

OSSERVATORIO ASTROFISICO DI ARCETRI

TMR Network

Laser Guide Star

for 8-meter class Telescopes

Application Builder Annotated Code

Document: LGS/Arcetri/REP/WPB/1.1/18.06.1999/L. Fini/S. Esposito

Institute: Osservatorio Astrofisico di Arcetri

Type: Report

Task: Work Project B

Date: June 18, 1999

Number: 1

Issue: 1

Number of pages: 88

Written by: L. Fini

Approved by: S. Esposito

Application Builder Annotated Code

L. Fini

Osservatorio Astrofisico di Arcetri

June 18, 1999

Abstract

This document contains the complete source code of the “Application Builder” for the “Laser Guide Star Simulation package” with detailed documentation.

Acknowledgements

The author is grateful to the LGS-TMR groups in Arcetri and at ESO for various discussions and suggestions related to the structure and the functionalities of the Application Builder. In particular M. Carillet, S. Esposito, B. Femenía and A. Riccardi (from this side of the Alps) and F. Delplancke and E. Viard (on the other side) were an efficient “testing team” and were very helpful in finding out the bugs and sometimes in suggesting fixes. A. Riccardi also provided the OS dependent code for the support of `Windows` based IDL implementations and helped me very much with his deeper insight into IDL dirty tricks.

Contents

Glossary	9
1 Introduction	11
1.1 How to Read this Document	11
1.2 Generalities	11
1.3 Application Builder Functionalities	11
1.4 Structure of a Project	12
1.5 Run-Time	13
2 Software Architecture	13
2.1 File Organization	13
2.2 Structure of the AB Code	13
2.3 Notes to the Annotated Code	13
2.4 Portability Issues	14
3 Objects (I)	14
3.1 Module	14
3.1.1 Object Description	14
3.1.2 Method: <code>INIT</code>	15
3.1.3 Method: <code>GetData</code>	18
3.1.4 Method: <code>GetID</code>	19
3.1.5 Method: <code>GetSlot</code>	19
3.1.6 Method: <code>GetGraph</code>	19
3.1.7 Method: <code>GetHandle</code>	19
3.1.8 Method: <code>GetIn</code>	20
3.1.9 Method: <code>GetInCoord</code>	20
3.1.10 Method: <code>GetInputs</code>	21
3.1.11 Method: <code>GetOut</code>	21
3.1.12 Method: <code>GetOutCoord</code>	21

3.1.13	Method: <code>ChangeDType</code>	21
3.1.14	Method: <code>Offset</code>	22
3.1.15	Method: <code>PutLine</code>	22
3.1.16	Method: <code>SetID</code>	23
3.1.17	Method: <code>SetInType</code>	23
3.1.18	Method: <code>SetOutType</code>	23
3.1.19	Method: <code>SetParams</code>	23
3.1.20	Method: <code>SetStatus</code>	24
3.1.21	Method: <code>SetLink</code>	25
3.1.22	Method: <code>DelLink</code>	25
3.1.23	Method: <code>List</code>	25
3.1.24	Method: <code>Cleanup</code>	26
3.1.25	Data Structure	26
3.2	<code>FdbStop</code> Special Module	27
3.2.1	Object Description	27
3.2.2	Method: <code>INIT</code>	28
3.2.3	Data Structure	28
3.3	<code>Combiner</code> Special Module	28
3.3.1	Object Description	28
3.3.2	Method: <code>INIT</code>	29
3.3.3	Method: <code>ToggleSign0</code>	30
3.3.4	Method: <code>ToggleSign1</code>	30
3.3.5	Method: <code>GetSigns</code>	30
3.3.6	Method: <code>SetSigns</code>	31
3.3.7	Method: <code>GetHandle</code>	31
3.3.8	Data Structure	32
3.4	<code>Link</code>	32
3.4.1	Object Description	32
3.4.2	Method: <code>INIT</code>	32
3.4.3	Method: <code>GetX</code>	33
3.4.4	Method: <code>GetY</code>	33
3.4.5	Method: <code>xyOfst</code>	33
3.4.6	Data Structure	33
3.5	<code>Grid</code>	33

3.5.1	Object Description	33
3.5.2	Method: INIT	34
3.5.3	Method: GetSize	35
3.5.4	Method: EnclosingBox	35
3.5.5	Method: Screen2slot	35
3.5.6	Method: Slot2screen	36
3.5.7	Method: Put	36
3.5.8	Method: GetModule	37
3.5.9	Method: Remove	37
3.5.10	Method: Cleanup	37
3.5.11	Data Structure	38
3.6	Project	38
3.6.1	Object Description	38
3.6.2	Method: INIT	39
3.6.3	Method: GiveName	39
3.6.4	Method: GetName	39
3.6.5	Method: GetGraph	40
3.6.6	Method: AddGraph	40
3.6.7	Method: RemoveGr	40
3.6.8	Method: PushModule	40
3.6.9	Method: PopModule	41
3.6.10	Method: GetBox	41
3.6.11	Method: Translate	41
3.6.12	Method: GetModFromSlot	42
3.6.13	Method: List	42
3.6.14	Method: DelModule	43
3.6.15	Method: FindModule	43
3.6.16	Method: SetMaxID	44
3.6.17	Method: GetMaxID	44
3.6.18	Method: GetList	44
3.6.19	Method: Cleanup	45
3.6.20	Data Structure	45
4	Miscellaneous Utilities (II)	45
4.1	OS Utilities	45

4.1.1	Function: <code>mkdir</code>	45
4.1.2	Function: <code>rename</code>	46
4.1.3	Function: <code>rmdir</code>	46
4.2	Access to Module Info	47
4.2.1	Procedure: <code>find_mod_info</code>	47
4.2.2	Function: <code>mod_list</code>	48
4.2.3	Procedure: <code>mod_list_crea</code>	49
4.3	Interaction support	51
4.3.1	Function: <code>askconfirm</code>	51
4.3.2	External Entry Point: <code>SelectFile</code>	51
4.3.3	Event handler: <code>newline_event</code>	51
4.3.4	Event handler: <code>selfromlist_event</code>	52
4.3.5	Function: <code>SelectFile</code>	52
4.4	Project Management	53
4.4.1	Function: <code>GetPrjList</code>	53
4.4.2	External Entry Point: <code>OpenProject</code>	54
4.4.3	Procedure: <code>ScanInput</code>	54
4.4.4	Procedure: <code>ScanModules</code>	56
4.4.5	Function: <code>doRESTORE</code>	57
4.4.6	Function: <code>OpenProject</code>	59
4.4.7	Procedure: <code>RmProject</code>	60
4.4.8	External Entry Point: <code>SaveProject</code>	60
4.4.9	Procedure: <code>ModuleCode</code>	60
4.4.10	Procedure: <code>CombinerCode</code>	61
4.4.11	Procedure: <code>FdbCode</code>	62
4.4.12	Function: <code>Resolve</code>	63
4.4.13	Algorithm	63
4.4.14	Function: <code>WriteCode</code>	64
4.4.15	Function: <code>SaveProject</code>	68
4.5	Link Management	69
4.5.1	Function: <code>CheckLink</code>	69
4.5.2	Function: <code>ComputeLine</code>	70
4.5.3	Function: <code>JoinMods</code>	70
4.5.4	Procedure: <code>RmLinks</code>	71

4.6	Widget Utilities	72
4.6.1	Main Entry Point: LoopCtrl	72
4.6.2	Procedure: loopctrl_event	72
4.6.3	Function: LoopCtrl	72
4.6.4	Main Entry Point: mod_menu	73
4.6.5	Procedure: mod_menu_event	73
4.6.6	Procedure: mod_menu	74
4.6.7	Procedure: NewWorksheet	74
4.6.8	Procedure: SetOverlay	75
4.6.9	Procedure: SetUpMoveProject	75
4.6.10	Procedure: UnsetOverlay	76
4.6.11	Procedure: WritePrjStatus	76
5	Main Procedure (III)	76
5.1	Event Handlers	77
5.1.1	Procedure overlay_event	77
5.1.2	Procedure worksheet_event	78
5.1.3	Procedure edt_event	82
5.1.4	Procedure file_event	83
5.1.5	Procedure edt_event	84
5.1.6	Procedure hlp_event	85
5.2	The Worksheet	85
5.2.1	Procedure: worksheet	85
	References	88

Glossary

Handle

The ending point of a **link**, graphically represented as a colored rectangle in the input or output area of a module (see fig. 3 at page 14).

Link

A line connecting the output **handle** of a module to the input **handle** of the following one. It represents the data flow in the simulation program.

Module

The elementary building block for any simulation program. A detailed description of modules can be found in section 3.1 of this report. Modules are implemented as three IDL procedures: a) the **Module Information Routine**, b) the **Module GUI**, and c) the **Module Computation Procedure**. Each module is identified by a unique three letter code. More details on Module implementation will be provided in a specific report.

Module Computation Procedure

`xxx()` (where `xxx` stands for the module identification code) Is the function where the actual computation is performed. It is called by the simulation code generated by the AB.

Module GUI

`xxx_gui()` (where `xxx` stands for the module identification code) This function is called by the AB on request of the user to allow him/her to specify all parameters needed by the module during the computation by means of a proper graphical interface.

Module Identification

(also: Module ID) an integer number which uniquely identifies a module within a project.

Module Information Routine

`xxx_info()` (where `xxx` stands for the module identification code) This function is used by the **Application Builder** to gather various pieces of information related to the module structure (number and type of inputs and output variables, initialization requirements and so on).

1 Introduction

1.1 How to Read this Document

A large portion of this document is directly generated from the source files of the `Application Builder` which contain extensive comments where \LaTeX formatting commands are also used. The files as such can be directly used within the IDL environment as executable procedures, while the \LaTeX source is obtained by a simple preprocessing procedure. In this manner a strict alignment between the source code comments and this document is guaranteed.

When the document is formatted the actual source code appears in fragments appropriately interspersed in the text which are typed in a different typeface in order to be easily identified. Each fragment of code begins with a separation line which also contains the name of the source file containing the code in order to ease in locating portions of the code for debugging or code inspecting purposes.

1.2 Generalities

One of the aims of the LGS-TMR project was to provide a powerful environment for the development of Adaptive Optics simulation programs. It was thus decided to gather the pieces of software developed by the various participant teams and to create a pool of general routines capable to simulate the various component of a generic Adaptive Optics system.

In the very first phases of the project it was also recognized the utility to provide the scientists with a Graphical Programming Environment in which elementary building blocks could be assembled together to create complex applications in a straightforward manner, so that the user could concentrate on the scientific aspects of his/her problem, while mundane coding problems were managed by some automatic tool.

During various discussions within the participants to the “Work Package B” part of the project the functionalities and the overall architecture of the Graphical Programming Environment (lately named “Application Builder”) were designed as the result of a tradeoff among various requirements, and mainly of three principal goals:

- The programming effort for the development of the AB had to be *small*; i.e.: much lower than the programming effort devoted to actual module development.
- The AB must have marginal impact on the actual simulation programs; i.e.: the coding of modules must not be affected significantly, and the time efficiency at run-time of the code produced must not be increased significantly.
- The requirements on single module coding imposed by the use of the AB must not prevent the use of modules in the traditional way; i.e.: as usual routines to be called by a user written program.

Because the IDL language was selected as the best choice for the implementation of modules, it was also decided to implement the Application Builder in the same language, although not fully suited to this particular task.

1.3 Application Builder Functionalities

The AB appears to the user, as shown in figure 1, as a graphical window (the *worksheet*) provided with a number of rectangular slots and with a number of pull-down menus.

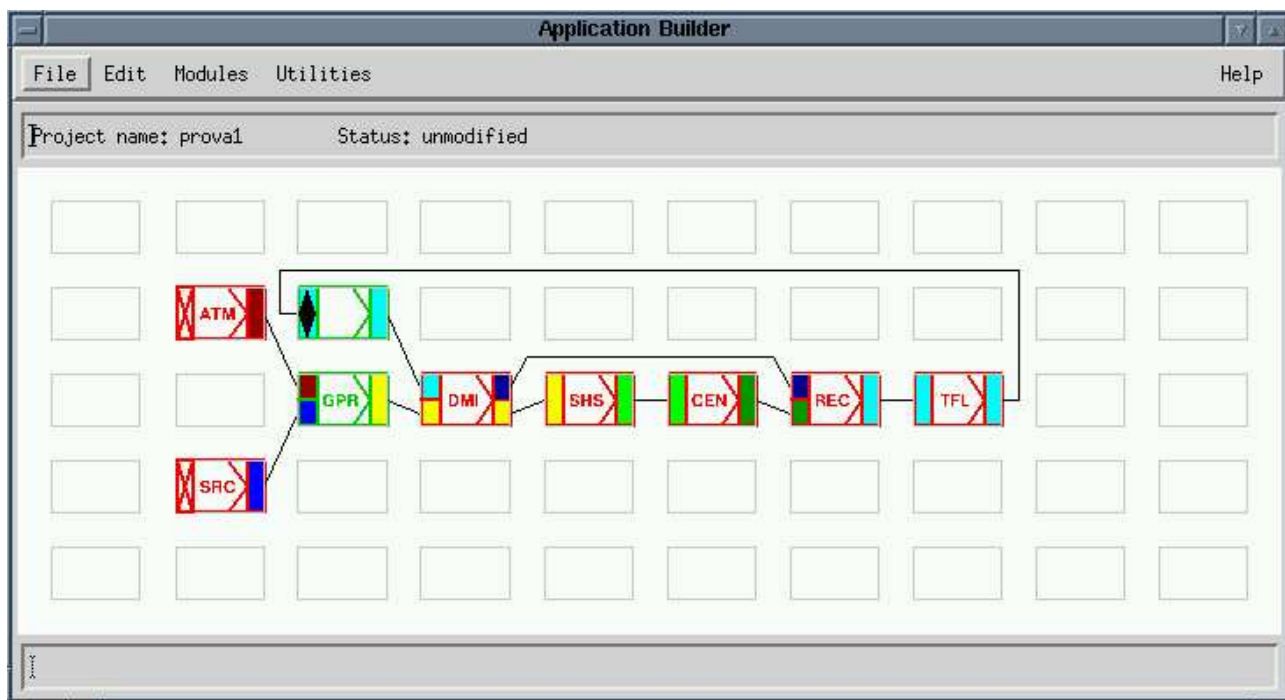


Figure 1: The Application Builder Worksheet

The user can choose from a list of available modules the building block for his/her simulation program (called a **project**); modules can be put into any free slot on the worksheet and then inputs and outputs can be joined by means of **links** which represents the data flow in the program.

The user can then activate, for any module, a Graphical User interface which helps in the definition of parameters to be used in the computation of the program.

When the project is finished it can be saved on disk. The save operation will produce the source code which implement the simulation program together with a textual description of the graphic layout of the project by which the project may be restored in the **Application Builder** for subsequent use.

1.4 Structure of a Project

When a project is stored onto disk a project specific subdirectory is created in the `./Projects` directory. Each subdirectory has the same name as the corresponding Project.

A number of files are created in the project subdirectory to store Project data:

- **project.pro**. This file is the main procedure which will be called to run the simulation program. It contains the code to initialize the main procedure, read parameters from parameter files, etc. It also contain a comment section with a textual description of the graphical representation of the project which is used to activate a worksheet containing the project drawing. As part of the project open operation the project description is read from this file and the corresponding data structures are recreated.
- **mod_calls.pro**. This file contains the list of calls to the procedures implementing modules. It is included via the IDL `@` operator by the procedure `project.pro`.
- ***.sav**. Parameter files. These files are generated by the module specific parameter input GUI procedures activated by the user. They contain module specific IDL data structures which are restored prior of the run of the simulation program by the main procedure.
- ***.bak**. Whenever a “save project” command is issued by the user the previous versions of both `project.pro` and `mod_calls.pro` are stored as, respectively, `project.bak` and `mod_calls.bak`.

1.5 Run-Time

When the user is satisfied with the structure of a project and has defined all the parameters required by the modules, the simulation program may be executed by running the procedure `project.pro` on the project directory.

When the project is running there is no connection whatsoever with the `Application Builder`, i.e.: the project code can run as an independent IDL application and no other services are needed.

2 Software Architecture

2.1 File Organization

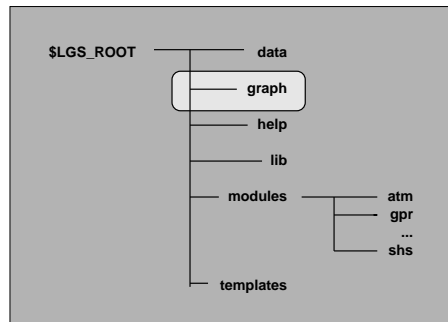


Figure 2: The LGS-TMR Code Directory Structure

The LGS-TMR code is organized in a directory tree with the structure shown in figure 2. The `Application Builder` related code is entirely contained in the subdirectory `graph` which is enhanced in the figure.

2.2 Structure of the AB Code

The structure of the software modules which implement the `Application Builder` can be divided into three parts, as follows:

- I - **Objects.** The application builder heavily uses the object oriented features of IDL 5.x. In this part are gathered the definitions of all objects and related methods.
- II - **Miscellaneous Utilities.** Various procedures and functions to be used within the main procedure. These include: OS utilities (mainly file and directory management), module info utilities, widget utilities, etc.
- III - **The Main Procedure.** The `Application builder` is started by creating a blank “worksheet” where to put together the modules as required by the user. This part includes the `worksheet` main procedure and all the related procedures.

The routines have been subdivided into files both to gather together related routines and in order to enforce the automatic compilation mechanism provided by IDL.

2.3 Notes to the Annotated Code

In the following pages we have adopted some typing conventions to enhance the readability of the code.

- Sections correspond to the three main parts quoted above.
 - Subsections correspond to lower level logical subdivisions of group of routines.
-

- Subsubsections correspond to routines (procedures or functions in the IDL context).
- Sometimes a particular operation has been subdivided into a group of routines, usually gathered into a single source file, but only a single entry point of the set is actually used by code outside the group. In this case the group is introduced by a subsection named **Main Entry Point** to stress the fact that the other entry points are to be considered “local” to the group.
- Each fragment of source code starts with a separation line which also shows the source file name for ease of reference.

2.4 Portability Issues

Although the Workpackage /B team from the very beginning of the project selected Unix as the target operating system for both the development and the final use of the simulation software, during the development of the **Application Builder** it became clear that by adopting some care in the developing of the code the package could be made portable to other IDL implementations and notably to the MS-Windows environment¹.

As it later resulted the only system dependent operations needed in the development of the **Application Builder** are those related to file and directory manipulation, both because IDL is lacking of native procedures for the purpose and because of the slightly different interpretation of “wildcards” in the two environments.

The only system dependent fragments of code are thus limited to the following routines: `find_mod_info.pro` (Sect. 4.2.1), `getprjlist.pro` (Sect. 4.4.1), `rename.pro` (Sect. 4.1.2), `rmdir.pro` (Sect. 4.1.3).

3 Objects (I)

The heart of the **Application Builder** software is the set of objects described in the following paragraphs.

3.1 Module

The graphical appearance of a generic module is show in figure 3.

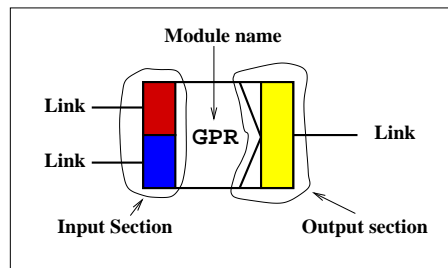


Figure 3: A Typical Module Icon

3.1.1 Object Description

The following code block defines the `Module` object. The term “module” in this context refers to the IDL Object which has been defined to implement the building blocks (also called “modules”) for the **Application Builder**. The definition of `Module` objects follows the IDL guidelines for object programming and contains all the details needed to manage the module representation within the AB. The `Module` object programming interface, ; i.e.: the initialization call and the “methods” defined for the object are, as usual, defined in the first part of the code fragment.

¹We arbitrarily decided not to attempt a port to MacOS.

Modules are characterized by a “Module Name” (a three letters identification code) which identifies the type of the module, i.e.: specifies the action performed by the module within the simulation program; the type of module must be declared when the object is created.

