message input boards.

34. cannot init boolean output board	<i>application</i>
---	---------------------------

A boolean output board init has failed. Check the I/O connection in the workbench and the parameters of your boolean output boards.

35. cannot init analog output board	<i>application</i>
--	---------------------------

An analog output board init has failed. Check the I/O connection in the workbench and the parameters of your analog output boards.

36. cannot init message output board	<i>application</i>
---	---------------------------

A message output board init has failed. Check the I/O connection in the workbench and the parameters of your message output boards.

37. cannot input boolean board	<i>application</i>
---------------------------------------	---------------------------

An error has been detected while refreshing a boolean input board. Check the I/O connection in the workbench and board parameters.

38. cannot input analog board	<i>application</i>
--------------------------------------	---------------------------

An error has been detected while refreshing an analog input board. Check the I/O connection in the workbench as well as board parameters.

39. cannot input message board	<i>application</i>
---------------------------------------	---------------------------

An error has been detected while refreshing a message input board. Check the I/O connection in the workbench and board parameters.

40. cannot output boolean output variable	<i>application</i>
--	---------------------------

An error has been detected while updating an output boolean variable. Check the I/O connection in the workbench and board parameters.

41. cannot output analog output variable	<i>application</i>
---	---------------------------

An error has been detected while updating an output analog variable. Check the I/O connection in the workbench and board parameters.

42. cannot output message output variable	<i>application</i>
--	---------------------------

An error has been detected while updating an output message variable. Check the I/O connection in the workbench and board parameters.

43. cannot operate boolean variable	<i>application</i>
--	---------------------------

An error has been detected executing an OPERATE call to a boolean variable. Verify your OPERATE parameters and board user's note.

44. cannot operate analog variable	<i>application</i>
---	---------------------------

An error has been detected executing an OPERATE call to a analog variable. Verify your OPERATE parameters and board user's note.

45. cannot operate message variable	<i>application</i>
--	---------------------------

An error has been detected executing an OPERATE call to a message variable. Verify your OPERATE parameters and board user's note.

46. cannot open board	<i>application</i>
------------------------------	---------------------------

The application is using a board reference which is unknown in the target. Check the I/O connection in the workbench. Your workbench library may not correspond to your target version.

47. cannot close board	<i>application</i>
-------------------------------	---------------------------

The application is using a board reference which is unknown in the target. Check the I/O connection in the workbench.

50. cannot overwrite boolean output variable	<i>program</i>
---	-----------------------

Two SFC sequences are writing the same boolean output variable in the same target cycle. This should be avoided to prevent hazardous behavior of the I/Os. In case of such a conflict, the priority is given to the highest program in the hierarchy. If the two SFC programs are located at the same level, the result is unpredictable.

51. cannot overwrite analog output variable	<i>program</i>
--	-----------------------

Two SFC programs are writing the same analog output variable in the same target cycle. See above comment.

52. cannot overwrite message output variable	<i>program</i>
---	-----------------------

Two SFC programs are writing the same message output variable in the same target cycle. See above comment.

61. unknown system request code	<i>program</i>
--	-----------------------

A program is using the SYSTEM call with an invalid code.

62. sampling period overflow***program***

The target cycle period is longer than specified in the workbench menu.

On a multitasking system, this means that there is not enough CPU time to execute a cycle, even if the 'current cycle duration' is less than the specified period.

On a single task system, this always means that there are too many operations in one of the target cycle.

There are many possible ways to remove this warning:

- reduce the number of operations performed at the instant where the warning is detected.
- reduce number of tokens, of valid transitions, optimize complex processing, etc.
- reduce other tasks CPU load to give more time to ISaGRAF.
- reduce communication traffic to give more time to ISaGRAF.
- use dynamic cycle duration modification to adapt the cycle duration to different process stages.
- set cycle duration to zero to let the ISaGRAF kernel run as fast as possible, without any overflow checking.

63. user function not implemented***application***

A program is using a C function which is unknown in the target. Your workbench library may not correspond to your target version.

64. integer divided by zero***program***

A program tries to divide an integer analog by zero. Your application should prevent such an event which may have unpredictable effects.

When this occurs, ISaGRAF places the maximum analog value as the result.

When the operand is negative, the result is inverted.

65. conversion function not implemented***application***

A program is using a C conversion function which is unknown in the target. Your workbench library may not correspond to your target version.

When this occurs, ISaGRAF does not convert the value.

66. function block not implemented***application***

A program is using a C function block which is unknown in the target. Your workbench library may not correspond to your target version.

67. standard function not implemented***application***

A program is using a function block which is unknown in the target, although it is supposed to be available on most targets. Ask your supplier.

68. real divided by zero***program***

A program tries to divide a real analog by zero. Your application should prevent such an event which may have unpredictable effects.
When this occurs, ISaGRAF places the maximum real analog value as the result.
When the operand is negative, the result is inverted.

69. invalid operate parameters	<i>application</i>
---------------------------------------	---------------------------

Your application uses an OPERATE call with wrong parameters. This is normally filtered by the compiler. It could be a timer parameter, or a variable which is not an input or output.

72. application symbols cannot be modified	<i>application</i>
---	---------------------------

Trying to make an application update, the modified application cannot be started because the symbols are different. One or more variables or instances of function blocks may have been added, removed or modified, compared to the current application.

