

# TACO Manual

*version 1.2*

November 2000

# Abstract

TACO is a toolkit for implementing distributed object oriented control systems originally developed at the European Synchrotron Radiation Facility (ESRF)<sup>1</sup> in Grenoble (FRANCE). In TACO all control points are represented as devices. Devices are objects which belong to a control class. The class implements the control logic necessary to control the device hardware/software. Devices are served by processes called device servers. Device servers are distributed over one or any number of machines. Clients which need to access devices do so through a application programmer's interface. The clients can access devices synchronously, asynchronously or by using events. The network layers are kept entirely hidden from the device server and client programmer's by TACO. TACO supports a database (based on gdbm or Oracle) for storing persistent information and keeping track of where devices are running and an archiving database (based on Oracle). There are 7 levels of security for controlling client-server access. TACO supports the notion of multiple TACO control systems. This facilitates management of a large number of devices on a large site. TACO is available free of charge without warranties under the GNU Public Licence.

---

<sup>1</sup> <http://www.esrf.fr>



# Contents

<b>1</b>	<b>Introduction</b>	<b>9</b>
	<i>by A.Götz</i>	
<b>2</b>	<b>What is TACO ?</b>	<b>11</b>
	<i>by A.Götz</i>	
2.1	Introduction . . . . .	11
2.2	System architecture . . . . .	11
2.3	Manager . . . . .	11
2.4	Database . . . . .	11
2.5	Device Server Model . . . . .	13
2.6	Application Programmer's Interface . . . . .	13
2.7	Data Collector . . . . .	13
2.8	Archiving . . . . .	14
2.9	Security . . . . .	14
2.10	Multiple Control Systems . . . . .	14
2.11	Uses of TACO . . . . .	14
<b>3</b>	<b>Changes</b>	<b>17</b>
<b>4</b>	<b>Acknowledgements</b>	<b>19</b>
<b>5</b>	<b>Getting Started</b>	<b>21</b>
<b>6</b>	<b>Installing</b>	<b>23</b>
	<i>by A.Götz</i>	
6.1	Introduction . . . . .	23
6.2	Getting Started . . . . .	23
6.2.1	GNU make . . . . .	23
6.2.2	configure . . . . .	23
6.2.3	make all . . . . .	24
6.2.4	make install . . . . .	24
6.2.5	make test . . . . .	24
6.2.6	make clean . . . . .	24
6.2.7	make clobber . . . . .	24
6.2.8	Libraries . . . . .	24
6.3	System processes . . . . .	25
6.4	Database tools . . . . .	25
6.5	Testing . . . . .	25
6.6	Problems . . . . .	25
6.7	Windows . . . . .	26
<b>7</b>	<b>Platforms</b>	<b>27</b>

**8 Device Servers in C++***by A.Götz and E.Taurel***29**

8.1	Introduction . . . . .	29
8.2	Device Server Model ++ . . . . .	29
8.3	Device root class . . . . .	31
8.3.1	Device.h - include file . . . . .	31
8.3.2	Device.cpp - source code file . . . . .	34
8.4	PowerSupply class - an example superclass . . . . .	36
8.4.1	PowerSupply.h - include file . . . . .	37
8.4.2	PowerSupply.cpp - source code file . . . . .	38
8.5	AGPowerSupply class - an example derived class . . . . .	38
8.5.1	AGPowerSupply.h - include file . . . . .	39
8.5.2	AGPowerSupply.cpp - source code . . . . .	39
8.6	startup.cpp - an example startup file . . . . .	42
8.7	OIDDevice wrapper class . . . . .	43
8.7.1	OIDDevice.h - include file . . . . .	44
8.7.2	startup.cpp - an example . . . . .	46
8.8	Implementation . . . . .	48
8.9	Compilers . . . . .	49
8.10	Template Class . . . . .	49
8.11	C++ Programming Style . . . . .	50
8.12	Advantages of C++ . . . . .	50
8.13	Disadvantages of C++ . . . . .	50
8.14	Future developments . . . . .	50
8.15	Conclusion . . . . .	51
8.16	Suggested Reading . . . . .	51

**9 DSAPI***by J.Meyer and A.Götz***53**

9.1	Introduction . . . . .	53
9.2	What is DSAPI ? . . . . .	53
9.3	Getting Started . . . . .	54
9.3.1	"Hello World" (synchronous) example . . . . .	54
9.3.2	"Hello World" (asynchronous) example . . . . .	57
9.3.3	Common Pitfalls . . . . .	61
9.3.4	Nethost . . . . .	61
9.3.5	Shared Libraries . . . . .	61
9.3.6	Makefiles . . . . .	62
9.3.7	Memory Allocation . . . . .	64
9.3.8	Advanced Features . . . . .	66
9.3.9	Timeouts . . . . .	66
9.3.10	Protocol . . . . .	66
9.4	C Library . . . . .	67
9.4.1	Synchronous Client API . . . . .	67
9.4.2	ASynchronous Client API . . . . .	71
9.4.3	Server . . . . .	72
9.4.4	General Purpose Functions . . . . .	74
9.5	XDR types . . . . .	77
9.5.1	Kernel Types . . . . .	77
9.5.2	Simple C Types . . . . .	77
9.5.3	Combinations of Simple Types . . . . .	78
9.5.4	Variable Length Arrays . . . . .	78
9.5.5	Exotic Types . . . . .	80
9.6	Changes . . . . .	81

