Automatic Class Generation Tool User's Guide

Laurent Claustre

May 14 1993
DSMU002 Revision: 2.0

Contents

1 Introduction **3**

1.1 Template files . . . 3

1.2 The input files . . . 4

1.3 Files to generate . . . 5

2 About ASCII version: classgen **7**

2.1 Write an input file . . . 7

2.1.1 Respect a well defined grammar . . . 7

2.1.2 The header . . . 9

2.1.3 **The command list** . . . 9

2.1.4 The state machine . . . 10

2.1.5 The classfields list . . . 11

2.1.6 The objectfields list . . . 12

2.1.7 The descriptions . . . 12

2.1.8 An example of inputfile . . . 13

2.2 Run the generator . . . 13

2.2.1 Location . . . 13

2.2.2 Use the generator . . . 13

2.2.3 Error messages . . . 14

3 About the X version: xclassgen **19**

3.1 Location . . . 19

3.2 Use the generator . . . 20

3.2.1 The menus 20

3.2.2 How to modifie characteristics of the class 21

4 The two process Device Server 23

5 Generated files: where to append code 25

5.1 for all the Device servers 25

5.1.1 Class.c 25

5.1.2 Class.res 25

5.1.3 Class_menu.c 26

5.1.4 Class.tex 26

5.2 For the Two Process Device Servers 27

5.2.1 ClassP.h 27

5.2.2 Class_sub.c 27

5.2.3 Class.c 28

5.3 Conclusion 29

Chapter 1

Introduction

The Control Group has written roughly 90 high level device classes for equipment interfaced to the accelerators and it seems obvious that we should make the implementation of the future device servers easier. This is possible with an automatic class generation tool.

The Generator should, starting from a set of input specifications, automatically produce all the files necessary to implement a device class. It is not possible to automatically generate the device specific code, only skeleton files which require editing in order to implement the full functionality.

The main goal of a such generator is to prepare the work for the device class programmer in the most efficient way and to provide a standard representation for all the classes (e.g. source, includes, user's guide, directory structure ...).

The generator can provide two kinds of classes. One class type contains one process; the other class contains two processes. For more informations about the two process Device Server Model see chapter 4.

There are two version of the generator,an ascii version and a X version. The generator should take all the characteristics of the device class from an **input file** (the specifications file) and generate the target files starting from a set of **template files**. The notion of template files is important for maintenance of the generator. By modifying the template files evolutionary changes to the device classes can be incorporated without any modifications to the generator.

The X generator can also take current characteristics to generate the target files and it can save the characteristics in a file.

The figure 1.1 shows the input and output of the the generator.

1.1 Template files

The template files are the following:

- **Template.c**, source file.
- **TemplateP.h**, private include file.

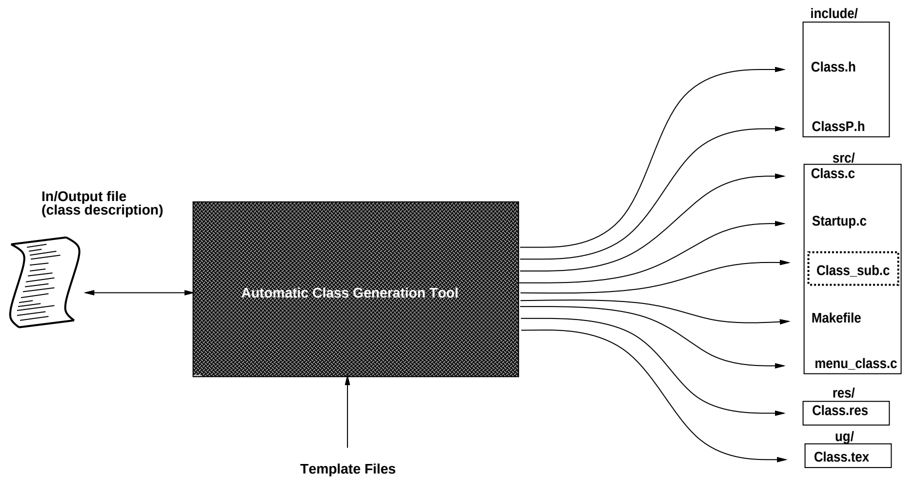

Figure 1.1: I/O of the generator

- **Template.h**, public include file.
- **Template.res**, resources file (database).
- **Startup.c**, containing the startup function should be called directly by main stub of the server. In the startup the server will create all necessary devices, intialise and export them.
- **Template_menu.c**, a small menu driver program used for debugging.
- **Makefile**.
- **Template.tex**,a LaTeX format user's guide.
- **Template_sub.c**, only use for two processes device server, containing the main() routine of the subprocess.