73. cannot update: different set of boolean variables	<i>application</i>
--	---------------------------

The modified application cannot be started because some boolean variables have been added or removed, compared to the current application.

74. cannot update: different set of analog variables	<i>application</i>
---	---------------------------

The modified application cannot be started because some analog variables have been added or removed, compared to the current application.

75. cannot update: different set of timer variables	<i>application</i>
--	---------------------------

The modified application cannot be started because some timer variables have been added or removed, compared to the current application.

76. cannot update: different set of message variables	<i>application</i>
--	---------------------------

The modified application cannot be started because some message variables have been added or removed, compared to the current application.

77. cannot update: cannot find new application	<i>application</i>
---	---------------------------

The modified application cannot be found in memory, something wrong may have happened during the download.

D. Glossary

Action	List of statements or assignments executed when a step of an SFC program is active.
Action (FC)	Symbol of a Flow Chart diagram. An action represents a list of instructions to be performed when the dynamic flow encounters the action symbol.
Activity of a step	Attribute of a step which is marked by an SFC token. The actions attached to the step are executed according to its activity.
Analog	Type of variables. These are continuous integer or real variables.
Attribute	Class of variables. Available variable attributes are internal, input or output.
Begin section	Group of cyclic programs executed at the beginning of each target cycle.
Beginning step	First step of the body of a macro step. A beginning step is not linked to any preceding transition.
Boolean	Type of variables. Such variables can only take true or false values.
Boolean action	SFC action: a boolean variable is assigned with the activity signal of a step.
Breakpoint	Mark placed by the user at debug time, on an SFC step or transition. The target system stops when an SFC token is moved on a breakpoint.
C function	Function written with the "C" language, called from the ISaGRAF programs (written with other languages), in a synchronous way. C functions are delivered by CJ International, or developed by the user.
C language	High level literal language used to describe the computer operations, such as C functions and conversion functions.
C source code	Text file which contains the "C" source code of a function or a conversion function.
C source header	Text file which contains the "C" definitions and types required for the programming of a C function or a conversion function.
Cell	Elementary area of the graphic matrix for graphic languages such as SFC, FBD or LD.
Child SFC program	SFC program controlled by another SFC program, called its father.

Clearing a transition	Run time operation: all the tokens existing in the preceding steps are removed. A token is created into each of the following steps.
Coil	Graphic component of an LD program, used to represent the assignment of an output variable.
Comment	Text included in a program, having no impact on the execution of the program.
Comment (SFC)	Text attached to an SFC step or transition, having no impact on the execution of the program.
Common	Range of defined words. Such objects can be used in any program of any project.
Condition (for a transition)	Boolean expression attached to an SFC transition. The transition cannot be cleared when its condition is false.
Connector (FC)	FC graphic component, which represents a link, from a point of the flow chart to a FC action or test. The graphic symbol of a jump is an small circle, numbered with the reference of the destination element.
Constant expression	Literal expression used to describe a constant value. A constant expression is dedicated to one type.
Contact	Graphic component of an LD program. It represents the status of an input variable.
Conversion	Filter attached to an input or output analog variable. The conversion is automatically applied each time the variable is input or output.
Conversion function	"C" written function which describes a conversion. Such a conversion can be attached to any input or output analog variable.
Conversion table	Set of points which defines a linear (by segment) conversion. Such a conversion can be applied to any input or output analog variable.
Cross references	Information calculated by the ISaGRAF workbench about the dictionary of variables, and where those variables are used in a project.
Current result (IL)	Result of an instruction in an IL program. The current result can be modified by an instruction, or used to set a variable.
Cycle timing	Duration of the target execution cycle.
Cycle to cycle mode	Execution mode: in this mode, cycles are executed one by one, according to the orders given by the user of the debugger.
Cyclic	Attribute of a program which is always executed.

Decision (FC)	(Also called test) Flow chart symbol attached to a boolean expression. The flow is directed to either YES or NO symbol output depending on the state of the expression.
Defined word	Unique identifier used to replace any expression in a program.
Delayed operation (IL)	Operation of an IL program, executed when the "(" instruction occurs, later in the program.
Diary	Text file which contains all the notes about the changes made to one program. Each note is completed with its editing date.
Dictionary	Set of internal, input or output variables, and defined words, used in the programs of one project.
Edge	Change of a boolean variable. A rising edge means a change from false to true. A falling edge means a change from true to false.
End section	Group of cyclic programs executed at the end of each target cycle.
Ending step	Last step of the body of an SFC macro step. An ending step is not linked to any following transition.
Expression	Set of operators and identifiers which represents the evaluation of a value.
Father SFC program	SFC program which controls other SFC programs, called its children.
FBD	Functional Block Diagram language.
FC	Stand for "Flow Chart".
Flow Chart	Graphical language used to design a flow. The chart consists in action to be performed and decision allowing the selection between various paths in the flow. The Flow Chart language enables the drawing of loops to be executed on consecutive cycles
Function block	Graphic component of the FBD language, which represents a standard elementary function from the ISaGRAF libraries.
Functional Block Diagram	Graphic language: the equations are built with standard elementary blocks from the ISaGRAF library, linked together in the diagram.
Global	Range of variables or defined words. Such objects can be used in any program of one project.
Hierarchy	Architecture of a project, divided into several programs. The hierarchy tree represents the links between father programs and children programs.