9.6.1	Version 8.0 . . . . .	81
9.6.2	Version 7.0 . . . . .	81
9.6.3	Version 6.0 . . . . .	81
9.6.4	Version 5.1 . . . . .	81
9.6.5	Version 4.1 . . . . .	81
9.6.6	Version 3.37 . . . . .	81

## 10 Database guide - ndbm

<i>by E.Taurel</i>		<b>85</b>
10.1	Introduction . . . . .	85
10.2	Device and resource definition . . . . .	85
10.2.1	The devices list . . . . .	85
10.2.2	Resource definition . . . . .	86
10.2.3	Domain names and NDBM files . . . . .	87
10.3	Greta . . . . .	87
10.3.1	The device window . . . . .	88
10.3.2	The server window . . . . .	91
10.3.3	The resource window . . . . .	91
10.3.4	The new server window . . . . .	93
10.3.5	The load file window . . . . .	93
10.3.6	The Option menu . . . . .	95
10.3.7	Other features . . . . .	95
10.4	Resource file . . . . .	95
10.5	Utilities . . . . .	96
10.6	Database administration commands . . . . .	96
10.6.1	db_fillup . . . . .	96
10.6.2	db_info . . . . .	96
10.6.3	db_read . . . . .	97
10.7	Database user commands . . . . .	97
10.7.1	db_update . . . . .	97
10.7.2	db_devres . . . . .	98
10.7.3	db_devinfo . . . . .	98
10.7.4	db_servinfo . . . . .	98
10.7.5	db_devdel . . . . .	98
10.7.6	db_resdel . . . . .	99
10.7.7	db_servdel . . . . .	99
10.7.8	db_servunreg . . . . .	99
10.8	Security commands . . . . .	99
10.8.1	dbm_sec_passwd . . . . .	99
10.8.2	dbm_sec_objinfo . . . . .	99
10.8.3	dbm_sec_userinfo . . . . .	99
10.9	The C library . . . . .	100
10.10	Resource oriented calls . . . . .	100
10.10.1	db_getresource() . . . . .	100
10.10.2	db_putresource() . . . . .	100
10.10.3	db_delresource() . . . . .	100
10.11	Exported device list oriented calls . . . . .	101
10.11.1	db_getdevexp() . . . . .	101
10.11.2	db_fredevexp() . . . . .	101
10.12	Device oriented calls . . . . .	101
10.12.1	db_getdevlist() . . . . .	101
10.12.2	db_dev_import() . . . . .	101
10.12.3	db_dev_export() . . . . .	102
10.12.4	db_deviceinfo() . . . . .	102

10.12.5 db_devicedelete()	102
10.12.6 db_devicedeleteres()	102
10.12.7 db_getpoller()	103
10.13 Server oriented calls	103
10.13.1 db_svc_unreg()	103
10.13.2 db_svc_check()	103
10.13.3 db_servinfo()	103
10.13.4 db_servdelete()	104
10.13.5 db_servunreg()	104
10.14 Database browsing oriented calls	104
10.14.1 db_getdevdomainlist()	104
10.14.2 db_getdevfamilylist()	104
10.14.3 db_getdevmemberlist()	104
10.14.4 db_getresdomainlist()	105
10.14.5 db_getresfamilylist()	105
10.14.6 db_getresmemberlist()	105
10.14.7 db_getresresolist()	105
10.14.8 db_getresresoal()	106
10.14.9 db_getdsserverlist()	106
10.14.10 db_getdspersnamelist()	106
10.14.11 db_gethostlist()	106
10.15 Pseudo device oriented calls	106
10.15.1 db_psdev_register()	106
10.15.2 db_psdev_unregister()	107
10.16 Database update calls	107
10.16.1 db_analyse_data()	107
10.16.2 db_upddev()	107
10.16.3 db_updres()	107
10.17 Miscellaneous calls	108
10.18 Multi TACO control system access	108
<b>11 Events</b>	
by A.Götz	111
11.1 Introduction	111
11.2 Events	111
11.3 API	112
11.3.1 Client side	112
11.3.2 Server side	112
11.4 Implementation	113
11.5 Timeouts	113
11.6 Examples	113
11.7 Performance	114
11.8 Known problems	115
<b>12 The Signal Interface</b>	
by J.Meyer and J-L.Pons	117
12.1 Introduction	117
12.2 Conventions on Signals	117
12.3 The Signal Properties	118
12.4 The Server Side	118
12.4.1 The Commands to Access Signals	118
12.4.2 Coding Example using a Multi Signal Object	120
12.5 Reading the Signal Properties without Accessing the Device	124
12.5.1 dev_get_sig_config()	124