1.2 The input files

This file, by convention, must have a **.dat** suffix. It must contain all the useful information specific to the desired future classes. It should respect a well defined grammar (see the section 2.1.1).

A example of input file is available in *libra:/users/d/dserver/system/classgen/ examples/inputfile.dat* file.

1.3 Files to generate

All the files are places in different directories, include, src, res and ug, there are :

- **Class.c**
- **ClassP.h**
- **Class.h**
- **startup.c**
- **Makefile**
- **Class_menu.c**
- **Class.tex**
- **Class_sub.c**

All the generated files will be correctly commented (using comments from the data file) and will contain a header adapted to be managed under RCS.

Chapter 2

About ASCII version: classgen

2.1 Write an input file

The generator provides an alpha numeric menu whose use is very simple. The difficulty will be to write an input file without syntax errors. It's very important that you read this section since your input file isn't correctly written you will not be able to use the generator.

2.1.1 Respect a well defined grammar

What is a grammar?

The syntax of a program can be defined by a *context-free grammar*. A context-free grammar is essentially a hierarchical structure that indicates the relationship of various language constructs.

The description of this grammar is in the form of *production rules*. Such production rules simply consist of a name or nonterminal, the left-hand side, followed by its definition. The definition, or right-hand side, consists of zero or more symbols which are nonterminals in that they point to other rules or terminals that correspond to tokens. For example, here is a simple grammar:

1. list $\longrightarrow$ object | list object
2. object $\longrightarrow$ string | number
3. string $\longrightarrow$ TEXT | COMMENT | COMMAND
4. number $\longrightarrow$ NUMBER | '+' NUMBER | '-' NUMBER | NUMBER '.' NUMBER

In this example, the capitalized words represent the terminals and the remaining words the nonterminals.

The first rule states that a *list* is made up either of an *object* or of a *list* and an *object*. fundamentally, then, this is a shorthand for two possible rules:

1. list $\longrightarrow$ object

2. list $\longrightarrow$ list object

The second form of the rule is a *recursive* definition. This allows a complex concept such like a list, to be described in a compact form. Since we do not know in advance how many elements of this list there will be, we can not conveniently define what a list is without this recursive form. Simply put, this definition states that a list is a sequence of objects, one after the other.

The | is a union operator. Note for example, its use in the last rule.

The construction of a grammar is a bottom-up process, including each grouping in progressively larger groupings until there is a single top-level grouping that includes all other groupings. This top-level construct is referred to as the *start* symbol. In our sample example above, "list" is the start symbol.

The well defined grammar

The nonterminal symbols are lower case and terminal symbols (or tokens) are capitalized.

ID (identifier) refers string which must begin with a letter and contain letters (upper or lower case), numbers, '/', '*', '[', ']' or '_'.

QSTRING must be a quoted string and can contain any ASCII character. Other terminal symbols refer to the following keywords:

Terminal Symbol	Keyword
CLASSNAME	classname
SUPERCLASS	superclass
ROOTDIR	rootdir
PROJECT	project
AUTHOR	author
DATE	date
COMMANDNAME	commandname
ARGINTYPE	argintype
ARGOUTTYPE	argouttype
DEVERRORS	deverrors
STATENAME	statename
ALLOWEDCOMMANDS	acommands
CLASSFIELD	classfield
OBJECTFIELD	objectfield
TYPE	type
NAME	name
RESOURCENAME	resourcename
DEFAULTVAL	defvalue
RESVAL	resvalue
END	end
DESCR	descr

The grammar is described in the figure 2.1.

If you respect this grammar the input file should contain five parts; the header, the commands list, the state machine, the classfields list and the objectfields list.

2.1.2 The header

In this part we find the general information about the device server.

→ **Super classes**

If you uses one or more super classes, write :

superclass $superclass_1$ ... $superclass_n$

Where $superclass_n$ is the exact name of a superclass with upper and lower case latters. These informations will be use to generate the classP.h file.

It is not necessary to give the DevServer class as root class.