I/O board	Hardware resource. An I/O board is characterized by a type and a direction (input or output). The parameters of an I/O board are described into the ISaGRAF library.
I/O channel	Single connection point of an I/O board. An I/O channel may receive one I/O variable.
I/O connection	Definition of the links between the variables of the application and the channels of the boards existing on the target system.
I/O variable	Variable connected to an input or output device. An I/O variable must be connected on a channel of an I/O board.
Identifier	Unique word used to represent a variable or a constant expression in the programming.
IL	Instruction List language.
Initial situation	Set of the initial steps of an SFC program, which represents the context of the program when it is started.
Initial step	Special step of an SFC program, which is activated when the program starts.
Input	Attribute of a variable. Such variables are linked to an input device.
Instruction	Elementary operation of an IL program, entered on one line of text.
Instruction List	Low level literal language, entered as a sequential list of elementary operations.
Integer	Class of analog variables, stored in a signed integer 32 bit format.
Internal	Attribute of a variable, which is not linked to an input or output device.
Jump to a step	SFC graphic component, which represents a link, from a transition to a step. The graphic symbol of a jump is an arrow, numbered with the reference of the destination step.
Keyword	Reserved word of the language.
Label (IL)	Identifier put at the beginning of an IL instruction line, which identifies the instruction, and can be used as an operand for the JMP operations.
Ladder Diagram	Graphic language mixing contacts and coils, for the design of boolean equations.
LD	Ladder Diagram language.
Level 1 of the SFC	Main description of an SFC program. Level 1 groups the chart (steps and transitions), and the attached comments.
Level 2 of	Detailed description of an SFC program. It is the description of the actions

the SFC	within the steps, and the boolean conditions attached to the transitions.
Library	Set of hardware or software resources, which can be directly inserted in any application.
Local	Range of variables or defined words. Such objects can be used in only one program of one project.
Locked I/O	Input or output variable, disconnected logically from the corresponding I/O device, by a "Lock" command sent by the user from the debugger.
Macro step	SFC graphic component. A macro step is a unique group of steps and transitions, represented as a unique symbol in the main chart, and described separately.
Matrix	Logical division of the editing area into rectangular cells, while editing a graphic language program.
Message	Type of variable. Such variables contains variable-length character strings.
Modbus	Master-Slave protocol. An ISaGRAF target system can be a Modbus slave for the link with an external system (such as supervisory systems) in a complete architecture.
Modifier (IL)	Single character put at the end of an IL operation keyword, which modifies the meaning of the operation.
Network address	Optional hexadecimal address freely defined for each variable. This address is used by the Modbus protocol when the target system is connected to an external system.
Non-stored action	SFC action: it is a list of statements, executed at each target cycle, when the corresponding step is active.
OEM key code (I/O board)	Hexadecimal 16 bit code defined for each I/O board of the ISaGRAF library. The OEM code identifies the supplier of the board.
OEM parameter (I/O board)	I/O board parameter, defined by the designer of the board. It can be a constant, or a variable parameter entered by the user during the I/O connection.
Operand (IL)	Variable or constant expression processed by an elementary IL instruction.
Operation (IL)	Basic instruction of the IL language. An operation is generally associated to an operand in an instruction.
Output	Attribute of a variable. Such variables are linked to an output device of the target machine.

Parameter (C function)	Value given as an input to a "C" function. A parameter is characterized by a type.
Parameter (I/O board)	User defined or constant parameter of a standard I/O board. A user defined parameter is entered by the programmer during the I/O connection.
Parent program	Program written in any language, which controls (calls) another non-SFC program, called its sub-program.
Power rail	Main left and right vertical rails at the extremities of an ladder diagram.
Program	Basic programming unit in a project. A program is described with one language, and is placed in the hierarchy architecture of the project.
Project	Programming area, which groups all the information (programs, variables, target code...) for one ISaGRAF application.
Pulse action	SFC action: it is a list of statements executed only once when the corresponding step is activated.
Range	Set of programs that can use an object. Pre defined ISaGRAF ranges are common, global and local.
Real	Class of analog variables, stored in a floating IEEE single precision 32 bit format.
Real board	I/O board physically connected to an I/O device on the target machine.
Real time mode	Run time normal execution mode: the target cycles are triggered by the programmed cycle timing.
Reference number (SFC)	Decimal number (from 1 to 65535) which identifies an SFC step or transition in an SFC program.
Register (IL)	Current result of an IL sequence.
Return value of a sub-program	Value returned by a sub-program at the end of its execution. The return value is used in the operations of the owner program.
Run time error	Application error detected by the ISaGRAF target system at run time.
Section	Group of programs executed with the same dynamic rules.
Separator	Special character (or group of characters) used to separate the identifiers in a literal language. A separator may represent an operation.