12.5.2	<code>dev_get_sig_config_from_name()</code>	124
12.5.3	<code>dev_get_sig_list()</code>	125
12.5.4	<code>dev_get_sig_set_list()</code>	125
12.5.5	<code>dev_get_sig_setread_from_name()</code>	125
12.6	The Client Side	126
12.7	The Signal Interface to HDB	128
12.8	Conclusion	129
<b>13</b>	<b>Access Control and Security</b>	
	<i>by J.Meyer</i>	<b>131</b>
13.1	Introduction	131
13.2	The Problem	131
13.3	The Model	132
13.3.1	Users, Groups and Networks	132
13.3.2	Access Rights	132
13.3.3	Domain, Family or Member	133
13.3.4	Verification Speed and Reliability	134
13.4	Integration into TACO	134
13.5	Complex Access Handling	135
13.6	Conclusion	136
13.6.1	The Current Implementation	137
13.6.2	How to get started?	138
13.6.3	Pending Problems	141
<b>14</b>	<b>Standard Makefiles using GNU make (gmake)</b>	
	<i>by A.Götz</i>	<b>143</b>
14.1	Introduction	143
14.2	Philosophy	143
14.3	GNU Make Commands	143
14.4	Standard Symbols	144
14.5	Standard Targets	144
14.6	Scripts	145
14.7	Example Makefile	145
14.8	Further Reading	152
<b>15</b>	<b>Basic steps to install and configure a device server</b>	
	<i>by A.Götz</i>	<b>153</b>
<b>16</b>	<b>A tool to test a TACO control system</b>	
	<i>by E.Taurel</i>	<b>155</b>
16.1	Introduction	155
16.2	Usage	155
16.3	Usage example	156
16.4	Testing a device server	156
16.5	Testing control system kernel servers	156
16.6	Testing all the device server running on a host	157
16.7	Testing a complete control system	158
<b>17</b>	<b>Adding Private Commands, Errors and XDR Data Types</b>	
	<i>by J.Meyer and A.Götz</i>	<b>161</b>
17.1	Introduction	161
17.2	Dynamic Errors	161
17.3	Error Numbers	162
17.4	Command Numbers	163



17.5 Database Support . . . . .	164
17.6 Time Stamp for Error Messages . . . . .	164
17.7 The restructured XDR concept . . . . .	165
17.8 Private XDR Data Types . . . . .	166
17.8.1 Data Type Numbers . . . . .	166
17.8.2 What is a Complete XDR Data Type Definition? . . . . .	166
17.8.3 How to Integrate a New Data Type? . . . . .	169
17.8.4 Available Data Types . . . . .	169
17.9 Numbering Limits . . . . .	170
17.9.1 Master Copies . . . . .	170
17.10 Conclusion . . . . .	171
<b>18 Interfaces</b>	<b>173</b>
<b>A Licence</b>	<b>175</b>

# Chapter 1

## Introduction

*by A. Götz*

TACO is a toolkit for implementing distributed object oriented control systems. It has been used at the European Synchrotron Radiation Facility (ESRF) in Grenoble (FRANCE) to control the accelerator complex and all beamlines. It is also used at FRM II<sup>1</sup> in Munich (Germany) to control the beamlines and at the Hartebeesthoek Radio Astronomy Observatory (HartRAO)<sup>2</sup> in Hartebeesthoek (South Africa) to control a 26 meter radio telescope.

TACO can be compared to other distributed object toolkits like CORBA, DCOM and OPC (on Windows) with the main differences being : (1) TACO is easy to use and understand, (2) TACO is freely available, (3) TACO is based on ONC/RPC (now part of the GNU C library), (4) TACO is multi-platform.

In TACO all control points are represented as devices. Devices are objects which belong to a control class. The class implements the control logic necessary to control the device hardware/software. Devices are served by processes called device servers. Device servers are distributed over one or any number of machines. Clients which need to access devices do so through a application programmer's interface. The clients can access devices synchronously, asynchronously or by events. The network layers are kept entirely hidden from the device server and client programmer's by TACO. TACO supports a database for storing persistent information and keeping track of where devices are running.

TACO is used to control an accelerator complex, experimental setups on beamlines (using synchrotron radiation and neutrons), a radio telescope and other smaller projects. It is ideal for adding Ethernet control to embedded and non-embedded devices in a research, industrial or home environment. Refer to the appendix for a list of existing device servers.

This manual is a compendium of all important TACO documents which have been written over the years by the various TACO programmers. This way there is only one single TACO manual for all important TACO documentation. The information is brought up to date on a regular basis and should be useful to new and experienced users of TACO.

TACO can be downloaded from the TACO website<sup>3</sup> and installed from the source code. TACO is made available under the GNU Public Licence (see Licence) without warranties. For news about recent developments in TACO go to the website.

This manual is organised as follows :

---

<sup>1</sup><http://www.frmii.de>

<sup>2</sup><http://www.hartrao.ac.za>

<sup>3</sup><http://www.esrf.fr/computing/cs/taco>

1. Changes - list of changes to this manual.
2. Introduction - this text, should be read by everyone (it's so short !).
3. What is TACO ? - provides a brief overview of what TACO is, useful for newcomers to TACO.
4. Getting Started - for those who want to get going quickly without having to read the manual.
5. Installing - how to install TACO from source code (basically the README distributed with the source code).
6. Device Servers in C++ - how to write device servers in C++.
7. Device Server Application Programmer's Interface - describes how to write TACO clients in C and C++.
8. Database - describes the TACO ndbm database and how to write clients for the TACO database.
9. Events - how to use and program events.
10. Signals - how to use and program normalised data types called signals.
11. Access Control and Security - a full description of TACO security.
12. Standard Makefiles - how to write TACO Makefiles using GNU make to maintain multiple platforms.
13. How to install a device server - basic steps on how to install a device server.
14. testcs - how to test a running TACO system.
15. Private commands, errors and xdr types - how to extend TACO to add private commands, errors and data types. system.
16. Licence - the full text of the GPL licence.

For more information about TACO refer to the website regularly or subscribe to `taco@esrf.fr` by sending an email to `majordomo@esrf.fr` with `subscribe taco` in the body of the email.

## Chapter 2

# What is TACO ?

*by A. Götz*

### 2.1 Introduction

TACO is an object oriented control system originally developed at the European Synchrotron Radiation Facility. The basic idea behind TACO is to treat every control element as an object on which commands can be executed. The objects are called devices and they are available network wide. Devices are created and stored in device servers. The commands which can be executed on a device are implemented in the device class. Device classes can be written in C (using a methodology called OIC) or C++. The commands are accessed via a small set of C calls referred to as the application programmer's interface (DSAPI).