→ **Root directory**

This information defines where you want to generate the target files. This directory should be created before using the generator. The generator will create all the subdirectories (src, include, RCS, ...). An example:

rootdir "/users/a/dserver/dev/TeslaHall"

→ **Project**

To describe the project where the device server will be used. an example:

project "Check and test to the new AGPowerSupply"

→ **Author**

Give your name, example:

author "Laurent Claustre"

→ **Date**

Give the original starting date, example:

date "October 20 1992"

→ **Class description**

Here you can describe your device server (if you want), this description will be inserted in the header of the file class.c, and in the chapter 'Description' in the user's guide. Example:

descr "Huber motor controller class"

2.1.3 The command list

All the required commands should be described in the command list. The information given for each command will allow generation of the command headers, the commandlist structure and so on.

n.b. Is not necessary to give the commands **DevState** and **DevStatus**. Both will be generated automatically .

→ **Command name**

You can only use commands names defined in the *libra:/users/d/dserver/include/DevCmds.h* file. Example:

commandname DevSetValue

→ **command description**

Give a description of the command. This description will be used to generate the command header and the user's guide. Example:

descr "Stop all the motors"

→ **Argin type**

You can only use the types available in the *libra:/users/d/dserver/include/xdr_typelist.h* file . Example:

argintype D_STRING_TYPE

→ **Argin description**

Give a description of the argin. This description will be used to generate the command header and the user's guide. Example:

descr "The motor position in step units"

→ **Argout type**

You can only use the types defined in the file *libra:/users/d/dserver/include/xdr_typelist.h* . Example:

argouttype D_VAR_STRINGARR

→ Argout description

As argin description.

→**Errors list**

You can give the error codes that the command return in case of failure. These errors are defined in the file *libra:/users/d/dserver/include/DevErrors.h*. This list will be inserted in the command header and in the user's guide. Example:

deverrors DevErr_DeviceWrite DevErr_DeviceRead

2.1.4 The state machine

All the data which are given to the state machine will be used to generate the state_handler method and the chapter State machine of the user's guide.

→ **State name**

You can only use the defined states found in the file *libra:/users/d/dserver/include/DevStates.h* . Example:

statename DEVSTANDBY

→ **State description**

This description will be inserted as comment in the state_handler() routine and in the users's guide of the device server (section State machine). An example:

descr "The device is on"

→ **Allowed commands**

It's the list of the allowed commands for the current state, if a command changes the state of the machine, you must indicate what is the new state. Example:
acommand DevState DevMoveReference {DEVMOVING}

In this example for the DevMoveReference command the new state will be DEVMOVING and for the DevState command the state is unchanged.

2.1.5 The classfields list

This part allows you to define the fields of the class (i.e. the typedef structure classClassPart) . The generator will use this information to make the classP.h file, the class.c file (to implement the resource table, the class_initialize method) and the class.res file (to implement the resource value of the class fields).

Don't forget to write the keyword **classfield** if you want to describe class fields. Example:
classfield name timeout **type** long ...

N.B.: The generator will automatically generate a **state** resource called state with the value DEVON. The field ds->devserver.state is set with this value by the class_initialize method.

→ **Name**

This is the name of the classfield, it will be used in the classP.h and class.c files. Example:
name timeout

→ **Type**

This is the type of the class field. If a resource is not specified for this field you can directly use C types or types defined in the *libra:/users/d/dserver/include/DevServerXdr.h* file. Example: **type** long
type DevString

If you define a resource, you must use only the types defined in the *libra:/users/d/dserver/include/xdr_typelist.h* file contains. Example:
type D_LONG_TYPE

→ **Default value**

A hardwired default value for class fields used in the class_initialize method. Example:
defvalue "0"

→ **Field description**

you can give a description of classfields. It will be inserted in the classP.h file, example:
descr "timeout for the serial line"

→ **Resource name**

If you want to define a resource for this field you must give a name for this resource; if possible, use the same name for the class field and the resource. Example:
resourcename timeout

Resource value

Defined the resource value (useful only if a resource name has been previously defined). Example:
resvalue “254”

2.1.6 The objectfields list

Here, you must describe the object fields of your class. The data provided will allow implemention of the classP.h (to code the typedef structure classPart) and the class.c (to code the resource table for objects and to code the object_initialize method) files.

Don't forget to write the keyword **objectfield** before to describe the fields. Example:
objectfield name io ...

→ **Name**

This is the name of the object field, it will be use in the classP.h and class.c files, example:
name io

→ **Type**

See section type of classfield.

