



Gem Drive Programming Guide

WARNING

This is a general manual describing a series of servo amplifiers having output capability suitable for driving AC brushless sinusoidal servo motors.

Instructions for storage, use after storage, commissioning as well as all technical details require the MANDATORY reading of the manual before getting the amplifiers operational.

Please see [GD1 Installation Guide for the installation](#) and the [GD1 User Guide for the operation of the amplifier \(commissioning, configuration...\)](#).

Maintenance procedures should be attempted only by highly skilled technicians having good knowledge of electronics and servo systems with variable speed (EN 60204-1 standard) and using proper test equipment.

The conformity with the standards and the "CE" approval is only valid if the items are installed according to the recommendations of the amplifier manuals. Connections are the user's responsibility if recommendations and drawings requirements are not met.



Any contact with electrical parts, even after power down, may involve physical damage. Wait for at least 5 minutes after power down before handling the amplifiers (a residual voltage of several hundreds of volts may remain during a few minutes).

**ESD INFORMATION (ElectroStatic Discharge)**

INFRANOR amplifiers are conceived to be best protected against electrostatic discharges. However, some components are particularly sensitive and may be damaged if the amplifiers are not properly stored and handled.

STORAGE

- The amplifiers must be stored in their original package.
- When taken out of their package, they must be stored positioned on one of their flat metal surfaces and on a dissipating or electrostatically neutral support.
- Avoid any contact between the amplifier connectors and material with electrostatic potential (plastic film, polyester, carpet...).

HANDLING

- If no protection equipment is available (dissipating shoes or bracelets), the amplifiers must be handled via their metal housing.
- Never get in contact with the connectors.

**ELIMINATION**

In order to comply with the 2002/96/EC directive of the European Parliament and of the Council of 27 January 2003 on waste electrical and electronic equipment (WEEE), all INFRANOR devices have got a sticker symbolizing a crossed-out wheel dustbin as shown in Appendix IV of the 2002/96/EC Directive.

This symbol indicates that INFRANOR devices must be eliminated by selective disposal and not with standard waste.

INFRANOR does not assume any responsibility for any physical or material damage due to improper handling or wrong descriptions of the ordered items.

Any intervention on the items, which is not specified in the manual, will immediately cancel the warranty.

Infranor reserves the right to change any information contained in this manual without notice.

Contents

CONTENTS.....	3
CHAPTER 1. GEM DRIVE PROGRAMMING	5
1. INTRODUCTION	5
2. PROGRAMMING STRUCTURE	5
2.1. Fast Cyclical Task: FCT	5
2.2. User Cyclical Task: UCT	5
2.3. Movement Task: MVT	5
CHAPTER 2. INTEGRATED DEVELOPMENT ENVIRONMENT	6
1. INTRODUCTION	6
2. PROJECT MANAGEMENT	6
3. PROGRAMMING ENVIRONMENT INTERFACE	7
3.1. File Menu Commands	8
3.2. Edit Menu Commands	9
3.3. View Menu Commands	11
3.4. Program Menu Commands	12
3.5. Autocompletion	13
4. DRIVE PARAMETERS: EEDS, MNEMONIC, INDEX/SUBINDEX	14
5. MAKE A NEW USER PROGRAM	14
Step 1: Create a New User Program	14
Step 2: Edit and Add Project Source Files	17
Step 3: Customize the user program configuration	20
Step 4: Build and correct a User Program: Program Compilation	20
Step 5: Load the generated building output into the drive	22
Step 6: Start the program execution	22
Step 7: Finally, monitor the program execution	22
CHAPTER 3. ELEMENTS OF THE PROGRAMMING LANGUAGE: IEC 1131-3	23
1. OVERVIEW	23
2. USER PROGRAM ORGANIZATION	23
3. PROGRAMMING LANGUAGE INSTRUCTION LIST	25
4. PROGRAMMING LANGUAGE REFERENCE	27
4.1. Comment	27
4.2. Data Type	27
4.3. Constants	27
4.4. User Variables	27
4.4.1 User Variable Naming	27
4.4.2 User Variable Type	28
4.4.3 User Variable declaration	28
4.4.4 Array Variable	29
4.4.5 User Names Report	29
4.4.6 Local axis parameters versus remote device objects (SDO): ap, rdo, wdo	31
4.5. Assignment	32
4.6. Saving / Restoring user variables	32
4.7. Language Operators	33
4.7.1 Arithmetic Operators	33
4.7.2 Comparison Operators	34
4.7.3 Bits Handling	34
4.8. Conditional Statement: if_then_else	36
4.9. Iteration Statement (Loop): while, for	38
4.9.1 Count Controlled Loop	38
4.9.2 Condition Controlled Loop	39
4.10. Flow Control instructions: exit, goto label, return, halt	39
4.10.1 "exit" statement	39
4.10.2 "goto label" statement	40
4.10.3 "return" statement	40
4.10.4 "halt" statement	40

- 4.11. Wait a delay time 41
- 4.12. Wait until Condition 41
- 4.13. Start / Stop cyclical tasks 42
- 4.14. Math functions 42
 - 4.14.1 ABS 42
 - 4.14.2 SQRT 43
 - 4.14.3 SIN 43
 - 4.14.4 COS 44
 - 4.14.5 ATAN..... 44
- 4.15. Main block: Begin, End..... 45
- 4.16. Function: Definition, call, arguments, parameters, return 46

Chapter 1. Gem Drive Programming

1. Introduction

The **Gem Drive** is the newest **INFRANOR** product. One of the differences with the former Infranor drive ranges is its programmability. A basic programming language was defined for this purpose. This gives the **Gem Drive** more autonomy. This programming language is based on **IEC 1131-3** syntax.

To make system integration easy, Gem Drive offers a comprehensive and powerful high-level application programming interface, as well as our Gem Drive Studio for its setting, tuning and programming. These user friendly software tools are designed to help the user getting up-and-running quickly.

2. Programming structure

When a user program is defined for an axis, it may contain maximum three tasks:

- Movement Task : MVT
- Fast Cyclical Task : FCT
- User Cyclical Task : UCT

Only the MVT task is mandatory. The drive multitask kernel will execute all the tasks simultaneously.

2.1. Fast Cyclical Task: FCT

The « Fast Cyclical Task » or FCT is a very fast cyclical task. It is in correlation with regulation loops. As the Gem Drive position loop is sampled at 500µs the code size must be reduced to 40 basic operations. Blocking instructions or temporization is prohibited as well.

2.2. User Cyclical Task: UCT

The « User Cyclical Task » is less time critical than FCT. This task allows the user to cyclically execute larger codes. The task sample is fixed by the user according to the application need. Nevertheless, to avoid the system locking, blocking instructions and temporization are also prohibited for this task.

2.3. Movement Task: MVT

Contrarily to the above ones, the « Movement Task » is executed as a background task. Without any temporal restriction, all language operations are allowed.

Chapter 2. Integrated Development Environment

1. Introduction

Gem Drive Studio is an intuitive Windows® based, Integrated Development Environment (**IDE**) for developing motion applications. It is dedicated to the newest generation of **INFRANOR** programmable drives, called **Gem Drive**. The IDE provides flexible project management environment making the system easy to program.

2. Project management

A project is designed in a single entity for all user applications. It is organized by axis. For each axis, the user can setup the drive parameters and, at need, make an application program with an unique software. **Gem Drive Studio** includes also diagnostic tools like the oscilloscope, helping the user during the whole development process. This section will deal with the programming tools only. This includes:

- Edit text files. From the IDE, the user can create and modify source. The IDE provides extensive editing features such as Undo/Redo, Copy/Paste Find and Go To Line.
- Define and manage project program. Within the IDE, the user specifies the files that the compiler Tool processes when building application projects. He can create this project definition once or modify it to meet changing development needs.
- The IDE provides dialogs through which the user specifies options for the compiler Tool. These options control how the tool processes inputs and generates outputs. The user can define these options once or modify them to meet changing development needs.
- View and respond to project build results. The user can then go to the line of a compilation error message within a source file.
- Once the program successfully built, an executable output is generated. The user can then load it into the target drive.
- In the drive, the executable program can then be launched, depending on the initial configuration. The program execution may be controlled later by using the execution control tool from the PC.
- Finally, the monitoring tool helps the user to monitor the loaded program execution.

All dialog windows are easy-to-use and make configuring, changing and managing projects easy. Commands for programming the axes in the **Gem Drive Studio** environment are located at five places:

- The File menu,
- The Edit menu,
- The View menu
- The Program menu,
- The Short-cut menu, which is accessible with left and right mouse button clicks in the Project navigator window.

The project operations available at these places are as follows:

- “File Menu Commands”

This pull-down menu lets you handle or create a new project, as well as to handle source files for axis programming.

- “Edit Menu Commands”

This pull-down menu provides the user with all what he needs when editing source files.

- “View Menu Commands”

This pull-down menu includes some helpful commands for the program editor.

- “Program Menu Commands”

This pull-down menu allows the user to handle a user program, to control its building, loading and then executing and monitoring.

- “Short-Cut Menu Commands”

From the Program menu, left and right mouse button clicks on the program items of the project navigator offer the user a direct access to the programming windows.

3. Programming environment interface

User programs can be created, viewed or modified by using the IDE editor. This is a powerful editor including a lot of interesting features helping the user to develop complex applications. The main editor features are:

- Language syntax highlighting
- OLE Drag and Drop
- Bookmarks
- Standard editing functionalities such as cut, copy and paste, find and replace, undo and redo.

The **Gem Drive Studio** environment commands are available through the menus shown in Figure 3-1.

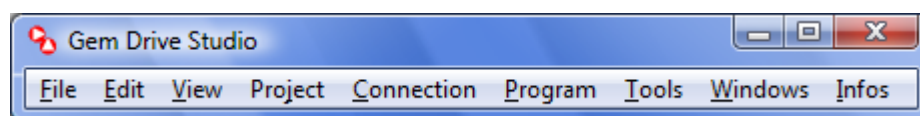


Figure 3-1: **Gem Drive Studio** Menus.

3.1. File Menu Commands

The **GDS File Menu** is shown in Figure 3-2. By using this menu, the user can create a new program file, open an existing one or save file changes. He can also open a list of files by using a drag-and-drop method applied to files dragged e.g. from a Windows Explorer and dropped on Gem Drive Studio.

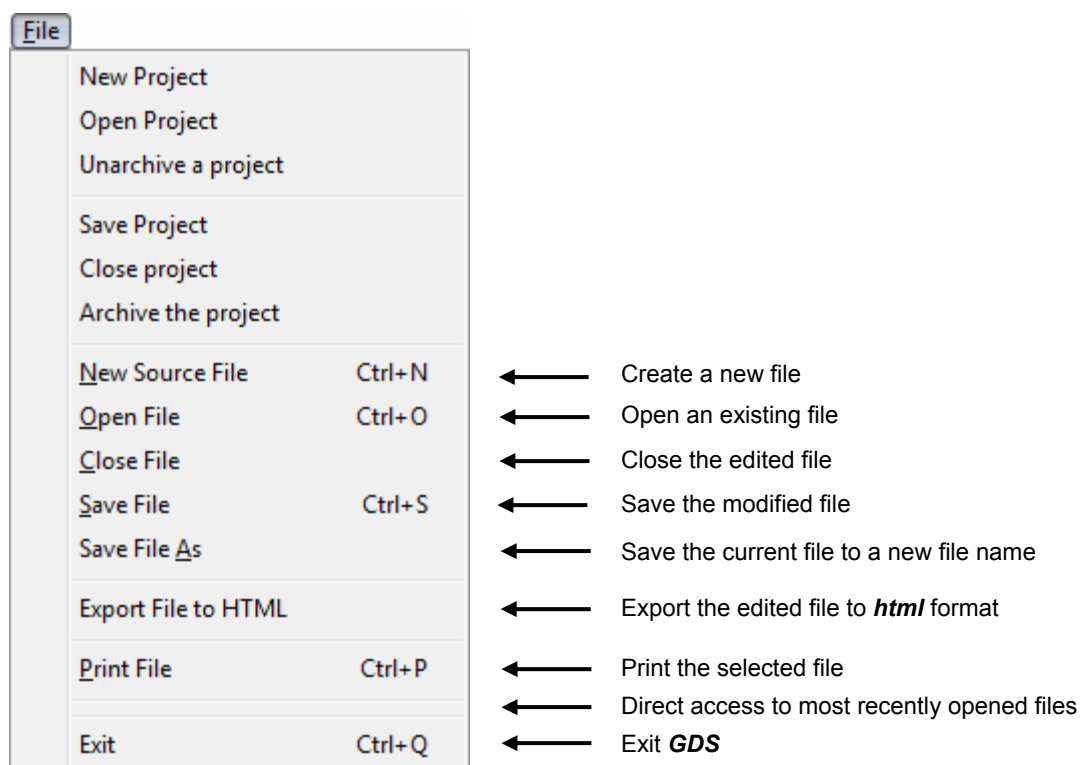


Figure 3-2: **GDS File Menu**.

An edited file can be exported to html format by using the **“Export File to HTML”** command. As programs are edited with syntax coloring, the html format allows keeping the program’s syntactic color, giving the user comprehensible printable programs. Programs can also be viewed by external Windows applications like “Microsoft Windows Explorer”.

The **“Print File”** command is used to print an active file. This command opens a Print dialog window where the user can select a destination printer and specify the range of pages to be printed, the number of copies and other printer setup options.