### 2.2 System architecture

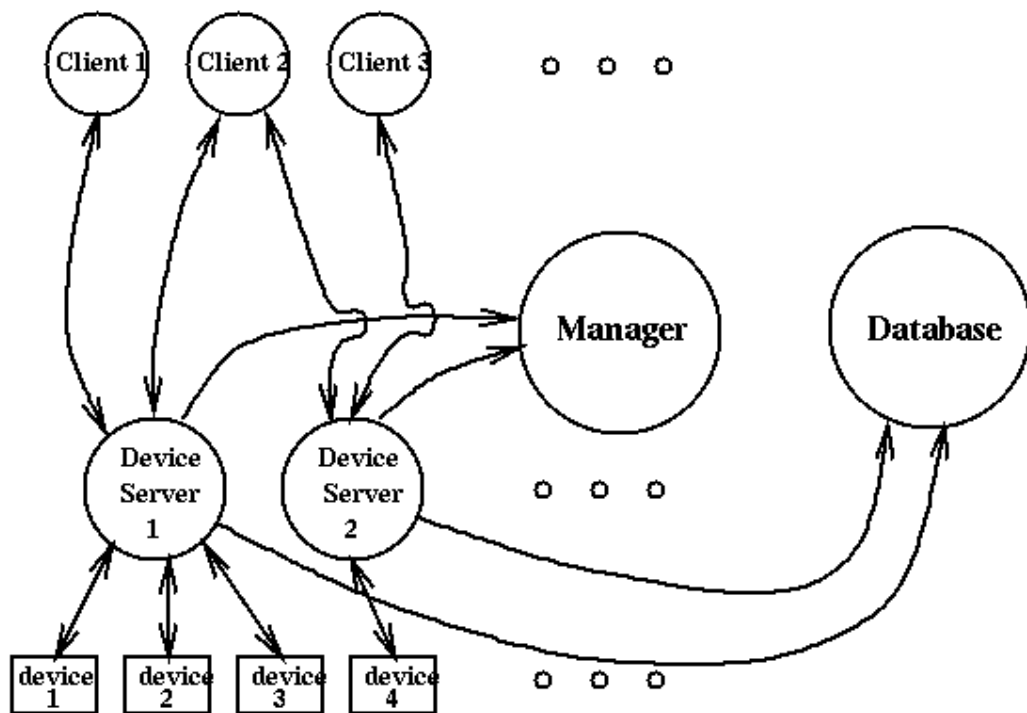
TACO is based on a client-server model. All devices are created and served by device servers. Clients access devices via a network transparent application programmer's interface (DSAPI). In addition to device servers there are so-called system servers—the manager and database which provide system services. There is no a-priori limit to the number of device servers and clients. This makes TACO very scalable.

### 2.3 Manager

The manager is the only fixed point in the whole TACO control system. It is used as a single entry point to start and stop the control system. All clients (including device servers) of the control system connect to the Manager before anything else.

### 2.4 Database

TACO supports a simple database called the resource database where all configuration parameters for devices are stored. The database is served by a database server. All values are stored as ascii strings which are then converted to the correct types at runtime in the calling process (device server). All C simple types and array of simple types are supported. The GNU ndbm database available under Unix and Windows is used as database.



### TACO – basic system architecture

Figure 2.1: TACO system architecture

## 2.5 Device Server Model

One of the most fundamental aspects of TACO can be found in the implementation of device access in the device servers. All device control (input/output) is implemented in the device servers. Device servers are implemented according to a model known as the device server model (DSM). In the DSM all devices (physical and logical) are treated as objects. Each object belongs to a device class. The class implements the actions necessary for each device type. The actions (referred to as commands) can be executed locally or via the network.

The device class implements methods and actions. The actions can be considered as special methods which can be executed by local and remote clients. They have a fixed number of input and output parameters where the parameters can be simple or complex (self-defined) types.

## 2.6 Application Programmer's Interface

The device server model is used for implementing device access in TACO. Users of the control system on the other hand have a "black box" view of the control system. They access the control system either via a high-level programming language (C, C++, Tcl, Spec) using the device server Application Programmer's Interface (DSAPI) or using one of the graphical applications which have been written.

The DSAPI consists the following basic calls :

1. `dev_import()` - import or build up a connection to a device
2. `dev_putget()` - execute a command on a device
3. `dev_putget_async()` - execute a command n a device asynchronously
4. `dev_free()` - free the device

In addition to these calls there are a number of calls for modifying the network communication parameters, interrogating the state of an asynchronous command execution and for managing device security. All network calls to and from the device server are implemented using the Sun Open Network Computing / Remote Procedure Call (ONC/RPC). The ONC/RPC is available on all platforms where the Network File System (NFS) is implemented. The ONC/RPC uses the eXternal Data Representation (XDR) format to encode data sent on the network.

## 2.7 Data Collector

The data collector is a huge distributed shared memory for storing intermediate results of commands from "real" and "pseduo" devices. Real devices are devices which are served by a device server. Pseudo devices are devices which only exist in the data collector. They have no corresponding device class or server. The data collector system is distributed over multiple computers. It is used to cache command results for multiple clients. The pseduo devices are a very useful mechanism for distributing information normally stored in applications or calculated values. Because the data is cached the data collector can be used to solve bottlenecks which arise when many clients request the same value from a device.