→ **Default value**

You must state default values, if you want the object_initialize method to initialize the object fields before to get the database resources. An example:
defvalue “1”

→ **Field description**

You can give a description of object fields. It will be inserted in the classP.h file. Example:
descr “GPIB interface”

→ **Resource name**

If you want to provide a resource for this field you must give a name for this resource. If possible, use the same name for the object field and the resource. Example:
resourcename io

There is no resource value, since you must supply the list of object (in the class.res file) and the resource values for each object.

2.1.7 The descriptions

All the descriptions (command description, class description ...) are quoted strings. If a description contains more than 200 characters we must used several quoted string as following:
descr “This is the command .... ”
“and it ....”
“.....via the powersupply.”

2.1.8 An example of inputfile

Using the grammar directly to write an input file is not trivial. As an aid to the reader an example of input file is presented (see figure 2.2) in the following sections.

Note that several tabs are used. tabulation is not mandatory, but improves the readability of the file.

The input file has been used to generate the Huber Motor Controller Device Server.

2.2 Run the generator

When your input file is ready, run the generator. If you get some syntax errors, don't panic; read the next chapters.

2.2.1 Location

Classgen for different platforms can be found under:

- libra:/users/d/dserver/bin/s300
- libra:/users/d/dserver/bin/s700
- libra:/users/d/dserver/bin/s800
- libra:/users/d/dserver/bin/sun4
- libra:/usr/local/bin/s300
- libra:/usr/local/bin/s700
- libra:/usr/local/bin/s800
- libra:/usr/local/bin/sun4

Warning: classgen software can be used on a machine if:

- libra is mounted,
- the TEMPLATEPATH environment variable is set to /libra/users/d/dserver/classes/template (and exported), the home directory of the template files.

2.2.2 Use the generator

To run the generator, symply type classgen following by the options and the input file name:

classgen [-sub] <inputfile.dat>

Option(s):

-sub Use the special template files to make device servers with an associated subprocess.

First, the generator will extract data from your input file. If a syntax error is detected, the generator will exit (see the section 2.2.3). If there are no syntax errors, the generator will create the required target files. Two message lines appear:

```
Input file <inputfile.dat> is OK and the data have been extracted.
```

```
Interactive update of target files (y/n)?:
```

The generator waits for the answer (y or n). If you type 'y', an attempt to create a target file (device server) will be made.

after this answer, a menu appears:

```
Which file(s) would you like generated ?
                a .  Quit
                b .  Makefile
                c .  Class.c
                d .  ClassP.h
                e .  Class.h
                f .  Class_menu.c
                g .  Startup.c
                h .  Class.res
                i .  Class.tex
                j .  All
```

```
Your choice ?:
```

enter the required option.

2.2.3 Error messages

This section discusses correcting problems and errors reported by the generator.

1. **syntax error**
   classgen: Syntax error at line n in the input file

   There is an syntax error at the line (n) to correct this error help you with the grammar.

2. **<inputfile>: Error**
   A component of the path prefix is not a directory

3. **<inputfile>: Error**
   The named directory doesn't exist

4. **<inputfile>: Error**
   Search permission is denied for a component of the path

5. **<inputfile>: Error**
   The root directory doesn't exist

   These four errors are caused by an uncorrect root directory. Perhaps you have forgotten that the root directory must already exists before running the generator.

6. **classgen can't open <inputfile.dat>**

   You have given a bad name for your input file.

7. **<inputfile>: Error**
   It's not necessary to define the DevState command.

8. **<inputfile>: Error**
   It's not necessary to define the DevStatus command.

   The DevState and DevStatus commands will be generated automatically, so it's not necessary to define them in the commands list.

9. **<inputfile>: Error**
   <STATE>, allowed command(s) undefine in the commands list:
   - <allowedcommand>
   - <allowedcommand>
   - ....