Most recently used files are listed at the bottom of the *File Menu*. The list contains the last four documents opened in the order last opened, first listed. Each file can be opened by clicking on its name within the list.

Use the **“Exit”** command to end your environment session. The IDE environment will prompt you to save any modified documents or project changes.

3.2. Edit Menu Commands

The **Gem Drive Studio** environment *Edit Menu* provides most standard windows edit commands. The Edit contains the commands that appear in Figure 3-3:

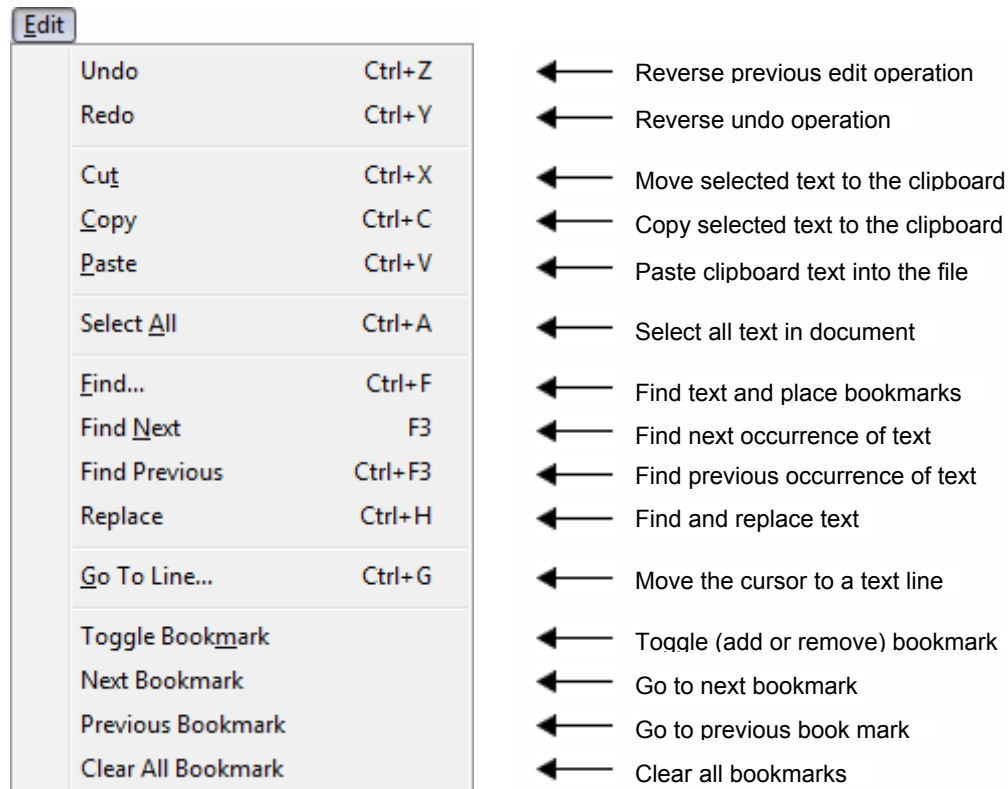


Figure 3-3: **GDS Edit Menu**.

Use the multilevel **“Undo”** command (*Ctrl Z*), to reverse the last editing action and use the **“Redo”** command (*Ctrl Y*), to reverse the previous **“Undo”** commands in reverse order.

The **“Cut/Copy/Past”** commands perform the standard tasks of cutting, copying text to the clipboard and pasting from the clipboard contents into the document at its insertion point.

With the **“Select All”** command (*Ctrl A*), the user can select all the document text before cutting or copying for example.

The **“Find”** commands (*Ctrl F...*) are used to search a text or a regular expression in the active document.

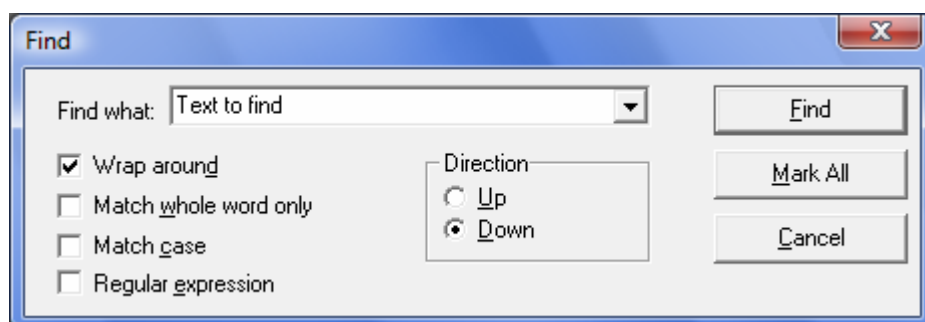


Figure 3-4: **GDS Find dialog box**.

This dialog box allows the user to set the **“Find What”** text, case sensitivity and search direction features for the Find command. The **“Wrap Around”** selection lets the user wrap around the end of the file.

The user can also set bookmarks by using the **“Mark All”** command. This lets the user mark all instances of text, wherever they occur, in his code (see explanation below).

The **“Replace”** command (*Ctrl H*) is used to replace a specified text with a given one.

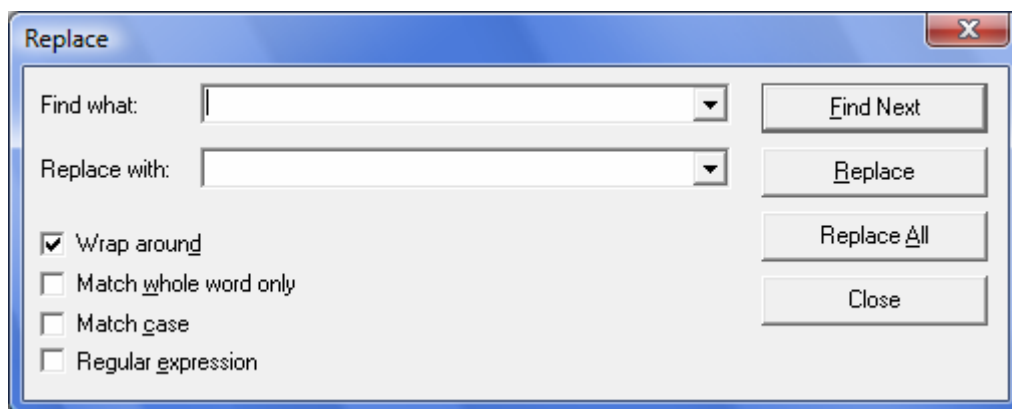


Figure 3-5: GDS Replace dialog box.

Regular expressions are non-alphabetic characters that are used to control a search in Find/Replace operations as indicated in the table below:

.	Matches any character
\ (Marks the start of a region for tagging a match.
\)	Marks the end of a tagged region.
\ n	Where n is 1 through 9 refers to the first through nine tagged regions when replacing. For example, if the search string was <code>Axis\[1-9\]XXX</code> and the replace string was <code>Slave\1YYY</code> , when applied to <code>Axis2XXX</code> this would generate <code>Slave2YYY</code> .
\ <	Matches the start of a word.
\ >	Matches the end of a word.
\ x	Allows to use a character x that would otherwise have a special meaning. For example, <code>\[</code> would be interpreted as <code>[</code> and not as the start of a character set.
[...]	Indicates a set of characters, for example, <code>[abc]</code> means any of the characters a, b or c. Ranges can also be used, e.g. <code>[a-z]</code> for any lower case character.
[^ ...]	The complement of the characters in the set. For example, <code>[^A-Za-z]</code> means any character except an alphabetic character.
^	Matches the start of a line (unless used inside a set. See above).
\$	Matches the end of a line.
*	Matches 0 or more times. For example, <code>Axis2*_S</code> matches <code>Axis_S</code> , <code>Axis2_S</code> , <code>Axis22_S</code> , <code>Saaam Axis222_S</code> and so on.
+	Matches 1 or more times. For example, <code>DInput1+_M</code> matches <code>DInput1_M</code> , <code>DInput11_M</code> , and so on.

Table 3-1: Regular Expression Operators.

The **“Go To Line”** command (*Ctrl G*), is used to move the cursor to the specified line and/or column number in the active document window. This is helpful when searching, for example, an error line after a program compilation.

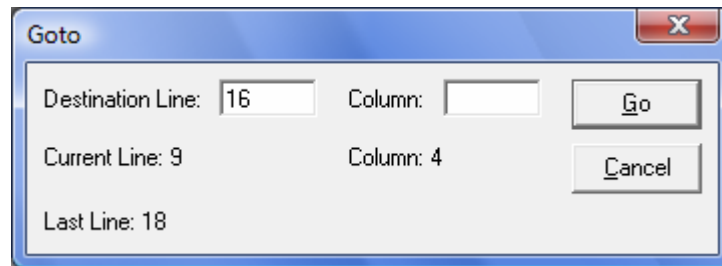



Figure 3-6 : Goto line dialog box.

Using Bookmarks

Bookmarks allows marking instances of text wherever they occur in the code. The user defines bookmarks by using the **“Toggle Bookmark”** in the *Edit Menu* or with the **“Find”** dialog box, **“Mark All”** button (see above). For instance, if the user wants to see every instance where he uses `MyVar` in his code, he can Find the first occurrence of `MyVar`, Mark All, and then quickly jump forward or backward through his file by using **“Next Bookmark”/“Previous Bookmark”** to see every occurrence of `MyVar`. Bookmarks are indicated by a  on the left of the editor window, but the user has to previously check the **“Bookmarks”** item in the *View Menu*. The **“Clear All Bookmarks”** command is used to clear all bookmarks.

3.3. View Menu Commands

This menu allows to configure the editor display.



Figure 3-7: **GDS** View Menu.

The **“Word Wrap”** command allows the control of the line wrapping. When the user enables line wrapping, lines wider than the window width are continued on the following lines. Lines are broken after space or tab characters. The horizontal scroll bar does not appear when wrap mode is on. By default, line wrapping is off.

Use the **“Line Numbers”** command to display Line Numbers and the **“Bookmarks”** command to display Bookmarks in the left margin of the text. The current line and column are also shown in the status bar of the GDS main window when the editor is active.

3.4. Program Menu Commands

This menu provides all programming commands for the selected axis.

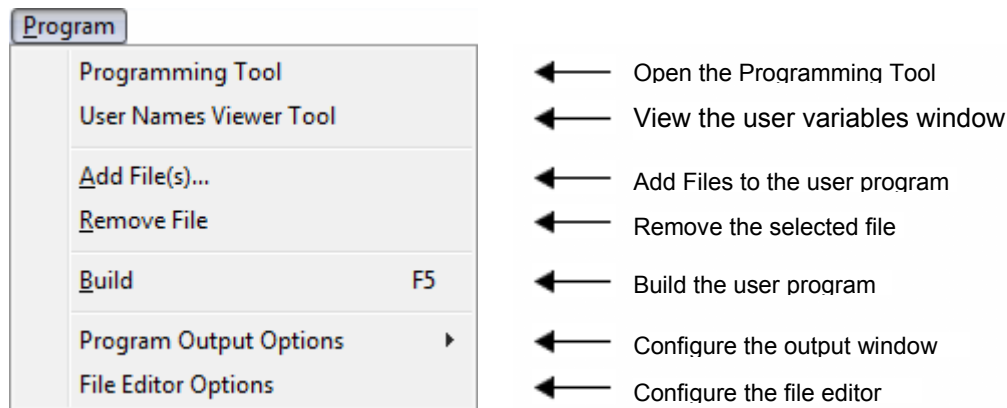


Figure 3-8: **GDS Program Menu.**

The **“Programming Tool”** command opens the Programming tool window while the **“User Names Viewer Tool”** command is used to view user variables window ([see more information in section 5](#)).

“Add Files” and **“Remove File”** commands have to be used for adding/removing files to/from the user program. To remove a file, select it first in the navigator window.

The **“Build”** (F5) command compiles all user program files and generate the executable output file.

Use the **“Program Output Options”** command to configure the output window display as the text font and colors, while the **“File Editor Options”** command provides you the possibility to configure the style of different elements of the file editing as keywords, numbers font and colors... ([see Figure 3-10](#)).