The data collector has accessed through an object oriented API very similar to the DSAPI.

## 2.8 Archiving

The long term data archiver in TACO (HDB) is based around a commercial database (Oracle). Using HDB it is possible to do long term archiving over years with a minimum time resolution of 10 seconds. HDB supports 6 different modes of archiving for single values and/or groups of values. HDB offers tools for configuring the database and extracting data. The extracted data are available directly from a C program via a C API or from a Wingz spreadsheet. HDB also offers tape archiving for offlining parts of the data base.

## 2.9 Security

TACO supports secure device access in a network environment. Security is implemented at the device command level. Each device command has its own level of security. Six levels of security are defined :

- READ
- WRITE
- SINGLE\_WRITE
- SUPER\_USER
- SINGLE\_SUPER\_USER
- ADMIN

It is possible using TACO security to "protect" devices from illegal accesses in a networked environment (e.g. Intranet or Internet) and to allow only those users who are authorised and who are logged onto authorised computers to access devices.

## 2.10 Multiple Control Systems

TACO supports the concept of multiple control systems. Each control system has its own database and device servers. Clients and servers of different control systems can communicate with each other as if they were part of the same control system. To specify a device in a different control system a device must be specified with its full name :

`//nethost/d/f/m`

Where nethost is the name of the host where the database of the second control system is running. This concept is sometimes referred to as *multi-nethost* in the documentation.

## 2.11 Uses of TACO

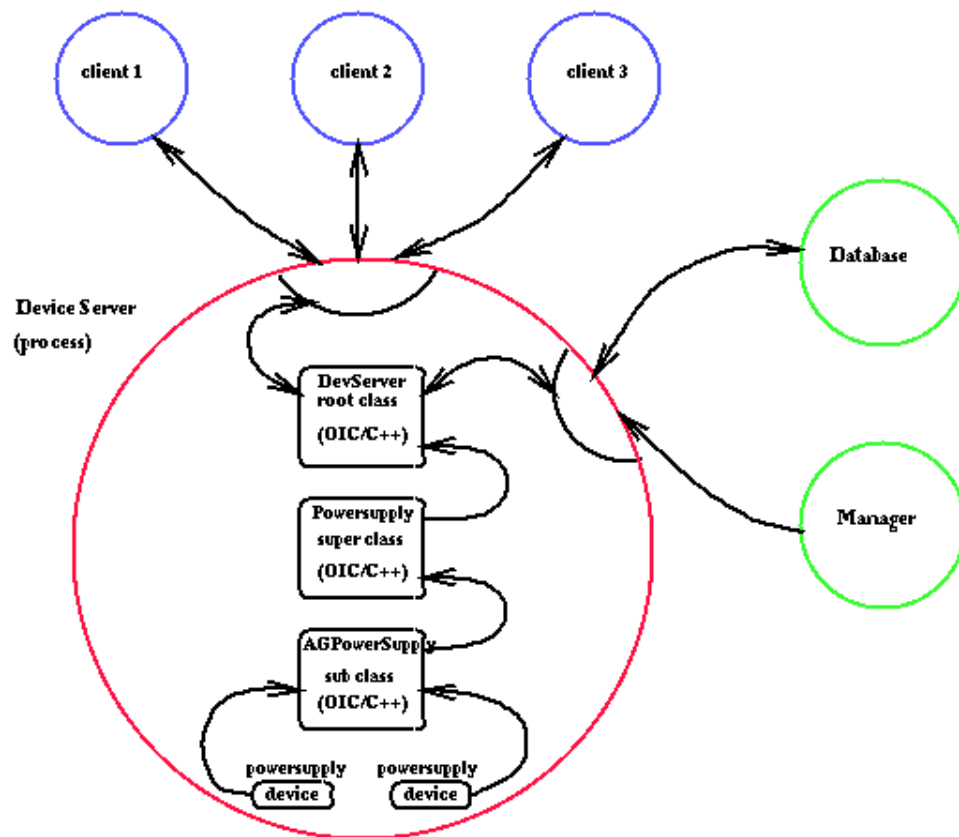
TACO is a toolkit for building distributed objects. Any application which can profit from encapsulating functionality into objects and distributing them over more than one host on the network can find a use for TACO. Control systems are one very good example of this and TACO was developed mainly for doing distributed control. All control systems need to control hardware. The hardware can be in the same computer or more often than not in a variety of computers and black-boxes. It is the job of the control system to coordinate the different hardware. Examples

of hardware are stepper motors, cameras, powersupplies, detectors, adc's, dac's but could even be coffee machines or light switches in the case of home automation. TACO is ideal for encapsulating hardware functionality in a device server and exporting it on the network e.g. for embedded controllers. These are called *tacoboxes* amongst TACO users. GNU/Linux is an ideal candidate as underlying operating system.

TACO can also be used to distribute pure logic where no hardware is involved e.g. for doing image processing, or for sharing data between applications.

TACO has been used in the research environment (synchrotron radiation sources, reactors and telescopes) but is also being used to control robots and soon in the home to automate light switches, heaters, messaging systems etc.





### Schematic of Device Server Model (DSM) for a typical powersupply

Figure 2.2: TACO Device Server Model for a typical PowerSupply

## Chapter 3

# Changes

Here is a list of changes in the TACO manual :

- **V1.1**
  - added section on Changes (this section).
  - documented the use of dynamic error messages (cf. chapter on DSAPI and chapter on Private Command and Errors).