Sequential Function Chart	Graphic language: the process is described as a set of steps, linked by transitions. Actions are attached to the steps. Transitions are detailed as boolean conditions.
Sequential section	Group of the programs of a project. The execution of those programs follows the dynamic rules of the SFC language.
SFC	Sequential Function Chart language.
ST	Structured Text language.
Statement	Basic ST complete operation.
Step	Basic graphic component of the SFC language. A step represents a steady situation of the process, and is drawn as a square. A step is referenced by a number. The activity of a step is used to control the execution of the corresponding actions.
String	Set of characters stored in a message variable.
Structured Text	High level structured literal language, combining assignments, high level structures such as If/Then/Else, and function calls.
Sub-program	Program written with any language but SFC, and called by another program, called its owner program.
Target	ISaGRAF target machine, which supports the ISaGRAF kernel software.
Target cycle	Set of the operations executed each time the ISaGRAF target system is activated. The cycles are triggered with programmable cycle timing.
Technical note	User's guide for an element of the ISaGRAF libraries (C function or function block, conversion function or I/O board). The technical note is written by the designer of the element.
Test (FC)	(Also called decision) Flow chart symbol attached to a boolean expression. The flow is directed to either YES or NO symbol output depending on the state of the expression.
Timer	Type of variables. Such variables contain time values, and can be automatically refreshed by the ISaGRAF system at run time.
Token (SFC)	Graphical marker used to show the active steps of an SFC program.
Toolbox	Small child window of an graphic editing tool window, which groups the main buttons for the selection of the graphic components.
Top level program	Program put at the top of the hierarchy tree. A top level program is activated by the system.
Transition	Basic graphic SFC component. A transition represents the condition between

different SFC steps. A transition is referenced by a number. A boolean condition is attached to each transition.

Type Class of variables which have the same format. Basic types are boolean, analog, timer and message.

Validity of a transition Attribute of a transition. A transition is validated when all the preceding steps are actives.

Variable Unique representation of elementary data which is processed in the programs of project.

Virtual board I/O board which is not physically connected to an I/O device of the target machine.

E. General index

-, B-238
 \$ sequence, B-171
 %, A-85, B-172
 &, B-235
) operation (IL), B-229
 *, B-239
 /, B-239
 :=, A-127
 := (ST assignment), B-214
 +, B-237
 <, B-243
 <=, B-244
 <>, B-246
 =, B-246
 =1, B-236
 >, B-244
 >=, B-245
 >=1, B-236
 l gain, B-233

A

ABS, B-271
 Absolute value, B-271
 Access right, A-147
 ACOS, B-275
 Action, A-42, A-46, B-182, B-187, B-192, B-193, D-409
 Activate, A-102
 Activity duration, B-178, B-220
 Activity of a step, B-177, B-178, B-190, B-220, D-409
 Addition, B-237
 Addition of messages, B-251
 Alias, A-55
 ANA, B-248
 Analog, B-169, B-170, B-173, C-353, C-354, D-409
 AND, B-235
 AND_MASK, B-240

AnyTarget, A-96
 appli.tst, C-316, C-325, C-336, C-344
 appli.x6m, C-325, C-336
 appli.x8m, C-316, C-344
 Application size, C-350
 Application size limit, C-317
 Arc cosine, B-275
 Arc tangent, B-277
 Archive, A-22, A-135, A-141, A-148
 Archive drive, A-142
 Archive file, A-142
 ARCREATE, B-298
 Argument, A-139
 Array creation, B-298
 Array reading, B-299
 Array writing, B-299
 ARREAD, B-299
 ARWRITE, B-299
 ASCII, B-289
 Assignment, B-233
 Assignment (in ST, =), B-214
 ATAN, B-277
 Attribute, D-409
 AVERAGE, B-265

B

Background picture, A-113
 Backup, A-22, A-135, A-141, A-142, A-148
 Backup file unit (VxWorks), C-328, C-331
 Bargraph, A-113
 Base, B-169
 Baud rate, A-31
 Begin, A-23, B-191
 Begin section, D-409
 Beginning step, A-34, A-36, B-182, D-409

Binary selector, B-288
 BinaryFile, A-95
 Bit field, A-114
 Bitmap, A-113
 BLINK, B-269
 Board, A-83, A-84
 Board parameter, A-85, A-137
 Board type, A-84
 Body of a macro step, B-182
 BOO, B-247
 Boolean, A-76, B-173, D-409
 Boolean action, A-39, B-182, D-409
 Breakpoint, A-103, A-105, D-409
 BY, B-218