Source file: `module_define.pro`

```

; NAME:
;
;   Module      - Module object
;
; Usage:
;
;   MyObject = Obj_New('Module',Type)      - See the INIT function
;
; Methods:
;
;           Part I: Get module related data
;
;   dataStruct = Module->GetData()         - Get module data
;   ModID = Module->GetID()                - Get module ID
;   slot = Module->GetSlot()               - Get module position (slot)
;   GraphObj = Module->GetGraph()          - Get Graphics
;   Handle = Module->GetHandle(XY)         - Get Data handle
;   InOut=Module->GetIn(Handle)            - Get input handle
;   [x,y]=Module->GetInCoord(Handle)       - Compute input handle coord.
;   InpArray=Module->GetInputs()           - Get array of input links
;   InOut=Module->GetOut(Handle)           - Grt output handle
;   [x,y]=Module->GetOutCoord(Handle)      - Compute output handle coord.
;
;           Part II: Set module characteristics
;
;   status = Module->ChangeDType(dtype)    - Change data type
;   Module->Offset, slotofst,xyofst        - Translate module slot
;   Module->PutLine,Input,Line             - Add a line to module input
;   Module->SetID, ID                      - Set module ID
;   Module->SetInType, handle, dtype       - Set input handle data type
;   Module->SetOutType, handle, dtype      - Set output handle data type
;   Status = Module->SetParams(ProjectName) - Call param interf.
;   Module->SetStatus, Status              - Set module to specified status
;   Module->SetLink(FromMod,FromID,
;                  FromHandle,Input)      - Defines a link
;
;           Part III: Miscellaneous
;
;   Module->DelLink(FromOut,Input)         - Deletes a link
;   Module->List(Unit)                     - Print module structure

```

3.1.2 Method: INIT

Here follows the INIT entry point, i.e.: the standard module creation procedure. In this procedure the object is created, internal data are initialized and the graphic “structure” of the object is defined.

Source file: `module_define.pro`

```

FUNCTION Module::INIT, type                ; type: a 3 character string specifying
;                                          ; the module identification code

COMMON ModuleList, ListPtr, TypeList, Generic_dtype, IOcolors
COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
;                               SlotOXY, FileVersion, AB_Version, AB_Date
COMMON GenDims, ModWidth, ModHeigth, Slotspace

mod_inf=MOD_LIST(1,type)

```

```

self.Graph.Model = OBJ_NEW('IDLgrModel', /SELECT_TARGET)

self.ID = ModIDGen                                ; Generate a unique Module ID
ModIDGen = ModIDGen+1

IF (*mod_inf).rdpar THEN                          $
    self.myColor = [250,0,0]                      $ ; Preset color (RED)
ELSE                                              $
    self.myColor = [0,200,0]                      $ ; Preset color (GREEN, because
                                                ; module doesn't require parameters)

self.myDims=[ModWidth,ModHeighth]
self.myslot= [0,0]
self.myPos = SlotOXY

Xvect = [0, 10, 30, 40, 50, 50, 40, 40, 30, 10, 0] ; Auxiliary vector for
Yvect = [0, 0, 0, 0, 0, 30, 30, 15, 30, 30, 30] ; drawing the module
Zvect = [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2] ; icon

self.Graph.Body = OBJ_NEW('IDLgrModel')          ; "Body" is the central
                                                ; part of the icon

self.Graph.Body->add, OBJ_NEW( 'IDLgrPolyline', Xvect, Yvect, $
    POLYLINES= [ 5, 0, 1, 9, 10, 0,          $
                4, 1, 3, 6, 9,              $
                4, 3, 4, 5, 6,              $
                3, 2, 7, 8,                  $
                ], $
    THICK=3, $
    COLOR=self.myColor, $
    LINSTYLE = 0 )

inp = (*mod_inf).inp_type
out = (*mod_inf).out_type

                                                ; Modules can have 0 or more
                                                ; input handles
cnt=N_ELEMENTS(inp)-1                          ; Initialize input handles
FOR i=0, cnt DO BEGIN
    aux = WHERE(inp[i] EQ TypeList)
    self.Inputs[i].dtype = aux[0]
    self.Inputs[i].Type =1
    self.Inputs[i].handle =i
    self.Inputs[i].ID = -1
    self.Inputs[i].Line = OBJ_NEW()
    self.Inputs[i].Box = OBJ_NEW()
ENDFOR
aux = WHERE(inp NE '',Ninp)
self.Ninp=Ninp

                                                ; Modules can have 0 or more
                                                ; output handles
cnt=N_ELEMENTS(out)-1                          ; Initialize output handles
FOR i=0, cnt DO BEGIN
    aux = WHERE(out[i] EQ TypeList)
    self.Outputs[i].dtype = aux[0]
    self.Outputs[i].Type =2
    self.Outputs[i].handle =i
    self.Outputs[i].ID = -1
    self.Outputs[i].Line = OBJ_NEW()
    self.Outputs[i].Box = OBJ_NEW()
ENDFOR
aux = WHERE(out NE '',Nout)
self.Nout=Nout

```

```

self.Graph.Inp = OBJ_NEW('IDLgrModel')           ; Set the graphic structure
                                                    ; for input and output handles

IF Ninp LE 0 THEN BEGIN                          ; No inputs
  self.Graph.Inp->add, OBJ_NEW('IDLgrPolyline', $
    [0,10, 0,10, 0, 0], $
    [0, 0,30,30, 0,30], $
    [1, 1, 1, 1, 1, 1], $
    THICK=3, $
    COLOR=self.myColor, $
    LINESTYLE = 0 )
ENDIF ELSE BEGIN                                ; One or more inputs
  x0 = 0
  x1 = self.myDims[0]*0.20
  y0 = 0
  Ystep=self.myDims[1]/Ninp
  FOR i=0, Ninp-1 DO BEGIN
    y1=y0+Ystep-0
    color=IOcolors[*,self.Inputs[i].dtype]
    self.Inputs[i].Box= OBJ_NEW( 'IDLgrPolygon', $
      [x0, x1, x1,x0], $
      [y0, y0, y1,y1], $
      [ 0, 0, 0, 0], $
      COLOR=color, $
      STYLE=2, $
      THICK=3, $
      LINESTYLE = 0 )
    self.Graph.Inp->add, self.Inputs[i].Box
    IF i GT 0 THEN self.Graph.Body->add, $
      OBJ_NEW('IDLgrPolyline', $
        [x0,x1], $
        [y0,y0], $
        [ 1, 1], $
        [ 2, 2], $
        COLOR=self.Mycolor, $
        THICK=3, $
        LINESTYLE = 0 )
    ;
    y0=y1
  ENDFOR
ENDELSE

IF Nout LE 0 THEN BEGIN                          ; No outputs
  self.Graph.Inp->add, OBJ_NEW( 'IDLgrPolyline', $
    [40,50,40,50,40,40], $ ; X coords
    [ 0, 0,30,30, 0,30], $ ; Y coords
    [1, 1, 1, 1, 1, 1], $
    THICK=2, $
    COLOR=self.myColor, $
    LINESTYLE = 0 )
ENDIF ELSE BEGIN                                ; One or more outputs
  x0 = self.myDims[0]*0.80
  x1 = self.myDims[0]
  y0 = 0
  Ystep=self.myDims[1]/Nout
  FOR i=0, Nout-1 DO BEGIN
    y1=y0+Ystep
    color=IOcolors[*,self.Outputs[i].dtype]
    self.Outputs[i].Box= OBJ_NEW( 'IDLgrPolygon', $
      [x0, x1, x1,x0], $
      [y0, y0, y1,y1], $
      [ 0, 0, 0, 0], $
      COLOR=color, $

```

```

                STYLE=2,
                LINSTYLE = 0
            )
self.Graph.Inp->add, self.Outputs[i].Box
IF i GT 0 THEN self.Graph.Body->add,
                OBJ_NEW('IDLgrPolyline',
                [x0,x1],
                [y0,y0],
                [ 1, 1],
                [ 2, 2],
                COLOR=self.Mycolor,
                THICK=2,
                LINSTYLE = 0
            )
;
                y0=y1
        ENDFOR
    ENDELSE

self.Type = type;

; Add the module type

Font = OBJ_NEW('IDLgrFont', 'Helvetica*Bold', SIZE = 9.0 )
self.Graph.Text = OBJ_NEW( 'IDLgrText', STRUPCASE(type),
                ALIGNMENT = 0.5,
                COLOR=self.myColor,
                FONT=Font,
                LOCATION = [24,12]
            )

self.Graph.Model->add, self.Graph.Inp
self.Graph.Model->add, self.Graph.Body
self.Graph.Model->add, self.Graph.Text

self.Graph.Model->translate,self.myPos[0],self.myPos[1],0

RETURN, 1
END

```

3.1.3 Method: GetData

The following function returns a structure containing all internal data relevant to the given module. For the sake of efficiency some of the data items returned by this function can also be retrieved singularly by specific methods.

Source file: `module_define.pro`

```

FUNCTION Module::GetData

RETURN, { Type:self.Type,
          ID:self.ID,
          Ninp:self.Ninp,
          Nout:self.Nout,
          Inputs:self.Inputs,
          Outputs:self.Outputs,
          myPos:self.myPos,
          mySlot:self.mySlot,
          myDims:self.myDims,
          Status:self.Status
        }

END

```

3.1.4 Method: GetID

The following function returns the module ID. Module ID is assigned upon the creation of the module by means of a function which ensures the uniqueness of module ID's within the project.

```
Source file: module_define.pro  
  
FUNCTION Module::GetID  
  
RETURN, self.ID  
  
END
```

3.1.5 Method: GetSlot

The following function returns the identification of the slot where the module is located. Slots in the worksheet are identified by couples of numbers in the fashion of two dimensional array elements (see section 3.5).

```
Source file: module_define.pro  
  
FUNCTION Module::GetSlot ; Returns an intarray  
  
RETURN, self.mySlot  
  
END
```

3.1.6 Method: GetGraph

The following function returns the graphic objects which define the aspect of the module on the worksheet. The function returns an IDLgrModel object.

```
Source file: module_define.pro  
  
FUNCTION Module::GetGraph  
  
RETURN, self.Graph.Model  
  
END
```

3.1.7 Method: GetHandle

The following function, given a position within the worksheet (XY coordinates of a point), returns the structure corresponding to the input/output handle found in that position (if any).

```
Source file: module_define.pro  
  
FUNCTION Module::GetHandle, XY ; Point position (intarray)  
; Returns InOut structure  
; The field type of the structure codes  
; the return status as follows:  
  
; -1: Input section, but handle is not free.  
; 0: Parametetr definition section  
; 1: Input section  
; 2: Output section  
  
XYr = XY-self.myPos ; Compute relative point position  
  
Ret={ InOut, Type:0, ID:-1, Handle:0, $
```

```

IF XYr[0] LT self.myDims[0]*0.2 THEN BEGIN      ; Check if in input section
  IF self.Ninp GT 0 THEN BEGIN
    IF self.Ninp EQ 1 THEN Ret.Handle=0 ELSE BEGIN
      IF XYr[1] LT self.myDims[1]*0.5 THEN      $
        Ret.Handle=0                             $
      ELSE                                         $
        Ret.Handle=1
    ENDELSE
  ENDELSE

  Ret.Type=1
  Ret.DType=self.Inputs[Ret.Handle].DType

      ; if Input handle is already in use
      ; return error
  IF self.Inputs[Ret.Handle].ID NE -1 THEN      $      ; Handle in use
    Ret.Type= -1
  Ret.ID=self.ID
ENDIF
ENDIF

IF XYr[0] GT self.myDims[0]*0.8 THEN BEGIN      ; Check if output section
  IF self.Nout GT 0 THEN BEGIN
    IF self.Nout EQ 1 THEN Ret.Handle=0 ELSE BEGIN
      IF XYr[1] LT self.myDims[1]*0.5 THEN      $
        Ret.Handle=0                             $
      ELSE                                         $
        Ret.Handle=1
    ENDELSE
    Ret.Type=2
    Ret.DType = self.Outputs[Ret.Handle].DType
    Ret.ID=self.ID
  ENDELSE
ENDIF
ENDIF

RETURN, Ret

END

```

3.1.8 Method: GetIn

The following functions, given an input handle index returns the corresponding InOut structure

```

Source file: module_define.pro

FUNCTION Module::GetIn, Handle                    ; Get input handle

RETURN, self.Inputs[Handle]

END

```

3.1.9 Method: GetInCoord

The following function, given an input handle structure index returns the corresponding coordinates in the current worksheet.

```

Source file: module_define.pro

FUNCTION Module::GetInCoord, Handle

```

```

step = self.myDims[1]/self.Ninp
start = 0.5*step

xy = [0, step*Handle+start] + self.myPos

RETURN, xy

END

```

3.1.10 Method: GetInputs

The following function, returns an array containing the input handles defined for the module

```

Source file: module_define.pro

FUNCTION Module::GetInputs

RETURN, self.Inputs

END

```

3.1.11 Method: GetOut

The following function, given an output handle index returns the corresponding InOut structure

```

Source file: module_define.pro

FUNCTION Module::GetOut, Handle          ; Get input handle

RETURN, self.Outputs[Handle]

END

```

3.1.12 Method: GetOutCoord

The following function, given an output handle index returns the corresponding coordinates in the current worksheet.

```

Source file: module_define.pro

FUNCTION Module::GetOutCoord, Handle

step = self.myDims[1]/self.Nout
start = 0.5*step

xy = [self.myDims[0], step*Handle+start] + self.myPos

RETURN, xy

END

```

3.1.13 Method: ChangeDType

Module inputs and output have specific data types depending on the type of the module. Some modules, anyway, are defined as “generic” in that they can be used in connection to many different data types (such modules are coded so that they can operate on different data types).

The following function is used to set the actual data type for a generic type module the first time a link is defined to a typed input or output handle.

```

FUNCTION Module::ChangeDType, dtype

COMMON ModuleList, ListPtr, TypeList, Generic_dtype, IOcolors
COMMON GenDims, ModWidth, ModHeight, Slotspace

                                ; Check that module is
                                ; actually generic
                                ; Set actual data type

FOR i=0, self.Ninp-1 DO BEGIN
  IF self.Inputs[i].DType EQ Generic_dtype THEN      $
    self->SetInType,i,dtype                          $
  ELSE RETURN, 0
ENDFOR

FOR i=0, self.Noutp-1 DO BEGIN
  IF self.Outputs[i].DType EQ Generic_dtype THEN    $
    self->SetOutType,i,dtype                        $
  ELSE RETURN, 0
ENDFOR

RETURN, 1

END

```

3.1.14 Method: Offset

When a module is created it is initially assigned to slot [0,0] and then moved to its final location.

The following procedure moves a module to an assigned slot, with given xy position and modifies the coordinates of the graphic elements accordingly.

This procedure assumes that the required slot is free, so the required check must be performed prior of the call.

This procedure only affects the module itself and does not relocate links which might be defined for the module.

Source file: module_define.pro

```

PRO Module::Offset, slotOfst, xyOfst
                                ; Translate module of given offset
                                ; slotOfst (intarray): slot offset
                                ; xyOfst (intarray): corresponding
                                ; coordinate offset

self.Graph.Model->Translate, xyOfst[0], xyOfst[1], 0.0

self.mySlot = self.mySlot + slotOfst
self.myPos = self.myPos + xyOfst

END

```

3.1.15 Method: PutLine

The following procedure adds a link line to the module graphics. The link line is a 'Link' object (see section 3.4)

Source file: module_define.pro

```

PRO Module::PutLine, Input, Line
                                ; Input: input handle specifier (int)
                                ; Line: line (Link obj)

self.Inputs[Input].Line = Line

END

```

3.1.16 Method: SetID

The following procedure assigns a value to the module identifier. The module ID is an identification integer number which must be unique throughout the project.

```
Source file: module_define.pro  
  
PRO Module::SetID, ID ; Set module ID  
self.ID = ID  
END
```

3.1.17 Method: SetInType

The following function is used to set the actual data type for a specified input handle

```
Source file: module_define.pro  
  
PRO Module::SetInType, handle, dtype  
  
COMMON ModuleList, ListPtr, TypeList, Generic_dtype, IOcolors  
COMMON GenDims, ModWidth, ModHeigth, Slotspace  
  
IF handle LT self.Ninp THEN BEGIN  
    self.Inputs[handle].dtype=dtype  
    color=IOcolors[* ,dtype] ; Change color  
    self.Inputs[handle].Box->SetProperty, COLOR=color  
ENDIF  
  
END
```

3.1.18 Method: SetOutType

The following function is used to set the actual data type for a specified output handle

```
Source file: module_define.pro  
  
PRO Module::SetOutType, handle, dtype  
  
COMMON ModuleList, ListPtr, TypeList, Generic_dtype, IOcolors  
COMMON GenDims, ModWidth, ModHeigth, Slotspace  
  
IF handle LT self.Nout THEN BEGIN  
    self.Outputs[handle].dtype=dtype  
    color=IOcolors[* ,dtype] ; Change color  
    self.Outputs[handle].Box->SetProperty, COLOR=color  
ENDIF  
  
END
```

3.1.19 Method: SetParams

Each module has an associated parameter input procedure which must be called to define running parameters.

This procedure calls the module specific GUI (`xxx_gui()`) to allow the user to specify run-time parameters for this instance of the module.

If the call of the parameter definition procedure is successful, the `SetStatus` method is also called to notify the change of status.

```

FUNCTION Module::SetParams, ProjectName      ; Returns 0 on success
                                           ;          1 No par set
                                           ;          2 No par required
                                           ; The name of the current
                                           ; project must be specified
                                           ; in the call.

COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
      SlotOXY, FileVersion, AB_Version, AB_Date
mod_inf=MOD_LIST(1,self.type)

IF (*mod_inf).rdpar THEN BEGIN
  ProcName = STRUPCASE(self.type + '_gui')

  Aux = WHERE(ROUTINE_INFO(/FUNCT) EQ ProcName, Count)

  IF COUNT EQ 0 THEN RESOLVE_ROUTINE, ProcName, /IS_FUNCTION

  LongName = filepath(ProjectName, ROOT=DirName)

  RetVal=CALL_FUNCTION(ProcName,self.ID,LongName)

  IF RetVal EQ 0 THEN self->SetStatus,1 ELSE RetVal=1

ENDIF ELSE RetVal=2

RETURN, RetVal

END

```

3.1.20 Method: SetStatus

This procedure sets the status of a module. The procedure both sets the module status word and modifies other module characteristics accordingly (E.g.: changes the module color to green when the “parameter defined” bit is set).

```

PRO Module::SetStatus, Status              ; The status word bits are defined
                                           ; as follows:
                                           ;
                                           ; Bit   Meaning
                                           ; 1     Set params has been called

                                           ; Other bits are currently undefined
                                           ; and may be used in following releases

IF (Status AND 1) EQ 1 THEN BEGIN          ; Status = Parameter defined
  self.myColor=[0,200,0]

  Parts=Self.Graph.Body->Get(/ALL)

  FOR i=0, N_ELEMENTS(Parts)-1 DO          $
    Parts[i]->SetProperty, COLOR=self.myColor

  Self.Graph.Text->SetProperty, COLOR=self.myColor

  self.Status = self.Status OR 1
ENDIF

END

```

3.1.21 Method: SetLink

The following function, given the specification of the endpoints of a link (a connection between an output handle and an input handle), sets the related information in the module.

```
Source file: module_define.pro

PRO Module::SetLink, FromOutID, FromOutHandle, Input

COMMON ModuleList, ListPtr, TypeList, Generic_dtype, IOcolors

self.Inputs[Input].ID=FromOutID
self.Inputs[Input].Handle=FromOutHandle

END
```

3.1.22 Method: DelLink

The following function, given the specification of an input handle, removes the link (actually sets the ID field to -1) and returns the associated link object which must be explicitly destroyed by the caller.

```
Source file: module_define.pro

FUNCTION Module::DelLink,InputHandle,Input           ; Deletes a link

IF self.Inputs[InputHandle].ID GE 0 THEN BEGIN
    self.Inputs[InputHandle].ID = -1
    line = self.Inputs[InputHandle].Line
    RETURN, Self.Inputs[InputHandle]
ENDIF ELSE RETURN, OBJ_NEW()

END
```

3.1.23 Method: List

The following procedure outputs to an assigned logical unit a textual description of the object.