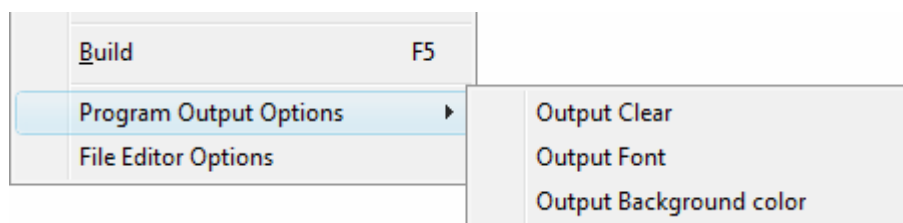
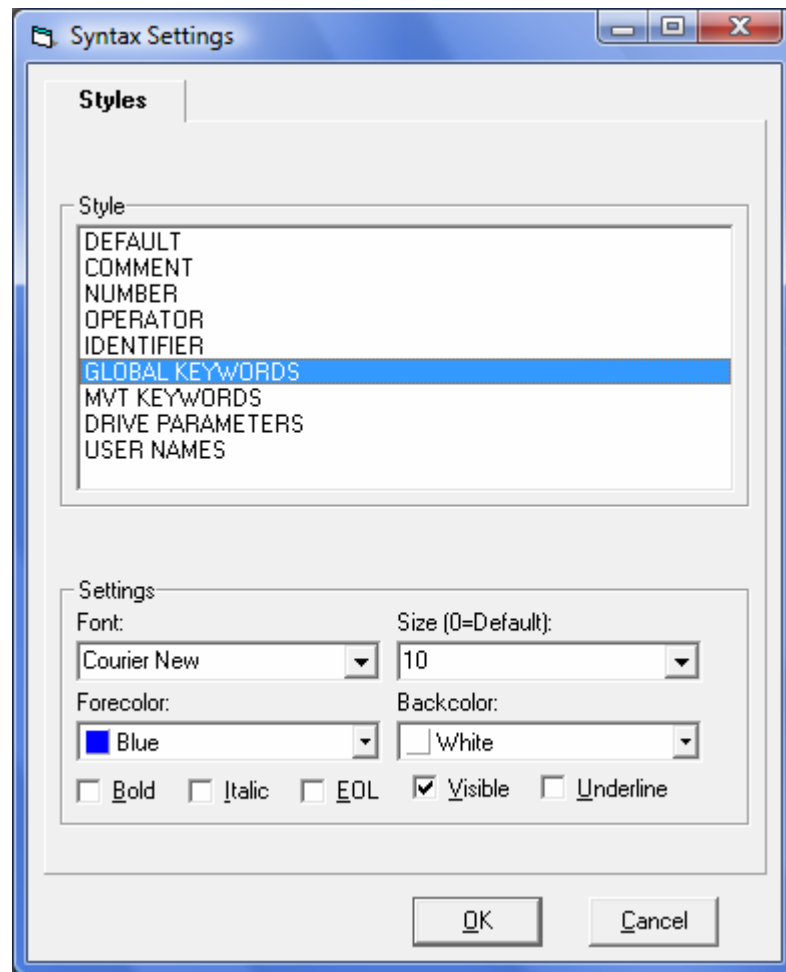
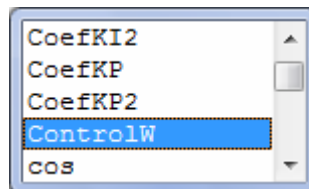


Figure 3-9: **GDS Output Window Configuration.**

Figure 3-10: **GDS File Editor Setting.**

3.5. Autocompletion

Autocompletion displays a list box showing likely identifiers based upon the user's typing. Autocompletion is launched by pressing "**Ctrl+Space**". The user chooses the currently selected item by pressing the tab or return key. The pop-up list will be then cancelled and the selected text from the autocompletion list will be inserted at the file cursor position.



The autocompletion list contains all language keywords, drive parameters and user variable names that the user can insert into the selected file. Remember that it depends on the selected axis *EEDS* (see next section).

4. Drive parameters: EEDS, Mnemonic, Index/SubIndex

Programming the Gem Drive is mainly based on its parameters and user variables handling. Nevertheless, the drive programming language offers the user all he needs to structure his program tasks. The access to the drive parameters or the user variables in the user program must be easy and without any ambiguity. Therefore, all drive parameters are organized clearly and user friendly in a special structure which is called EEDS (Extended Electronic Data Sheet). This is an extension to the CANopen EDS specification.

The EEDS file is in **XML** format and can be viewed by any **XML** editor, e.g. Microsoft Explorer®.

At a project creation with *Gem Drive Studio*, an EEDS file must be selected for each axis integrated in the application.

The drive parameters are structured in groups and sub-groups. (Each parameter can be accessed by using its EEDS Mnemonic called "ParameterName" or by its Index/SubIndex address. Ex: The CAN communication cycle period parameter can be accessed by "Period" mnemonic or by the 0x1006,0x00 address.

5. Make a new user program

Although there are many ways to go about developing programs in the *Gem Drive Studio* environment, all program developments within the environment should include, for each axis, the following steps:

- "Step 1: Create a New Program"
- "Step 2: Edit and Add Project Source Files"
- "Step 3: Customize the user program configuration"
- "Step 4: Build and correct a user program" on page
- "Step 5: Load the generated building output into the drive"
- "Step 6: start the program execution"
- "Step 7: and finally monitor the program execution"

By following these steps, the user's projects will be built consistently and accurately with minimum project management. This process reduces the development time and lets the user concentrate on actual code development.

Step 1: Create a New User Program

Because all development in the *Gem Drive Studio* environment occurs within a project, creating a project is the first step in the application development process. The user has then to add to the project the axes involved in the application. This procedure is not described in this section.

When a new axis is added to the project, an axis item is added in the project navigator window with several sub-items which are organized in a way to facilitate the axis management. Axis programming is grouped under "Device Programming" item.

At its creation, the user program is empty by default.

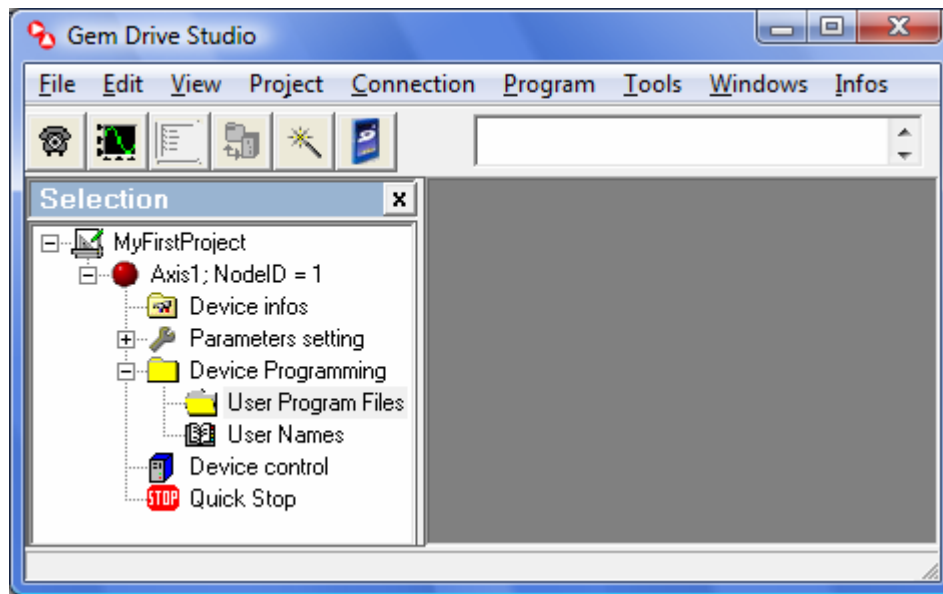


Figure 4-1: Project structure at its creation.

To go on with the next steps, the user must open the **Gem Drive Studio “Programming Tool”** by clicking on the project navigator “Device Programming” item or “User Program Files” sub-item (see Figure 4-1) or by selecting “Programming Tool” in the “Program” menu.

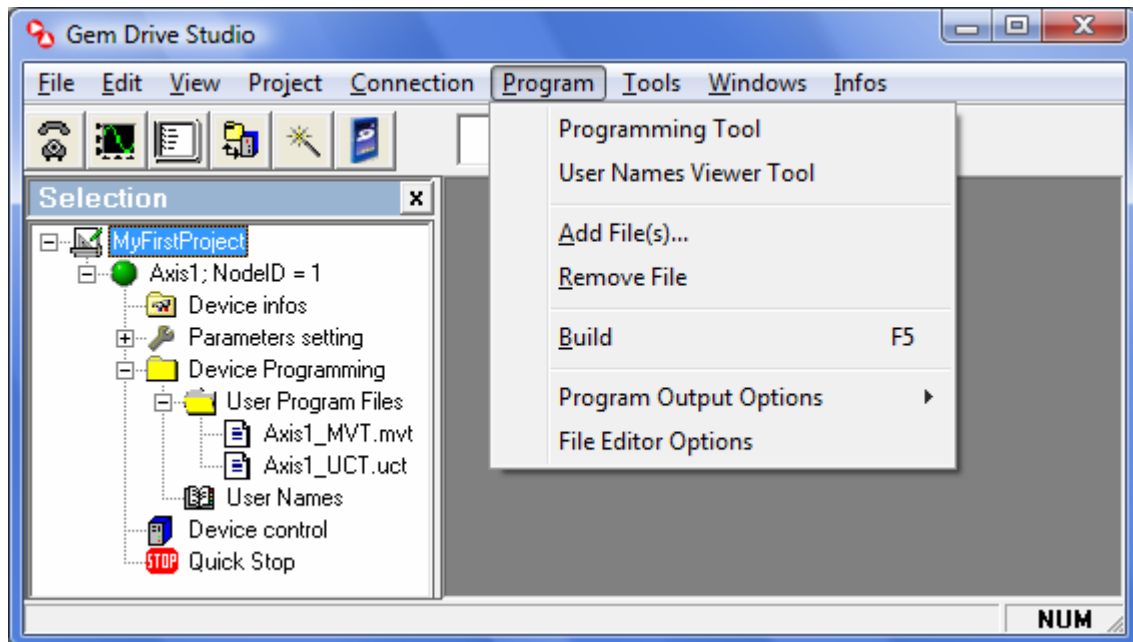


Figure 4-2: Program Menu.

The “Programming Control” tool is the window dialog which allows the user to manage the axis program:

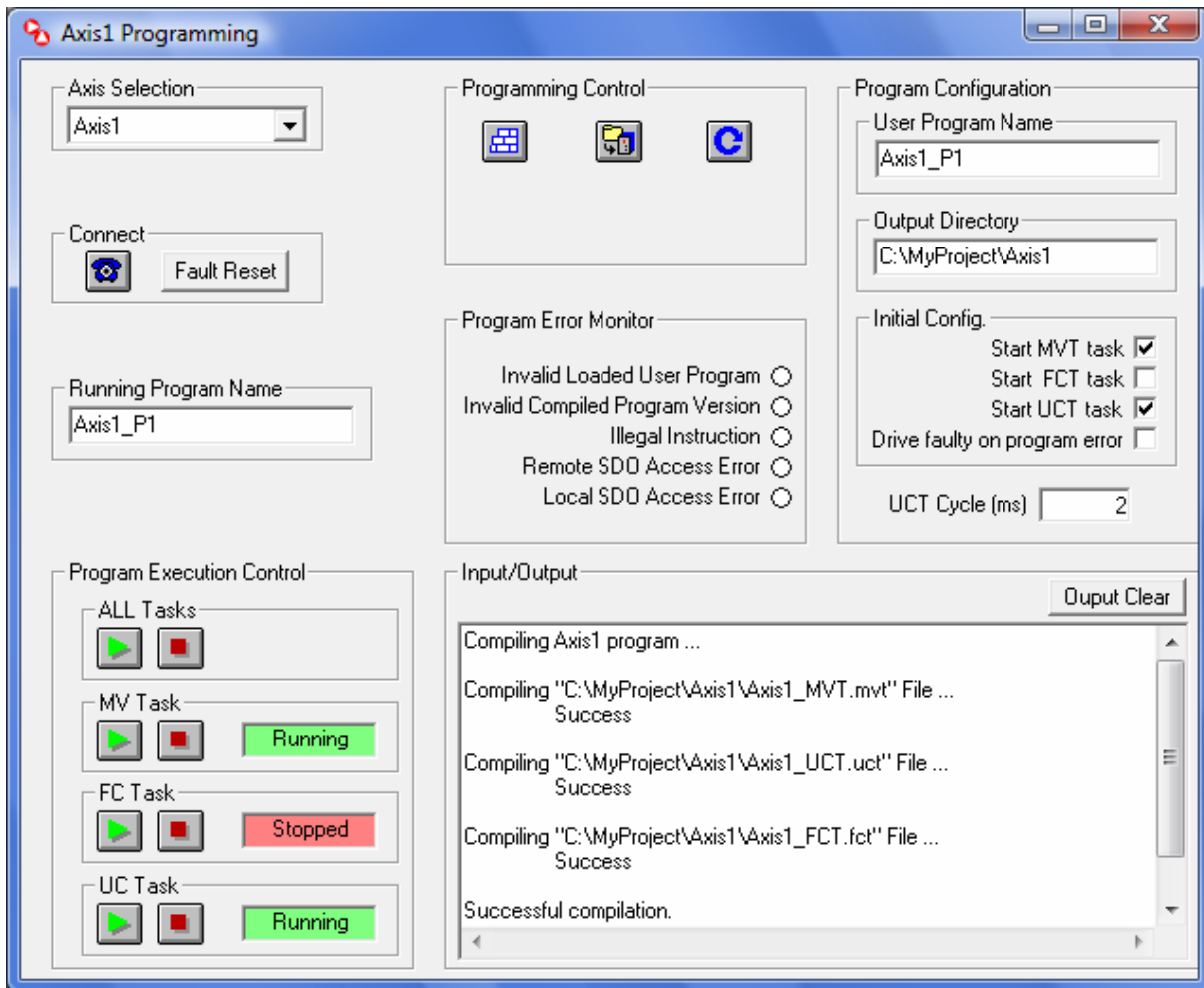


Figure 4-3: Programming Control Tool.

Step 2: Edit and Add Project Source Files

The user can now add to the user program the source files involved in the application. He can edit new files by using the “editor” tool or use existing files.