C

C code, A-93, A-135
 C compiler, C-352, C-381
 C function, A-140, C-352, C-358, D-409
 C function block, A-140, C-352
 C language, C-352, C-354, C-356, C-360, C-368, C-370, C-381, D-409
 C source code, C-356, C-361, C-370, C-381, D-409
 C source header, C-354, C-360, C-368, C-381, D-409
 CAL operator (IL), B-231
 CASE, B-216
 Cat, B-251
 Cell, D-409
 Channel, A-84, A-85, A-86, A-137, A-149
 Channel comment, A-85
 CHAR, B-289
 Child, A-24, B-165
 Child SFC program, B-190, D-409
 Clearing a transition, B-189, D-410
 CLKRATE, C-327
 CMP, B-263
 Code generation, A-28, A-90
 Coil, A-49, A-58, B-202, D-410
 Coil direct, B-204
 Coil inverted, B-204
 Coil negative, B-207
 Coil positive, B-206
 Coil reset, B-206
 Coil set, B-205
 Coil type, A-53
 Comment, B-174, B-194, B-210, B-225, D-410
 Comment (SFC), B-177, B-178, D-410
 Common, A-141, D-410
 Communication, A-31, A-104, A-118, A-153, C-314, C-318, C-319, C-320, C-323, C-327, C-332, C-339, C-348, C-350
 Communication logical number, C-320, C-321, C-331
 Comparison, B-263
 Compile, A-28, A-90, A-134, A-138
 Compiler message, A-93
 Compiler option, A-28, A-119
 Compiler options, A-91
 Compression, A-142
 Condition, B-192
 Condition (for a transition), B-187, B-188, D-410
 Connection, A-59, A-60
 Connector, A-44, B-193, D-410
 Constant expression, B-169, D-410
 Contact, A-49, A-58, B-202, D-410
 Contact direct, B-202
 Contact inverted, B-203
 Contact negative edge, B-203
 Contact Positive edge, B-203
 Contact type, A-53
 Control panel, A-101
 Convergence, A-33, A-35, B-179
 Conversion, A-88, D-410
 Conversion ASCII -> character, B-289
 Conversion character -> ASCII, B-289
 Conversion function, A-140, C-352, C-353, D-410
 Conversion table, A-88, A-89, D-410
 Convert to boolean, B-247
 Convert to integer, B-248
 Convert to message, B-250
 Convert to real, B-249
 Convert to timer, B-250

Copy FBD, A-61
 Copy FC, A-46
 Copy LD, A-54
 Copy library, A-134
 Copy program, A-27
 Copy SFC, A-36
 Copy text, A-65
 Copy variable, A-75
 Corner, A-59
 COS, B-277
 Cosine, B-277
 Counter down, B-259
 Counter up, B-258
 Counter up/down, B-260
 Cross reference, A-99
 Cross references, A-29, D-410
 CTD, B-259
 CTU, B-258
 CTUD, B-260
 Current result (IL), B-225, B-226, D-410
 Curve, A-113, A-114, A-117
 Cut FBD, A-61
 Cut FC, A-46
 Cut LD, A-54
 Cut SFC, A-36
 Cut text, A-65
 Cut variable, A-75
 Cycle, A-129, B-164, B-168, C-312
 Cycle profiler, A-124
 Cycle time, C-317, C-326, C-337, C-346
 Cycle timing, A-28, A-103, A-124, B-252, C-354, C-358, C-365, D-410
 Cycle to cycle, A-28, A-103
 Cycle to cycle mode, D-410
 Cyclic, B-164, D-410

D

DAY_TIME, B-297
 DDE, A-109
 DDE (NT target), C-344, C-348, C-350
 Debug, A-30
 Debug workspace, A-30
 Debugger, A-101, A-121
 Decimal, B-170

Decision, A-42, A-45, A-46, B-192, D-411
 Declaration, A-26, A-72
 Defined word, A-72, A-76, B-175, D-411
 Delayed operation (IL), B-226, B-229, D-411
 DELETE, B-290
 Delete board, A-84
 Delete FBD, A-61
 Delete FC, A-46
 Delete LD, A-54
 Delete library, A-134
 Delete program, A-27
 Delete SFC, A-36
 Delete text, A-65
 Deleted style, A-63
 DERIVATE, B-268
 Descriptor, A-19, A-29
 Diagnosis, A-121
 Diary, A-26, D-411
 Dictionary, A-26, A-67, A-72, A-99, A-139, C-354, C-365, D-411
 Differentiation, B-268
 Direct coil, B-204
 Direct contact, B-202
 Directly represented variable, A-85, B-172
 Directory, A-153
 Disabled transition, B-189
 Disk, A-12
 Dissociate, A-115
 Divergence, A-33, A-35, B-179
 Divergence (FBD), B-197
 Division, B-239
 DO, B-217, B-218
 Document, A-20, A-29, A-144
 Download, A-102
 Dump, A-110

E

Edge, D-411
 Edge contact, B-203
 Edit project descriptor, A-20

ELSE, B-215, B-216
 ELSIF, B-215
 Embedded source code, A-119
 EN, A-50
 Enabled transition, B-189
 End, A-23, A-132, B-191
 End of cycle control (VxWorks), C-328, C-331
 End section, D-411
 END_CASE, B-216
 END_FOR, B-218
 END_IF, B-215
 END_REPEAT, B-217
 END_WHILE, B-217
 Ending step, A-36, B-182, D-411
 ENO, A-50
 EPROM, C-325, C-336
 EQ operator (IL), B-246
 Error, A-93
 Error message, A-70
 Ethernet, A-31
 Execute one cycle, A-103
 Execution order, A-61
 EXIT, B-219
 Exit key (NT target), C-346
 Exit key (on target), C-317
 Exponent, B-272
 Export, A-79
 Export function, A-28
 Export function block, A-28
 Expression, D-411
 EXPT, B-272

F

F_CLOSE, B-302
 F_EOF, B-302
 F_ROPEN, B-300
 F_TRIG, B-256
 F_WOPEN, B-301
 FA_READ, B-304
 FA_WRITE, B-305
 FALSE, A-76, B-169
 Father SFC program, B-190, D-411