This method is commonly used to save the module status onto disk.

```
Source file: module_define.pro

PRO Module::List, Unit
IF self.type EQ '+++' THEN
    type=self.type+self.SignChr[0]+self.SignChr[1]
ELSE
    type=self.type

slot = self.mySlot

PRINTF, Unit, '; MODULE: ', type
PRINTF, Unit, ';',Self.ID, ' - ID'
PRINTF, Unit, ';',Self.Status, ' - STATUS'
PRINTF, Unit, ';',slot[0],slot[1],' - SLOT'
PRINTF, Unit, ';',self.Ninp,' - Ninputs'
FOR i=0, self.Ninp-1 DO BEGIN
    IF self.Inputs[i].ID GE 0 THEN BEGIN
        Line = self.Inputs[i].Line
        Xvect=Line->GetX()
        Yvect=Line->GetY()
        NPoints=N_ELEMENTS(Xvect)
    ENDIF ELSE NPoints=0
```

```

        PRINTF,Unit,';',self.Inputs[i].ID,self.Inputs[i].Handle,' - Input',i,$
            '      Nptns:',NPoints,'      Dtype:',Self.Inputs[i].DType
    IF NPoints GT 0 THEN BEGIN
        FOR k=0, NPoints-1 DO
            PRINTF, Unit, ";          ", FIX(Xvect[k]), $
                FIX(Yvect[k])
        ENDIF
    ENDFOR
    PRINTF, Unit, ';',self.Nout,' - Noutputs'
    FOR i=0, self.Nout-1 DO BEGIN
        PRINTF,Unit,';',i,' - Output', '      Dtype:',Self.Outputs[i].DType
    ENDFOR

END

```

3.1.24 Method: Cleanup

The IDL object oriented programming specification requires that any object is provided with a `cleanup` method which is called when the object is destroyed. It is used to destroy other objects which might be contained in the current one.

Source file: `module_define.pro`

```

PRO Module::Cleanup

IF self.Graph.Model NE OBJ_NEW() THEN OBJ_DESTROY, self.Graph.Model
IF self.Graph.Body NE OBJ_NEW() THEN OBJ_DESTROY, self.Graph.Body
IF self.Graph.Inp NE OBJ_NEW() THEN OBJ_DESTROY, self.Graph.Inp
IF self.Graph.Text NE OBJ_NEW() THEN OBJ_DESTROY, self.Graph.Text
FOR i=0, self.Ninp-1 DO BEGIN
    IF self.Inputs[i].Line NE OBJ_NEW() THEN OBJ_DESTROY, self.Inputs[i].Line
    IF self.Inputs[i].Box NE OBJ_NEW() THEN OBJ_DESTROY, self.Inputs[i].Box
ENDFOR
FOR i=0, self.Nout-1 DO BEGIN
    IF self.Outputs[i].Line NE OBJ_NEW() THEN OBJ_DESTROY, self.Outputs[i].Line
    IF self.Outputs[i].Box NE OBJ_NEW() THEN OBJ_DESTROY, self.Outputs[i].Box
ENDFOR

END

```

3.1.25 Data Structure

The following procedure is the required structure definition for the module object.

Source file: `module_define.pro`

```

PRO Module_define          ; Module data structure definition

struct = { Module, Type:' ', $ ; Module type
          ID:-1,          $ ; Module id
          Ninp:0,         $ ; Number of Inputs
          Nout:0,         $ ; Number of outputs
          NinpDef:0,      $ ; Number of Inputs defined
          NoutDef:0,      $ ; Number of Outputs defined
          Inputs: REPLICATE({InOut},2), $ ; Input links
          Outputs: REPLICATE({InOut},2), $ ; Output links
          myPos:[0,0],    $ ; Current position
          mySlot:[-1,-1], $ ; Current slot
          myDims:[0,0],   $ ; Icon dimensions
          Status:0,       $ ; Module status
          myColor:[0,0,0], $ ; Current color

```

```

Graph:{gr, Model:OBJ_NEW(), $; Graphical aspect (Model)
      Body:OBJ_NEW(), $; Box
      Inp:OBJ_NEW(), $; Input graphics (Links)
      Text:OBJ_NEW() } $
}
END

```

3.2 FdbStop Special Module

The graphical appearance of the FdbStop special module is show in figure 4.

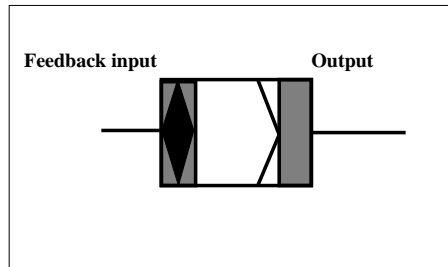


Figure 4: The Feedback Stop special module

3.2.1 Object Description

This code block defines the `Feedback stop` object. It is a special Module and is actually defined as a subclass of the class `Module`.

This module must be used when the project implements a system with feedback as a targeted for the link which actually "closes" the loop. If the loop is closed without the use of this module an error (infinite loop) is issued in the code generation phase.

NOTE: this module will likely be eliminated from the final version of the `Application Builder` to be substituted by the `Combiner` special module (see section 3.3).

The only difference between the `FdbStop` special module and any other module is a special simbol which identifies the input of this module as a possible target for a feedback. For this reason the `FdbStop` special module shares with plain modules all the methods and only differs for the `INIT` one.

The `FdbStop` module type is "generic".

```

Source file: fdbstop_define.pro

; NAME:
;
;       FdbStop      - Feedback stop object
;
; Usage:
;
;       MyFdbStop = Obj_New('FdbStop');
;
; Methods:
;
;       See Module
;

```

3.2.2 Method: INIT

Here follows the INIT entry point. This method creates a module object and then makes the small modifications which distinguish the FeedBack Stop module from a plain module.

```
Source file: fdbstop_define.pro

FUNCTION FdbStop::INIT

COMMON ModuleList, ListPtr, TypeList, Generic_dtype, IOcolors
COMMON GenDims, ModWidth, ModHeigth, Slotspace
COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
        SlotOXY, FileVersion, AB_Version, AB_Date

IF self->Module::INIT('s*s') THEN BEGIN                ; Create module
    self.Graph.Model->remove, self.Graph.Text          ; Remove name

                                                ; Add feedback input
    self.Graph.Body->add,                               $
        OBJ_NEW( 'IDLgrPolygon',                       $
        [5,10,5,0],                                   $      ; X coords
        [0,15,30,15],                                 $      ; Y coords
        [1,1,1,1],                                   $
        STYLE=2,                                       $
        COLOR=[0,0,0],                                 $
        LINSTYLE = 0      )                            ; Add central mark

    RETURN, 1
ENDIF ELSE RETURN, 0

END
```

3.2.3 Data Structure

The following procedure is the required structure definition for the fdbstop object. The data structure is exactly the same as for plain methods.

```
Source file: fdbstop_define.pro

PRO FdbStop_define                ; Module data structure definition

struct = { FdbStop, INHERITS Module }

END
```

3.3 Combiner Special Module

The graphical appearance of a Combiner special module is show in figure 5. The combiner is to be used as a classical summing element in a project. On of the two inputs is marked with a lozenge symbol which identifies the point where the feedback input must be applied. To add flexibility to the design, the module has been provided with the capability to toggle the sign associated with each input.

3.3.1 Object Description

The following code block defines the Combiner object. It is a special Module used to manage the feedback of a closed loop.

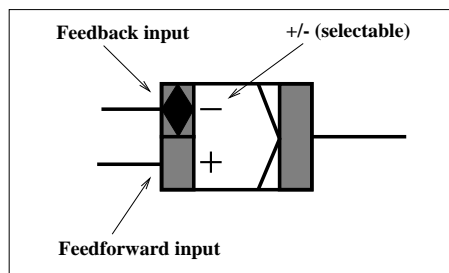


Figure 5: The Combiner special module

This object is defined as a subclass of the module object from which inherits all the methods adding a few of them as specified later.

Source file: `combiner_define.pro`

```

;
; NAME:
;
;   Combiner      - Combiner object
;
; Usage:
;
;   MyCombiner = Obj_New('Combiner');
;
; Methods:
;
;   See Module
;
;   Combiner->ToggleSign0      ; Toggle sign to input 0
;   Combiner->ToggleSign1      ; Toggle sign to input 1
;   Combiner->SetSigns          ; Set both signs
;   signs = Combiner->GetSigns  ; get signs
;   Handle = Combiner->GetHandle, XY ; Get handle

```

3.3.2 Method: INIT

The INIT entry points call the corresponding method of the superclass and then adds a few features (mainly needed for the management of input signs).

Source file: `combiner_define.pro`

```

FUNCTION Combiner::INIT

COMMON ModuleList, ListPtr, TypeList, Generic_dtype, IOcolors
COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
          SlotOXY, FileVersion, AB_Version, AB_Date
COMMON GenDims, ModWidth, ModHeight, Slotspace

IF self->Module::INIT('+++') THEN BEGIN      ; Create module
self.Graph.Model->remove, self.Graph.Text    ; Remove name

                                          ; Add feedback input
self.Graph.Body->add,                      $
          OBJ_NEW( 'IDLgrPolygon',         $
          [5,10,5,0],                      $      ; X coords
          [15,22.5,30,22.5],              $      ; Y coords
          [1,1,1,1],                      $
          STYLE=2,                          $
          COLOR=[0,0,0],                   $

```

LINESTYLE = 0)

```
self.SignChr[0]='+'
self.SignChr[1]='+'
Font = OBJ_NEW('IDLgrFont', 'Helvetica*Bold', SIZE = 12.0 )
self.SignObj[0]=OBJ_NEW('IDLgrText', self.SignChr[0],
                        ALIGNMENT = 0.5,
                        COLOR=[0,0,0],
                        FONT=Font,
                        LOCATION = [18,5] )
self.SignObj[1]=OBJ_NEW('IDLgrText', self.SignChr[1],
                        ALIGNMENT = 0.5,
                        COLOR=[0,0,0],
                        FONT=Font,
                        LOCATION = [18,20] )
self.Graph.Body->add,self.SignObj[0]
self.Graph.Body->add,self.SignObj[1]

RETURN, 1
ENDIF ELSE RETURN, 0

END
```

3.3.3 Method: ToggleSign0

This procedure toggle the sign associated with input 0.

Source file: `combiner_define.pro`

```
PRO Combiner::ToggleSign0

IF self.SignChr[0] EQ '+' THEN BEGIN
    self.SignChr[0]= '-'
ENDIF ELSE BEGIN
    self.SignChr[0]= '+'
ENDELSE
self.SignObj[0]->SetProperty,STRING=self.Signchr[0]
END
```

3.3.4 Method: ToggleSign1

This procedure toggle the sign associated with input 1.

Source file: `combiner_define.pro`

```
PRO Combiner::ToggleSign1

IF self.SignChr[1] EQ '+' THEN BEGIN
    self.SignChr[1]= '-'
ENDIF ELSE BEGIN
    self.SignChr[1]= '+'
ENDELSE
self.SignObj[1]->SetProperty,STRING=self.Signchr[1]
END
```

3.3.5 Method: GetSigns

This function returns the currently defined signs for input 0 and 1

```

FUNCTION Combiner::GetSigns                                ; Returns signs

RETURN, self.SignChr

END

```

3.3.6 Method: SetSigns

This procedure sets the signs for both input 0 and 1.

Source file: combiner_define.pro

```

PRO Combiner::SetSigns,Signs                                ; Set signs

IF STRMID(Signs,0,1) EQ '-' THEN s0='- ' ELSE s0='+'
IF STRMID(Signs,1,1) EQ '-' THEN s1='- ' ELSE s1='+'

self.SignChr = [s0,s1]
self.SignObj[0]->SetProperty,STRING=self.SignChr[0]
self.SignObj[1]->SetProperty,STRING=self.SignChr[1]

END

```

3.3.7 Method: GetHandle

The following function, overrides the module superclass function to allow the detection of “clicks” in the input sign area. It actually changes the response of the object when the pointer is set to the central part of the module icon. Because the combiner has not an associated parameter definition GUI routine, codes to identify the nearest sign on the icon are returned via the InOut structure.

Source file: combiner_define.pro

```

FUNCTION Combiner::GetHandle, XY                            ; Event position
                                                            ; Returns InOut structure

XYr = XY-self.myPos                                        ; Compute relative event position

IF (XYr[0] LT self.myDims[0]*0.2) OR $
  (XYr[0] GT self.myDims[0]*0.8) THEN BEGIN ; Sezione ingresso/uscita
  RETURN, self->Module::GetHandle(XY)
ENDIF ELSE BEGIN ; Central area
  Ret={ InOut, Type:0, ID:-1, Handle:0, $
        DType:0, Box:OBJ_NEW(), Line:OBJ_NEW() }
  Ret.type=5
  IF XYr[0] LT self.myDims[0]*0.5 THEN BEGIN ; Sign area
    IF XYr[1] LT self.myDims[1]*0.5 THEN $
      ret.Type=3 $
    ELSE $
      ret.Type=4
    ENDIF
  ENDELSE

RETURN, Ret

END

```

3.3.8 Data Structure

The following procedure is the required structure definition for the combiner object.

```
Source file: combiner_define.pro

PRO Combiner_define          ; Module data structure definition

struct = { Combiner, SignObj:[OBJ_NEW(),OBJ_NEW()],          $
          SignChr:['+', '+'], INHERITS Module }

END
```

3.4 Link

3.4.1 Object Description

This code block defines the Link object. It is a subclass of the IDLgrPolyline object used to define links connecting module outputs to module inputs.

```
Source file: link_define.pro

; NAME:
;
;   Link      - Link object
;
; Usage:
;
;   MyLink = Obj_New('Link',VectX,VectY);
;
; Methods:
;
;   See: IDLgrPolyline
;
;   Xvec = Link->GetX
;   Yvec = Link->GetY
;   Link->Translate,xyOfst
;
```

3.4.2 Method: INIT

The link object is actually a subclass of the IDL standard graphic 'IDLgrPolyline' to which a few features have been added, as explained below.

```
Source file: link_define.pro

FUNCTION Link::INIT, VectX, VectY

IF self->IDLgrPolyline::INIT( VectX,      $
                             VectY,      $
                             THICK=2,    $
                             COLOR=[0,0,0] ) THEN BEGIN
    self.LinkX=PTR_NEW(VectX)
    self.LinkY=PTR_NEW(VectY)
    RETURN, 1
ENDIF ELSE RETURN, 0

END
```

3.4.3 Method: GetX

This function returns the X coordinate vector defining the link line.

```
Source file: link_define.pro  
  
FUNCTION Link::GetX  
  
RETURN, (*self.LinkX)  
  
END
```

3.4.4 Method: GetY

This function returns the Y coordinate vector defining the link line.

```
Source file: link_define.pro  
  
FUNCTION Link::GetY  
  
RETURN, (*self.LinkY)  
  
END
```

3.4.5 Method: xyOfst

This Procedure translates the entire link to a new position.

```
Source file: link_define.pro  
  
PRO Link::Translate,xyOfst  
  
(*self.LinkX) = (*self.LinkX) + xyOfst[0]  
(*self.LinkY) = (*self.LinkY) + xyOfst[1]  
  
END
```

3.4.6 Data Structure

The following procedure is the required structure definition for the link object.

```
Source file: link_define.pro  
  
PRO Link__define          ; Link data structure definition  
  
struct = { Link, LinkX:PTR_NEW(), LinkY:PTR_NEW(), INHERITS IDLgrPolyline }  
  
END
```

3.5 Grid

3.5.1 Object Description

The graphical aspect of the AB worksheet is a rectangular area with a regular grid of **slots**.

The grid is defined as a subclass of the IDL standard graphic object 'IDLgrModel' by the following code.

```

;
; NAME:
;       Grid - Grid object
;
; This procedure defines the grid for building the application. It
; returns an object of type "IDLgrModel" which can be added to a view
;
; Usage:
;
;       MyGrid = Obj_New('Grid',Xslots,Yslots);
;
; Methods:
;
;       [xs,ys,xc,yc] = Grid::GetSize
;       [x0,x1,y0,y1] = Grid::EnclosingBox, PrjBox ; Returns the coordinates
;                                               ; of the given slot box
;       [xs,ys] = Grid::Screen2slot, xy ; returns slot number corresponding
;                                       ; to screen position
;       [xc,yc] = Grid::Slot2screen, slot ; returns the coordinates of given
;                                       ; slot
;       retstat = Grid::Put, Module, slot ; puts module into given slot
;       Module = Grid::GetModule, slot ; returns module in given slot
;       Module = Grid::Remove, slot ; Remove module from given slot
;

```

3.5.2 Method: INIT

The INIT method initializes a GRID object with a given number of slots in X and Y directions. If the numbers are not provided in the call a suitable default is assumed.

Note: Slot numbers are couples [x,y] with x increasing from left to right and y increasing from top to bottom. The upper left slot is numberd [0,0].

```

FUNCTION Grid::INIT, Xslots, Yslots

COMMON GenDims, ModWidth, ModHeighth, Slotspace

ModWidth=50
ModHeighth=30
SlotSpace=20

IF(self->IDLgrModel::INIT() NE 1) THEN RETURN, 0

self.Xslots=Xslots
self.Yslots=Yslots
self.Xstart=SlotSpace
self.Xspace=ModWidth
self.Xstep=self.Xspace+self.Xstart
self.Ystart=Yslots*(SlotSpace+ModHeighth)
self.Yspace=Modheighth
self.Ystep= SlotSpace+ModHeighth
self.Xsize= Xslots*self.Xstep+self.Xstart
self.Ysize= Yslots*self.Ystep+SlotSpace

Yvert = [self.Ystart, self.Ysize-self.Ystart]

self.Slots=PTR_NEW(OBJARR(self.Xslots,self.Yslots))
self.HConnect=PTR_NEW(INTARR(self.Yslots))
self.VConnect=PTR_NEW(INTARR(self.Xslots))

```

```

self.myColor = [200,200,200]

FOR i=0, self.Xslots-1 DO BEGIN                                ; Draw Boxes
    x0 = self.Xstart+i*self.Xstep
    x1 = x0+self.Xspace
    FOR j=0, self.Yslots-1 DO BEGIN
        y0 = self.Ystart-j*self.Ystep
        y1 = y0-self.Yspace

        box = OBJ_NEW( 'IDLgrPolygon',                        $
                        [x0, x1, x1, x0],                    $ ; X coords
                        [y0, y0, y1, y1],                    $ ; Y coords
                        COLOR=self.myColor,                  $
                        STYLE=1,                              $
                        LINESTYLE = 0                          )
        self->add, box
    ENDFOR
ENDFOR

RETURN, 1
END

```

3.5.3 Method: GetSize

This function returns the size of the grid. The value returned is a four elements integer array which gives both sizes in screen coordinates and in number of slots.

```

Source file: grid_define.pro

FUNCTION Grid::GetSize

RETURN, [self.Xsize, self.Ysize, self.Xslots, self.Yslots]

END

```

3.5.4 Method: EnclosingBox

The following function returns the coordinates (in screen units) of a given rectangular subset of the grid slots.

```

Source file: grid_define.pro

FUNCTION Grid::EnclosingBox, PrjBox ; Returns coordinates of Box enclosing
                                   ; the given slot box

xy1 = self->Slot2screen([PrjBox[0],PrjBox[2]])
xy2 = self->Slot2screen([PrjBox[1],PrjBox[3]])

RETURN, [xy1[0], xy2[0]+self.Xspace, xy1[1]+self.Yspace, xy2[1]]

END

```

3.5.5 Method: Screen2slot

The following function returns the slot number of the grid slot containing the given screen point. The function returns [-1, -1] if the point is not contained in any slot.

```

Source file: grid_define.pro

```

```

FUNCTION Grid::Screen2slot, xy          ; returns the Number of the slot
                                       ; containing given point

xslot=FLOOR((xy[0]-self.Xstart)/self.Xstep)

IF xslot GE self.Xslots THEN RETURN, [-1, -1]
IF xslot LT 0.0          THEN RETURN, [-1, -1]

rel = (xy[0]-self.Xstart) MOD self.Xstep

IF rel GT self.Xspace THEN RETURN, [-1, -1]

yslot=FLOOR((self.Ystart-xy[1])/self.Ystep)

IF yslot LT 0.0          THEN RETURN, [-1, -1]
IF yslot GE self.Yslots THEN RETURN, [-1, -1]

rel = (self.Ystart-xy[1]) MOD self.Ystep
IF rel GT self.Yspace THEN RETURN, [-1, -1]

RETURN, [xslot,yslot]

END

```

3.5.6 Method: Slot2screen

The following function returns the coordinates of the lower left corner of given slot.

```

Source file: grid_define.pro

FUNCTION Grid::Slot2screen, slot        ; returns the coordinates of the
                                       ; lower left point of given slot

IF slot[0] LT 0 OR slot[0] GE self.Xslots THEN RETURN, [-1, -1]
IF slot[1] LT 0 OR slot[1] GE self.Yslots THEN RETURN, [-1, -1]

x = slot[0]*self.Xstep + self.Xstart
y = self.Ysize - (slot[1]+1)*self.Ystep

RETURN, [x, y]

END

```

3.5.7 Method: Put

The following function puts a module into a given slot

```

Source file: grid_define.pro

FUNCTION Grid::Put, Module, slot        ; Put module into given slot
                                       ; return 1 on success

IF (*self.Slots)[slot[0], slot[1]] NE OBJ_NEW() THEN BEGIN
    r=DIALOG_MESSAGE('Slot is not empty')
    RETURN, 0
ENDIF

xy = self->Slot2screen(slot)
offst = xy - Module.myPos
Module.mySlot = slot

```

```
Module.Graph.Model->Translate, offst[0], offst[1], 0.0
```

```
(*self.Slots)[slot[0],slot[1]] = Module
```

```
RETURN, 1
```

```
END
```

3.5.8 Method: GetModule

The following function returns the module contained in given slot.