   These allowedcommands (defined for the state <STATE>) are not found by the commands list.

```
class            ←  classname classcontent classterminator
classname        ←  CLASSNAME ID
classterminator  ←  END ID
classcontent     ←  superclasses header commands states classfields objectfiels

superclasses     ←  /*nothing*/ | SUPERCLASS superclass
superclass       ←  ID | superclass ID

header           ←  superclasses rootdir project author date classdesc
rootdir          ←  ROOTDIR QSTRING
project          ←  PROJECT qstrings
author           ←  AUTHOR QSTRING
date             ←  DATE QSTRING
classdesc        ←  /*nothing*/ | desc

commands         ←  /*nothing*/ | commands command
command          ←  commandname commanddesc argintype argindesc
                    argouttype argoutdesc errorlist
commandname      ←  COMMANDNAME ID
commanddesc      ←  /*nothing*/ | desc
argintype        ←  ARGINTYPE ID
argindesc        ←  /*nothing*/ | desc
argouttype       ←  ARGOUTTYPE ID
aroutdesc        ←  /*nothing*/ | desc
errorlist        ←  /*nothing*/ | DEVERRORS deverrors
deverrors        ←  ID | deverrors ID

states           ←  /*nothing*/ | states state
state            ←  statename statedesc allowedcommands
statedesc        ←  /*nothing*/ | desc
statename        ←  STATENAME ID
allowedcommands  ←  /*noting*/ | ALLOWEDCOMMANDS allowedcommand
allowedcommand   ←  ID newstate | allowedcommand ID newstate
newstate         ←  /*nothing*/ | '{' ID '}'

classfields      ←  /*nothing*/ | CLASSFIELD classfield
classfield       ←  name type defaultval fielddesc resourcename resourceval
objectfields     ←  /*nothing*/ | OBJECTFIELD objectfield
objectfield      ←  name type defaultval fielddesc resourcename
name             ←  NAME ID
type             ←  TYPE ID
defaultval       ←  /*nothing*/ | DEFAULTVAL QSTRING
fielddesc        ←  /*nothing*/ | desc
resourcename     ←  /*nothing*/ | RESOURCENAME ID
resourceval      ←  /*nothing*/ | RESVAL QSTRING

desc             ←  DESCR qstrings
qstrings         ←  QSTRING | qstrings QSTRING
```

Figure 2.1: The grammar

```
                    October 20 1992                huber.dat

classname huber

      superclass      StepperMotor
      rootdir         "/users/a/dserver/dev/huber"
      project         "Beam line"
      author          "Rey Vicente"
      date            "October 20 1992"
      descr           "Huber Motor Controller class"


      commandname DevReadPosition
           descr      "Sends a P command - get the actual position"
           argintype  D_VOID_TYPE
              descr   "none"
           argouttype D_LONG_TYPE
              descr   "The motor position in step units"
           deverrors   DevErr_DeviceWrite DevErr_DeviceRead

                     .
                     .
                     .

      commandname DevAbortCommand
           descr      "Stops all the motors"
           argintype  D_VOID_TYPE
              descr   "none"
           argouttype D_VOID_TYPE
              descr   "none"
           deverrors  DevErr_CommandFailed


      statename DEVON

           acommands   DevMoveReference {DEVMOVING}
                       DevMoveRelative  {DEVMOVING}
                       DevMoveAbsolute  {DEVMOVING}
                       DevSetContinuous {DEVMOVING}
                       DevReadState
                       DevReadPosition
                       DevReadSwitches
                       DevSetVelocity
                       DevSetAcceleration
                       DevSetFirstStepRate
                       DevLoadPosition
                       DevAbortCommand

      statename DEVFAULT

            acommands       DevReset

      statename DEVMOVING

            acommands  DevReadState
                       DevReadPosition
                       DevReadSwitches
                       DevSetVelocity
                       DevSetAcceleration
                       DevSetFirstStepRate
                       DevLoadPosition
                       DevAbortCommand {DEVON}

     /* No classfield */

      objectfield
                       name         io
                       type         devserver
                       descr        "GPIB interface"

                       name         unitnum
                       type         D_SHORT_TYPE
                       descr        "Unit number for the gpib interface"
                       resourcename unitnum

                       name         axe
                       type         D_SHORT_TYPE
                       descr"       Axe number for this particular motor"
                       resourcename axe

                       name         gpibdev
                       type         D_STRING_TYPE
                       descr        "GPIB device name "
                       resourcename gpibdev

                       name         velocity
                       type         D_LONG_TYPE
                       descr        "Velocity value"
                       resourcename velocity

                       name         lastcmd
                       type         long
                       descr        "Last issued command"

end huber
```

Figure 2.2: Example of input file

Chapter 3

About the X version: xclassgen

the Graphical User Interface of xclassgen was made within the Motif widgets. Its utilization is very friendly, so we are not developed in details how to use it.