FBD, A-57, B-196, C-359, C-367, D-411
 FBD comment, A-59
 FBD editor, A-57, A-67
 FC, A-42, B-191, D-411
 FC comment, A-44
 FC connector, A-44
 FC editor, A-42
 FC link, A-44
 FC sub-program, A-24, B-193
 FEDGE, B-213
 File
 end of file detection, B-302
 File close, B-302
 File open, B-300, B-301
 File read, B-304, B-307
 File write, B-305, B-309
 Find, A-37, A-46, A-54, A-61, A-65, A-70
 FIND, B-291
 Flow, A-44, B-191, B-193, B-195
 Flow Chart, A-42, B-191, D-411
 Flow Chart editor, A-42
 FM_READ, B-307
 FM_WRITE, B-309
 Font, A-146
 FOR, B-218
 From, A-97
 Function, A-23, A-26, A-134, A-138, B-165
 function block, B-212
 Function block, A-23, A-26, A-50, A-58, A-62, A-73, A-76, A-134, B-166, B-196, C-365, D-411
 Function block call in IL, B-231
 Function block instance, C-365
 Function call (ST), B-211
 Function call in IL, B-230
 Functional Block Diagram, B-196, D-411

G

gain 1, B-233
 Gallery, A-41

GE operator (IL), B-245
 GFREEZE, B-190, B-222
 GKILL, B-190, B-222
 Global, B-171, D-411
 Go to, A-37, A-46, A-65
 Goto, A-130
 Graphic, A-113, A-117
 Greater or equal, B-245
 Greater than, B-244
 Grid, A-51
 Group, A-13, A-21, A-115
 GRST, B-190, B-223
 GSTART, B-190, B-221
 GSTATUS, B-190, B-223
 GT operator (IL), B-244

H

Hierarchy, A-23, A-26, B-164, B-190,
 D-411
 History, A-20, A-29
 HYSTER, B-266
 Hysteresis, B-266, B-267

I

I/O, A-29, A-83, A-84, A-85, A-99, A-
 103, A-122, A-135, A-136, A-137, A-
 149, A-150
 I/O board, A-137, D-412
 I/O channel, D-412
 I/O channel OPERATE, B-253
 I/O complex equipment, A-136
 I/O configuration, A-19, A-135
 I/O connection, A-83, D-412
 I/O specific, B-192, B-193
 I/O variable, D-412
 I/O wiring, A-29
 Icon, A-114
 Icons, A-13
 Identifier, D-412
 If, A-131
 IF, B-194, B-215
 IL, A-65, A-112, B-187, B-188, B-225,
 D-412

IL editor, A-67
 Import, A-79
 Import function, A-27
 Import function block, A-27
 Initial situation, B-178, B-189, D-412
 Initial step, B-178, B-189, D-412
 Input, A-83, A-99, A-122, A-124, A-
 137, B-168, D-412
 INSERT, B-292
 Insert coil, A-52
 Insert contact, A-52
 Insert FBD, A-62
 Insert FBD element, A-59
 Insert FC element, A-43
 Insert file, A-65
 Insert rung, A-53
 Insert slot, A-83
 Insert variable, A-38
 Installation, A-12
 Instance, A-73, A-76
 Instruction, B-225, D-412
 Instruction List, B-225, D-412
 Integer, A-76, B-169, D-412
 INTEGRAL, B-267
 Interface, A-26, A-139
 Internal, D-412
 Inverted coil, B-204
 Inverted contact, B-203
 IO variable, C-353, C-354
 Is equal, B-246
 Is not equal, B-246
 ISA task (OS9), C-318
 ISA.EXE, C-314
 ISA.O (VxWorks), C-327, C-328
 isa_main, C-329, C-332
 isa_register_slave, C-328
 ISAGRAF.INI (NT target), C-338
 ISAKER task (OS9), C-319
 ISAKERET.O (VxWorks), C-327, C-
 330
 ISAKERSE.O (VxWorks), C-327, C-
 330
 ISAMOD (VxWorks), C-327
 ISAMOD.EXE, C-314
 ISANET task (OS9), C-320

ISASSR.O (VxWorks), C-327
 ISATST task (OS9), C-319
 ISAx0, C-324
 ISAx1, C-315, C-324
 ISAx1 (NT target), C-343
 ISAx1 (VxWorks), C-335
 ISAx2, C-324
 ISAx3, C-324
 ISAx4, C-324
 ISAx5, C-324
 ISAx6, C-316, C-324
 ISAx6 (NT target), C-343
 ISAx6 (VxWorks), C-335

J

JMP operator (IL), B-228
 Jump, A-50, A-58, B-197, B-208
 Jump to a step, A-34, B-179, D-412

K

Keyword, B-171, B-226, D-412

L

Label, A-58, A-130, B-197, B-208
 Label (IL), B-225, D-412
 Ladder Diagram, B-200, D-412
 Language, A-24, B-167
 LD, A-40, A-47, A-49, A-57, B-200, D-412
 LD editor, A-49
 LD operator (IL), B-227
 LE operator (IL), B-244
 LEFT, B-293
 Less or equal, B-244
 Less than, B-243
 Level 1 of the SFC, B-177, B-178, D-412
 Level 2, A-37, A-46
 Level 2 of the SFC, B-182, D-413
 Level of protection, A-147
 Library, A-22, A-27, A-28, A-84, A-99, A-123, A-133, A-141, C-352, D-413