Note: Empty slots are “filled” with NULLObjects, so if the object returned is OBJ_NEW(), the slot is empty.

Source file: `grid_define.pro`

```
FUNCTION Grid::GetModule, slot          ; returns the module  
                                       ; in given slot
```

```
theObj = (*self.Slots)[slot[0],slot[1]]
```

```
RETURN, theObj
```

```
END
```

3.5.9 Method: Remove

The following function removes a module from a slot. It returns the module object just removed.

Source file: `grid_define.pro`

```
FUNCTION Grid::Remove, slot            ; Remove module from given slot  
                                       ; returns the module removed
```

```
theObj = (*self.Slots)[slot[0],slot[1]]
```

```
IF theObj NE OBJ_NEW() THEN BEGIN
```

```
    offst = -1 * theObj.myPos
```

```
    theObj.Graph.Model->Translate, offst[0], offst[1], 0.0
```

```
    theObj.myPos=[0,0]
```

```
    theObj.mySlot= [-1,-1]
```

```
    (*self.Slots)[slot[0],slot[1]] = OBJ_NEW()
```

```
ENDIF
```

```
RETURN, theObj
```

```
END
```

3.5.10 Method: Cleanup

Source file: `grid_define.pro`

```
PRO Grid::Cleanup
```

```
self->IDLgrModel::Cleanup
```

```
END
```

3.5.11 Data Structure

The following procedure is the required structure definition for the grid object.

```
PRO Grid_define ; Grid data structure definition

struct = { Grid, INHERITS IDLgrModel, $
          Xslots:0, $
          Yslots:0, $
          Xstart:0, $
          Xspace:0, $
          Xstep:0, $
          Xsize:0, $
          Ystart:0, $
          Yspace:0, $
          Ystep:0, $
          Ysize:0, $
          Slots:PTR_NEW(), $
          Hconnect:PTR_NEW(), $
          Vconnect:PTR_NEW(), $
          myColor:[0,0,0] }

END
```

3.6 Project

3.6.1 Object Description

The project object is the support for the management of the set of modules which together represent the simulation application program.

It is actually a dynamic list of modules with the operations needed for list management implemented as methods.

```
Source file: project_define.pro

; NAME:
;
; Project - Project define block
;
; Usage:
;
; Project = Obj_New('Project')
;
; Methods:
; Project->GiveName,'ProjectName' ; Give a name to project
; name=Project->GetName() ; Get project name
; Project->List ; List a Project
; Project->PushModule, Module ; Add a module to project
; graph=Project->GetGraph() ; Returns project graphics
; Project->RemoveGr() ; Removes graphic element
; Project->AddGraph,gr ; Returns project graphics
; Module = Project->PopModule() ; Delete topmost module from
; Project->DelModule, Module ; delete a module from a project
; Module = Project->GetModFromSlot(slot) ; Find the module in given slot
; Module = Project->FindModule(ModID) ; Find the module With given ID
; ModArray= Project->GetList() ; Get module list
; ModArray= Project->ConnList(ID) ; Get list of modules with
; ; given module as input.
; maxModID = Project->GetMaxID() ; Get Maximum ID
; Project->SetMaxID, MaxID ; Set Maximum ID
```

```

;      Box = Project->GetBox()          ; Returns Project box corners:
;                                          ; [XslotMin,XslotMax,YslotMin,YslotMax]
;      Project->Translate,SlotOfst,xyOfst    ; Translate project
;
;
; Notes:
;      A project is a linked list of Modules
;
; \subsubsection{Internal Methods}
;
; The following procedure is not part of the interface of the {\tt project}
; module in that it is only used internally to manage the linked list
; where modules are stored. It is actually used only to define a list
; element as an IDL structure.

PRO ModListElm__define          ; Module List Element structure definition

struct = { ModListElm, Module:OBJ_NEW(),          $ ; This Module
           Next:PTR_NEW()          $ ; Pointer to Next Module
         }
END

```

3.6.2 Method: INIT

Source file: `project_define.pro`

```

FUNCTION Project::INIT, name          ; Initialize Project

self.PrjName = 'newproject'
self.Graph=OBJ_NEW('IDLgrModel')
self.ModGraph=OBJ_NEW('IDLgrModel')

RETURN, 1
END

```

3.6.3 Method: GiveName

This procedure gives a name to the project.

Source file: `project_define.pro`

```

PRO Project::GiveName, name          ; Change name to the project

self.PrjName = name

END

```

3.6.4 Method: GetName

The following function returns the current name of the project.

Source file: `project_define.pro`

```

FUNCTION Project::GetName          ; Returns project name

RETURN, self.PrjName

END

```

3.6.5 Method: GetGraph

Graphic details associated to the project are stored into internally maintained graphic objects of the type `IDLgrModel`. Two objects are used: one maintains the graphic elements of modules in the project and the other keeps other elements (link lines, comments, and the like).

This function returns the two graphic objects as a two elements array.

```
Source file: project_define.pro  
  
FUNCTION Project::GetGraph                                ; Returns project graphics  
  
RETURN, [self.Graph, self.ModGraph]  
  
END
```

3.6.6 Method: AddGraph

The following procedure adds a graphic element to the project

```
Source file: project_define.pro  
  
PRO Project::AddGraph,gr                                ; Returns project graphics  
  
self.graph->add,gr  
  
END
```

3.6.7 Method: RemoveGr

The following procedure removes a graphic element from the project

```
Source file: project_define.pro  
  
PRO Project::Removegr,gr                                ; Removes graphic element  
  
self.graph->remove,gr  
  
END
```

3.6.8 Method: PushModule

The following procedure pushes (adds) a new module to a project.

```
Source file: project_define.pro  
  
PRO Project::PushModule, Module                          ; Add a module to project  
  
NewElm = OBJ_NEW('ModListElm')                          ; Create new list element  
NewElm.Module=Module  
NewElm.Next=Self.Body  
Self.Body=PTR_NEW(NewElm)  
  
Self.counter = Self.counter+1                            ; Update module counter  
gr=Module->GetGraph()  
Self.ModGraph->add,gr                                    ; Add module to project graphics  
  
END
```

3.6.9 Method: PopModule

The following function pops (removes) the topmost module from a project. The removed module object is returned to the caller.

```
Source file: project_define.pro

FUNCTION Project::PopModule                                ; Delete topmost module from
                                                         ; a project list. Return popped
                                                         ; module

RetMod=OBJ_NEW()

IF self.Body NE PTR_NEW() THEN BEGIN
    Current=self.Body
    RetMod=(*Current).Module
    self.ModGraph->remove,RetMod->GetGraph()
    self.Body=(*Current).Next
    OBJ_DESTROY,*Current
    self.Counter = self.Counter-1
ENDIF

RETURN, RetMod

END
```

3.6.10 Method: GetBox

The following function finds the slot box where the project is located and returns it as a four elements array [minX,maxX,minY,maxY].

```
Source file: project_define.pro

FUNCTION Project::GetBox                                  ; Returns Project box dimension
                                                         ; (in slots)

Current = self.Body
Maxx=0
Minx=100000
Maxy=0
Miny=100000

WHILE Current NE PTR_NEW() DO BEGIN
    TheSlot=(*Current).Module->GetSlot()
    IF TheSlot[0] LT Minx THEN Minx=TheSlot[0]
    IF TheSlot[1] LT Miny THEN Miny=TheSlot[1]
    IF TheSlot[0] GT Maxx THEN Maxx=TheSlot[0]
    IF TheSlot[1] GT Maxy THEN Maxy=TheSlot[1]

    Current = (*Current).Next
ENDWHILE

RETURN, [Minx,Maxx,Miny,Maxy]

END
```

3.6.11 Method: Translate

The following procedure translates a project of given amount (both a slot offset and the corresponding screen coordinate offset must be provided).

```
Source file: project_define.pro
```

```

PRO Project::Translate,SlotOfst,xyOfst          ; Translate project

Current = self.Body

WHILE Current NE PTR_NEW() DO BEGIN           ; Translate modules
  (*Current).Module->Offset,SlotOfst,xyOfst
  ModInp=(*Current).Module->GetInputs()
  FOR i=0, N_ELEMENTS(ModInp)-1 DO $
    IF ModInp[i].ID GE 0 THEN ModInp[i].line->Translate,xyOfst
  Current = (*Current).Next
ENDWHILE

                                           ; Translate other graphics
self.graph->Translate,xyOfst[0],xyOfst[1],0

END

```

3.6.12 Method: GetModFromSlot

The following function finds the module of the project which is contained in given grid slot.

```

Source file: project_define.pro

FUNCTION Project::GetModFromSlot, slot          ; Find the module in given slot

RetMod=OBJ_NEW()
Current = self.Body

WHILE Current NE PTR_NEW() DO BEGIN
  TheSlot=(*Current).Module->GetSlot()
  IF Total(TheSlot EQ slot) EQ 2 THEN BEGIN ; Module found
    RetMod=(*Current).Module
    GOTO, LoopEnd ; This terminates WHILE loop
  ENDIF
  Current = (*Current).Next
ENDWHILE
LoopEnd:

RETURN, RetMod

END

```

3.6.13 Method: List

The following function outputs onto a given logical unit a printable representation of the project. It is mainly used to store the project structure on top of the project.pro procedure file (see Sect. 1.4).

```

Source file: project_define.pro

PRO Project::List,Unit,SlotOfst,xyOfst        ; List project

COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
SlotOXY, FileVersion, AB_Version, AB_Date

Box = self->GetBox();

Current = self.Body

PRINTF, Unit,',FILEVER: ', FileVersion
PRINTF, Unit,',PROJECT: ', self.PrjName
PRINTF, Unit, ', ', self.Counter, ' - NMODS'

```

```
PRINTF, Unit,',' , Box[0],Box[1],Box[2],Box[3], '- Box'
```

```
WHILE Current NE PTR_NEW() DO BEGIN  
    (*Current).Module->List,Unit  
    Current = (*Current).Next  
ENDWHILE  
  
END
```

3.6.14 Method: DelModule

The following procedure deletes a module from a project.

Source file: `project_define.pro`

```
PRO Project::DelModule, Module                ; delete a module from a project  
  
Previous=PTR_NEW()  
Current = self.Body  
  
WHILE Current NE PTR_NEW() DO BEGIN  
  
    IF (*Current).Module EQ Module THEN BEGIN ; Module found, delete it  
        IF Previous EQ PTR_NEW() THEN      $  
            self.Body = (*Current).Next    $  
        ELSE                                $  
            (*Previous).Next = (*Current).Next  
  
        TheMod=(*Current).Module  
        self.ModGraph->remove,TheMod->GetGraph()  
        OBJ_DESTROY, *Current  
        OBJ_DESTROY, TheMod  
        self.counter = self.counter-1  
        GOTO, LoopEnd ; This terminates WHILE loop  
    ENDIF  
  
    Previous=Current  
    Current=(*Previous).Next  
ENDWHILE  
  
LoopEnd:  
  
END
```

3.6.15 Method: FindModule

This function scans the module list to find module with given Module ID.

Source file: `project_define.pro`

```
FUNCTION Project::FindModule, ModID  
  
Current = self.Body  
  
EndLoop=0  
Ret=OBJ_NEW()  
  
REPEAT BEGIN  
    IF Current NE PTR_NEW() THEN BEGIN  
        IF (*Current).Module->GetID() EQ ModID THEN BEGIN  
            EndLoop=1  
        ENDIF  
    ENDIF  
END
```

```

                Ret=(*Current).Module
            ENDIF ELSE Current = (*Current).Next
        ENDIF ELSE EndLoop=1
    ENDREP UNTIL EndLoop

    RETURN, Ret

END

```

3.6.16 Method: SetMaxID

For the purpose of generating unique module ID's this function is used to set the max value of ID among all the modules in the project.

```

Source file: project_define.pro

PRO Project::SetMaxID, MaxID                ; Set Maximum ID

self.MaxID = MaxID

END

```

3.6.17 Method: GetMaxID

This function returns the max ID value among project modules.

```

Source file: project_define.pro

FUNCTION Project::GetMaxID                ; Get Maximum ID

RETURN, self.MaxID

END

```

3.6.18 Method: GetList

This function returns an array containing all modules in the project.

```

Source file: project_define.pro

FUNCTION Project::GetList

ModArray = OBJARR(self.Counter)

Current = self.Body

i=0

WHILE Current NE PTR_NEW() DO BEGIN
    ModArray[i]=(*Current).Module
    Current = (*Current).Next
    i=i+1
ENDWHILE

RETURN, ModArray

END

```

3.6.19 Method: Cleanup

Source file: `project_define.pro`

```
PRO project::Cleanup

Current = self.Body

WHILE Current NE PTR_NEW() DO BEGIN
    hold = (*Current).Next
    OBJ_DESTROY, (*Current).Module
    OBJ_DESTROY, *Current
    Current = hold
ENDWHILE

OBJ_DESTROY, self.graph

END
```

3.6.20 Data Structure

The following procedure is the required structure definition for the module object.

Source file: `project_define.pro`

```
PRO Project__define          ; Project data structure definition

struct = { Project,          $
    PrjName: '',             $ ; Project name
    MaxID: 0,                 $ ; Remember max Mod ID
    Counter: 0,               $ ; Internal Counter
    ModGraph: OBJ_NEW(),     $ ; Module graphics
    Graph: OBJ_NEW(),        $ ; other Graphical details
    Body: PTR_NEW()          $ ; Project contents
}

END
```

4 Miscellaneous Utilities (II)

4.1 OS Utilities

Due to the fact that IDL lacks of built-in routines to perform some simple OS operations needed for the implementation of the `Application Builder`, they had to be implemented by means of OS specific commands issued via the `spawn` call. This arises obviously a portability issue and limits the usability of the package to operating system for which a specific version has been developed. At the moment Only Unix-like and Windows 9x based IDL version are supported (see Sect. 2.4).

The following IDL callable procedures and function have been defined for the management of files and directories.

Note: A few other routines to be used for similar purposes have been included among the general library procedures described in a specific document.

4.1.1 Function: `mkdir`

The following procedure creates a new directory relative to the current working directory.

```

; NAME:
;     mkdir
;
; PURPOSE:
;     creates a subdirectory of current working directory.
;
; CATEGORY:
;
;     Miscellaneous
;

PRO mkdir, filename

cmd = 'mkdir ' + filename

spawn, cmd

END

```

4.1.2 Function: `rename`

The following procedure renames a file or a directory

Source file: `rename.pro`

```

PRO rename, fromname, toname

; ----- BEGIN SYSDEP
case !VERSION.OS_FAMILY of
  "unix": begin
    cmd = 'mv -f ' + fromname + ' ' + toname
    spawn, cmd
  end

  "Windows": begin
    cmd = 'move ' + fromname + ' ' + toname
    spawn, cmd
  end

  else: begin
    message, "Operating System not supported"
  end
endcase
; ----- END SYSDEP

END

```

4.1.3 Function: `rmdir`

The following procedure deletes a file or a directory. In the latter case the directory content is removed too.

Source file: `rmdir.pro`

```

PRO rmdir, filename

; ----- BEGIN SYSDEP
case !VERSION.OS_FAMILY of
  "unix": begin
    cmd = 'rm -fr ' + filename

```

```

        spawn, cmd
    end

    "Windows": begin
        cmd = 'echo y|del '+filename+'\*.*'
        spawn, cmd
        cmd = 'rd '+filename
        spawn, cmd
    end

    else: begin
        message, "Operating System not supported"
    end
endcase
; ----- END SYSDEP

END

```

4.2 Access to Module Info

The Application Builder needs to know various details related to the modules and in order to ease the independent development of modules a well defined API has been established.

The following procedures rely on module info API to manage module information.

4.2.1 Procedure: find_mod_info

In order to allow the definition of new modules without the need to modify the AB code, the module list is automatically generated from the info provided by the modules currently generated. The following function scans the module directory for all defined modules and creates a list of all modules found.

Source file: find_mod_info.pro

```

FUNCTION find_mod_info, COUNT=count

; ----- BEGIN SYSDEP
CASE !VERSION.OS_FAMILY of
    "unix": BEGIN
        dir_wildcard = ''
        file_wildcard = '*'
        prefix=!LGS_ENV.modules      ; To override different behaviours
                                     ; of FINDFILE()
    END

    "Windows": BEGIN
        dir_wildcard = STRING(replicate((byte("?")) [0], !LGS_ENV.module_len))
        file_wildcard = dir_wildcard
        prefix=''
    END

    ELSE: BEGIN
        MESSAGE, "Operating System ("+!VERSION.OS_FAMILY+") not supported"
    END
ENDCASE
; ----- END SYSDEP

path = FILEPATH(ROOT=!LGS_ENV.modules, dir_wildcard)
mod_path = FINDFILE(path, COUNT=mod_count)

IF mod_count EQ 0 THEN BEGIN

```

```

        count = 0L
        info_vector = ""
ENDIF ELSE BEGIN
    info_vector = strarr(mod_count)
    count = 0
    FOR i=0,mod_count-1 DO BEGIN
        info_path = FILEPATH(ROOT=prefix+mod_path[i],    $
                               file_wildcard+"_info.pro")
        the_file = FINDFILE(info_path, COUNT=info_count)
        IF info_count EQ 1 THEN BEGIN
            info_vector[count]=the_file
            count = count+1
        ENDIF
    ENDFOR
    IF count eq 0 THEN BEGIN
        info_vector = ""
    ENDIF ELSE BEGIN
        IF count lt mod_count THEN info_vector = info_vector[0:count-1]
    ENDELSE
ENDELSE

RETURN, info_vector
END

```

4.2.2 Function: mod_list

The following function is used to manage the module list. A detailed description of the module list can be found in the related section (sect. 4.2.3).

The function returns the list of modules, or a specified module information structure

Source file: mod_list.pro

```

FUNCTION MOD_LIST, Mode, Spec

COMMON ModuleList, ListPtr, TypeList, Generic_dtype, IOcolors

CASE Mode OF

0: RETURN, ListPtr

1: BEGIN
    Spec = STRLOWCASE(Spec)
    aux=WHERE((*listPtr).type EQ Spec, cnt)
    IF cnt GT 0 THEN ElmPtr = PTR_NEW((*ListPtr)[aux[0]]) ELSE BEGIN
        PRINT, "The requested module (",Spec,") is not available!"
        PRINT, ""
        PRINT, "You ought to upgrade the TMR Application Builder to the"
        PRINT, "Latest version"
        EXIT
    END

    RETURN, ElmPtr
END

ENDCASE

END

```

4.2.3 Procedure: mod_list_crea

The following procedure defines the data structure required for the management of the “module list”.

The routine builds a list of module by calling find_mod_info() and then adds the two special modules (the combiner and the FdbStop) at the end of the list.

The module data are stored into an array of structures allocated in the heap.

Each array element is described in the structure MOD_INFO defined below.

Warning: A table of input/output data types and corresponding handle color is coded statically in the following piece of code, so the code must be modified whenever a new data type is required by any newly defined module.