## Chapter 4

# Acknowledgements

A lot of people have contributed to TACO since its beginning. The following people have contributed to the kernel of TACO in the form of system programming, bug fixes, ports etc :

- Martin Diehl (FRMII) - bug fixes
- Andy Götz (ESRF) - device server model, asynchronism, events, dsapi
- Markku Karhu (ESRF) - (original) ndbm database server
- Wolf-Dieter Klotz (ESRF) - Windows port
- Jens Meyer (ESRF) - dsapi, dsxdr, security, manager
- Jon Quick (HartRAO) - bug fixes
- Björn Pederson (FRMII) - bug fixes, improvements to events
- Emmanuel Taurel (ESRF) - rtldb, Oracle and ndbm database server, dbapi, hdb

The following people have written client interfaces to TACO :

- Marie-Christine Dominguez (ESRF) - Python clients
- Laurent Farvacque (ESRF) - Mathlab
- Andy Götz (ESRF) - Labview
- Jens Meyer (ESRF) - Python servers
- Gilbert Pepellin (ESRF) - Tcl
- Faranguiss Poncet (ESRF) - xdevmenu
- Gerry Swislow (CSS) - SPEC

TACO would not be of much use without the device servers therefore it is only fair to mention the (long and incomplete) list of device server programmers :

- A.Beteva (ESRF), D.Carron (ESRF), J.M.Chaize (ESRF), M-C.Dominguez (ESRF), F.Epaud (ESRF), L.Farvacque (ESRF), D.Fernandez (ESRF), A.Götz (ESRF), S.Hunt (SLS), W.D.Klotz (ESRF), M.Konijnenberg (AFOM), P.Mäkijärvi (ESRF), J.Meyer (ESRF), J.Neuhaus (FRM II), W.Öhme (Rossendorf), B.Pederson (FRM II), C.Penel (ESRF), M.Perez (ESRF), M.Peru (ESRF), J.L.Pons (ESRF), J.Quick (HartRAO), B.Regad (ESRF), V.Rey (ESRF), L.Roussier (Lure), B.Scaringella (ESRF), M.Schofield (ESRF), F.Sever (ESRF), E.Taurel (ESRF), P.Verdier (ESRF), R.Wilcke (ESRF), H.Witsch (ESRF)



## Chapter 5

# Getting Started

How to get started with TACO ? The best way is to download it and install it first. Once it is compiled for your platform start the TACO manager and database servers. Start a test device server and client to see if everything is working. The final step is to write your own device server for your hardware and own client for your application and start them. Voila you have a working TACO control system ! Here is a step by step description of the above recipe :

1. *downloading* - TACO can be downloaded from

```
ftp://ftp.esrf.fr/pub/computing/cs/taco/src_release_Vx.y.tar.gz
```

where x.y is the latest version of the TACO source code release (2.6 in July 2000). Download using anonymous ftp (login=anonymous, password=your email address) e.g.

```
cd ~/pub/cs/taco
bin
get src_release_Vx.y.tar.gz
quit
```

2. *unpacking* - unpack the source code in a directory where you have sufficient free space for compiling using tar e.g.

```
tar -xzf src_release_Vx.y.tar.gz
```

3. *compiling* - position your TACO home directory (DSHOME) to the place where you want TACO to be installed (normally the same directory where you unpacked it), run configure and then make and make install :

```
export DSHOME='pwd'
./configure
make all
make install
```

4. *testing* - test TACO has correctly compiled and installed :

```
make test
```

*write a device server* - copy the test device server or a template and adapt it to your hardware, compile it

5. *install device* - create a device entry in the TACO database :

```
db_update TEST/mydevice.res
```

*start TACO* - start TACO manager and database :

```
etc/taco.startup
```

6. *start device server* - position NETHOST and start your device server

```
export NETHOST='hostname'  
mys test&
```

7. *start your client* - start your client and test your device server !

# Chapter 6

## Installing

*by A. Götz*

### 6.1 Introduction

TACO has been developed at the ESRF about 10 years ago but has only recently been started to be used by groups external to the ESRF. It is obvious that to give these external groups as much autonomy as possible they need access to the source code. To satisfy this request the TACO source code release has been prepared. It is basically a copy of the source code development tree maintained at the ESRF. In order to make a quick release not much effort has gone into changing up the directory tree structure and source code. What you have on your disk is a copy of the latest release of the Unix development tree. The main aim is to allow external users to have access to the source code and (re)compile for whatever (Unix) platform they need to. For Windows compilation look under WINDOWS.

### 6.2 Getting Started

#### 6.2.1 GNU make

The release is organised with a main Makefile which calls the underlying Makefiles for compiling the different packages. All the underlying Makefiles are based on the GNU Make which supports conditional statements. Before trying to compile anything you must have a version of GNU make which is accessible from your \$PATH environment when you type "make". GNU make is standard with Linux. For other platforms you can find a release of GNU make in the directory "gmake" with this release. Configure, compile and install it for your platform if you don't have it.

#### 6.2.2 configure

In order to simplify compilation + installation a simple script called "configure" is provided which prompts for what platform you want to compile on. Run configure by typing "./configure" and answer the questions. Before running configure set the environment variable DSHOME It will also prompt for the TACO home directory (\$DSHOME) where you plan to keep all the TACO libraries and include files. This could be anywhere. At the ESRF we normally have a user account "dserver" which we use as home directory for TACO.



If you need the TACO libraries to be compiled with additional CFLAGS (e.g.) -D\_REENTRANT) for your system then it is possible to set an environment variable EXTRACFLAGS before calling configure. This will be added to CFLAGS during compilation of all libraries (DSAPI, DSXDR, DBAPI). The configure script prompts for this flag.