The user can edit a new file:

1. From a *File Menu: New Source File* (see Figure 4-4),
2. From these shortcuts:

a. Keyboard Shortcut:



b. Accelerator Keys:

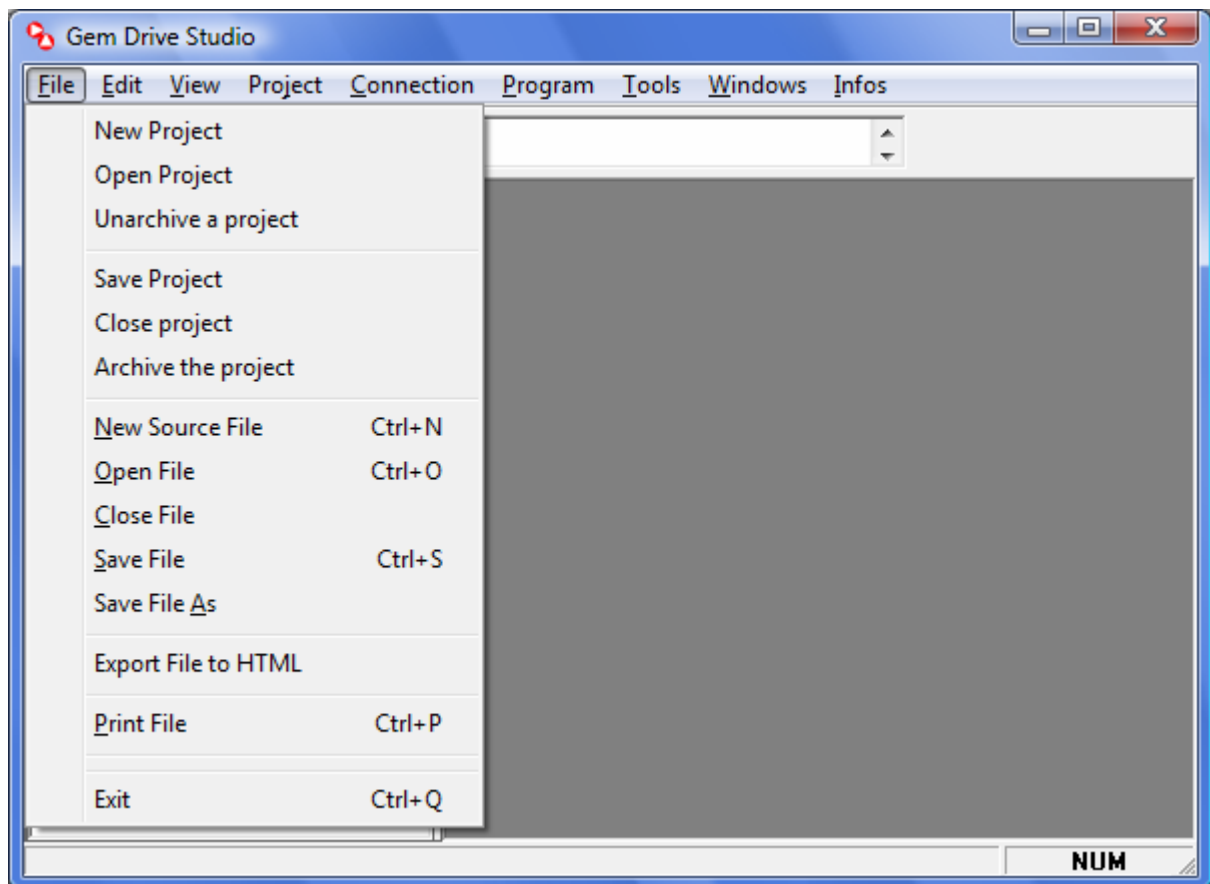


Figure 4-4: *Gem Drive Studio* File Menu.

The Edit Menu provides extensive editing features such as Undo/Redo, Copy/Paste Find and Go To Line to help the user when editing a source file.

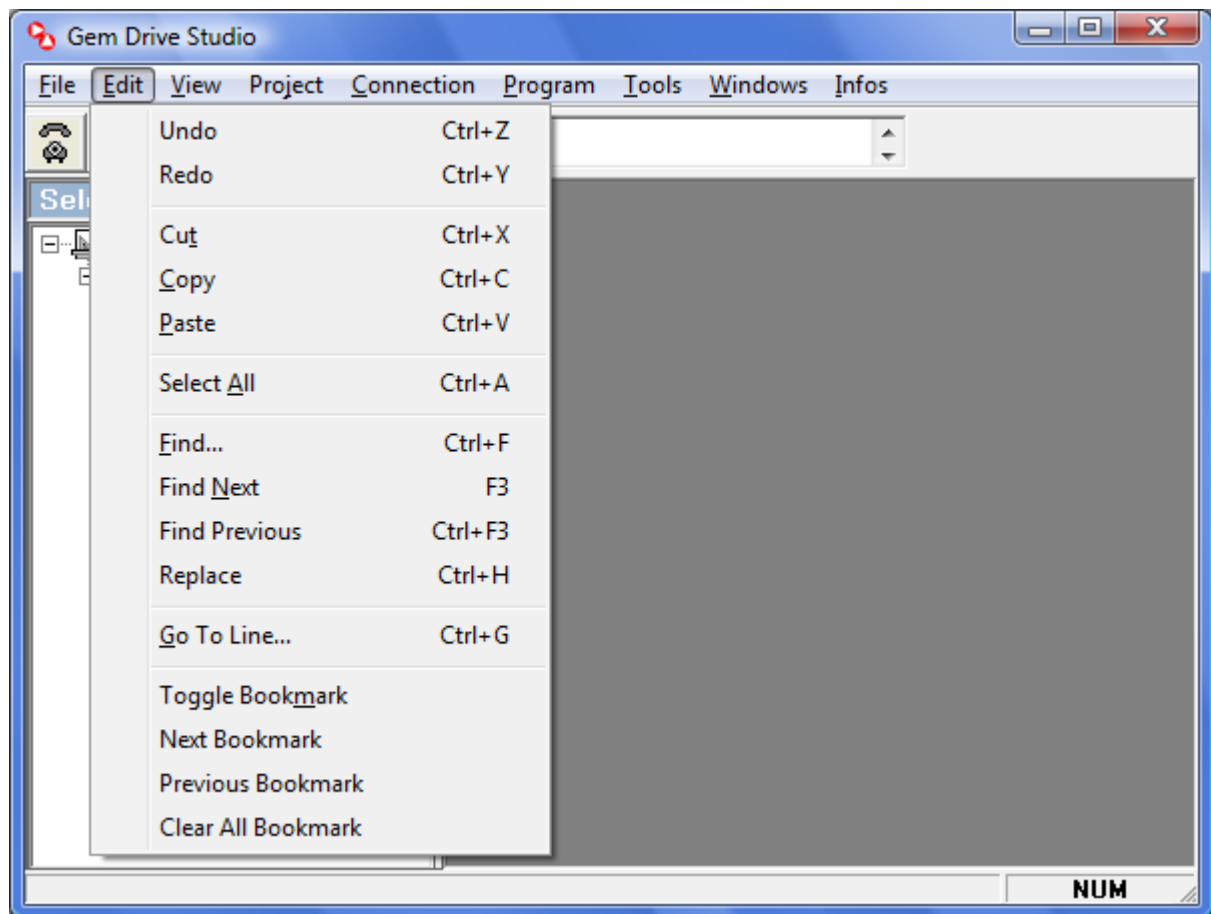


Figure 4-5. Edit Menu.

File and output window text can also be customized by configuring their options in the Program Menu (see [Figure 4-2](#)).

Once the source files created, the user can add them to the project. The user program may contain up to three files per axis, one for each task. Only the “MVT” task is mandatory.

To add a file to the project, the user can use one of the following methods:

1. Right click in the project navigator window and then select “Add Source File” in the popup menu (see [Figure 4-6](#))
2. Select “Add File” from the environment “Program Menu” (see [Figure 4-2](#)).

The environment displays the file `Open` dialog, prompting the user to select a file to be added to the project.

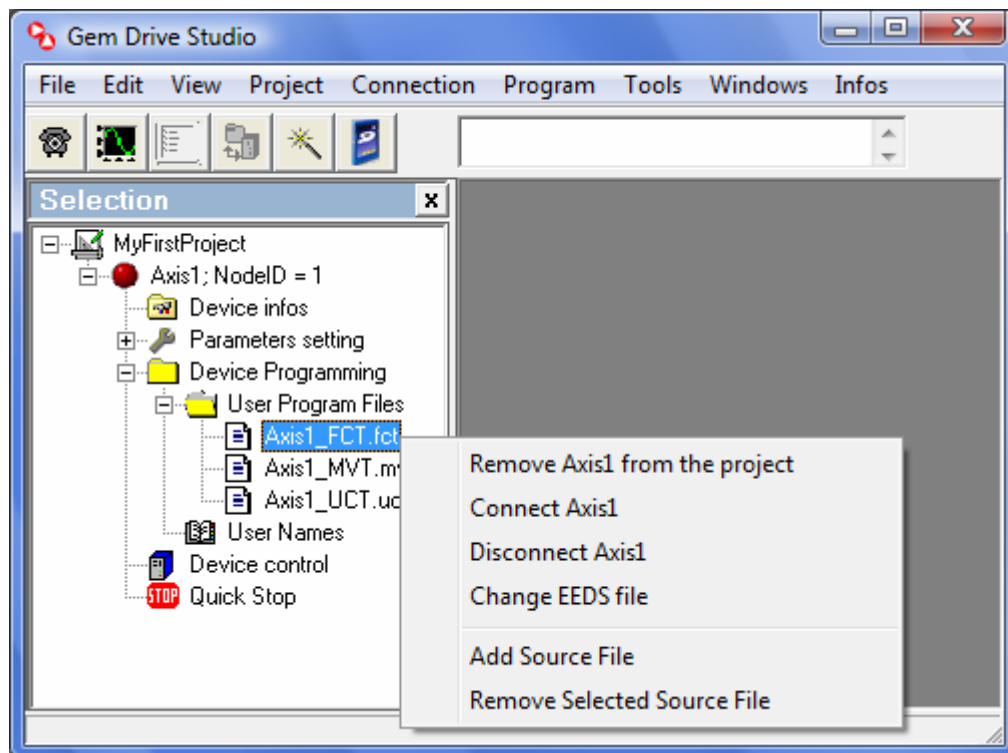



Figure 4-6: Add/Remove a file to/from a project.

The type of a source file depends on its extension, as follows:

1. “mvt” for the MVT task,
2. “fct” for the FCT task and
3. “uct” for the UCT task.

The user can then edit the file(s) added to the project as required. To open a source file with the Gem Drive Studio editor, he can either:


1. Double-click on a document file icon  in the project navigator or
2. Open a source file from a *File Menu: Open Source File* (see [Figure 4-4](#)) or
3. Use the following shortcuts:

a. Keyboard Shortcut:



b. Accelerator Keys:



The user can still remove from a project a previously added file. He must at first select it by clicking on its project navigator icon  (see [Figure 4-2](#)) and then:

1. Right click in the project navigator window and then select “Remove Selected Source File” in the popup menu (see [Figure 4-6](#)) or
2. Select “Remove File” from the environment “Program Menu” (see [Figure 4-2](#)).

Step 3: Customize the user program configuration

Once the user has added the project source files, he can set any necessary custom build project options. Note that this step is optional if default option settings are used. Configuration setting is made in the “Program Configuration” frame of the Programming Tool (see Figure 4-3). It consists of the *UCT* cycle in ms and the tasks starting mode. At power ON, the user can choose, for each task, if it must start automatically or if he will start it later on from the PC by using either the “Program Execution Control” frame of the Programming Tool (see Figure 4-3) or the START/STOP statements in the program (see section 4.13).