Source file: mod_list_crea.pro

```
PRO MOD_LIST_CREA

COMMON ModuleList, ListPtr, TypeList, Generic_dtype, IOcolors

TypeList = [    '',                $      ; Null type
               'atm_t',           $      ; Atmosphere type
               'gen_t',           $      ; Generic type
               'shi_t',           $      ; SHS struc. type
               'img_t',           $      ; image/PSF type
               'mir_t',           $      ; mirror geom. struc. type
               'src_t',           $      ; Source type
               'wfp_t',           $      ; propagated Wavefront type
               'mes_t',           $      ; Centroiding meas. type
               'com_t',           $      ; Commands type
               'stf_t',           $      ; Structure Fct type
               ]

Generic_dtype = 2                ; Set this to index of 'gen_t' into TypeList

IOcolors = [    [255,255,255],     $      ; Null type
               [150, 0, 0],       $      ; Atmosphere type
               [120,120,120],     $      ; Generic type
               [ 0,255, 0],       $      ; SHS struc. type
               [255, 0,255],     $      ; image/PSF type
               [ 0, 0,150],       $      ; mirror geom. struc. type
               [ 0, 0,255],       $      ; Source type
               [255,255, 0],     $      ; propagated Wavefront type
               [ 0,150, 0],       $      ; Centroiding meas.. type
               [ 0,255,255],     $      ; commands type
               [255, 0, 0]       $      ; Structure Fct type
               ]

                                   ; Get list of info files
Modules = find_mod_info(COUNT=cnt)

Modules = Modules[Sort(Modules)]   ; Sort modulenames (for WIN 95 compat.)

IF cnt LE 0 THEN BEGIN
    print, "No modules found in directory: ",!lgs_env.Modules
    retall
ENDIF

IF cnt LE 0 THEN BEGIN
    print, "No modules found in directory: ",file
    EXIT
ENDIF

ALL_MOD_INFO = REPLICATE({MOD_INFO, type:''}, $
```

```

        Ninp:0,          $$
        Nout:0,         $$
        inp_type:['',''],  $$
        out_type:['',''],  $$
        init:OB,        $$
        calib:OB,       $$
        rdpar:OB,       $$
        time:OB,        $$
        ver:fix(0),     $$
        Descr:''        }, cnt+2 )

```

FOR i=0, cnt-1 DO BEGIN

```

    Filnam = sep_path(Modules[i])
    ProcAux=STR_SEP(Filnam, '.')
    ProcName=STRLOWCASE(ProcAux[0])

    RESOLVE_ROUTINE, ProcName, /IS_FUNCTION

    RetVal=CALL_FUNCTION(ProcName)

    inp = STR_SEP(RetVal.inp_type, ',')
    IF inp[0] EQ '' THEN Ninp=0 ELSE Ninp=N_ELEMENTS(inp)

    out = STR_SEP(RetVal.out_type, ',')
    IF out[0] EQ '' THEN Nout=0 ELSE Nout=N_ELEMENTS(out)

    ALL_MOD_INFO[i].type      = STRLOWCASE(RetVal.mod_type)
    ALL_MOD_INFO[i].Ninp      = Ninp
    ALL_MOD_INFO[i].Nout      = Nout
    ALL_MOD_INFO[i].inp_type  = inp
    ALL_MOD_INFO[i].out_type  = out
    ALL_MOD_INFO[i].Descr     = RetVal.descr
    ALL_MOD_INFO[i].ver       = RetVal.ver
    ALL_MOD_INFO[i].init      = RetVal.init
    ALL_MOD_INFO[i].time      = RetVal.time
    ALL_MOD_INFO[i].calib     = RetVal.calib
    ALL_MOD_INFO[i].rdpar     = 1 ; NOTA: da aggiungere nella
                                ; descrizione dei moduli

```

ENDFOR

```

                                ; Add Combiner special module at the end of list
ALL_MOD_INFO[cnt].type      = '+++'
ALL_MOD_INFO[cnt].Ninp      = 2
ALL_MOD_INFO[cnt].Nout      = 1
ALL_MOD_INFO[cnt].inp_type  = ['wfp_t','wfp_t']
ALL_MOD_INFO[cnt].out_type  = ['wfp_t']
ALL_MOD_INFO[cnt].Descr     = 'Combiner'
ALL_MOD_INFO[cnt].ver       = 0
ALL_MOD_INFO[cnt].init      = OB
ALL_MOD_INFO[cnt].time      = OB
ALL_MOD_INFO[cnt].calib     = OB
ALL_MOD_INFO[i].rdpar       = OB
cnt=cnt+1

                                ; Add Feedback-stop special module at the end of list
ALL_MOD_INFO[cnt].type      = 's*s'
ALL_MOD_INFO[cnt].Ninp      = 1
ALL_MOD_INFO[cnt].Nout      = 1
ALL_MOD_INFO[cnt].inp_type  = ['gen_t']
ALL_MOD_INFO[cnt].out_type  = ['gen_t']
ALL_MOD_INFO[cnt].Descr     = 'Feedback stop'
ALL_MOD_INFO[cnt].ver       = 0
ALL_MOD_INFO[cnt].init      = OB

```

```

ALL_MOD_INFO[cnt].time      = OB
ALL_MOD_INFO[cnt].calib     = OB
ALL_MOD_INFO[i].rdpar       = OB

ListPtr = PTR_NEW(ALL_MOD_INFO,/NO_COPY)

END

```

4.3 Interaction support

The following routines are used to manage dialog boxes and similar interaction devices.

4.3.1 Function: askconfirm

The following function asks for a generic confirmation. It returns either 1 or 0 if the answer is, respectively “yes” or “no”.

Source file: `askconfirm.pro`

```

FUNCTION AskConfirm, Prompt, Parent

Res = DIALOG_MESSAGE(Prompt,/DEFAULT_NO,DIALOG_PARENT=Parent,/QUESTION)

IF Res EQ 'Yes' THEN r=1 ELSE r=0

RETURN, r
END

```

4.3.2 External Entry Point: SelectFile

The following group of routines of code implement an event driven widget for the selection of a filename among a list. Figure 6 shows the select file widget appearance: the filename can be either selected from the filelist or entered within an editable text field.

The AB code uses the main entry point: `SelectFile` (Sect. 4.3.5).

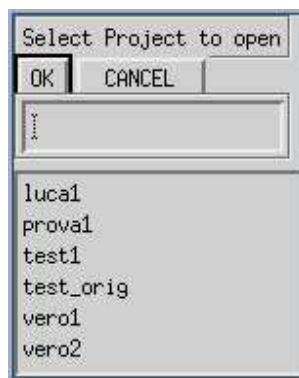


Figure 6: The file Selection Widget

4.3.3 Event handler: newline_event

The following procedure is the event handler for “newline” events generated in the file selection widget. The newline event is usually equivalent to clicking on the “OK” button.

```

PRO newline_event, event

COMMON for_askforfile_only, Ok_ID, Can_ID, File_Id, Field_ID, $
        FileList, Selection

WIDGET_CONTROL, Field_id, get_value=Selection

WIDGET_CONTROL, event.top, /destroy

END

```

4.3.4 Event handler: `selfromlist_event`

The following procedure is the event handler from the “file selected” event. The event is fired when the user presses the mouse button onto the widget. The Event ID value is then used to select among various possible events.

Source file: `selectfile.pro`

```

PRO selfromlist_event, event

COMMON for_askforfile_only, Ok_ID, Can_ID, File_Id, Field_ID, FileList, Selection

IF event.id EQ File_ID THEN BEGIN
        Selection=FileList[event.index] ; Filename selected
        widget_control, Field_id, set_value=Selection ; Remeber file index
ENDIF

IF event.id EQ Can_ID THEN BEGIN
        Selection='' ; Cancel button pressed
        widget_control, event.top, /destroy ; return from widget
ENDIF

IF event.id EQ Ok_ID THEN BEGIN
        IF Selection EQ '' THEN BEGIN ; Ok button pressed
                widget_control, Field_id, $ ; This to retrieve
                get_value=Selection ; the edited value
        ENDIF
        widget_control, event.top, /destroy ; return from widget
ENDIF

END

```

4.3.5 Function: `SelectFile`

The following function is the externally visible entry point. When called it displays the file selection widget and goes to a wait loop for event management. The loop is terminated when the appropriate event is received.

Source file: `selectfile.pro`

```

FUNCTION SelectFile, Title, PrjList, PrjName, Parent ; returns a filename

COMMON for_askforfile_only, Ok_ID, Can_ID, File_Id, Field_ID, FileList, Selection

FileList=PrjList
Selection=''

base = widget_base(TITLE = Title, $ ; setup the base widget

```

```

        Group_leader=Parent,    $
        /modal                  )
                                ; Add a title
Ok_id = Widget_Label(base,     $
        FRAME=3,               $
        Value=Title            )
                                ; Add the OK button
Ok_ID = Widget_Button(base,    $
        value='OK',           $
        YOFFSET=20,           $
        XSIZE=30,             $
        /ALIGN_CENTER,        $
        event_pro='selffromlist_event')
                                ; Add the CANCEL button
Can_ID = Widget_Button(base,   $
        value='CANCEL',       $
        YOFFSET=20,           $
        XOFFSET=30,           $
        XSIZE=70,             $
        /ALIGN_CENTER,        $
        event_pro='selffromlist_event')
                                ; Add the editable text field
field_id = widget_text(base,   $
        event_pro='newline_event', $
        /EDITABLE,           $
        VALUE=PrjName,       $
        FRAME=2,             $
        YOFFSET=40,          $
        XSIZE = 20)
                                ; add the list of files
File_ID = widget_list(base,    $
        event_pro='selffromlist_event', $
        VALUE=PrjList,       $
        XSIZE=20,            $
        YSIZE=5,             $
        YOFFSET=80,          $
        /NO_COPY             )

WIDGET_CONTROL, base, /realize    ; display the widget

XMANAGER, 'SelectFile', base     ; Loop on events

aux=SIZE(Selection)
IF aux[0]>0 THEN RETURN, Selection[0] ELSE RETURN, Selection

END

```

4.4 Project Management

The routines in this section perform various operations for the management of projects.

4.4.1 Function: GetPrjList

The following function returns an array containing a list of currently defined projects. Its main purpose is to isolate operating system dependency in the file scan IDL procedure (FINDFILE) which has slightly different behaviours under Unix and under Windows.

```

FUNCTION GetPrjList                                ; Returns a list of current projects

PrjDir = filepath(ROOT='.',SUB='Projects','')

; ----- BEGIN SYSDEP
CASE !VERSION.OS_FAMILY of
  "unix": BEGIN
    dir_wildcard = ''
    prefix=!LGS_ENV.modules                        ; To override different behaviours
                                                    ; of FINDFILE()
  END

  "Windows": BEGIN
    dir_wildcard = '*'
    prefix=''
  END

  ELSE: BEGIN
    MESSAGE, "Operating System (!VERSION.OS_FAMILY) not supported"
  END
ENDCASE
; ----- END SYSDEP

PrjList = FINDFILE(PrjDir+dir_wildcard, COUNT=cc)

FOR i=0,cc-1 DO BEGIN
  fname = sep_path(PrjList[i], SUB=sub)
  IF fname EQ '' THEN                               $
    PrjList[i] = sub[n_elements(sub)-1]           $
  ELSE                                             $
    PrjList[i]=fname
ENDFOR

idx = WHERE((PrjList NE '.') AND (PrjList NE '..'), cc)
IF cc EQ 0 THEN PrjList = '' ELSE PrjList = PrjList[idx]

RETURN, PrjList

END

```

4.4.2 External Entry Point: OpenProject

The following group of routines are used to open a new project reading data from a project directory. The structure of a project directory has been dealt with in a preceding section (see sect. 1.4). The AB code uses the main entry point: `OpenProject` (Sect. 4.4.6).

4.4.3 Procedure: ScanInput

The following procedure is called by `ScanModules` (see below) to decode the input section of the module textual representation.

```

PRO ScanInput, Mode, Unit, Version, Module, InputN, xyOfst, info
; Scans the Input handle description
; For mode=1, recreates the project
;          structre
; For mode=2, fills in the input data

Fline=''
Px=INTARR(100)

```

Source file: `openproject.pro`

Py=INTARR(100)

Id = Module->GetID()

READF, Unit, Fline

Fline=STRMID(Fline,1,80)

READS, Fline, FromID, FromHandle

IF Mode EQ 1 THEN

IF FromID GE 0 THEN Module->SetLink,FromID, \$
FromHandle,InputN \$

IF Version LE 3 THEN \$

n=STRPOS(Fline,'Extra:') \$; Search for multipoint

ELSE \$; line

n=STRPOS(Fline,'Npts:') ; Search for # of points

IF n GT 0 THEN BEGIN ; line

Fline=STRMID(Fline,n+6,40) ; Extract # of points

READS, Fline, Nxy

ENDIF ELSE Nxy=0

n=STRPOS(Fline,'Dtype:') ; Search input datatype

IF n GT 0 THEN BEGIN

Fline=STRMID(Fline,n+6,40) ; Extract Input type

READS, Fline, Dtype

IF Mode EQ 1 THEN Module->SetInType,InputN,Dtype

ENDIF

IF Version LE 3 THEN BEGIN ; Up to version 3 only extra

FirstK=1 ; line coordinates where stored

LastK= Nxy

ENDIF ELSE BEGIN ; After version 3 all line

FirstK=0 ; coordinates are stored

LastK= Nxy-1

ENDELSE

IF Nxy GT 0 THEN BEGIN

FOR k=FirstK, LastK DO BEGIN ; Read line coordinates

READF, Unit, Fline

Fline=STRMID(Fline,1,80)

READS, Fline, xx,yy

Px[k]=xx

Py[k]=yy

ENDFOR

ENDIF

IF FromID GE 0 THEN BEGIN

IF Mode EQ 2 THEN BEGIN

FromModule=(*info).Project->FindModule(FromID)

ToModule=(*info).Project->FindModule(Id)

IF Version LE 3 THEN BEGIN ; Put in coordinates of

; input and output handles

xy=FromModule->GetOutCoord(FromHandle)

Px[0]=xy[0]

Py[0]=xy[1]

xy=ToModule->GetInCoord(InputN)

Nxy=Nxy+1;

Px[Nxy]=xy[0]

Py[Nxy]=xy[1]

Nxy=Nxy+1;

ENDIF

Px=Px[INDGEN(Nxy)]

Py=Py[INDGEN(Nxy)]

IF Version GT 3 THEN BEGIN ; Relocate coordinates

```

                Px = Px + xyOfst[0]
                Py = Py + xyOfst[1]
            ENDIF
            Line=ComputeLine(0,Px,Py)
            ToModule->PutLine,InputN,Line
            (*info).Project->AddGraph, Line
        ENDIF
    ENDIF

END

```

4.4.4 Procedure: ScanModules

The following procedure is called by doRESTORE (see below) to decode the module textual representation at the beginning of the module section of the project textual representation. It calls ScanInput for each input defined to decode the input specific textual representation.

Source file: `openproject.pro`

```

PRO ScanModules, Mode, Unit, Version, NModules, xyOfst, info
    ; Scans the module description
    ; For mode=1, recreates the project
    ;           structure
    ; For mode=2, fills in the input data

COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
    SlotOXY, FileVersion, AB_Version, AB_Date

Fline=''
slot=INTARR(2)
xyPos=INTARR(2)
Status=0

FOR ModN=0, Nmodules-1 DO BEGIN
    READF, Unit, Fline
    ModType=STRMID(Fline,10,3)           ; Extract mod type from line
    IF ModType EQ ' ' THEN ModType='+++'; For compatibility with
                                           ; FileVersions < 3

    IF Mode EQ 1 THEN BEGIN
        CASE ModType OF
            '+++': BEGIN                     ; Combiner
                Module=OBJ_NEW('Combiner')
                Signs=STRMID(Fline,13,2)
                Module->SetSigns,Signs
            END
            's*s': BEGIN                     ; Feedback stop
                Module=OBJ_NEW('FdbStop')
            END
            ELSE: Module=OBJ_NEW('Module', ModType) ; Generic module
        ENDCASE
    ENDIF

    READF, Unit, Fline
    Fline=STRMID(Fline,1,80)             ; Read Module ID
    READS, Fline, Id
    IF Mode EQ 1 THEN                   $
        Module->SetID,Id                 $
    ELSE                                 $
        Module=(*info).Project->FindModule(Id)
    ENDIF

    IF ModIDGen LT Id THEN ModIDGen=Id+1

```

```

IF Version GT 1 THEN BEGIN
    READF, Unit, Fline
    Fline=STRMID(Fline,1,80)
    READS, Fline, Status          ; Read module status
    IF Mode EQ 1 THEN Module->SetStatus,Status    ; Set module status
END

READF, Unit, Fline
Fline=STRMID(Fline,1,80)          ; Read Slot position
READS, Fline, slot

IF Version LT 3 THEN BEGIN
    READF, Unit, Fline
    Fline=STRMID(Fline,1,80)          ; Read XY position (unused)
    READS, Fline, xyPos              ; Eliminate from FileVersion 3
ENDIF

xyPos = (*info).oGrid->slot2screen(slot)-SlotOXY
IF Mode EQ 1 THEN Module->Offset,slot,xyPos
IF Mode EQ 1 THEN (*info).Project->PushModule,Module

READF, Unit, Fline
Fline=STRMID(Fline,1,80)          ; read number of inputs
READS, Fline, Ninputs

IF Version GT 3 THEN xyOfst = SlotOXY

FOR InputN=0, Ninputs-1 DO BEGIN
    ScanInput, Mode, Unit, Version, Module, InputN, $
        xyOfst, info
ENDFOR

IF Version GT 0 THEN BEGIN          ; Check for file version
    READF, Unit, Fline
    Fline=STRMID(Fline,1,80)          ; read number of outputs
    READS, Fline, Nout
    FOR OutN=0, Nout-1 DO BEGIN
        READF, Unit, Fline
        n=STRPOS(Fline,'Dtype:')          ; Search output datatype
        Fline=STRMID(Fline,n+6,40)        ; Extract # of points
        READS, Fline, Dtype
        IF Mode EQ 1 THEN Module->SetOutType,OutN,Dtype
    ENDFOR
ENDIF
ENDFOR

END

```

4.4.5 Function: doRESTORE

The following procedure is called by `OpenProject` (see below) to decode the project textual representation. After reading the project general parameters the function creates the required objects and then it calls `ScanModules` to decode the modules textual representation.

Note: The scanning of project textual representation is done in two passes: in the first pass the project structure is created, while some data items (notably the data flow links) must be filled in in the second pass.

Source file: `openproject.pro`

```

FUNCTION doRESTORE, savefile, info          ; Returns either '' or error message

COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $

```

```

(*info).Project=OBJ_NEW('Project')

OPENR, Unit,savefile,/GET_LUN

Fline=''
Goon=1
Version=0
WHILE Goon DO BEGIN                                ; Look for Project start
    READF, Unit, Fline
    IF STRMID(Fline, 1, 8) EQ 'PROJECT:' THEN BEGIN
        Goon=0
    ENDIF
    IF STRMID(Fline, 1, 8) EQ 'APPBLDR:' THEN BEGIN
        Fline=STRMID(Fline,10,30)
        READS, Fline, Version
    ENDIF
    IF STRMID(Fline, 1, 8) EQ 'FILEVER:' THEN BEGIN
        Fline=STRMID(Fline,10,30)
        READS, Fline, Version
    ENDIF
ENDWHILE

PrjName=STRMID(Fline,10,70)
PrjName=STRTRIM(PrjName)

(*info).Project->GiveName,PrjName

READF, Unit, Fline
Fline=STRMID(Fline,1,80)                            ; Get rid of comment char
READS, Fline, NModules

PrjBox=INTARR(4)
READF, Unit, Fline
Fline=STRMID(Fline,1,80)                            ; Get rid of comment char
READS, Fline, PrjBox

PrjSize=[PrjBox[1]-PrjBox[0]+1, $                    ; Get Project size
        PrjBox[3]-PrjBox[2]+1]

Gsize = (*info).oGrid->GetSize()                    ; Get worksheet size

IF (Gsize[2] LT Prjsize[0]) OR $
   (Gsize[3] LT PrjSize[1]) $
   THEN RETURN, 'Project too big to fit into worksheet'

POINT_LUN, -Unit, BeginModules                      ; Get file pointer for rewinding

xyOfst=[0,0]

ScanModules, 1, Unit, Version, NModules, $         ; First scan
        xyOfst, info

                                                ; After version 3, graphic elements
                                                ; coordinates are relative to project
                                                ; origin

IF Version GT 3 THEN $
    xyOfst = (*info).oGrid->Slot2screen([PrjBox[0],PrjBox[3]])

;
; Now set up link lines (It is done a separate loop
; because not all the modules are available during
; the first scan)

```

```

POINT_LUN, Unit, BeginModules          ; reset file pointer to beginning
                                        ; of modules description

ScanModules, 2, Unit, Version, NModules, $          ; Second scan
        xyOfst, info

ProjectModified=0

WritePrjStatus, info

PrGraphs = (*info).Project->GetGraph()
(*info).oView->Add,PrGraphs[0]
(*info).oView->Add,PrGraphs[1]

(*info).oWin->DRAW, (*info).oView

specinfo='Drag project box to desired position'
SetUpMoveProject,info

WIDGET_CONTROL, (*info).oTxt, SET_VALUE=specinfo
(*info).oWin->SetCurrentCursor, (*info).cursor          ; Change cursor

RETURN, ''

END

```

4.4.6 Function: OpenProject

This is the main entry point called when an existing project must be opened. It manages the project selection interaction with the user and then calls doRESTORE to restore the project.

```

Source file: openproject.pro

FUNCTION OpenProject, Parent, info          ; Asks for project and opens it

COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
        SlotOXY, FileVersion, AB_Version, AB_Date

prjname=''

PrjList = GetPrjList()

Goon=1

prjname=SelectFile('Select Project to open',PrjList,prjname,Parent)

IF prjname EQ '' THEN RETURN, 'No project opened'

fullpath = filepath(prjname,ROOT=DirName)

found=is_a_dir(fullpath)

IF found THEN BEGIN
        savefile = filepath('project.pro',ROOT=fullpath)

        ret=doRESTORE(savefile,info)

ENDIF ELSE RETURN, 'Project not found'

RETURN, ret

```

END

4.4.7 Procedure: RmProject

The following routine deletes a project from the ./Projects directory.