3.1 Location

xclassgen for different platforms can be found under:

- libra:/users/d/dserver/bin/s300
- libra:/users/d/dserver/bin/s700
- libra:/users/d/dserver/bin/s800
- libra:/users/d/dserver/bin/sun4
- libra:/usr/local/bin/s300
- libra:/usr/local/bin/s700
- libra:/usr/local/bin/s800
- libra:/usr/local/bin/sun4

Warning: xclassgen software can be used on a machine if:

- libra is mounted (/users/d volume),
- the TEMPLATEPATH environment variable is set to /libra/users/d/dserver/classes/template (and exported), the home directory of the template files,
- the XAPPLRESDIR environment variable is set to /usr/local/lib/X11/app-defaults and exported,
- /usr/local/lib/X11/app-defaults directory contains the resource file of xclassgen (XClassGen),
- the /usr/local/lib/X11/app-help directory contains the help file (XClassGen).

On the mars, libra, pavo and civa machines these files are already installed. And its will be automatically updated via the source files on libra.

You can copy the resource file (XClassGen) and the help file (XClassGen) respectively from libra:/users/d/dserver/system/classgen/motif/app-defaults and libra:/users/d/dserver/system/classgen/motif/app-help.

The help file contains the text used by the help menus. The generator had been built with the Help library, realized by Deavid Carron at the European Synchrotron Radiation Facility.

3.2 Use the generator

The main window of xclassgen contains a menu bar and a scrolled workspace. The workspace will allow the user to build his class description wwithin elements. These elements are the superclasses, the objectfields the classfields, the states the allowed commands, the forbidden commands of the states and the commands. The elements will be groups under list in the workspace, The list and the elements are represented by push buttons.

3.2.1 The menus

For all the menus and all the menu items we dispose of accelerators and mnemonics. Any menu items are unsensitive when you begin a session.

⊕ **File menu**

- *Open*: you can load a class description from a input file. This file must have the ".dat" suffixe and it has been generated by xclassgen. If you want to load a classgen format input file, you must do some modifications, because the grammar is not the same. for more informations about the convertion, please call me.
- *New*: if you select this option the current class description is deleted.
- *Save*: saves your current class description in a xclassgen readable format, the name of the file is the current file name.
- *Save As ...*: As Save option, and you can specifie another file name.
- *Generate Code*: generates the target files.To select the target files see the option "File To Generate" in the "Options" menu.
- *Exit*: exits xlcassgen.

⊕ **Palette menu**

This menu will allow you to add new elements to the current class description.

- *Superclass*: adds a superclass element.
- *ClassField*: adds a classfield element.
- *ObjectField*: adds a objectfield element.

- *State*: adds a state element.
- *Allowed Command*: adds a allowed command element to the current state. The current state is the selected state in the workspace.
- *Forbidden Command*: adds a forbidden commadn element to the current state. The current state is the selected state in the workspace.

N.B.: if no one state is selected the options “Allowed Command” and “Forbidden Command” are not sensitive.

⊕ Edit menu

The options of this menu will are sensitive if an element in the workspace has been selected.

- *Copy*: duplicates the selected element with its content.
- *Delete*: deletes the selected element.

⊕ Options menu

This menu will allow you to specifie the template files (templates for one process device server or for two process device server) and the target files.

- *For One Process*: It is a toggle button, the generator will use the one process device server template files.
- *For Two Process*: it is a toggle button, the generator will use the two process device server template files.
- *File To Generate*: it pops up the target files windows.

⊕ Help menu

This menu will apply you all the needed informations about several topics, the generator, the elements and their content, the template and the target files. You can find all the necessary informations to complete the content of each element without do errors.

So it is very important that the user read all the helps before to generate the target files.

N.B.: The content window of the element (see the next section) will provide you helps. These helps are the same in the “Help” menu.

3.2.2 How to modifie characteristics of the class

⊕ The content windows

When you want to modifie the characteristics of a element of the current class description, you must pop up its content window. A content window provide the fields of a element. For example the content window of a “Command” element contains:

- the Command Name field
- the Command Description field

- the Argument In Type field
- the Argument In Description field
- the Argument Out Type field
- the Argument Out Description field
- Error Code List field

To valid a element content press "OK", the "CanCel" button close the content window and the 'Help" button pops up the help about this element. the small push button at the right side of the window will allow you to pop up text editor windows.

⊕ The text editors

Some element contents require descriptions (e.i. class decription, command description ...), so if a description will contains several lines we can use a simple text editor. To pop up the text editor wimdow of a description field, use the small push button at the right side of the window.

The "Apply" pushbutton valids the text, the "Reset" pushbutton delete the text int the text editor and the "Dismiss" pushbutton close the window.