Library manager, A-133, C-352, C-354, C-358, C-366
 LIM_ALARM, B-267
 LIMIT, B-283
 Link, A-31, A-59, A-60, A-104, A-118, A-153, B-191, B-193, B-195
 Link (FBD), B-197
 Link (LD), B-200
 Link (SFC), B-178
 List of variables, A-110, A-112, A-115
 Local, A-139, B-171, D-413
 Lock, A-103, A-150
 Locked I/O, D-413
 LOG, B-273
 Logarithm, B-273
 LT operator (IL), B-243

M

Macro step, A-34, A-36, B-181, D-413
 Make, A-28, A-90
 Mask on integer bits (and), B-240
 Mask on integer bits (not), B-242
 Mask on integer bits (or), B-241
 Mask on integer bits (xor), B-242
 Matrix, D-413
 MAX, B-283
 Maximum, B-283
 Memory, A-12
 Message, A-76, A-110, B-170, B-174, D-413
 Message concatenation, B-251
 Message length, B-294
 Metafile, A-113
 MID, B-294
 MIN, B-282
 Minimum, B-282
 MLEN, B-294
 MOD, B-284
 Modbus, D-413
 MODBUS, A-78, C-387
 Modification tracking, A-63
 Modified style, A-63
 Modifier (IL), B-225, B-226, D-413
 Modify variable, A-75

Modulo, B-284
 Move board, A-83
 Move FBD, A-60
 Move FC, A-45
 Move program, A-26
 Move project, A-19
 Move SFC, A-36
 Move SpotLight, A-115
 MSG, B-250
 Multi-applications, C-324, C-335, C-343
 Multiplexer with 4 entries, B-285
 Multiplexer with 8 entries, B-286
 Multiplication, B-239
 MUX4, B-285
 MUX8, B-286

N

N qualifier, A-38
 NE operator (IL), B-246
 NEG, B-234
 Negated link, A-59, A-60
 Negation, B-234
 Negation (FBD), B-198
 Negative coil, B-207
 Negative contact, B-203
 Network address, A-74, A-75, A-78, D-413
 New function, A-25
 New function block, A-25
 New library element, A-133
 New program, A-25
 New project, A-19
 New rung, A-51
 New variable, A-75
 Non stored, A-38
 Non-stored action, B-184, D-413
 Normal style, A-63
 NOT, A-59, A-60
 NOT_MASK, B-242
 NT (protection key), A-14

O

ODD, B-287

OEM key code, A-137, D-413
 OEM parameter, A-137
 OEM parameter (I/O board), D-413
 OF, B-216
 Off-delay timing, B-262
 On Line, A-30, A-101
 On line modification, A-103, A-106
 Open program, A-25, A-100
 Open project, A-20
 Operand (IL), B-225, B-226, D-413
 OPERATE I/O channel, B-253
 Operation (IL), B-225, B-226, D-413
 Optimiser, A-92
 OR, A-57, B-236
 OR_MASK, B-241
 OS-9 shell, C-326
 Other program, A-68
 Output, A-83, A-99, A-122, A-137, B-168, D-413
 Output window, A-70

P

P qualifier, A-38
 P0 qualifier, A-39
 P1 qualifier, A-39
 Page, A-146
 Parameter, A-26, A-139
 Parameter (C function), C-359, D-414
 Parameter (function block), C-367
 Parameter (I/O board), D-414
 Parent program, D-414
 Parenthesis, B-211, B-225, B-226
 Parity, A-31
 Parity test odd/even, B-287
 Password, A-20, A-87, A-134, A-147
 Paste FBD, A-61
 Paste FC, A-46
 Paste LD, A-54
 Paste SFC, A-36
 Paste text, A-65
 Paste variable, A-75
 Point, A-88, A-89
 Positive coil, B-206
 Positive contact, B-203

POW, B-273
 Power calculation, B-273
 Power rail, A-49, A-50, A-57, B-200, D-414
 Print, A-20, A-29, A-74, A-128, A-144, A-146
 Print program, A-68
 PrintTime, A-129
 Priority, C-348
 Priority level (NT target), C-342
 Program, A-23, A-67, A-124, B-164, D-414
 Program comment, A-25
 Program manager, A-23
 Program syntax, A-67
 Project, A-19, A-141, D-414
 Project descriptor, A-20, A-29
 Project document, A-20, A-29, A-144
 Project group, A-21
 Project list, A-19, A-21
 Project manager, A-19
 Project separators, A-19
 PROM, C-325, C-336
 Protection, A-20, A-87, A-134, A-147
 Protection key, A-14
 Protection level, A-147
 Pulse, A-38
 Pulse action, B-183, D-414
 Pulse timing, B-262

Q

Quick LD, A-40, A-47, A-49
 Quick LD editor, A-67

R

R (reset) operator (IL), B-228
 R_TRIG, B-256
 RAND, B-287
 Random number, B-287
 Range, A-72, A-74, D-414
 Real, A-76, B-170, D-414
 REAL, B-249
 Real board, A-84, D-414