Source file: `rmproject.pro`

```
PRO RmProject, Parent          ; Removes a project

prjname=''

PrjList=GetPrjList()

prjname=SelectFile('Select Project to delete',PrjList,prjname,Parent)

IF prjname EQ '' THEN RETURN

fullpath = FILEPATH(prjname,ROOT='.',SUB='Projects')

found=is_a_dir(fullpath)

IF found THEN BEGIN
    qst = "Ok to delete project " + prjname + "?"
    IF AskConfirm(qst,Parent) THEN rmdir, fullpath
ENDIF

END
```

4.4.8 External Entry Point: SaveProject

The following set of routines are used to save the status of a project onto disk. Details on the structure of project directories have been dealt with in a preceding section (see sect. 1.4). The AB code uses the main entry point: `SaveProject` (Sect. 4.4.15).

Note: The `SaveProject` function generates the two files `project.pro` and `mod_calls.pro`. Parameter files are created as the result of calling each module parameter definition GUI routine.

4.4.9 Procedure: ModuleCode

The following procedure is called by `WriteCode` (see below) to generate and write onto the output file the function call to invoke a module.

Source file: `saveproject.pro`

```
PRO ModuleCode, ModData, ModInfo, is, Fdp

Nargs = ModData.Ninp + ModData.Nout + 1      ; How many args ?
IF (*ModInfo).init EQ 1B THEN Nargs = Nargs + 1
IF (*ModInfo).time EQ 1B THEN Nargs = Nargs + 1

ModId = STRING(ModData.ID,FORMAT='(I5.5)')

prefix = "ret = " + ModData.Type + "("

FOR k=0, ModData.Ninp-1 DO BEGIN
    IF Nargs LE 1 THEN postfix = ")" ELSE postfix = ",          $"
    Nargs = Nargs-1
    FeedFromId=ModData.Inputs[k].ID
```

```

FeedFromHandle=ModData.Inputs[k].handle
VarPf="0_"
IF FeedFromId LT 0 THEN BEGIN
    FeedFromId=is
    FeedFromHandle=k
    VarPf="I_"
ENDIF
var = VarPf + STRING(FeedFromId,FORMAT='(I3.3)') + '_' + $
    STRING(FeedFromHandle,FORMAT='(I2.2)')

PRINTF, fdP, prefix,var,postfix
prefix = "      "
ENDFOR

FOR k=0, ModData.Nout-1 DO BEGIN
    IF Nargs LE 1 THEN postfix = ")" ELSE postfix = ",          $"
    Nargs = Nargs-1
    var = "0_" + STRING(ModData.ID,FORMAT='(I3.3)') + '_' + $
        STRING(k,FORMAT='(I2.2)')
    PRINTF, fdP, prefix,var,postfix
    prefix = "      "
ENDFOR

;;test to support the case in which there aren't INIT and TIME
IF Nargs LE 1 THEN postfix = ")" ELSE postfix = ",          $"

var=ModData.Type + '_' + ModId + '_p'
PRINTF, fdP, prefix,var,postfix
prefix = "      "
Nargs = Nargs-1

IF (*ModInfo).init EQ 1B THEN BEGIN      ; Module requires
                                        ; initialization
    IF Nargs LE 1 THEN postfix = ")" ELSE postfix = ",          $"
    Nargs = Nargs-1
    var='INIT='+ ModData.Type + '_' + ModID + "_c"
    PRINTF, fdP, prefix,var,postfix
    prefix = "      "
ENDIF

IF (*ModInfo).time EQ 1B THEN BEGIN      ; Module requires
                                        ; time data
    IF Nargs LE 1 THEN postfix = ")" ELSE postfix = ",          $"
    Nargs = Nargs-1
    Var='TIME='+ ModData.Type + '_' + ModID + "_t"
    PRINTF, fdP, prefix,var,postfix
    prefix = "      "
ENDIF

PRINTF, fdP, 'IF ret NE 0 THEN ProjectMsg, "', ModData.Type, ''
PRINTF, fdP, ""

END

```

4.4.10 Procedure: CombinerCode

The following procedure is called by WriteCode (see below) to generate and write onto the output file the code which implements the Combiner special module.

Source file: saveproject.pro

```
PRO CombinerCode, ModData, ModInfo, Signs, Fdp
```

```

Nargs = ModData.Ninp + ModData.Nout + 1          ; How many args ?

ModId = STRING(ModData.ID,FORMAT='(I5.5)')

Outvar = "0_" + STRING(ModData.ID,FORMAT='(I3.3)') + '_' +      $
          STRING(0,FORMAT='(I2.2)')

FeedFromId=ModData.Inputs[0].ID                ; Setup first operand
FeedFromHandle=ModData.Inputs[0].handle
VarPf= "0_"
IF FeedFromId LT 0 THEN BEGIN
    FeedFromId=is
    FeedFromHandle=0
    VarPf="I_"
ENDIF
DirectIn = VarPf + STRING(FeedFromId,FORMAT='(I3.3)') + '_' +    $
          STRING(FeedFromHandle,FORMAT='(I2.2)')

FeedFromId=ModData.Inputs[1].ID                ; Setup second operand
FeedFromHandle=ModData.Inputs[1].handle
VarPf= "0_"
IF FeedFromId LT 0 THEN BEGIN
    FeedFromId=is
    FeedFromHandle=0
    VarPf="I_"
ENDIF
FeedBackIn = VarPf + STRING(FeedFromId,FORMAT='(I3.3)') + '_' +  $
          STRING(FeedFromHandle,FORMAT='(I2.2)')

sf = ".screen"                                ; Set structure field name

PRINTF, fdP, ";----- Loop is closed Here"
PRINTF, fdP, "IF status EQ !LGS_STATUS.run THEN BEGIN           ; RUN status"
PRINTF, fdP, "  IF this_iter LE 1 THEN                          $"
PRINTF, fdP, "    ",Outvar," = ", DirectIn,"                      $"
PRINTF, fdP, "    ",Outvar,sf," = ", Signs[0],DirectIn,sf,"      $"
PRINTF, fdP, "  ELSE                                          $"
PRINTF, fdP, "    ",Outvar," = ", DirectIn,"                      $"
PRINTF, fdP, "    ",Outvar,sf," = ",                               $
          Signs[0],DirectIn,sf,                                   $
          Signs[1],FeedBackIn,sf
PRINTF, fdP, "ENDIF ELSE BEGIN"
PRINTF, fdP, "  IF status EQ !LGS_STATUS.init THEN BEGIN       ; INIT status"
PRINTF, fdP, "    ",Outvar," = ", DirectIn
PRINTF, fdP, "  ENDIF ELSE BEGIN                                   ; CALIB status"
PRINTF, fdP, "    ",Outvar," = ", DirectIn
PRINTF, fdP, "    IF (",FeedBackIn,".data_status EQ !LGS_DATA.valid) THEN BEGIN"
PRINTF, fdP, "      ",Outvar,sf," = ", Signs[0],DirectIn,sf,',+',FeedBackIn,sf
PRINTF, fdP, "      ",Outvar,".data_status = !LGS_DATA.valid"
PRINTF, fdP, "    ENDIF"
PRINTF, fdP, "  ENDELSE"
PRINTF, fdP, "ENDELSE"
PRINTF, fdP, ";-----"
PRINTF, fdP, ""

END

```

4.4.11 Procedure: FdbCode

The following procedure is called by WriteCode (see below) to generate and write onto the output file the code which implements the FeedBack special module.

```

PRO FdbCode, ModData, ModInfo, Fdp

ModId = STRING(ModData.ID,FORMAT='(I5.5)')

Outvar = "0_" + STRING(ModData.ID,FORMAT='(I3.3)') + '_' + $
          STRING(0,FORMAT='(I2.2)')

FeedFromId=ModData.Inputs[0].ID           ; Setup first operand
FeedFromHandle=ModData.Inputs[0].handle
VarPf= "0_"
IF FeedFromId LT 0 THEN BEGIN
    FeedFromId=is
    FeedFromHandle=0
    VarPf="I_"
ENDIF
Invar = VarPf + STRING(FeedFromId,FORMAT='(I3.3)') + '_' + $
          STRING(FeedFromHandle,FORMAT='(I2.2)')

PRINTF, fdP, ";----- Loop is closed Here"
PRINTF, fdP, "IF N_ELEMENTS(",Invar,") GT 0 THEN ", Outvar," = ", Invar
PRINTF, fdP, ";-----"
PRINTF, fdP, ""

END

```

4.4.12 Function: Resolve

The following procedure is called by WriteCode (see below) to derive from the project structure the correct sequence of module calls.

On success modules in the list are associated with a sequence number which represents the order of the function calls in the simulation program.

The function fails if an infinite loop is detected, i.e.: a feedback link is not connected to the special input of any of the "Feedback" special modules (either FdbStop or Combiner).

4.4.13 Algorithm

Note: In the following description we call *ancestors* of a given module all the modules whose output is connected to an input of the latter.

- 1 Initialization. The list of module is scanned once and sequence number 0 is assigned to modules with no input required. These will be computed first.

Then the list of modules is scanned repeatedly, performing the following operations:

- 2.1 To each module with any input connected to a module output it is assigned the sequence number with the following rule:

$$SeqNumber = MAX(SeqNumbers\ of\ ancestors) + 1$$

- 2.2 Any input which is a "feedback input" is ignored in rule 2.1.

- 3 If during the last scan no module has had its sequence number modified the loop is terminated.

```

FUNCTION Resolve, ModArray          ; RETURN:
                                   ; 1 on success
                                   ; 0 on failure (the project has an
                                   ;           infinite loop)

MaxDepth=-1
ToQuote=1
LastMod = N_ELEMENTS(ModArray)-1
WHILE ToQuote DO BEGIN
    ToQuote=0
    FOR i=0, LastMod DO BEGIN          ; Adjust Module depths
        MaxDepth=ModArray[i].Depth
        CASE ModArray[i].Type OF
            '+++': LastInp=ModArray[i].Ninp-2      ; Combiner Ignore close loop inp.
            's*s': LastInp=-1                      ; Feedb. Ignore input
            ELSE: LastInp=ModArray[i].Ninp-1
        ENDCASE

        FOR j=0, LastInp DO BEGIN
            ParentID = ModArray[i].Inputs[j].ID
            Aux=WHERE(ModArray.ID EQ ParentID,cnt1)
            IF cnt1 GT 0 THEN BEGIN
                ParentIndex = Aux[0]
                NewDepth=ModArray[ParentIndex].Depth+1
                IF NewDepth GT MaxDepth THEN BEGIN
                    MaxDepth=NewDepth
                    ToQuote=1
                ENDIF
            ENDIF
        ENDFOR
        ModArray[i].Depth=MaxDepth
    ENDFOR
    IF MaxDepth GT LastMod THEN RETURN, 0      ; Detect infinite loops
ENDWHILE

RETURN, 1

END

```

4.4.14 Function: WriteCode

This procedure scans a project structure and writes out the corresponding IDL code

Source file: saveproject.pro

```

FUNCTION WriteCode, Project, dd, LoopSpec, $ ; RETURN:
    PrjDir, procfile, fdM, fdP              ; 0 on success.
                                             ; 1 Project is empty
                                             ; 2 infinite loop has
                                             ;   been detected

COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
    SlotOXY, FileVersion, AB_Version, AB_Date

IF LoopSpec LE 1 THEN LoopSpec = 1

PrjArray = Project->GetList()

Cnt = N_ELEMENTS(PrjArray)

IF Cnt LE 0 THEN RETURN, 1

```

```

ModArray = MAKE_ARRAY(cnt,
                        VALUE={ Type:'',
                                Ninp:0,
                                Nout:0,
                                ID:0,
                                Depth:0,
                                Inputs: Replicate({InOut}, 2) } )
; Populate module array with data
FOR i=0, cnt-1 DO BEGIN
    modData = PrjArray[i]->GetData()
    ModArray[i].Type=ModData.Type
    ModArray[i].ID=ModData.ID
    ModArray[i].Ninp=ModData.Ninp
    ModArray[i].Nout=ModData.Nout
    ModArray[i].Inputs=ModData.Inputs
ENDFOR

ret=Resolve(ModArray) ; Compute module calling order

if ret EQ 0 THEN RETURN, 2 ; Infinite loop detected

SortIds = SORT(ModArray.Depth)

PRINTF, fdM, ""
PRINTF, fdM, ";;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;"
PRINTF, fdM, "; Error message procedure ;"
PRINTF, fdM, ";;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;"
PRINTF, fdM, ""
PRINTF, fdM, "PRO ProjectMsg, TheMod"
PRINTF, fdM, ";
                                Common Block definition"
PRINTF, fdM, ";
                                ====="
PRINTF, fdM, ";
                                Total nmb Current Current unique"
PRINTF, fdM, ";
                                of iter. iteration status signature"
PRINTF, fdM, ";
                                -----"
PRINTF, fdM, "COMMON lgs_ao_block, tot_iter, this_iter, status, signature"
PRINTF, fdM, ""
PRINTF, fdM, "MESSAGE,""', "Error calling Module: ", $
                                '','', "+TheMod+", '','', " at iteration:", $
                                '','', "+STRING(this_iter)"

PRINTF, fdM, "END"

TheSign=LONG(SYSTIME(1)) ; Generate file signature

PRINTF, fdM, ""
PRINTF, fdM, "COMMON lgs_ao_block, tot_iter, this_iter, status, signature"
PRINTF, fdM, ""
PRINTF, fdM, "tot_iter = ", STRING(LoopSpec)
PRINTF, fdM, "signature = ", TheSign
PRINTF, fdM, "this_iter = -1"
PRINTF, fdM, ";;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;"
PRINTF, fdM, "; Load Parameter variables ;"
PRINTF, fdM, ";;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;"
PRINTF, fdM, ""

FOR i=0, cnt-1 DO BEGIN ; Printout Initialization lines

    is=SortIds[i]

    ModData = PrjArray[is]->GetData()
    ModInfo=MOD_LIST(1, ModData.Type)
    IF (*ModInfo).rdpar THEN BEGIN ; If module requires parameter

```

```

ModuleName = ModData.Type + '_' + $
             STRING(ModData.ID,FORMAT='(I5.5)')

PRINTF, fdM, ""
PRINTF, fdM, "; - Module: ", ModuleName

IF (*ModInfo).init EQ 1 THEN BEGIN                ; initializ.
                                                    ; required
    PRINTF, fdM, ModuleName, '_c=0'
ENDIF

IF (*ModInfo).time EQ 1B THEN BEGIN              ; Time data
                                                    ; required
    PRINTF, fdM, ModuleName, '_t=0'
ENDIF
ParFile= mk_par_name(ModData.type,             $
                    ModData.ID,               $
                    PROJ_NAME=PrjDir)
PRINTF, fdM, "RESTORE, '" + ParFile + '"'
PRINTF, fdM, ModuleName, '_p=par'
ENDIF
ENDFOR

PRINTF, fdM, ""
PRINTF, fdM, ";;;;;;;;;;;;;"
PRINTF, fdM, "; Initialization;"
PRINTF, fdM, ";;;;;;;;;;;;;"
PRINTF, fdM, ""

PRINTF, fdM, "status = !LGS_STATUS.init          ; INIT status"
PRINTF, fdM, 'print, "=== START INITIALIZATION... ==="'
PRINTF, fdM, '@', procfile

PRINTF, fdM, ""
PRINTF, fdM, ";;;;;;;;;;;;;"
PRINTF, fdM, "; Calibration Loop ;"
PRINTF, fdM, ";;;;;;;;;;;;;"
PRINTF, fdM, ""
PRINTF, fdM, "status = !LGS_STATUS.calib_idle      ; CALIBRATION status"

PRINTF, fdM, "REPEAT BEGIN                          ; Calibration Loop"
PRINTF, fdM, ' print, "=== CALIBRATION ITERATION... ==="'
PRINTF, fdM, ' @', procfile
PRINTF, fdM, ' this_iter = 0'
PRINTF, fdM, "ENDREP UNTIL status eq !LGS_STATUS.calib_idle"

PRINTF, fdM, ""
PRINTF, fdM, ";;;;;;;;;;;;;"
PRINTF, fdM, "; Loop Control ;"
PRINTF, fdM, ";;;;;;;;;;;;;"
PRINTF, fdM, ""
PRINTF, fdM, "status = !LGS_STATUS.run            ; RUNNING status"

IF LoopSpec LE 1 THEN BEGIN
    PRINTF, fdM, "; -- Single shot ;"
    PRINTF, fdM, 'print, "=== RUNNING SINGLE SHOT... ==="'
    PRINTF, fdM, ''
    PRINTF, fdM, 'this_iter=1'
    PRINTF, fdM, ''
ENDIF ELSE BEGIN
    PRINTF, fdM, 'print, "=== RUNNING... ==="'
    PRINTF, fdM, "FOR this_iter=1, tot_iter DO BEGIN                ; Begin Main Loop"
    PRINTF, fdM, ' print, "=== ITER. #" +strtrim(this_iter)+"/"+strtrim(tot_iter)+"..."'

```

```

ENDELSE

PRINTF, fdM, ' @', procfile

IF LoopSpec GT 1 THEN PRINTF, fdM, "ENDFOR" ; End Main Loop"

PRINTF, fdM, ""
PRINTF, fdM, ";;;;;;;;;;;;;"
PRINTF, fdM, "; End Main      ;"
PRINTF, fdM, ";;;;;;;;;;;;;"
PRINTF, fdM, ""
PRINTF, fdM, "END"

                                ; Generating routine calls

PRINTF, fdP, "; --"
PRINTF, fdP, "; -- TMR-WPB Project. Application builder. Version " + AB_Version
PRINTF, fdP, "; --"
PRINTF, fdP, "; -- file: mod_calls.pro"
PRINTF, fdP, "; --"
PRINTF, fdP, "; -- Module procedures sequence file for project: ", Project->GetName()
PRINTF, fdP, "; -- Automatically generated on: ", dd
PRINTF, fdP, "; --"
PRINTF, fdP, ""
PRINTF, fdP, "; -- This procedure is invoked at the beginnning of execution (with";
PRINTF, fdP, "; -- the global variable status in lgs_ao_block common set to";
PRINTF, fdP, "; -- !LGS_STATUS.init) to do initialization and then once for every";
PRINTF, fdP, "; -- iteration through the module sequence loop."
PRINTF, fdP, "; -- "
PRINTF, fdP, ""
PRINTF, fdP, "COMMON lgs_ao_block, tot_iter, this_iter, status, signature"
PRINTF, fdP, ""
PRINTF, fdP, "IF signature NE ",TheSign," THEN BEGIN" ; Verify routine conguence
PRINTF, fdP, " PRINT, 'PROCEDURE FILES NOT ALIGNED!'"
PRINTF, fdP, " RETURN"
PRINTF, fdP, "ENDIF"
PRINTF, fdP, ""

FOR i=0, cnt-1 DO BEGIN ; Printout Project computation lines

    is=SortIds[i]

    ModData = PrjArray[is]->GetData()
    ModInfo=MOD_LIST(1,ModData.Type)

    CASE ModData.Type OF
    '+++': BEGIN
        Signs=PrjArray[is]->GetSigns()
        CombinerCode, ModData, ModInfo, Signs, fdP
    END
    's*s': BEGIN
        FdbCode, ModData, ModInfo, fdP
    END
    ELSE: ModuleCode, ModData, ModInfo, is, fdP
    ENDCASE

ENDFOR

RETURN, 0
END

```

4.4.15 Function: SaveProject

This is the main entry point called when a project must be saved onto disk. It manages the project name specification interaction with the user and then creates the project code (files: `project.pro` and `mod_calls.pro`, see detailed description is sect. 1.4).