Chapter 4

The two process Device Server

The generator can provide two kinds of classes. One class contains a single process (see figure 4.1) and another class contains two processes (see figure 4.2), the server process and the subprocess. The server process and the subprocess communicate via Dataports (shared memory), and the subprocess works continiously to update the shared memory. We use the interprocess communication model provided by David Carron. He has made a library of five routines (dataport library) to use this model. For more details see the mars:/users/a/dserver/rel/utils/dataport/doc/dataport.tex document.

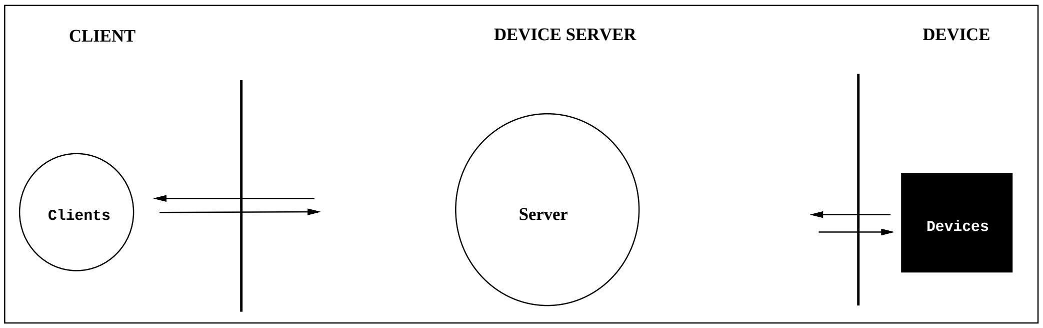

Figure 4.1: One process Divice Server Model

The main goal of a two processes Device Server is to make the subprocess perform a continious task and communicate directly with the device. The server process communicates with the client and always waits

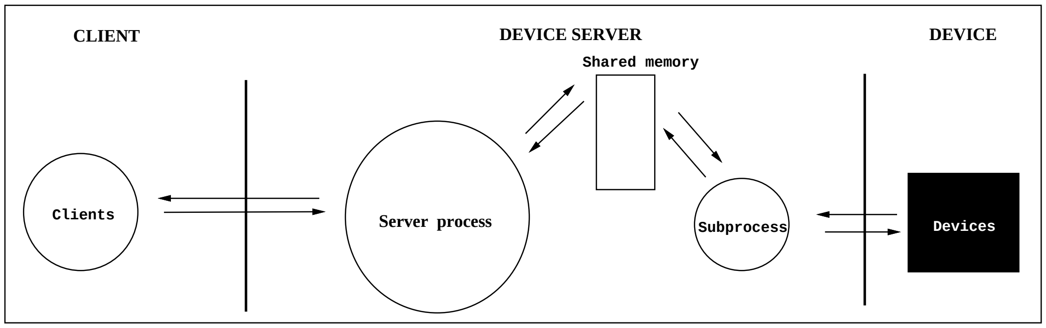

Figure 4.2: Two process Device Server Model

for a command from clients. The subprocess is a infinite loop.

The source code of the subprocess should be contained in the Class_sub.c file. The shared structure will be defined in the ClassP.h header file. The reader could consult the template files (on libra:/users/d/dserver/classes/template/subproc) to get a better understanding.

The subprocess will be created and the Dataport opened by the class_initialize() routine. All the necessary source code has been added to check if the subprocess already exists and if the shared memory is allocated.

When a Device Server is generated, the reader should complete the shared structure with variables shared by the two processes to make it possible for the user to use the dataport and implement the commands which will use the shared memory. Some examples are provided in the section 5.2.

Chapter 5

Generated files: where to append code

The generator cannot entirely code your device server. Nevertheless, the generated code is directly compilable and executable.

If you complete the class.res file (objects list, resource values for each object),you can run the server and the menu. The commands will not work (they have not implemented), except DevStatus and Devstate commands.

If you are happy with the generated stubs, you can edit the files to complete them.

In the below sections we are going to present only the files which must be completed.

In the section 5.1 the reader is going to find help for the two kinds of Device Server and in the section 5.2 only help for the two processes Device Server.

5.1 for all the Device servers

5.1.1 Class.c

We can consider two kinds of code, the code we must add in all cases (code for the commands) and the code to add only if you uses superclasses.

If you use resources for object and class fields of superclasses, you must complete the both tables res_object[ ] and res_class[ ], and the both methods class_initialize and object_initialize.

5.1.2 Class.res

The generator has only got the resources values for the class fields, you should add the list of objects, and the resource values for each object. If you uses superclasses the resources for their fields should be added.