Step 4: Build and correct a User Program: Program Compilation

The user can now build the program output file from the created project by using any of the following methods:

1. Build axis program from a *Program Menu: Build* (see Figure 4-2) or
2. Use the following shortcuts:

a. Keyboard Shortcut:

F5

b. Accelerator Keys:

Alt

P

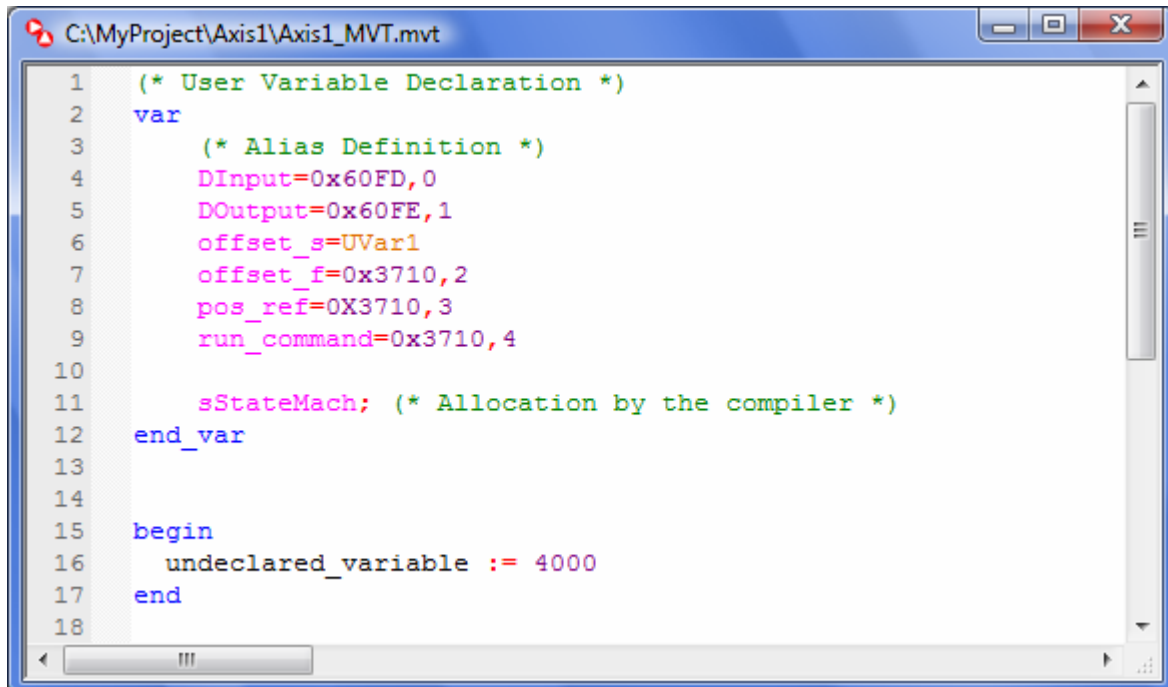
Then

B

c. Programming Control Tool Icon:



By invoking this command, the compiler will check a syntactic and semantic correctness of a user program. Any possible message errors are displayed in the Programming Tool output window, see the example below:



```

1  (* User Variable Declaration *)
2  var
3      (* Alias Definition *)
4      DInput=0x60FD,0
5      DOutput=0x60FE,1
6      offset_s=UVar1
7      offset_f=0x3710,2
8      pos_ref=0x3710,3
9      run_command=0x3710,4
10
11      sStateMach; (* Allocation by the compiler *)
12  end_var
13
14
15  begin
16      undeclared_variable := 4000
17  end
18

```

Figure 4-7: Error example.

As we can see, we are trying to initialize a not declared variable. The compiler will report an error message with the source line and a description of the error. Note that in the editor window, all unknown names are in black, giving a visible programme checking before its compilation.

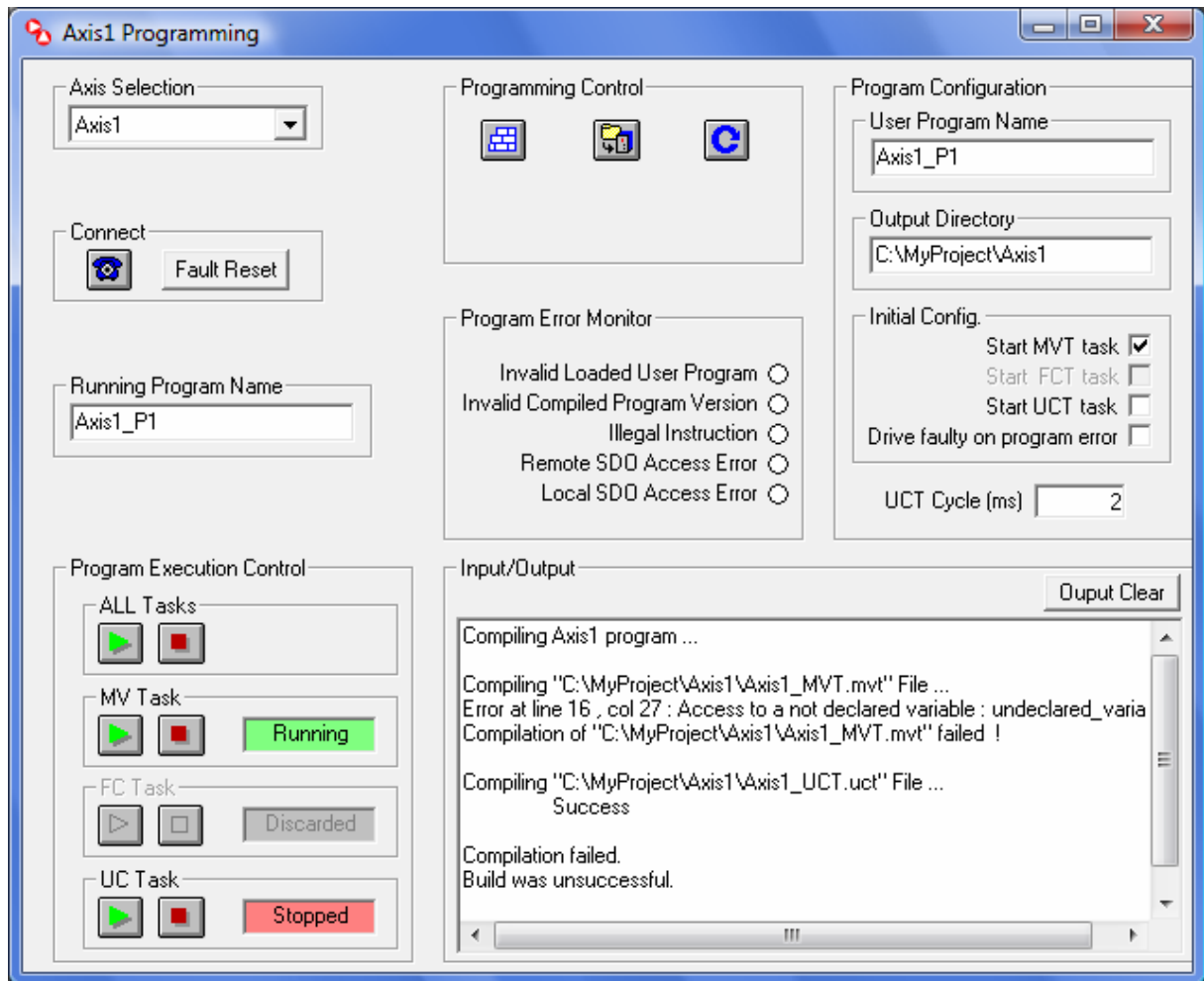




Figure 4-8: Compiler error message report.

The user can go to the error source line by either:

1. double clicking on the error line in the Programming Tool output window (see Figure 4-8) or
2. using the **go to line** command:
 - a. Goto Line Command from a *Edition Menu*: *Go To Line* (see Figure 4-5) or
 - b. Keyboard Shortcut:  

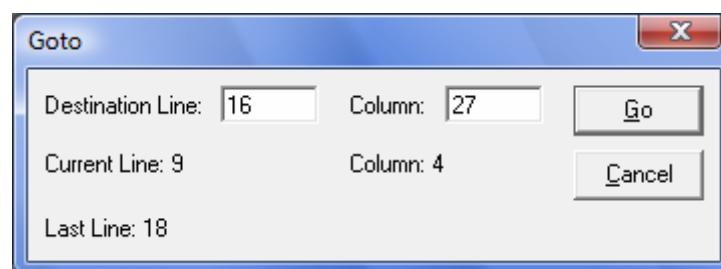



Figure 4-9: Go To Line dialog.


Step 5: Load the generated building output into the drive

When the user program is correct, the compiler generates a unique executable file called **USERPROG.OUT** which can be then loaded into the drive.

The user has to use the Programming Tool program transfer button  (see Figure 4-3).

If an executable file is already loaded into the drive, the user has to confirm its deletion. If the user answers OK, an indication message will be displayed in the Programming Tool during the program loading.

Step 6: Start the program execution

To start a loaded program, the user must use the Programming Tool start program button  (see Figure 4-3). This will stop the currently executed program if any and then launch the new one. The user will be asked for confirmation. The program tasks start immediately or not, depending on the program initial configuration. The user can see the current program execution status in the “Program Execution Control” frame of the Programming Tool.

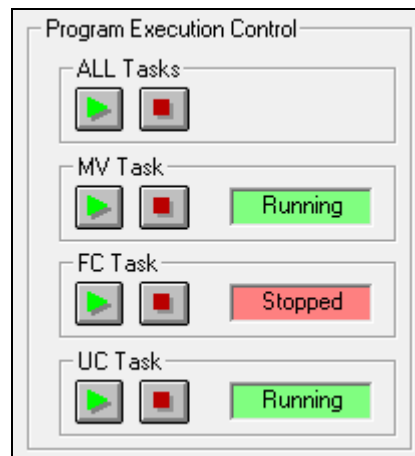


Figure 4-10: User program execution control.

The user can, at any time, use this window to manually start or stop each task.

Step 7: Finally, monitor the program execution

By starting a user program execution, possible execution errors are displayed in this Programming Tool window. When the error occurs, the corresponding led becomes red.

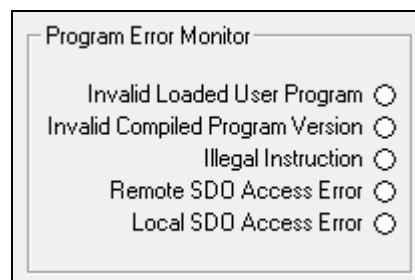


Figure 4-11: User program error monitor.

Chapter 3. Elements of the programming language: IEC 1131-3

1. Overview

The programming language defined for programming the Gem Drive is a specific language made by INFRANOR. It is a BASIC language following the IEC 1131-3 syntax. So, users who are familiar with the IEC 1131-3 PLC languages will quickly get up programming by using this language. It is an easy structured language which allows the user to make from simple to very complex application programs.

As described in chapter 2, a user program may contain up to three tasks, “MVT”, UCT and FCT task. Only the “MVT task is mandatory. UCT and FCT are cyclic. FCT is cycled at 500 μ s whereas UCT is cycled at the cycle time given by the user in the project configuration. The MVT task is executed continuously in background. The Gem Drive kernel is multitask and schedules all tasks simultaneously.

Some of the language instructions are forbidden in the cyclical tasks, like the blocking statements. But the MVT task accepts all language instructions. In the next sections, the authorized tasks will be indicated for each instruction.

Each task is edited in a single file in *Gem Drive Studio* and is recognized by the compiler according to its extension: “.mvt” for MVT, “.uct” for UCT and “.fct” for FCT task.

2. User program organization

The edition of the various task files must follow some principles:

- The user variables must be defined in the MVT task
- Subroutines can be defined only in the MVT task
- Subroutines must be defined after variable declaration and before the main program
- Programs start with “begin” and finish with “end” keywords

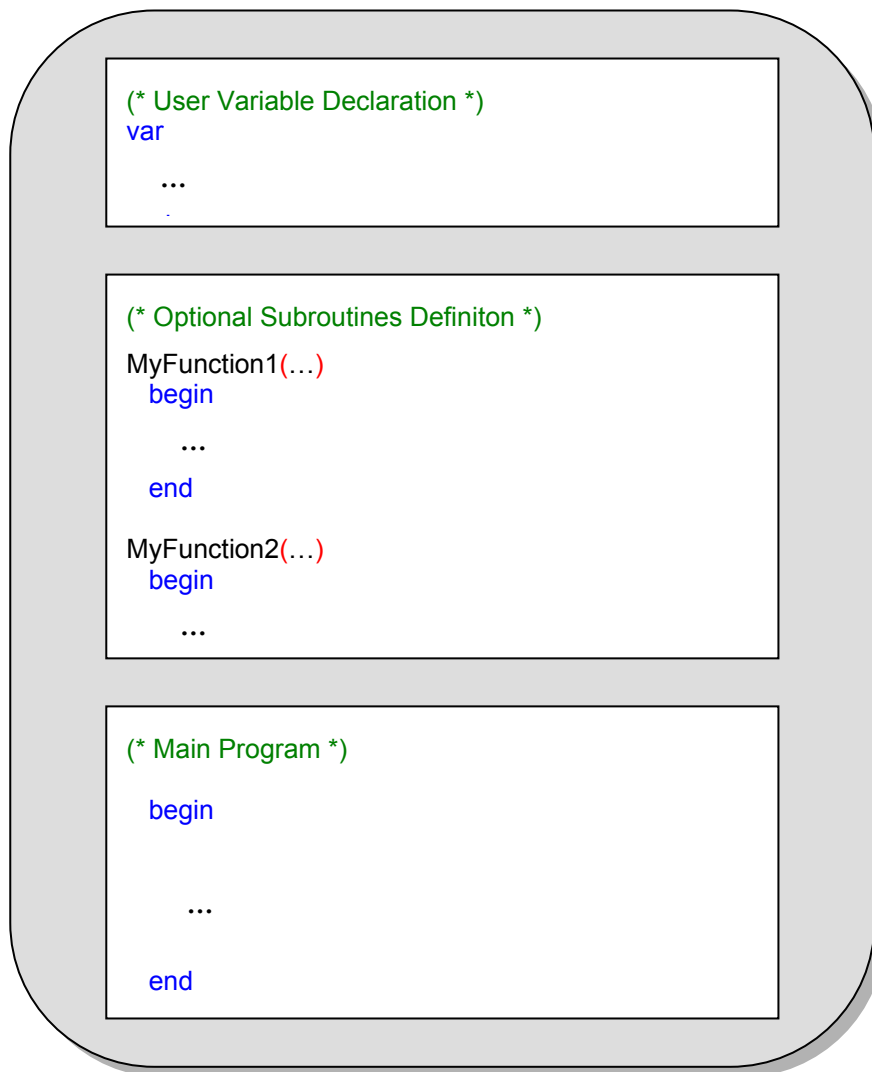


Figure 2-1: MVT program structure.