Source file: `saveproject.pro`

```
PRO SaveProject, info, Parent

COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
        SlotOXY, FileVersion, AB_Version, AB_Date

ThisProject = (*info).project
prjname = ThisProject->GetName()

IF NOT is_a_dir(DirName) THEN mkdir, DirName

PrjList = GetPrjList()

REPEAT BEGIN
    prjname=SelectFile('Specify a name to save Project',PrjList,prjname,Parent)

    IF prjname NE '' THEN BEGIN                ; A name has been specified
        yes=1

        fullname = filepath(prjname,ROOT=DirName)

        IF is_a_dir(fullname) THEN BEGIN
            yes=AskConfirm('Project ' + prjname + ' exists. Override?',Parent)
        ENDIF
    ENDIF ELSE BEGIN                          ; Cancel ...
        fullname=''
        yes=1
    ENDELSE
ENDREP UNTIL yes

IF fullname EQ '' THEN RETURN                ; Save the project structure

savefile = filepath('project.pro',ROOT=fullname)
procfile = filepath('mod_calls.pro',ROOT=fullname)
savetemp = filepath('project.tmp',ROOT=fullname)
proctemp = filepath('mod_calls.tmp',ROOT=fullname)
saveback = filepath('project.bak',ROOT=fullname)
procback = filepath('mod_calls.bak',ROOT=fullname)

IF NOT is_a_dir(fullname) THEN mkdir,fullname

ThisProject->GiveName,prjname

ThisProject->setMaxID, ModIDgen                ; Save Module ID generator

PrjBox=ThisProject->GetBox()
SlotOfst= [PrjBox[0],PrjBox[2]]

xyOfst= (*info).oGrid->Slot2screen(SlotOfst)

dd=SYSTIME(0)

OPENW, fdM,savetemp,/GET_LUN
OPENW, fdP,proctemp,/GET_LUN
PRINTF, fdM, "; --"
```

```

PRINTF, fdM, "; -- TMR-WPB Project. Application builder. Version "+AB_Version
PRINTF, fdM, "; --"
PRINTF, fdM, "; -- file: project.pro"
PRINTF, fdM, "; --"
PRINTF, fdM, "; -- Main procedure file for project: ", Prjname
PRINTF, fdM, "; -- Automatically generated on: ", dd
PRINTF, fdM, "; --"
PRINTF, fdM, ";"

                                ; Write project structure
ThisProject->Translate,-Slot0fst,-xy0fst
ThisProject->List,fdM

                                ; Write Project code
Status=WriteCode(ThisProject,dd,(*info).LoopSpec,fullname,procfile,fdM,fdP)

FREE_LUN,fdM
FREE_LUN,fdP

CASE Status OF
0: BEGIN
    rename, savefile, saveback
    rename, procfile, procbck
    rename, savetemp, savefile
    rename, proctemp, procfile
    r=DIALOG_MESSAGE(['Project saved to files:',savefile,procfile],/INFORMATION)
    ProjectModified=0;
    WritePrjStatus, info
    END
1: BEGIN
    r=DIALOG_MESSAGE(['Project is empty'],/WARNING)
    END
2: BEGIN
    r=DIALOG_MESSAGE(['Infinite loop detected','Project file has not been created'],/ERROR)
    END
ENDCASE

END

```

4.5 Link Management

The following set of routines is used by the main routine worksheet to perform various operations related to creation and deletion of links.

4.5.1 Function: CheckLink

The following function performs the tests required to check if a link can be actually created.

Note: A legal link must join a module output to a free module input, and either data types at the two ends of the link are the same or one of the modules joined is of type generic.

```

                                Source file: checklink.pro

FUNCTION CheckLink, FromM, FromOut, ToM, ToIn

COMMON ModuleList, ListPtr, TypeList, Generic_dtype, IOcolors

OutSpec = FromM->GetOut(FromOut)           ; Get output specification
InSpec = ToM->GetIn(ToIn)                   ; Get input specification

IF OutSpec.dtype EQ InSpec.dtype THEN RETURN, 1

```

```

IF OutSpec.dtype EQ Generic_dtype THEN BEGIN
    rt=FromM->ChangeDType(InSpec.DType)
ENDIF ELSE BEGIN
    IF InSpec.DType EQ Generic_dtype THEN BEGIN
        rt=ToM->ChangeDType(OutSpec.DType)
    ENDIF ELSE rt=0
ENDIF ELSE
RETURN, rt

END

```

4.5.2 Function: ComputeLine

The following function is used to rearrange intermediate points of a link by setting them in the proper order, irrespective of the order defined by the user. Links, in fact, can be defined using either an output or an input as starting point, but are stored in the data structure in the direction *output* → *input*.

As a result of the call a new "Link" object is returned.

Source file: `computeline.pro`

```

FUNCTION ComputeLine, Direction,          $ ; Computes link line coordinates
    VecX, VecY

Nxy=N_ELEMENTS(VecX)

IF Direction EQ 1 THEN BEGIN              ; Reverse order of line points
    LastI = FIX((Nxy-1)/2)
    FOR i=1, LastI DO BEGIN
        exi = Nxy-1-i
        sx = VecX[exi]
        sy = VecY[exi]
        VecX[exi]=VecX[i]
        VecY[exi]=VecY[i]
        VecX[i]=sx
        VecY[i]=sy
    ENDFOR
ENDIF

Line=OBJ_NEW( 'Link', VecX, VecY )

RETURN, Line

END

```

4.5.3 Function: JoinMods

The following function is called to actually create a link when the user terminates the definition of it by clicking on the other end of the link.

Note: A link definition begins when the user clicks on either an input or an output handle of a module and terminates when the user clicks on the other endpoint (output or input, respectively). In between by clicking on other points of the worksheet other intermediate points can be defined.

Source file: `joinmods.pro`

```

FUNCTION JoinMods , FromOut, ToIn, info ; Returns error message

ToM = (*info).Project->FindModule(ToIn.ID)

```

```

FromM = (*info).Project->FindModule(FromOut.ID)

Rt=CheckLink(FromM,FromOut.Handle,    $
             ToM, ToIn.Handle        )

IF rt GT 0 THEN BEGIN    ; Set up link info

    ToM->SetLink,FromOut.ID,          $
             FromOut.Handle,        $
             ToIn.Handle

    IF (*info).LinkDir EQ 0 THEN BEGIN
        Fp= FromM->GetOutCoord(FromOut.Handle)
        Lp= ToM->GetInCoord(ToIn.Handle)
    ENDIF ELSE BEGIN
        Fp= ToM->GetInCoord(ToIn.Handle)
        Lp= FromM->GetOutCoord(FromOut.Handle)
    ENDELSE
    Nxy=(*info).VecIx
    (*info).KeepX[0]=Fp[0]
    (*info).KeepY[0]=Fp[1]
    (*info).KeepX[Nxy]=Lp[0]
    (*info).KeepY[Nxy]=Lp[1]
    Nxy=Nxy+1
    Line=ComputeLine((*info).LinkDir,          $
                    (*info).KeepX[INDGEN(Nxy)], $
                    (*info).KeepY[INDGEN(Nxy)])
    ToM->PutLine, ToIn.Handle, Line
    (*info).Project->AddGraph, Line
    Redraw=1
    msg=''
ENDIF ELSE msg='Link Error!'

RETURN, msg

END

```

4.5.4 Procedure: RmLinks

The following procedure removes all links related to a given module. Both input and output links are removed.

Note: Links are stored in the input section data structure of a module. In order to find output links of a given module a search through all modules must be performed to find modules which receive input from the current one.

```

PRO RmLinks, Module, info                ; Delete all links related to a module

Modlist=(*info).Project->GetList();      Get Module list
cnt=N_ELEMENTS(ModList)-1

MyID = Module->GetID()                   ; Get module ID

                                        ; The following loop deletes input
                                        ; links of all the other modules
                                        ; connected to this one

FOR i=0,cnt DO BEGIN                     ; For each module
    ModData=ModList[i]->GetData()        ; Get module data
    FOR j=0,ModData.Ninp-1 DO BEGIN      ; Analyze module inputs
        IF ModData.Inputs[j].ID EQ MyID THEN BEGIN
            Obj=ModList[i]->DelLink(j)   ; Remove link
        ENDIF
    ENDFOR
ENDFOR

```

Source file: **rmlinks.pro**

```

                (*info).Project->RemoveGr, Obj.line
                OBJ_DESTROY, Obj.line
            ENDIF
        ENDFOR
    ENDFOR
    ModData=Module->GetData()          ; Now delete input links
                                        ; Get module data
    FOR j=0,ModData.Ninp-1 DO BEGIN
        IF ModData.Inputs[j].ID NE -1 THEN BEGIN
            Obj=Module->Dellink(j)      ; Delete link
            (*info).oGrid->remove, Obj.line
            OBJ_DESTROY, Obj.Line
        ENDIF
    ENDFOR
END

```

4.6 Widget Utilities

The following set of routines are used to define and manage various kinds of interactive widgets.

4.6.1 Main Entry Point: LoopCtrl

The following group of routines are used to generate the interaction widget to allow the user to define the number of iterations for the simulation project. The AB code uses the main entry point: LoopCtrl (Sect. 4.6.3).

4.6.2 Procedure: loopctrl_event

This is the event handler for the LoopCtrl widget.

```

Source file: loopctrl.pro

PRO loopctrl_event, event

COMMON for_loopctrl_only, Field_id, field_value

widget_control, Field_id, get_value = field_value

IF Field_value GT 0 THEN widget_control, event.top, /destroy

END

```

4.6.3 Function: LoopCtrl

This is the external entry point to the LoopCtrl function. When called the procedure displays an interaction widget for the definition of the number of iterations to be performed when the project is run.

```

Source file: loopctrl.pro

FUNCTION LoopCtrl, InitVal, Parent

COMMON for_loopctrl_only

base = widget_base(title = 'Define Loop Parameters',          $
                   Group_leader=Parent,                    /modal)

field_id = cw_field(base, title = 'Number of Iterations',    $
                   value = InitVal, xsize = 5,/INTEGER)

```

```

dummy = Widget_Button(base, value='OK',                                     $
                      event_pro='loopctrl_event')

widget_control, base, /realize

xmanager, 'loopctrl', base

RETURN, field_value
end

```

4.6.4 Main Entry Point: mod_menu

The following set of routines are used to generate the module selection menu based on the module list created by mod_list_crea (see sect. 4.2.3). The AB code uses the main entry point: mod_menu (Sect. 4.6.6).

Note: The list of modules generated by mod_list_crea is augmented with the special modules which are displayed properly separated from the other.

4.6.5 Procedure: mod_menu_event

This is the event handler for the Module menu

Source file: mod_menu.pro

```

PRO mod_menu_event, event

widget_control, event.id, get_value=value

; -- name=TAG_NAMES(event, /STRUCTURE_NAME)
; -- PRINT, 'MODMENU_EVENT: ', name

type=STRLOWCASE(STRMID(value,0,3))           ; Extract module type

; -- mod_inf = MOD_LIST(1, type)

widget_control, event.top, get_uvalue=info   ; Get info on window

CASE type OF
'+++': BEGIN
    (*info).KeepMod = OBJ_NEW('Combiner')
    dummy=dialog_message(["The Combiner module will be", $
                          "used in future releases."], $
                          DIALOG_PARENT=event.top)
    END
's*s': (*info).KeepMod = OBJ_NEW('FdbStop')
ELSE: (*info).KeepMod = OBJ_NEW('Module', type)
ENDCASE

(*info).Status='FR'
;(*info).KeepMod->status, (*info).Status

(*info).oWin->SetCurrentCursor, 'ICON'

WIDGET_CONTROL, (*info).oTxt, SET_VALUE='Put module into an empty slot'

; Widget_Control, event.top, SET_UVALUE=info   ; pass the info to window struc

RETURN
END

```

4.6.6 Procedure: mod_menu

This is the external entry point to the mod_menu procedure. When called the procedure displays a pull-down menu on the worksheet widget with the full list of currently available modules

Source file: mod_menu.pro

```
FUNCTION mod_menu, parent

modlptr = MOD_LIST(0)           ; Get module list

Nmods = N_ELEMENTS(*modlptr)

Nspecial=2                     ; Number of special modules

                                ; Note: the string array to describe the
                                ;       menu is built so that the
                                ;       combinator module (the last in
                                ;       module list) is separated from
                                ;       the others.

list = STRARR(Nmods+1)
list = ['0\' + STRUPCASE((*modlptr).type) + ' - ' + (*modlptr).Descr, '' ]

FOR i=Nmods, Nmods-Nspecial+1,-1 DO list[i]=list[i-1]

list[Nmods-Nspecial] = '1\Special'

Id = CW_PDMENU (parent, ['1\Modules\mod_menu_event', list ], $
               /RETURN_NAME, /MBAR
               )

RETURN, Id
END
```

4.6.7 Procedure: NewWorksheet

The following procedure is called whenever a new empty worksheet must be created. If a worksheet is already alive it is first destroyed.

Source file: newworksheet.pro

```
PRO NewWorksheet, info

COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
       SlotOXY, FileVersion, AB_Version, AB_Date

ModIDgen = 1                   ; Initialize Module ID Generator

IF OBJ_VALID((*info).oGrid) THEN OBJ_DESTROY, (*info).oGrid
IF OBJ_VALID((*info).oView) THEN OBJ_DESTROY, (*info).oView
IF OBJ_VALID((*info).Project) THEN OBJ_DESTROY, (*info).Project

(*info).oGrid = OBJ_NEW('Grid', GridD[0], GridD[1])

VwSize = (*info).oGrid->GetSize()

SlotOXY=(*info).oGrid->slot2screen([0,0])

(*info).oView = OBJ_NEW( 'IDLgrView', $
                       VIEWPLANE_RECT = [0,0,VwSize[0],VwSize[1]] )

(*info).oView->add, (*info).oGrid
```

```

(*info).Status = ''
ProjectModified=0

END

```

4.6.8 Procedure: SetOverlay

In order to allow a “project move” operation (to move an entire project around the worksheet) a graphic gadget has been defined which is substantially a shaded rectangular box surrounding the project which can be moved by dragging one of the corners.

The box is implemented as an “overlay widget” superimposed to the worksheet widget. The following procedure is used to set up the “overlay widget” and activate a specific event handler.

Source file: `setoverlay.pro`

```

PRO SetOverlay, Box, Scr, info                ; Set up the Overlay status

COMMON Over_common, Rect, Screen, XStep, YStep, GrPoly, GrRect, GrView

Rect=Box
Screen=Scr

Dxy = (*info).oGrid->slot2screen([1,1]) - (*info).oGrid->slot2screen([0,0])

XStep= Dxy[0]
YStep= Dxy[1]

VwSize=(*info).oGrid->GetSize()
GrView = OBJ_NEW( 'IDLgrView',                $
                  /TRANSPARENT,              $
                  VIEWPLANE_RECT = [0,0,VwSize[0],VwSize[1]] )

GrRect = OBJ_NEW('IDLgrModel')
GrView->add, GrRect

GrPoly = OBJ_NEW('IDLgrPolygon',              $
                  DATA=[ [Rect[0],Rect[2]],   $
                           [Rect[1],Rect[2]], $
                           [Rect[1],Rect[3]], $
                           [Rect[0],Rect[3]] ], $
                  THICK=1,                     $
                  COLOR=[0,0,0],               $
                  STYLE = 1,                   $
                  LINSTYLE = 0                 )

GrRect->add, GrPoly

(*info).oWin->DRAW, GrView

WIDGET_CONTROL, (*info).win, EVENT_PRO='overlay_event'
WIDGET_CONTROL, (*info).win, DRAW_MOTION_EVENTS=1

END

```

4.6.9 Procedure: SetUpMoveProject

The following procedures gathers the data required to define an “overlay widget” which implements the “move project” operation, then call `SetOverlay` (see sect. 4.6.8) to create the widget.

```

PRO SetUpMoveProject,info
    (*info).cursor='UP_ARROW'
    PrjBox=(*info).Project->GetBox()
    Rect = (*info).oGrid->EnclosingBox(PrjBox)
    GridBox=(*info).oGrid->GetSize()
    GBox=[0,GridBox[2]-1,0,GridBox[3]-1]
    Screen = (*info).oGrid->EnclosingBox(GBox)
    SetOverlay, Rect, Screen, info
END

```

4.6.10 Procedure: UnsetOverlay

The following procedure is used to terminate the “overlay widget” created by `SetOverlay` (see sect. 4.6.8).

Source file: `unsetoverlay.pro`

```

PRO UnsetOverlay, info                ; Exit the Overlay status

COMMON Over_common, Rect, Screen, XStep, YStep, GrPoly, GrRect, GrView

OBJ_DESTROY,GrView
OBJ_DESTROY,GrRect
OBJ_DESTROY,GrPoly

WIDGET_CONTROL, (*info).win, EVENT_PRO='worksheet_event'
WIDGET_CONTROL, (*info).win, DRAW_MOTION_EVENTS=0
WIDGET_CONTROL, (*info).oTxt, SET_VALUE=''

(*info).oWin->DRAW, (*info).oView

END

```

4.6.11 Procedure: WritePrjStatus

The following procedure updates the worksheet widget top line where the project status is displayed.

Source file: `writeprjstatus.pro`

```

PRO WritePrjStatus, info

COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
    SlotOXY, FileVersion, AB_Version, AB_Date

txt = 'Project name: ' + (*info).Project->GetName() + '          Status: '

IF ProjectModified THEN stat = 'modified' ELSE stat = 'unmodified'

WIDGET_CONTROL, (*info).Label, SET_VALUE=txt + stat

END

```

5 Main Procedure (III)

The main procedure of the Application Builder is called `worksheet` and its effect is to open a “worksheet widget” as explained above (see sect. 1.3).

From the point of view of software organization this group of routines includes all the event handler related to the worksheet operation and the worksheet procedure itself.

5.1 Event Handlers

5.1.1 Procedure overlay_event

The following procedure is the event handler to manage “overlay” events. Further details on overlays can be found in section 4.6.8.

Source file: `worksheet.pro`

```
PRO overlay_event, myEvent          ; Define the event manager for graphic
                                   ; overlays
COMMON Over_common, Rect, Screen, XStep, YStep, GrPoly, GrRect, GrView

WIDGET_CONTROL, myEvent.top, GET_UVALUE=info

XGuard = ABS(XStep*0.51)
YGuard = ABS(YStep*0.51)

WIDGET_CONTROL, myEvent.top, GET_UVALUE=info
WIDGET_CONTROL, myEvent.top, /CLEAR_EVENTS

CASE myEvent.type OF
0: BEGIN                            ; Button press event

    NewSlot=(*info).oGrid->Screen2slot([Rect[0]+5,Rect[2]-5])
    PrjBox=(*info).Project->GetBox()
    SlotOfst = NewSlot - [PrjBox[0],PrjBox[2]]
    xyOfst = (*info).oGrid->Slot2screen(NewSlot) -           $
              (*info).oGrid->Slot2screen([PrjBox[0],PrjBox[2]])

    (*info).Project->Translate,SlotOfst,xyOfst
    UnsetOverlay, info
END

2: BEGIN                            ; Motion event

                                   ; Drag the Box
IF ABS(myEvent.x-Rect[0]) LT ABS(myEvent.x-Rect[1]) THEN $
    XOfst=myEvent.X-Rect[0]          $
ELSE                                  $
    XOfst=myEvent.X-Rect[1]

IF ABS(myEvent.y-Rect[2]) LT ABS(myEvent.y-Rect[3]) THEN $
    YOfst=myEvent.y-Rect[2]          $
ELSE                                  $
    YOfst=myEvent.y-Rect[3]

IF XOfst LT 0 THEN XOfst=MAX([XOfst,Screen[0]-Rect[0]])
IF XOfst GT 0 THEN XOfst=MIN([XOfst,Screen[1]-Rect[1]])

IF YOfst LT 0 THEN YOfst=MAX([YOfst,Screen[0]-Rect[0]])
IF YOfst GT 0 THEN YOfst=MIN([YOfst,Screen[1]-Rect[1]])

ToMove=0
IF (ABS(XOfst) GT XGuard) THEN BEGIN
    IF XOfst GT 0 THEN XMove = XStep ELSE XMove = -XStep
    ToMove=1
ENDIF ELSE XMove=0
IF (ABS(YOfst) GT YGuard) THEN BEGIN
```

```

        IF Y0fst LT 0 THEN YMove = YStep ELSE YMove = -YStep
        ToMove=1
    ENDIF ELSE YMove=0

    IF ToMove THEN BEGIN
        (*info).oWin->DRAW, (*info).oView

        Rect[0] = Rect[0] + XMove
        Rect[1] = Rect[1] + XMove

        Rect[2] = Rect[2] + YMove
        Rect[3] = Rect[3] + YMove

        GrPoly->SetProperty, DATA=[ [Rect[0],Rect[2]],      $
                                     [Rect[1],Rect[2]],      $
                                     [Rect[1],Rect[3]],      $
                                     [Rect[0],Rect[3]] ]

        (*info).oWin->DRAW, GrView
    ENDIF

    END
ELSE:
ENDCASE

END

```

5.1.2 Procedure worksheet_event

The following procedure is the main event handler related to the worksheet It is used to process all the events related to the worksheet.