Real time, A-28, A-103
 Real time mode, D-414
 REDGE, B-212
 Reference number, B-177, B-178, B-179, B-182, D-414
 Register (IL), B-225, D-414
 Rename library, A-134
 Renumber, A-37, A-46
 REPEAT, B-194, B-217
 Replace, A-37, A-46, A-54, A-61, A-65
 REPLACE, B-295
 Resize FC, A-45
 Resize SpotLight, A-115
 Resource, A-29, A-94
 Resource definition file, A-94
 Restore, A-22, A-135, A-141, A-142, A-148
 RET operator (IL), B-229
 Retain, C-394
 Return, A-50, A-58
 RETURN, B-197, B-207, B-215
 Return value, D-414
 RIGHT, B-296
 ROL, B-279
 ROR, B-280
 Rotation left, B-279
 Rotation right, B-280
 RS, B-255
 Run time error, B-252, D-414
 Rung, A-49, A-50, A-54, A-60
 Rung comment, A-51, A-55
 Rung label, A-52
 Run-time, A-28
 Run-time error, A-28, A-104

S

S (set) operator (IL), B-227
 Save list, A-110
 Scientific, B-170
 Script, A-125, A-127
 Section, A-23, D-414
 SEL, B-288
 Select FBD element, A-59
 Select FC element, A-44

- Select SpotLight, A-115
 SEMA, B-257
 SEMAPHORE, B-257
 Separator, B-210, D-414
 Sequential, A-23, B-164, B-177
 Sequential Function Chart, B-177, D-415
 Sequential section, D-415
 Serial link, A-31
 Set coil, B-205
 SFC, A-32, A-91, A-105, A-146, B-177, C-359, D-415
 SFC child, A-24, A-39, B-165
 SFC editor, A-32, A-67
 SFC evolution rules, B-189
 SFC gallery, A-41
 Shift left, B-281
 Shift right, B-281
 SHL, B-281
 SHR, B-281
 SIG_GEN, B-270
 Signal generator, B-270
 Simulator, A-30, A-122, A-124, A-125, C-353, C-358, C-365
 SIN, B-278
 Sine, B-278
 Slave number, A-31, C-314, C-318, C-319, C-320, C-321, C-328, C-330, C-339, C-347, C-350
 SlavesLink, C-333
 Slot, A-84, A-86
 Sort, A-75
 Source code, A-135
 SpotLight, A-113
 Spy, A-110, A-112, A-113
 Spy variable, A-110
 SQRT, B-274
 Square root, B-274
 SR, B-254
 SSR[x][1].space, C-336
 ST, A-40, A-65, A-112, B-210, C-359, C-366, D-415
 ST editor, A-67
 ST operator (IL), B-227
 Stack of integer analogs, B-264
 STACKINT, B-264
 Start, A-102
 Statement, B-210, D-415
 Step, A-32, A-37, A-105, B-177, D-415
 Stop, A-102
 String, B-170, D-415
 String length, B-294
 Structured Text, B-210, D-415
 Style, A-63, A-115
 Sub-program, A-24, B-165, B-186, B-189, B-198, D-415
 Sub-Program, B-193
 Sub-program call (ST), B-211
 Sub-program call in IL, B-230
 Sub-string delete, B-290
 Sub-string extraction (left), B-293
 Sub-string extraction (middle), B-294
 Sub-string extraction (right), B-296
 Sub-string find, B-291
 Sub-string insert, B-292
 Sub-string replace, B-295
 Subtraction, B-238
 Symbol table, A-155
 Symbols (application symbols), C-316
 SYSTEM, B-252
 System clock rate (VxWorks), C-327
 SYSTEM function, C-395
- ## T
- Table of contents, A-144
 TAN, B-279
 Tangent, B-279
 Target, A-91, A-96, D-415
 Target architecture, C-313
 Target cycle, D-415
 Technical note, A-84, A-134, C-352, C-354, C-359, C-366, D-415
 Terminal mode, C-326
 Test, A-42, A-45, A-46, A-101, A-122, B-192, D-415
 Text display, A-113
 Text editor, A-65
 TextFile, A-96
 THEN, B-215

Time unit, B-170
Time-out, A-31
Timer, A-76, B-170, B-174, D-415
TMR, B-250
To, A-97
TO, B-218
TOF, B-262
Token (SFC), B-177, D-415
Toolbox, D-415
Tools menu, A-29
Top level, A-23
Top level program, D-415
Touch, A-28, A-90
TP, B-262
Transition, A-32, A-37, A-105, B-178, D-415
TRUE, A-76, B-169
TRUNC, B-275
Truncate decimal part, B-275
TSK_FUNIT, C-328, C-331
TSK_NBTCKSCHED, C-328, C-331, C-337
tst_main_ex, C-332
TSTART, B-220
TSTOP, B-221
Type, A-72, A-74, A-83, A-99, A-137, B-169, D-416

U

ULongData, A-94
Unlock, A-103
UNTIL, B-217
Update, A-103
Upload, A-118

Upload (options), A-119
Upload (prepare), A-119

V

Validity of a transition, D-416
Variable, A-26, A-38, A-53, A-59, A-61, A-65, A-72, A-99, A-104, A-139, A-149, B-171, B-196, D-416
Variable name, B-171
VarList, A-95
Verify, A-28, A-90, A-138
Version number, A-102
Virtual board, A-84, A-150, D-416
Virtual boards (simulation with NT target), C-348, C-350
Virtual boards (simulation with NT), C-341

W

Wait, A-129
WHILE, B-194, B-217
WISAKER.EXE (NT), C-338

X

XOR, B-236
XOR_MASK, B-242

Z

Zoom, A-48, A-55, A-62