5.1.3 Class_menu.c

This menu allows to test all the commands. This command line is:

class_menu <objectname>

It's a alpha-numeric menu, an example:

```
[mars]$ Simul_menu sy/simul/1
dev_import(sy/simul/1) returned 0

Select one of the following commands :


 0.Quit     1.State      2.Status
 3.GetName  4.ReadValue  5.SetValue
 6.Off      7.On

cmd ?
```

The menu import the object (dev_import). For each command it calls the dev_putget() routine. All the necessary parameters to call dev_putget() are created. For instance, with the command DevStatus we will have:

```
devstatus = dev_putget(ds, DevStatus,
                          NULL,
                          D_VOID_TYPE,
                          &status_argout,
                          D_STRING_TYPE,
                          &error);
```

The generator has find the status_argout variable. To recognize the variables it's very simple:

<commandname>_argout if the command <commandname> has an argument out
<commandname>_argin if the command has <commandname> an argument in.

So to complete the menu it's just necessary to control the variables (set and print them).

5.1.4 Class.tex

The User's Guide has been generated in a LaTeX format. It contains 7 sections:

1. Introduction
2. Description
3. State machine
4. Startup
5. Commands
6. Problems

7. Conclusions

The sections 3 and 5 are entirely generated.

5.2 For the Two Process Device Servers

5.2.1 ClassP.h

You should complete the shared structure provided in this file. When ClassP.h file will be generated it will contain the following structure including all variables (or elements) shared by the two process:

```
typedef struct {
        unsigned short pid;
        unsigned long time;
        long value; /*example of element */
        /* Insert here your elements */
    }
        Sharedstruct;
```

the 'value' element will be used such like an example to show how to use a shared memory with the **dataport** library. These elements must always be included in the structure.

The 'pid' element will contain the subprocess pid, and the 'time' element the time of the day. these two element will be used to manage the subprocess and the shared memory.

Time element tells the time in seconds since the subprocess last usesd the dataport.

5.2.2 Class_sub.c

Contains the main() routine of the subprocess. This routine is an infinite acquisition loop. It also contains one routine of acquisition:

```
static int StoreResult()
{
        struct timeval sub_time;
        struct timezone tzp;

        gettimeofday(&sub_time, &tzp);
        AccessDataport(dp_sub);
        shared-¿time = sub_time.tv_sec;

        shared-¿value++;

        ReleaseDataport(dp_sub);
}
```

The StoreResult() routine uptdates the time element and the value element.It uses two routines of the dataport library, AccessDataport() and ReleaseDataport(). **dp_sub** is the identifier of the shared memory of the subprocess.

You should implement your own acquisition routines to update the element you have added in the structure.

5.2.3 Class.c

As for a ordinary Class.c file you should complete the commands you are described in the input file. Certain commands access the Dataport, you can find two examples of this type of command to help you, at the end of the file. they are the following:

```
/*******************************************************************
static long dev_writevalue(ds, argin, argout, error)

Template ds;
DevLong *argin;
DevVoid *argout;
long *error;
{

        long time_diff;

        gettimeofday(&time_now, &tzp);
        AccessDataport(dp);
        shared->value = *argin;
        time_diff = time_now.tv_sec - shared->time;
        ReleaseDataport(dp);
        if(time_diff > 4)
        {
                *error = DevErr_NoData;
                return(DS_NOTOK);
        }

        return (DS_OK);
}
*******************************************************************/

/*******************************************************************
static long dev_readvalue(ds, argin, argout, error)

Template ds;
DevVoid *argin;
DevLong *argout;
long *error;
{

        long time_diff;

        gettimeofday(&time_now, &tzp);
        AccessDataport(dp);
        *argout = shared->value;
        time_diff = time_now.tv_sec - shared->time;
        ReleaseDataport(dp);
        if(time_diff > 4)
        {
```

```
                *error = DevErr_NoData;
                return(DS_NOTOK);
        }

        return (DS_OK);
}
*********************************************************************/
```

These two command functions manipulate the value element of the shared memory. The first sets it and the second reads it. Each time we want to access the shared memory, we should use AccessDataport() and ReleaseDataport() routines as in the examples.

5.3 Conclusion

The author would appreciate feed back form users concerning bugs and/or possible improvements. Some improvements are being considered: the generator should be more user friendly (writing a input file without syntax errors is not easy), so future versions will be based on X-Windows, e.g. a graphic users interface will be provided; the input file will be produce by the generator and it will allow to store the class description, even if not complete.