Source file: `worksheet.pro`

```

PRO worksheet_event, myEvent

COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
        SlotOXY, FileVersion, AB_Version, AB_Date

WIDGET_CONTROL, myEvent.top, GET_UVALUE=info

name=TAG_NAMES(myEvent, /STRUCTURE_NAME)

; -- PRINT, 'WORKSHEET_EVENT: ', name, ' (Status:', (*info).Status, ')'

IF name EQ 'WIDGET_KILL_REQUEST' THEN BEGIN
    WIDGET_CONTROL, myEvent.top, /DESTROY
    RETURN
ENDIF

Redraw=0
msg=''
wtd='' ; What to do

CASE myEvent.type OF ; Event loop
0: BEGIN ; Button press Events
    slot=(*info).oGrid->Screen2slot([myEvent.x,myEvent.y]) ; Get Slot

    CASE (*info).Status OF

'DL': BEGIN ; Delete Module request
        IF slot[0] LT 0 THEN RETURN

```

```

IF ProjectModified EQ 0 THEN BEGIN
    ProjectModified=1
    WritePrjStatus, info
ENDIF
Module = (*info).Project->GetModFromSlot(slot)
IF Module NE OBJ_NEW() THEN BEGIN
    Point=Module->GetHandle([myEvent.x, myEvent.y])

    CASE Point.type OF
    -1: msg='I' ; Delete input handle
    0: msg='M' ; Delete module
    2: msg='Cannot delete output handle!'
    3: msg='M'
    4: msg='M'
    5: msg='M'
    ELSE: BEGIN ; Input handle empty
        ; cannot delete
        msg='Nothing to do here!'
    END
    ENDCASE
    IF msg EQ 'M' THEN BEGIN ; Actually delete module
        IF ProjectModified EQ 0 THEN BEGIN
            ProjectModified=1
            WritePrjStatus, info
        ENDIF
        RmLinks,Module,info
        gr= Module->Getgraph()
        (*info).oView->Remove, gr ; Remove model from view
        (*info).Project->DelModule, Module ; remove module from project
        OBJ_DESTROY, Module ; Destroy
        (*info).Status='00'
        Redraw=1
        msg=''
    ENDIF
    IF msg EQ 'I' THEN BEGIN ; Actually delete link
        LinkToDelete=Module->DelLink(Point.Handle)
        (*info).Project->RemoveGr, LinkToDelete.Line
        OBJ_DESTROY, LinkToDelete.Line
        (*info).Status='00'
        Redraw=1
        msg=''
    ENDIF
    ENDIF ELSE BEGIN
        msg='Nothing to do here!'
    ENDELSE
        (*info).Status='00'
    END
'FR': BEGIN ; First positioning
    IF slot[0] LT 0 THEN RETURN
    IF (*info).Project->GetModFromSlot(slot) EQ OBJ_NEW() THEN BEGIN
        IF ProjectModified EQ 0 THEN BEGIN
            ProjectModified=1
            WritePrjStatus, info
        ENDIF
        SlotXY = (*info).oGrid->Slot2screen(slot)-SlotOXY
        id=ModIDGen ; Generate unique ID
        ModIDgen=ModIdGen+1
        (*info).KeepMod->SetID, id
        (*info).KeepMod->Offset, slot, SlotXY
        (*info).Project->PushModule,(*info).KeepMod
        (*info).Status='00'
        Redraw=1
    ENDIF

```

```

END

'SM': BEGIN                                ; Select Object for Moving
  IF slot[0] LT 0 THEN RETURN
  Modul=(*info).Project->GetModFromSlot(slot)
  IF Modul NE OBJ_NEW() THEN BEGIN
    IF ProjectModified EQ 0 THEN BEGIN
      ProjectModified=1
      WritePrjStatus, info
    ENDIF
    (*info).Status='MV'          ; Set status for next move
    (*info).KeepMod=Modul      ; save Module reference
    wtd='Move module to an empty slot'
    Redraw=1
  ENDIF
END

'MV': BEGIN                                ; Move to other cell
  IF slot[0] LT 0 THEN RETURN

  Module=(*info).KeepMod
  RmLinks,Module,info
  slot=(*info).oGrid->Screen2Slot([myEvent.x,myEvent.y])
  IF (*info).oGrid->Put((*info).KeepMod,slot) THEN BEGIN
    (*info).Status='00'
    RedrawAllLinks,Module,InAndOut,info
    Redraw=1
  ENDIF
END

'00': BEGIN                                ; Click over Module
                                          ; Join definition
                                          ; or parameter request
  IF slot[0] LT 0 THEN RETURN
  IF ProjectModified EQ 0 THEN BEGIN
    ProjectModified=1
    WritePrjStatus, info
  ENDIF
  Module = (*info).Project->GetModFromSlot(slot)
  IF Module NE OBJ_NEW() THEN BEGIN
    Point=Module->GetHandle([myEvent.x, myEvent.y])

    CASE Point.type OF
      -1: BEGIN                            ; Input handle active: error

          msg='Input Handle already used !'
          (*info).Status='00'
        END

      0: BEGIN                            ; Set param request
          name = (*info).Project->GetName()
          CASE Module->SetParams(name) OF
            0: Redraw=1
            1: wtd='Parameter setting cancelled by user'
            2: wtd='This module has no parameters to set'
          ENDCASE
          (*info).Status='00'
        END

      1: BEGIN                            ; Input selected
          (*info).LinkDir=1
          (*info).VecIx=1
          wtd='Click over module output to join'
    END
  END

```

```

        (*info).Status='EP'
        (*info).KeepPnt=Point
    END
2: BEGIN                                ; Output selected
    (*info).LinkDir=0
    (*info).VecIx=1
    wtd='Click over module input to join'
    (*info).Status='EP'
    (*info).KeepPnt=Point
    END
3: BEGIN                                ; Combiner sign 0 selected
    Module->ToggleSign0;
    (*info).Status='00'
    Redraw=1
    END
4: BEGIN                                ; Combiner sign 1 selected
    Module->ToggleSign1;
    (*info).Status='00'
    Redraw=1
    END
5: BEGIN                                ; Combiner unused area
    msg='Nothing to do here!'
    (*info).Status='00'
    END
    ENDCASE
ENDIF ELSE BEGIN
    msg='Nothing to do here!'
    (*info).Status='00'
ENDELSE
END

'EP': BEGIN                                ; Join end point
    IF ProjectModified EQ 0 THEN BEGIN
        ProjectModified=1
        WritePrjStatus, info
    ENDIF
    Module = (*info).Project->GetModFromSlot(slot)
    IF Module NE OBJ_NEW() THEN BEGIN
        Point=Module->GetHandle([myEvent.x, myEvent.y])

        CASE Point.type OF

            1: BEGIN                                ; Input Handle
                IF (*info).KeepPnt.Type NE 2 THEN BEGIN
                    Err=1
                ENDIF ELSE BEGIN
                    FromOut=(*info).KeepPnt
                    ToIn=Point
                    Err=0
                ENDELSE
            END

            2: BEGIN                                ; Output handle
                IF (*info).KeepPnt.Type NE 1 THEN BEGIN
                    Err=1
                ENDIF ELSE BEGIN
                    FromOut=Point
                    ToIn=(*info).KeepPnt
                    Err=0
                ENDELSE
            END

            0: BEGIN

```

```

        msg = 'No connection here!'
        Err=1
    END
-1: BEGIN
    msg = 'Input Handle already used !'
    Err=1
    END
ENDCASE
IF Msg EQ '' THEN BEGIN
    msg=JoinMods(FromOut,ToIn,info)
    IF Msg EQ '' THEN Redraw=1
ENDIF
(*info).Status='00'

    ENDF ELSE BEGIN
        ; Save intermediate points
        (*info).KeepX[(*info).VecIx]=FIX(myevent.x/5.0+0.5)*5
        (*info).KeepY[(*info).VecIx]=FIX(myevent.y/5.0+0.5)*5
        (*info).VecIx=(*info).VecIx+1
    ENDELSE
END

ELSE: msg= 'Event Loop, Unexpected state: ' + (*info).Status

ENDCASE

IF msg NE '' THEN r=DIALOG_MESSAGE(msg)

WIDGET_CONTROL, myEvent.top, SET_UVALUE=info

IF Redraw THEN (*info).oWin->DRAW, (*info).oView

WIDGET_CONTROL, (*info).oTxt, SET_VALUE=wtd

IF (*info).Status EQ '00' THEN BEGIN
    (*info).oWin->SetCurrentCursor, 'ARROW' ; Reset cursor default
ENDIF

END

4: (*info).oWin->DRAW, (*info).oView ; Expose Events

ELSE:
ENDCASE

RETURN
END

```

5.1.3 Procedure edt_event

The following procedure is the event handler for the management of the **edit** pull-down menu.

Source file: **worksheet.pro**

```

PRO edtEvent, myEvent ; Edit menu event routine

; DBG - name=TAG_NAMES(myEvent, /STRUCTURE_NAME)
; DBG - PRINT, 'FILE_EVENT: ', name

WIDGET_CONTROL, myEvent.id, GET_UVALUE=sel ; Get Menu item ID

WIDGET_CONTROL, myEvent.top, GET_UVALUE=info ; Get window info

```

```

(*info).Status=sel;                                ; Record status

CASE sel OF
'DL': BEGIN
    specinfo='Click: module name (delete module), handle (delete link)'
    (*info).cursor='UP_ARROW'
    END
'SM': BEGIN
    specinfo='Click over module to move'
    (*info).cursor='MOVE'
    END
'FL': BEGIN
    specinfo='This function is not implemented yet!'
    (*info).cursor='ORIGINAL'
    END
'MP': BEGIN
    specinfo='Drag project box to new position'
    SetUpMoveProject,info
    END
ENDCASE

WIDGET_CONTROL, (*info).oTxt, SET_VALUE=specinfo
(*info).oWin->SetCurrentCursor, (*info).cursor           ; Change cursor

WIDGET_CONTROL, myEvent.top, SET_UVALUE=info, /NO_COPY ; Update window info

END

```

5.1.4 Procedure file_event

The following procedure is the event handler for the management of the **file** pull-down menu.

Source file: `worksheet.pro`

```

PRO fileEvent, myEvent           ; dummy event handler

COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
    SlotOXY, FileVersion, AB_Version, AB_Date

; DBG - name=TAG_NAMES(myEvent, /STRUCTURE_NAME)
; DBG - PRINT, 'FILE_EVENT: ', name

WIDGET_CONTROL, myEvent.id, GET_UVALUE=sel

WIDGET_CONTROL, myEvent.top, GET_UVALUE=info

CASE sel OF
'EX': BEGIN
    ; Exit from program
    IF ProjectModified THEN SaveProject, info, myEvent.top

    WIDGET_CONTROL, myEvent.top, /DESTROY
    END
'NW': BEGIN
    IF ProjectModified THEN SaveProject, info, myEvent.top

    NewWorksheet, info
    (*info).Project=OBJ_NEW('Project')
    PrGraphs = (*info).Project->GetGraph()
    (*info).oView->Add,PrGraphs[0]
    (*info).oView->Add,PrGraphs[1]
    (*info).oWin->DRAW,(*info).oView
;    SetWindowName(*info)

```

```

        END
'SV': BEGIN
    SaveProject, info, myEvent.top
    (*info).Status='00'
    END
'OP': BEGIN
    IF ProjectModified THEN SaveProject, info, myEvent.top
    NewWorksheet, info
    msg = OpenProject(myEvent.top,info)
    IF msg NE '' THEN BEGIN
        r=DIALOG_MESSAGE(msg)
    ENDIF
    (*info).Status='00'
    END
'RM': BEGIN                                ; Remove project
    RmProject, myEvent.top
    END
'PR': BEGIN                                ; Print Project
    (*info).oPrinter->DRAW, (*info).oView
    (*info).oPrinter->NewDocument
    END
'PS': BEGIN                                ; Set printer
    ret = DIALOG_PRINTERSETUP((*info).oPrinter)
    END
ELSE:
ENDCASE

END

```

5.1.5 Procedure edt_event

The following procedure is the event handler for the management of the **utilities** pull-down menu.

```

PRO utl_event, myEvent                    ; Utilities menu event handler

COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
    SlotOXY, FileVersion, AB_Version, AB_Date

; DBG - name=TAG_NAMES(myEvent, /STRUCTURE_NAME)
; DBG - PRINT, 'UTL_EVENT: ', name

WIDGET_CONTROL, myEvent.id, GET_UVALUE=sel

WIDGET_CONTROL, myEvent.top, GET_UVALUE=info

CASE sel OF
'EX': BEGIN
    IF Resolve((*info).Project, (*info).LoopSpec) NE 0 THEN    $
        r=DIALOG_MESSAGE('Error from RESOLVE!')
    END
'LP': BEGIN
    (*info).LoopSpec=loopctrl(1,myEvent.top)
    END
ELSE: BEGIN
    r=DIALOG_MESSAGE('Utility not implemented yet!')
    END
ENDCASE

END

```

5.1.6 Procedure hlp_event

The following procedure is the event handler for the management of the edit pull-down menu.

```
Source file: worksheet.pro

PRO hlp_event, myEvent          ; Help menu event routine

COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
                               SlotOXY, FileVersion, AB_Version, AB_date

WIDGET_CONTROL, myEvent.id, GET_UVALUE=sel

WIDGET_CONTROL, myEvent.top, GET_UVALUE=info

CASE sel OF
'AB': r=DIALOG_MESSAGE([ '      TMR-LGS',          $
                        ' Project Builder',        $
                        ' Version '+AB_Version,     $
                        ' L. Fini - '+AB_Date ], $
                        /INFORMATION              )
ELSE: BEGIN
      r=DIALOG_MESSAGE('Utility not implemented yet!')
      END
ENDCASE

END
```

5.2 The Worksheet

5.2.1 Procedure: worksheet

The following procedure is the main entry point invoked by the user to activate the Application Builder.

When called it creates the needed data structures and sets up the worksheet widget, then calls NewWorksheet to activate an empty worksheet and finally starts the widget event driven loop.

```
Source file: worksheet.pro

PRO worksheet, Xslots, Yslots

COMMON Worksheet_Common, ModIDgen, DirName, ProjectModified, GridD, $
                               SlotOXY, FileVersion, AB_Version, AB_Date
COMMON ModuleList, ListPtr, TypeList, Generic_dtype, IOcolors

AB_Version='1.3'                ; Specifies version of Application Builder
AB_Date='May 1999'              ; Specifies date of Application Builder
FileVersion=4                   ; Specifies version of project.pro file

DirName = 'Projects'
ModIDgen = 1

IF N_PARAMS() LT 1 THEN Xslots = 10
IF N_PARAMS() LT 2 THEN Yslots = 10

GridD=[Xslots, Yslots]

LGS_INIT                        ; Initialize LGS system

MOD_LIST_CREA                   ; Create module list

RESOLVE_ROUTINE, 'rmdir'       ; To avoid some pitfalls
```

RESOLVE_ROUTINE, 'Module_define'

```
InOut = { InOut, Type:0,      $      ; Modyle type index
          ID:-1,           $      ; Module ID
          Handle:0,       $      ; Output module handle
          DType:0,        $      ; Data type index
          Box:OBJ_NEW(),  $      ; Associated colored box
          Line:OBJ_NEW() }      ; Associated line
                                ; Set up Widgets
```

```
info = PTR_NEW( {oWin:OBJ_NEW(),      $
                oView:OBJ_NEW(),      $
                oGrid:OBJ_NEW(),      $
                oPrinter:OBJ_NEW(),   $
                cursor:'' ,           $
                win:0,                $
                Wks:0,                $
                Label:0,              $
                KeepMod:OBJ_NEW(),     $
                KeepX:INTARR(100),    $
                KeepY:INTARR(100),    $
                VecIx:0,               $
                LinkDir:0,            $
                oTxt:0,               $
                Status:'' ,           $
                KeepPnt:{InOut,0,-1,0,0,OBJ_NEW(),OBJ_NEW()} , $
                LoopSpec:1,           $
                Project:OBJ_NEW(),     $
                PrjStatus:0           } )
```

```
wks = Widget_base(TITLE='Application Builder', $
;               ROW=3,                       $
;               /COLUMN, APP_MBAR = TopBar)
;               APP_MBAR = TopBar)
```

```
fileMenu = Widget_button(TopBar, VALUE='File', $ ; Set up File menu
                          EVENT_PRO='fileEvent', $
                          /MENU)
```

```
newMenuBut = Widget_button(fileMenu, $
                           VALUE='New Project', $
                           UVALUE='NW')
```

```
openMenuBut = Widget_button(fileMenu, $
                             VALUE='Open Project', $
                             UVALUE='OP')
```

```
svMenuBut = Widget_button(fileMenu, $
                           VALUE='Save Project', $
                           UVALUE='SV')
```

```
rmMenuBut = Widget_button(fileMenu, $
                           VALUE='Delete Project', $
                           UVALUE='RM')
```

```
prMenuBut = Widget_button(fileMenu, $
                           VALUE='Print Project', $
                           UVALUE='PR')
```

```
prMenuBut = Widget_button(fileMenu, $
                           VALUE='Set Printer', $
                           UVALUE='PS')
```

```
exitMenuBut = Widget_button(fileMenu, $
                             VALUE='Exit', $
                             UVALUE='EX', $
                             /SEPARATOR )
```

```
edtMenu = Widget_button(TopBar, VALUE='Edit', $ ; Set up edit menu
                        EVENT_PRO='edtEvent', $
                        /MENU)
```




```

(*info).oTxt = Widget_text(wks)

WritePrjStatus, info

Widget_Control, wks, /REALIZE

Widget_Control, (*info).win, GET_VALUE= oWin

(*info).oWin=oWin

(*info).oPrinter=OBJ_NEW('IDLgrPrinter',COLOR_MODEL=1) ; Preset printer object

oWin->draw, (*info).oView

Widget_Control, wks, SET_UVALUE=info ; Prepare to pass the info to objects

Xmanager, 'worksheet', wks, /NO_BLOCK ; Start the X manager

END

```

References

- M. Carbillet, F. Delplancke, B. Femenía, L. Fini, M. Le Louarn, A. Riccardi, E. Viard, "WPB Deliverable Report". LGS-TMR Report: LGS/ESO/DEL/WPB/1.1, 28 Jan 1999.
- F. Delplancke, M. Carbillet, N. Hubin, S. Esposito, F. Rigaut, E. Marchetti, A. Riccardi, E. Viard, R. Ragazzoni, M. Le Louarn, L. Fini, "Laser guide star simulation for 8-m class telescopes". *Proceedings of SPIE Conference on Adaptive Optical System Technologies*, Kona, Hawaii 23-26 march, Volume 3353, D.Bonaccini and R.K.Tyson (eds.), 1998, pp. 371-382.
- M. Carbillet, A. Riccardi, B. Femenía, L. Fini, S. Esposito, "Preliminary results of simulations for the adaptive optics system of the Large Binocular Telescope", *SAIt's Conf. Techno'99: Telescopes, instruments and data processing for astronomy in the year 2000*, Napoli, Maggio 1999.
- M. Carbillet, F. Delplancke, S. Esposito, B. Femenía, L. Fini, A. Riccardi, E. Viard, N. Hubin, M. LeLouarn, F. Rigaut, "A software package for laser guide star adaptive optics systems". *SPIE's Int. Symp. on Astronomical Telescopes and Instrumentation*, Denver, Colorado, July 1999.
-