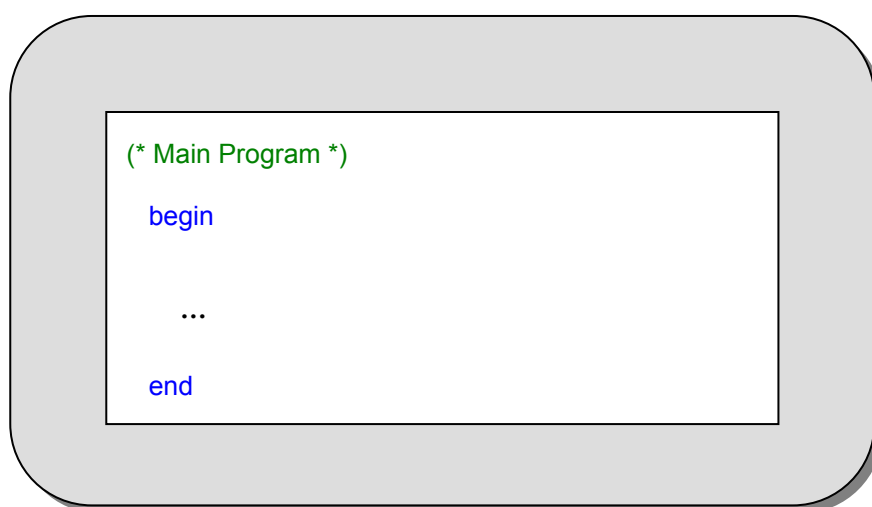


Figure 2-2: FCT and UCT program structure

3. Programming language instruction list

Instruction	Language syntax : IEC 1131-3
<u>Comment</u>	(* ... *) : Multilines
<u>Data Type</u>	Signed Long 32 bits
<u>Constant</u>	<ul style="list-style-type: none"> Decimal Hexadecimal : 0x / 0X...
<u>Blok</u>	<u>begin</u> .. <u>end</u>
<u>Variables and alias</u>	<p>Explicit declaration at the beginning of a « MVT » task.</p> <p>Possibility to define a user alias to a specific drive parameter or a user variable.</p> <p>Variable can be simple or mono-dimensional array</p> <pre> <u>var</u> (* Alias *) MyVar1=ControlW; MyVar2=0x6041,0 x; y (* User Variables *) MyVar3=Uvar17 (* An other Alias *) Tab1[size1]; Tab2[size2] (* Array *) Tab3[size3] ... <u>end var</u> </pre> <p>Array Access : <code>Tab1[expression]</code></p>
<u>Access to the drive parameters</u>	<p>Drive parameters may be accessed by EEDS Mnemonic, by user Alias or by their Index/SubIndex address:</p> <pre><u>ap</u>(Index, SubIndex)</pre> <p>Parameters of remote devices can be accessed by:</p> <ul style="list-style-type: none"> Read : <code><u>rdo</u>(devAddress, Index, SubIndex)</code> Write : <code><u>wdo</u>(devAddress, Index, SubIndex, Size)</code>
<u>Assignment</u>	<code>:=</code>
<u>Arithmetic Operator</u>	<code>- ; + ; - ; * ; / ; mod</code>
<u>Relational Operators</u>	<code>= ; <></code> <code>> ; >= ; < ; <=</code>
<u>Shift Operators</u>	<code><u>shl</u> ; <u>shr</u></code>
<u>Bit Handling Operators</u>	<code>and ; or ; xor ; not</code> <code>clr ; set ; tgl ; tst</code>

<u>Conditional statement</u>	<pre> if Expression then Instruction / Instructions Set elseif Expression Instruction / Instructions Set else Instruction / Instructions Set end if </pre>
<u>Loop statement</u>	<ul style="list-style-type: none"> <pre> while Expression do Instruction / Instructions Set end while </pre> <pre> for InitExpression to Expression [by Value] do Instruction / Instructions Set end for </pre>
<u>Program Flow Control</u>	<ul style="list-style-type: none"> <u>return</u> <u>exit</u> <u>goto</u> label label : <u>halt</u>
<u>Special Statements</u>	<ul style="list-style-type: none"> <u>wait until</u> Expression <u>delay</u>(ms) <u>start fct</u> ; <u>stop fct</u> <u>start uct</u> ; <u>stop uct</u> <u>uvsave</u> ; <u>uvrestore</u>
<u>Math Functions</u>	<ul style="list-style-type: none"> <u>abs</u>(expression) <u>sqrt</u>(expression) <u>sin</u>(expression) <u>cos</u>(expression) <u>atan</u>(expression)
<u>Subroutines</u>	<pre> Declaration : MyFunction(argument1, argument2...) Begin ... end Call: MyFunction(param1, param2...) </pre>

Table 3-1: Programming language instruction list.

4. Programming language reference

This section describes all what the programming language offers the user to make an application program. For each instruction, syntax, detailed description and examples will be given to help the user to get up quickly programming the drive.

4.1. Comment

For more readability, the user can write explanation notes in his program. Comments are used for this purpose. They can be placed everywhere in the program and are ignored by the compiler. A comment is multiline. It starts by “(* ” and ends by “*) ”.

Example:

```
(* This is a multiline comment  
   to describe something...   *)
```

Figure 4-1: Comment syntax.

4.2. Data Type

Since the drive parameter type is Integer, the language data type is a Signed Long 32 Bits. The float data type is not used.

4.3. Constants

The programmer can use constant numbers in the program operations. They are Signed Long 32 Bits values. A constant can be written as a decimal or hexadecimal value. Hexadecimal values must be prefixed by “0x” or “0X”.

4.4. User Variables

The user program is structured around variable handling. Variables are meaningful names that represent data in a program. A value or a calculation result can be assigned to a variable by using other variables.

These variables are called User Variables to distinguish them from the drive parameters which are reserved keywords used by the drive to perform a specific task.

4.4.1 User Variable Naming

User variables can have any name as long as it is not a reserved keyword. They can have any length allowing the user to give them a significant name. Nevertheless, names must begin with a letter or underscore, followed by any alphanumeric characters.

Note: Lower and uppercase is to be taken into account, `MyVar` is different from `myVAR`.

4.4.2 User Variable Type

Like the constants, all user variables are only Signed Long 32 Bit numbers. They are also global and can be accessed from all tasks. The user can deal with up to 128 user variables in a program. That is enough to develop more complex applications.

4.4.3 User Variable declaration

Unlike many implementations of BASIC language, variables must be declared previously to their use, otherwise an error will be generated at their compilation. This is to avoid a wrong use of drive parameter names as user variables.

A user variables declaration, as shown in “[Figure 2-1: MVT program structure](#)”, must be at the beginning of a file. Variable declaration must be between “`var`” and “`end_var`” keywords (see example).

Note: User variables are part of the EEDS file and have already defined EEDS mnemonic: `UVarx`, where `x` is the number of a user variable in the range of 1 to 128. They have also a specific CANopen object address at Index `0x3710`, SubIndex `1..128` (`0x1..0x80`).

A user variable name can also be associated to any drive parameter or user variable EEDS mnemonic or CANopen object address. This is referenced to **ALIAS** definition (see example of Figure 4-2).

```
(* User Variable Declaration *)
var

    (* Alias Definition *)
    offset_s=UVar1          (* Saved Offset Value *)
    pos_ref=0x3710,3
    offset_f=0x3710,2
    PGTacc=0x3514,3; PGTdec=0x3514,4
    PGProf_Vel=0x3514,2;
    PGTtarget=0x3514,1;
    TopZ=0x3127,0;
    speed_res=0x310A,0; speed_enc=0x312A,0;
    DInput=0x60FD,0
    PLUS=0x3710,0x71
    MINUS=0x3710,0x72
    REG=0x3710,0x73
    run_command=0x3710,0x74
    reset_offset_memory=0x3710,0x75

    sStateMach; TopZ_Memory  (* Allocation by the compiler *)
    teachArray[10]          (* Array of 10 elements *)

end_var
```

Figure 4-2: Variable declaration.

Remark: User variables are not initialized to zero at the program start, the user has to explicitly make variable initialisation.

4.4.4 Array Variable

A user variable can be simple, as described in the previous section, or a set of elements grouped in a structure called "array". Nevertheless, the language is limited to a mono-dimensional array. Each element is a numeric Signed Long 32 Bit. It can be used everywhere in expressions in the same way as simple variables. Elements are referenced by an index number within brackets "[]".

Example:


```
Offset_s := teachTab[0]
PGTarget := teachTab[2*i - 1] (* Assume i > 0 *)
```

Restriction: In cyclical tasks, array index must be a simple constant.

4.4.5 User Names Report

After Compilation, the compiler generates a variable binding structure which can be viewed by the Gem Drive User Names Viewer tool (see Figure 4-3). It lists, for each defined user variable, the associated EEDS mnemonic and the CANopen object address.

To open the *User Names Viewer* tool in *Gem Drive Studio*, the user can:

1. Simply click on a User Names icon  in the project navigator,

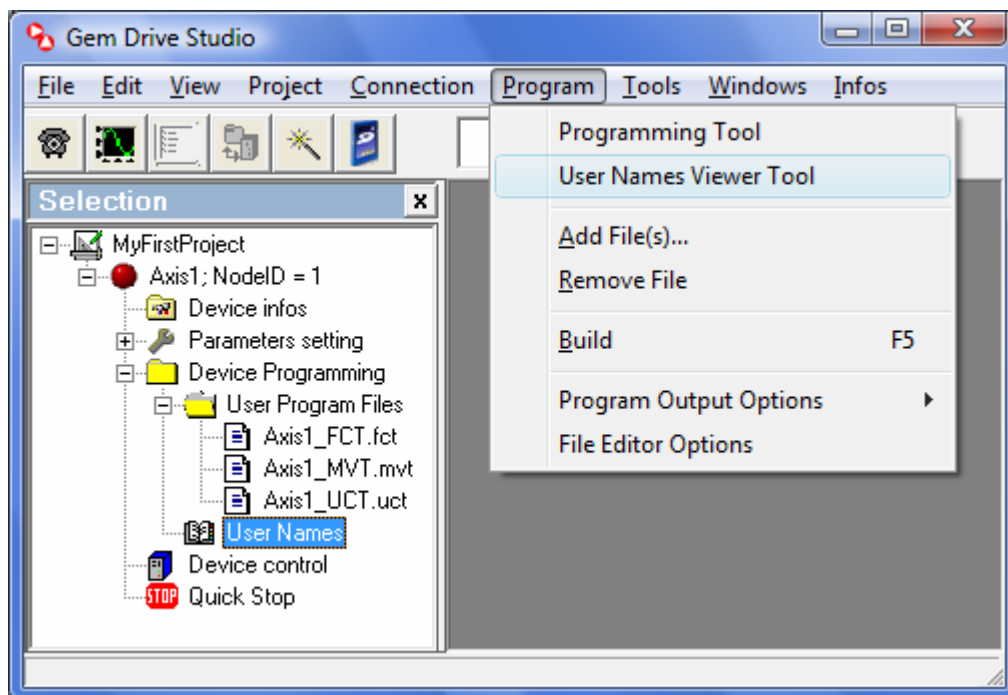


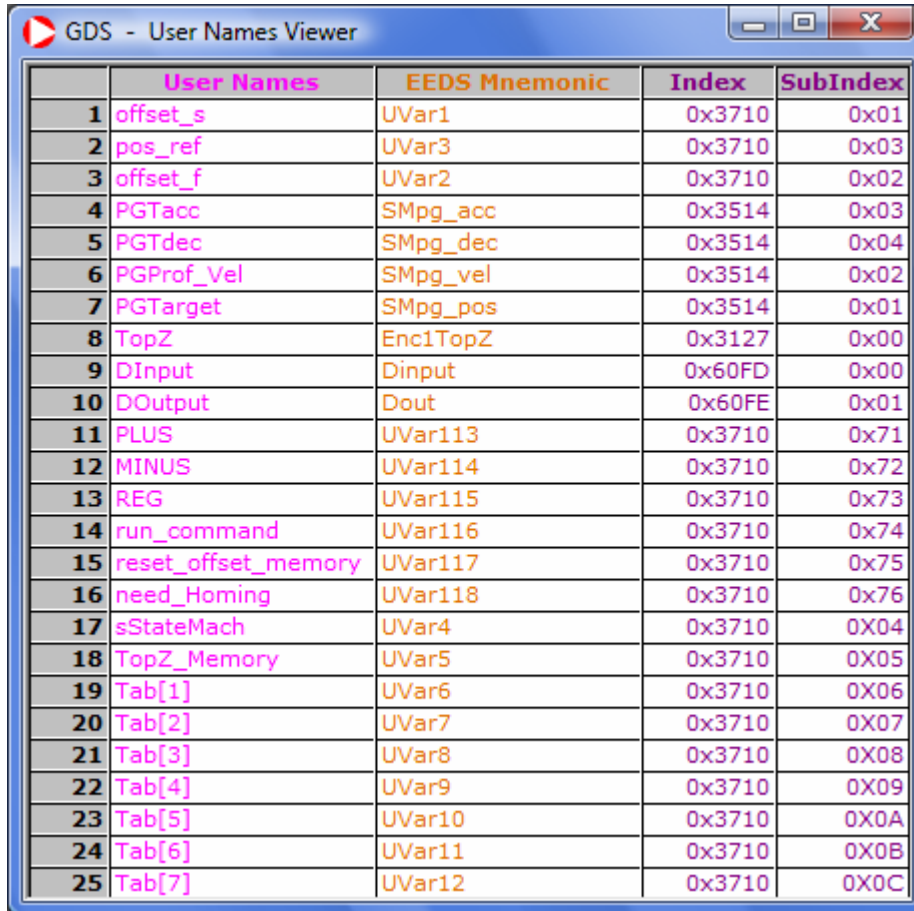
Figure 4-3: User variables binding.