### 6.2.3 make all

Once you have configured the platform you can call "make all" to make all the libraries and system processes.

### 6.2.4 make install

Will copy the libraries and include files to \$DSHOME/lib/\$OS and \$DSHOME/include. Some of the libraries and include files are copied when you do "make all" as part of the TACO boot-strapping process. Will also remake dsapi and dsapi++ because of the "make clean" rule in the makefile.

### 6.2.5 make test

Will fill the TACO database up with some default resources, start a TACO Manager and then start a test device server (Inst\_verify) and client (Inst\_verify\_menu).

### 6.2.6 make clean

Will remove all object files.

### 6.2.7 make clobber

Will do a clean and remove all libraries. It is a good idea to do a clobber before compiling on a new platform to avoid mixing object files and/or libraries.

### 6.2.8 Libraries

The TACO system has three fundamental libraries - DSAPI, DSXDR and DBAPI. These libraries are fundamental to creating any TACO server or client. The source code release contains all the source code for them and Makefiles for generating archive and shared library versions. They can be found in the following directories :

```
DSAPI - ./dserver/system/api/apilib/src
        ./dserver/classes/main/src
        ./dserver/classes++/device/src
```

```
DSXDR - ./dserver/system/xdr/src
```

```
DBAPI - ./dbase/src
```

The libraries are installed in :

```
./lib/$OS
```

The corresponding include files in :

```
./include
./include/private
```

## 6.3 System processes

TACO requires three system process to run - the Manager, Database and Message servers. The source code release contains the source code and Makefiles to generate them. They can be found in :

MANAGER - `./dserver/system/manager/src`

DBSRVR - `./dbase/server/src`

MSGSRVR - `./dserver/system/msg/src`

Once compiled they are installed in :

`./system/bin/$OS`

## 6.4 Database tools

TACO supports a simple database based on the GNU DBM library. DBM is based on a single key and one file per table. Some tools are provided for analysing the contents of the database. They can be found in :

DBTOOLS - `./dbase/tools/src`

Once compiled they are installed in :

`./system/bin/$OS`

## 6.5 Testing

This release assumes you have a running TACO installation and know a bit about TACO. If this is your case all you need to do is point your shared library path (`$LD_LIBRARY_PATH` on Linux/Solaris) to the directory where you have created the shared libraries and restart your device server/client. Alternatively you can recompile you device server/client if you are using archive libraries. The main advantage of the source code release is you will be able to modify and generate new versions of the TACO libraries at will now.

If you have never used TACO before then you better send an email to "taco@esrf.fr" for more detailed instructions. In brief you have to start setup a database, start the Manager and then start as many device server/clients as necessary. Device server/clients which know about your hardware will have to be written. An example for C++ can be found in `dserver/classes++/powersupply`. It consists of a superclass `PowerSupply.cpp` and the subclass `AGPowerSupply.cpp`. A second example of a real device server for controlling a serial line under Linux can be found in `dserver/classes++/serialline`. An example for C (using the Objects In C methodology) can be found in `dserver/classes/instverify`.

## 6.6 Problems

Of course you will have some. Please report them to "taco@esrf.fr". and we will do our best to answer you and include your problem in this section in the future.

Here is a (non-exhaustive) list of problems you can encounter :

- the database server does not compile correctly - the most likely reason is that you do not have the a version of the GNU C++ compiler which includes the standard C++ library. Make sure you have it. You can download it from the web for Solaris from <http://www.sunfreeware.com>. For Linux it comes packaged with the distributions SuSE 6.1 and RedHat 5.2. If you do not have one of these distributions you can download the egcs compiler (the new gcc) from <http://egcs.cygnus.com>.

## 6.7 Windows

This source code release is intended only for Unix platforms. If you need the Windows port which uses Visual C++ then refer to the web page <http://www.esrf.fr/computing/cs/taco/dsapiNT> where you can find a source code distribution for Windows (based on DSAPI V5.15).

## Chapter 7

# Platforms

TACO is actively supported and used on the following platforms :

- **Linux/x86** - following distributions have been tested
  - SuSE
  - Mandrake
  - RedHat
  - Debian

but there is no reason why TACO shouldn't compile and run on any Linux distribution.

- **Linux/68k** - using the Debian distribution on MVME-162's and MVME-167's Motorola's
- **Solaris** - versions 2.5 and 2.7 are supported using the native Solaris compilers and GNU gcc compilers
- **HP-UX** - version 9.x and 10.20
- **OS9** - version V3.03 on VME
- **Windows** - 95/98 and NT using Visual C++ 5.0

The following platforms have been ported to in the past but are not used anymore and are therefore not uptodate :

- **VxWorks** - version 5.x
- **LynxOS** - version ?
- **Irix** - version 6.5

The latter platforms could be updated if need arises.



## Chapter 8

# Device Servers in C++

*by A. Götz and E. Taurel*

### 8.1 Introduction

Device Servers are the distributed objects which form the heart of the TACO control system. They were designed to be written in C based on a technique called *Objects in C* (OIC). This technique was inspired by the Widget model in the X11 Intrinsics Toolkit (Xt). This chapter describes how to implement Device Servers in C++.

This chapter will describe the first C++ implementation of Device Servers taking as an example the AGPowerSupply class. The advantages and disadvantages of this new implementation will be discussed plus the possible future directions which sh/could be explored.