2. Select a *Program: User Names Viewer menu* (see above Figure) or

3. Use the following accelerator keys:



The corresponding variable binding to the above example is shown in the following Figure:



	User Names	EEDS Mnemonic	Index	SubIndex
1	offset_s	UVar1	0x3710	0x01
2	pos_ref	UVar3	0x3710	0x03
3	offset_f	UVar2	0x3710	0x02
4	PGTacc	SMpg_acc	0x3514	0x03
5	PGTdec	SMpg_dec	0x3514	0x04
6	PGProf_Vel	SMpg_vel	0x3514	0x02
7	PGTarget	SMpg_pos	0x3514	0x01
8	TopZ	Enc1TopZ	0x3127	0x00
9	DInput	Dinput	0x60FD	0x00
10	DOutput	Dout	0x60FE	0x01
11	PLUS	UVar113	0x3710	0x71
12	MINUS	UVar114	0x3710	0x72
13	REG	UVar115	0x3710	0x73
14	run_command	UVar116	0x3710	0x74
15	reset_offset_memory	UVar117	0x3710	0x75
16	need_Homing	UVar118	0x3710	0x76
17	sStateMach	UVar4	0x3710	0x04
18	TopZ_Memory	UVar5	0x3710	0x05
19	Tab[1]	UVar6	0x3710	0x06
20	Tab[2]	UVar7	0x3710	0x07
21	Tab[3]	UVar8	0x3710	0x08
22	Tab[4]	UVar9	0x3710	0x09
23	Tab[5]	UVar10	0x3710	0x0A
24	Tab[6]	UVar11	0x3710	0x0B
25	Tab[7]	UVar12	0x3710	0x0C

Figure 4-4: User variables binding example.

As we can see, defined user `offset_s` is bound to `UVar1` as specified by the user. Note that it corresponds to object `0x3710,0x01`, `pos_ref` to `UVar3` and so on. `sStateMach` and `TopZ_Memory`, since the user has left the compiler the responsibility to allocate them, are bound to the first free user variables not yet allocated, i.e. `UVar4` and `UVar5`.

Once a user variable defined in a MVT task, it can be used as operand of program operations everywhere in all tasks. A variable may be accessed by its user name, EEDS mnemonic or object address. So, in the example above, `offset_f`, `Uvar2` and `ap(0x3710, 0x02)` designate the same user variable.

The same identifiers are also used to access a user variable in all *Gem Drive Studio* tools, i.e. a dialog window and the oscilloscope which is very important for application debugging.

Care must be taken when using variables. Some parameters are protected from reading or writing. The user must refer to the compiler error output and the EEDS parameters description to correct his program. Care must also been taken when using operations between operands of different types, as drive parameters can be of 16 or 32 Bits. Conversion is automatically made by the compiler.

4.4.6 Local axis parameters versus remote device objects (SDO): **ap**, **rdo**, **wdo**

ap keyword (Axis Parameter) is used to access local drive parameters or user variables via their Index/SubIndex address. The user may also access to parameters of remote devices via CAN under the **SDO** communication protocol. The following syntax is used to access those objects:

- Local drive parameters: **ap** (Index, SubIndex), the same for reading and writing.
- Remote device objects:
 - **Read:** **rdo** (devAddress, Index, SubIndex)
 - **Write:** **wdo** (devAddress, Index, SubIndex, Size)

Access to local axis parameters is controlled by the compiler, depending on there existence and access type (see EEDS file) and possible errors are generated in *Gem Drive Studio* at compilation. But access to remote device objects cannot be controlled by the compiler at compilation. Remote access will generate a dynamic error at program execution if one of the following conditions is not fulfilled:

- Remote devices must have devAddress as address regarding the CAN addressing,
- Object Index/SubIndex must exist
- Remote object access must be complying with its access type and
- Object Size must be correct, relating to the SDO communication protocol. Thus, Size can take:
 - 1: one Byte data
 - 2: two Bytes data
 - 4: four Bytes data
 - 0: when the user does not know the object size or when the remote device, accepts SDO transfer without data size indication (see [CANOpen SDO protocol](#)).

When this error occurs, the error bit in the user program status word is set and the “Remote SDO Access Error” bit is set in the User Program Error word.

Remark: The Gem Drive accepts SDO transfer without data size indication, so the user can access remote drive parameters just by using 0 as Size value (see example).

Example:

```
(* Read a local parameter *)
x := ap(0x6041, 0)
(* Write a local parameter *)
ap(0x6040, 0) := 0
(* Read Object (0x6700,0) from node 3 (CAN Address) *)
y := rdo(3, 0x6700, 0)
(* Write Object (0x8100,0) of node 3 *)
wdo(3, 0x8100, 0, 2) := 0x0F
```

4.5. Assignment

Assignment lets the user transfer a value to a user variable or parameter. This is made by using a “:=” symbol. A value can be just a simple constant, a variable or a result of a complex calculation.

Variables must be declared previously to their use.

	MVT	UCT	FCT
Valid Tasks	Yes	Yes	Yes

Restriction: In cyclical tasks, operations must not exceed one operation whereas there is no restriction for MVT task, i.e. in MVT, the assignment can be the result of a complex calculation, while in cyclic tasks, operations are limited to one arithmetic operation. The value to be assigned must be constant, variable or the result of a simple operation between a simple constant or a variable, like:

```
dest := constant
dest := src1
dest := src1 opt src2
```

src1 and src2 are any constant or variable. opt is any operator of [table 4.7-1](#).

4.6. Saving / Restoring user variables

These statements are used to save and restore a number of user variables. By giving n as input parameter, user variables Uvar1 to Uvarn will be saved/restored. As these statements take some delay to be done, they are forbidden in cyclical tasks.

	MVT	UCT	FCT
Valid Tasks	Yes	No	No

```
(* Save the n first user variables *)
uvsave(n)
uvrestore(n)
```


4.7. Language Operators

GDL provides a wide range of relational and mathematical operators to construct more or less complex expressions. Table 4.7.1 lists all these operators.

<code>+ - * /</code>	Arithmetic operators
<code>-</code>	Unary minus
<code>mod</code>	Modulo
<code>=</code>	Equals
<code><></code>	Different
<code>></code>	Greater than
<code><</code>	Lower than
<code>>=</code>	Greater than or equal to
<code><=</code>	Lower than or equal to
<code>and</code>	Bitwise AND
<code>or</code>	Bitwise OR
<code>xor</code>	Bitwise exclusive OR
<code>not</code>	Unary NOT
<code>shl</code>	Left shift
<code>shr</code>	Right shift
<code>clr</code>	Bit clear
<code>set</code>	Bit set
<code>tgl</code>	Bit toggle
<code>tst</code>	Bit test

Table 4.7-1: *GDL* operators' list

4.7.1 Arithmetic Operators

Arithmetic operators are unary `+`, `-` and binary `+`, `-`, `*`, `/`, `mod` and operators. All operations are Signed Long 32 Bits. Expressions are evaluated from left to right respecting the following precedence laws:

Binary `+`, `-` < Binary `*`, `/`, `mod` < Unary `+`, `-` < `(,)`
 Use “(“ and “)” to force precedence as they have the highest priority.

For example, `2+5*6` will evaluate to `32` where `(2+5)*6` will evaluate to `42`.

	MVT	UCT	FCT
Valid Tasks	Yes	Yes	Yes

Restriction: In cyclical tasks, operations must not exceed one operation whereas there is no restriction for MVT task.

4.7.2 Comparison Operators

Comparison operators are used to make tests for conditional statements with respect of the following priority:

`=, <>` `<` `>`, `>=`, `<`, `<=` `<` **Arithmetic Operators**

Thus, expression "`pos - 100 > positiveLimit`" evaluates as "`(pos - 100) > positiveLimit`".

	MVT	UCT	FCT
Valid Tasks	Yes	Yes	Yes

Restriction: In cyclical tasks, operations must not exceed one operation whereas there is no restriction for MVT task.

4.7.3 Bits Handling

These operators allow the user to manipulate variable Bits.

	MVT	UCT	FCT
Valid Tasks	Yes	Yes	Yes

Restriction: In cyclical tasks, operations must not exceed one operation while there is no restriction for MVT task.

4.7.3.1 Bitwise Operators

Bitwise operations are made by `and`, `or` and `xor` operators. So, masks can be used to test, set, clear or toggle some Bits of a variable.

For example:

```
Output := Output or 0x0F
```

Will set the four first Bits, while

```
wait until Input and 0x11
```

Will pause the program until Input Bits 0 and 4 are set.

A **bitwise `xor`** or **exclusive or** takes two bit patterns of equal lengths and performs the logical xor operation on each pair of corresponding bits. The result in each position is 1 if the two bits are different, and 0 if they are the same.

The bitwise `xor` may also be used to toggle flags in a set of bits. By giving a bit pattern, Bits corresponding to the bit pattern containing 1 will be toggled simultaneously.

Example:

```
0101 xor 0011 = 0110 (* In Binary Representation *)
```

Toggles Bits 0 and 1.

Remark: For more clearness, the “=” symbol gives here just the result of the operation and must not be confused with “=” or “:=” *GDL* language symbols.

4.7.3.2. Operator NOT

A **logical not** is a unary operator which performs boolean negation on its operand. As specified in IEC 1131-3, this operator is a "logical or boolean not". It treats the entire value as a Boolean value, changing a false value, i.e. 0, to true, i.e. 1, and not null values to false (0).

This operator must not be confused with the "bitwise not" operator which performs the ones' complement or logical negation of the given binary value, i.e. bit 0 becomes 1 and vice versa. Unlike many other languages, this operator is not implemented in *GDL*. The user can actually make a "bitwise not" operation by using the *xor* operator with -1 value as a pattern.

Example:

```
not 0 = 1
not 1234 = 0
0101 xor -1(decimal) = 1010 (* In Binary Representation *)
```

4.7.3.3. Shift Operators

Shift operators allow the user to move or shift Bits to the left or right. This is an *arithmetic shift*. The bits that are shifted out of either end are discarded. Zeros are shifted in on the right, in the case of a left arithmetic shift (*shl*); in the case of a right arithmetic shift (*shr*), copies of the sign bit are shifted in on the left.

Example:

```
MyVar1 := Offset shl 2
MyVar2 := Inputs shr 3
```

In the first case, the two leftmost bits were shifted past the end of the binary variable value, and a new 0 was shifted into the two rightmost positions. In the second case, the three rightmost bits were shifted out, and a variable sign bit was copied into the three leftmost positions, preserving the sign of the variable. Thus, using a 8-bit binary value will give following results:

```
10110110 shl 1 = 01101100
10110110 shr 1 = 11011011
00110110 shr 1 = 00011011
```

A left arithmetic shift by n is equivalent to multiplying by 2^n , while a right arithmetic shift by n is equivalent to dividing by 2^n and rounding towards zero.

4.7.3.4. Single Bit Handling

These operators allow the user to access a single bit in a variable. He can use `set` operator to set, `clr` to clear, `tgl` to toggle or `tst` to test a single bit in a variable. Their syntax is as follows:

`opt(destination, BitNumber)` where `opt` = `set`, `clr`, `tgl` or `tst`

BitNumber is the number of the bit to be handled. It ranges from 0 to 31.

Example:

```
opt(MyVar, 2)      (* Bit 2 *)
(* Where opt = set, clr, tgl *)
if tst(Input, 5) then
  ...
```

Restriction: In the MVT task, the bit number can be the result of any calculation. The destination can also be a local user or parameter variable or a remote device object. But in the cyclical tasks, the bit number can only be a constant or variable and the destination must be local.

4.8. Conditional Statement: `if_then_else`

	<i>MVT</i>	<i>UCT</i>	<i>FCT</i>
Valid Tasks	Yes	Yes	Yes

Conditional or commonly called “`if_then_else`” statement, is used to execute a statement block depending on a conditional expression.

```
if condition_expression then
  true_statement_block
[ else      (* Optional *)
  false_statement_block ]
end_if
```

First, the boolean “`condition_expression`” is evaluated. If the condition is true, the statements `true_statement_block` up to the optional `else` are executed. Otherwise, the execution continues in the `false_statement_block` if the `else` block is present. In all cases, the execution will continue in the statements following the `end_if` keyword.

`if_then_else` statement can be nested, allowing the user to structure his application code as needed.

Example 1:

```

if pos < lowerLimit then
    DOutput := DOutput or 0x40
    Offset_s := lowerLimit - pos
else if pos > upperLimit then
    DOutput := DOutput or 0x80
    Offset_s := upperLimit - pos
else
    DOutput := DOutput and 0xFFFFFFF3F
    Offset_s := 0
end_if

```

Restriction: Care must be taken when using this statement in the cyclical tasks:

- The “condition_expression” must be a simple conditional operation, i.e. only one operation.
- Statement block may not exceed one operation.
- Nesting is not allowed, use **goto** statement to pass round this constraint, see the next example.

Example 2:

```

vTmp := StatusW and 0x221
if vTmp <> 0x221 then
    goto pNext
end_if

if DInput = 0 then
    goto pNext
end_if

if DInput = 0x01 then
    pos := pos + vPLUS
else
    pos := pos - vMINUS
end_if

if pos < lowerLimit then
    goto lowLimit
else
    goto upLimit
end_if

lowLimit:
DOutput := DOutput or 0x40
Offset_s := lowerLimit - pos
goto pNext

upLimit:
if pos <= upperLimit then
    goto noOffset
end_if

DOutput := DOutput or 0x80
Offset_s := upperLimit - pos
goto pNext

```

```

noOffset:
DOutput  := DOutput and 0xFFFFFFF3F
Offset_s := 0

pNext:
...

```

4.9. Iteration Statement (Loop): while, for

	<i>MVT</i>	<i>UCT</i>	<i>FCT</i>
Valid Tasks	Yes	No	No

Loop statements let you execute a statement block as long as a conditional expression is true. There are two kinds of loops, the first is a count-controlled-loop. The second one is a condition-controlled-loop.

In both cases, nesting is allowed without any restriction. You can also combine these statements as you need.

4.9.1 Count Controlled Loop

In the count-controlled-loop, the block is executed a number of a given count:

```

for init_expression to end_expression
  [ by step_expression ] do
    statement_block
end_for

```

Count starts at `init_expression` and ends at `end_expression` which are more or less complex expressions evaluated to integer value. If `init_expression` is greater than `end_expression` and `by` expression is discarded, the statement block is no more executed.

The `by` expression lets the user give a step by which the count is increased. After each loop iteration, the `step_expression` is evaluated and added to the count until the `end_expression` is reached. Note that if the `step_expression` is negative then count is rather decreased, hence the count will decrement from `init_expression` to `end_expression` which must be lower than the first one. If the `by` expression is not present, the count is incremented by default by one.

Example:

```

for i = 0 to 10 do
  wait until Dinput and 0x01
  teachTab[i] := pos
  wait until not(Dinput and 0x01)
end_for

for i = 15 to 1 by -2 do
  myTab[i] := 1 shl i
end_for

```

4.9.2 Condition Controlled Loop

In the condition-controlled-loop, the block is executed while a conditional expression is true:

```
While condition_expression do
    statement_block
end_while
```

“condition_expression” is a boolean expression which can be made by relational operators for example and evaluated first. As long as its value is true, the “statement_block » is executed.

A “forever” loop can be implemented by an infinite loop in which the condition is always true. Remember that you have to foresee break conditions to prematurely terminate the execution of the “statement_block ». The user has to use the “**exit**” statement for this purpose.

Example:

```
while DInput and 0x01 do
    ...
end_do

(* FOREVER Loop *)
while 1 do
    ...
    if exit_condition then
        exit
    end_if
    ...
end_do
```

4.10. Flow Control instructions: exit, goto label, return, halt

The program is executed in a sequential way. Statements are executed one after the other in the order of there appearance in the program. Flow control statements are used to modify this execution flow of the program.

4.10.1 "exit" statement

The “**exit**” keyword is used to force the termination of a loop. It is generally used in combination with the “**if**” statement (see example above). Note that in nested loops, it will terminate the execution of the inner loop only.

	<i>MVT</i>	<i>UCT</i>	<i>FCT</i>
<i>Valid Tasks</i>	Yes	No	No

4.10.2 "goto label" statement

The **goto** statement is an unconditional transfer of control. The flow control continues at statement appearing immediately after the specified label.

A label is an explicit name which must respect the Identifier rules. It must appear at the beginning of a line and immediately be followed by a colon ":".

```
goto label
...
label: ...
```

Care is to be taken when using this statement, in loop statement for example. The destination label must be also well controlled, as the branching cannot be made to anywhere in the program.

	MVT	UCT	FCT
Valid Tasks	Yes	Yes	Yes

Example:

```
if pos > maximum_Pos then
    goto errHandling
end_if

...

errHandling:
(* Process error cases *)
set(DOutput, 1)
...
```

4.10.3 "return" statement

A **return** statement is used to leave the current subroutine and resume at the point where the subroutine was called (see the function use in [section 4.15](#)). A value may be returned by this statement to the caller if the function returns a value.

	MVT	UCT	FCT
Valid Tasks	Yes	No	No

4.10.4 "halt" statement

A **halt** statement is used to stop the execution of the user program, i.e. to stop all tasks.

	MVT	UCT	FCT
Valid Tasks	Yes	Yes	Yes

Example:

```

if DInput and 0x01 then
    (* Some handling to do before halting the program *)
    myAtExitFunction()
    stop_fct
    stop_uct
    Doutput := 0x01
    halt
end_if

```

4.11. Wait a delay time

The **delay** statement is used for waiting a delay time. The program will pause for a given number of milli-seconds.

	<i>MVT</i>	<i>UCT</i>	<i>FCT</i>
Valid Tasks	Yes	No	No

Example:

```

(* wait 2000 ms *)
delay(2000)

```

4.12. Wait until Condition

Wait_until statement stops the program until a specified condition is true. This is useful for program synchronisation. Condition can be any boolean expression which evaluates to true or false.

	<i>MVT</i>	<i>UCT</i>	<i>FCT</i>
Valid Tasks	Yes	No	No

Example:

```

(* wait for a position value *)
wait until pos > 8000

(* wait a rising edge of an input *)
wait until not(Dinput and 0x01)
wait until Dinput and 0x01

```

4.13. Start / Stop cyclical tasks

User can start or stop a cyclical task by using these statements.

	<i>MVT</i>	<i>UCT</i>	<i>FCT</i>
<i>Valid Tasks</i>	Yes	Yes	Yes

```
start_fct
stop_fct
start_uct
stop_uct
```

4.14. Math functions

Some useful math functions are implemented in GD1 to meet the application needs.

- abs
- sqrt
- sin
- cos
- atan

As GD1 does not use floating numbers, the input and output of all these functions are integer values. For trigonometric functions, as explained below, a special scaling must be done when interpreting the values.

4.14.1 ABS

This function returns the absolute value of a given number.

Parameter:

Any valid expression valuated to a Signed Long 32 Bits number.

Value Range: -2147483648 .. 2147483647

Return value:

The absolute value of the input parameter.

Value Range: 0 .. 2147483647

Example:

```
(* Absolute value *)
MyVar := abs(-32000)    (* 32000 *)
offset := abs(CurrentPos * 2)
```

4.14.2 SQRT

This function returns the square root of a given number.

Parameter:

Any valid expression valuated to an Unsigned Long 32 Bits number.

Value Range: 0 .. 4294967295 (0xFFFFFFFF)

Return value:

The square root value of the input parameter.

Value Range: 0 .. 65535 (0xFFFF)

Example:

```
(* sqrt function *)
x := sqrt(9)      (* 3 *)
speed := sqrt(CurrentSpeed * filter) / 2
```

4.14.3 SIN

This function returns the sine of a given value.

Parameter:

Any valid expression valuated to a Signed Integer 16 Bits number.

Even though the input value is a long value, only the first 16 bits are valid.

Value Range: -32768 (0x8000) .. 32767 (0x7FFF)

This value range corresponds to the angle range $-\pi$.. $+\pi$

Return value:

The sine of the input parameter.

The returned value is a signed Integer 16 bits value corresponding to the range -1 to $+1$. So the result must be scaled.

As the type of the user variables is signed long 32-bits, when the result is assigned to a variable or is used in an arithmetic expression, the sign is extended to 32 bits signed value (see example below).

Value Range: -32768 .. 32767

Example:

```
(* sine function *)
x := sin(0)      (* x = 0 *)
y := sin(0x7FFF) (* Input=+ $\pi$ ; y = 0 *)
z := sin(0x3FFF) (* Input=+ $\pi/2$ ; z = 0x7FFF corresponding to +1 *)
t := sin(0xBFFF) (* Input=- $\pi/2$ ; Result = 0x8000 extended to
                  0xFFFF8000 corresponding to -1 *)
```

4.14.4 COS

This function returns the cosine value of a given value.

Parameter:

Any valid expression valuated to a Signed Integer 16 Bits number.

Even though the input value is a long value, only the first 16 bits are valid.

Value Range: -32768 (0x8000) .. 32767 (0x7FFF)

This value range corresponds to the angle range $-\pi..+\pi$

Return value:

The cosine value of the input parameter. The returned value is a signed Integer 16 bits value corresponding to the range $-1..+1$. So the result must be scaled.

As the type of the user variables is signed long 32-bits, when the result is assigned to a variable or is used in an arithmetic expression, the sign is extended to 32 bits signed value (see example below).

Value Range: -32768 .. 32767

Example:

```
(* cosine function *)
x := cos(0)           (* x = 0x7FFF : Corresponds to +1 *)
y := cos(0x3FFF)      (* Input=+ $\pi/2$ ; y = 0 *)
z := cos(0x7FFF)      (* Input=+ $\pi$ ; Result = 0x8000 extended to
                      0xFFFF8000 corresponding to -1 *)
t := cos(0x8000)      (* Input=- $\pi$ ; t = 0xFFFF8000 *)
```

4.14.5 ATAN

This function returns the arc tangent of a given number.

Parameter:

Any valid expression valuated to a Signed Long 32-Bits number. The input parameter is in the 16.16 format having 16 bits of scalar and 16 bits of fraction.

The scalar part (High 16-bits) corresponds to the integer value of the input parameter. With 16-bits in this part, we can represent 2^{16} (65536) discrete values (-32768 to +32767 signed values).

The fractional part gives again 2^{16} steps to represent the values from 0 to almost 1, specifically, 0 to 65535/65536 or approximately 0.99999. The fractional resolution is 1/65536, or about 0.000015.

Return value:

The arc tangent of the input parameter.

The returned value is a signed Integer 16 bits value corresponding to the range $-\pi..+\pi$. So the result must be scaled.

As the type of the user variables is signed long 32-bits, when the result is assigned to a variable or is used in an arithmetic expression, the sign is extended to 32 bits signed value (see example below).

Value Range: -32768 .. 32767

Example:

```
(* atan function *)
x := atan(0)           (* Input= 0.0           ⇒ x=0 *)
y := atan(0x10000)     (* Input= 1.0           ⇒ y=0x00001FFF ⇔  π/4 *)
z := atan(0xFFFF0000)  (* Input=-1.0          ⇒ z=0xFFFFFE001 ⇔ -π/4 *)
t := atan(0x000093CC)  (* Input= 0.57735      ⇒ t=0x00001555 ⇔  π/6 *)
u := atan(0xFFFF6C34)  (* Input=-0.57735     ⇒ u=0xFFFFEAAB ⇔ -π/6 *)
v := atan(0x7FFFFFFF)  (* Input= 32767.99999 ⇒ v=0x00004000 ⇔  π/2 *)
w := atan(0x80000000)  (* Input=-32767.99999 ⇒ w=0xFFFFC000 ⇔ -π/2 *)
```

4.15. Main block: Begin, End

As shown in [figures 2-1 and 2-2](#), a task main program starts with **begin** and ends with **end** keywords.

	<i>MVT</i>	<i>UCT</i>	<i>FCT</i>
<i>Valid Tasks</i>	Yes	Yes	Yes

Example:

```
(* User Variable Declaration *)
var

  (* Alias Definition *)
  offset_s=UVar1           (* Saved Offset Value *)
  pos_ref=0X3710,3

  sStateMach; TopZ_Memory  (* Allocation by the compiler *)

end_var

begin
  (* Variables initialisation *)
  offset_s := 100
  sStateMach := 0
  ...
end
```

4.16. Function: Definition, call, arguments, parameters, return

In an MVT task the user can structure a larger program by defining subroutines or functions and then using them in the program main block.

Functions must be defined first in the beginning of the program, i.e. before their use (see example below).

The functions must be labelled with the same rules as for an identifier, i.e. must start with an alphabetic character followed by a number of alphanumeric characters. “_” can be used also anywhere in the name. The name must be then followed by “()” to differentiate it from a variable name. The function name must not be a user variable name, a parameter name or any keyword, otherwise the compiler will generate an error message.

Optionally, a function may have parameters as inputs and returns a calculation result. Their data type is signed long values.

Input parameters are a list of variable names located between “(“ and “)” and separated by comas “,”. These variables can then be used only in the block of the defined function. Note that user defined variable and drive parameters are global and can be used anywhere in the program. Hence they can serve to transfer data between the various functions.

When a function returns a value, **return** statement should be used. A value is transferred to the caller.

Once a function defined, it can be called in the main block. This is done just by calling the function name followed by “()”. Arguments must be given between parenthesis if the called function is defined with input parameters. The number of parameters must be the same as in the definition.

If the function returns a value, the result can be assigned to a variable by the caller as in the example or used in an expression.

Example:

```
(* User Variable Declaration *)
var
    index
    filter
    tab1[10]
    tab2[10]
end_var

(* Function without parameters *)
init( )
begin
    DOutput := 0
    ControlW := 0
end

(* Function that returns square *)
square( n )
begin
    return n*n
end

(* Function with parameters *)
setFilter(a, b, c)
begin
    filter := square(a) + square(b) + square(c)
end

(* Main Block *)
begin
    init()

    setFilter (1, 2, 3)

    index := 1
    while index <= 10 do
        tab1[index] := square(index)
        tab2[index] := square(index) + (tab1[index] * filter)
        index := index + 1
    end_while

    ...
end
```