### 8.2 Device Server Model ++

The Device Server Model (DSM) provides a framework for implementing and distributing objects called devices in a networked environment. The original DSM (as described in *The Device Server Programmer's Manual*) was comprised of the following elements :

1. the device,
2. the server,
3. Objects in C,
4. the root class,
5. the device class,
6. the resource database,
7. the commands,
8. local access,
9. network access, and
10. the applications programmer's interface.

Because the DSM has proved to be successful and in order to stay backwards compatible the DSM has been kept as is and only the OIC part has been replaced. However replacing OIC by C++ has meant a new terminology and technology for implementing the individual elements of the DSM. In the C++ implementation the individual elements of the DSM are implemented as follows :

1. a **device** is an instantiation of the base C++ class `Device`,
2. a **server** is an individual process in the classical operating system sense (here nothing has changed),
3. the **root class** is the C++ base class `Device`,
4. a **device class** is a C++ class derived from the public base class `Device` (e.g. `AGPowerSupply`),
5. the **resource database** is a database accessed via a database api (here nothing has changed),
6. **commands** are C++ protected member functions implemented in the device class,
7. **local access** is implemented via the standard api call `dev_putget()` or via the virtual `Command` method implemented in the base class `Device` (the equivalent of the old `dev_cmd()` function),
8. **network access** is provided via the standard api call `dev_putget()`,
9. the **applications programmer's interface** is the same as before i.e. the client does not know if the server is implemented in OIC or a C++

In addition to the above basic elements the following additional points can be made about the C++ implementation of the DSM :

- the **class initialise** function (called once for every class) although not supported by the C++ language has been retained in order to allow efficient implementation e.g. for the retrieving of class resources, and is implemented as a virtual private method in the base class `Device`,
- the **object initialise** method has been suppressed,
- the **state machine** has (of course) been retained and is implemented as a virtual public method in the base class,
- a **get resource** method has been added as a standard method in all in order to retrieve resources from the static database.
- C++ does not support **class variables** in the same manner OIC does i.e. one copy of a variable per class and derived class common to all instantiations of that class. Class variables were therefore transformed into static class members, static variables (with file scope) or in the worst case a copy of the variable was stored in each object.

## 8.3 Device root class

All device classes **must** be derived from the `Device` base class (also known as the root class). The `Device` class replaces the old `DevServer` class. The server part is implemented in the rpc stubs and in the standard Device Server `main()`. This distinction between what is a device and what is a server creates a clean separation between two functionally different aspects of the DSM.

The following comments can be made about the present implementation :

- `Device` is implemented as an **abstract class** (one of its members, `GetResources`, is a **pure virtual function**). This means `Device` cannot be instantiated and can only serve as a base class for derived classes.
- the new type `DeviceCommandListEntry` replaces the old `DevCommandListEntry`.
- a command is defined as a pointer member functions of the `Device` class (or a class publicly derived from `Device`) which takes as arguments two void and one long pointer and returns a long status. The void pointers refer to argin and argout and have to be casted to the correct type inside the command.
- the standard commands `State()` and `Status()` are implemented as virtual methods in the base class. This means that any derived class which does not implement these commands automatically inherits the base class implementation.
- a dummy `StateMachine` method is implemented as virtual method which always returns `DEVOK`.
- as mentioned above the object initialise as something different from the object create method has been suppressed from the DSM. This has been done for simplicity reasons (in the past most Device Server Programmer's did not know what the difference between the two were) and also to be more in the spirit of C++. All initialisation is now done at object create time in the class constructor method.
- most of the variables required by the old `DevServer` implementation have been retained for compatibility reasons e.g. `class_name`, `dev_type`, these are also needed by the api when exporting a device.
- each instantiation object of a class derived from `Device` has a pointer to the commands list and the number of commands. This was unavoidable because C++ does not support the notion of class variables.

### 8.3.1 Device.h - include file

The `Device` interface is defined in the public include file `Device.h` and is listed below.

```
%\include{/segfs/dserver/dev/classes++/device/include/Device.h}
//static char RcsId[] = "$Header: /segfs/taco/doc/manual/cppdserver.tex,v 1.1 2000/07/24 09:4
//+*****
//
// File: Device.h
//
// Project: Device Servers in C++
//
```



```

// Description: public include file containing definitions and declarations
// for implementing the device server Device base class in C++
// (DeviceClass).
//
// Author(s): Andy Goetz
//
// Original: February 1995
//
// $Revision: 1.1 $
//
// $Date: 2000/07/24 09:42:46 $
//
// $Author: goetz $
//
// $Log: cppdserver.tex,v $
// Revision 1.1 2000/07/24 09:42:46 goetz
// Initial revision
//
//
//+*****

#ifndef _DEVICE_H
#define _DEVICE_H

// Some remarks about the Device class definition
//
// 1 - Members class_name and dev_type should not be defined as static members
//      otherwise, there will be only one copy of them for the device server
//      process and it is not possible to correctly handle device server
//      with several embedded classes
//      Therefore, don't forget to initialize them in the object constructor
//      and not in the class_initialise function which is executed only once
//      for a class.
//
// 2 - The State and Status member function are declared as public. This is due
//      to the OS-9 C++ compiler. To reuse them in a device derived class
//      (by specifying a pointer to them in the command list), the OS-9 compiler
//      needs the function to be declared as public !!
//

class Device {

//
// private members
//

private :

//
// private virtual functions which should be defined in each new sub-class
//

```

```

static short class_initd;

virtual long ClassInitialise( long *error );
virtual long GetResources (char *res_name, long *error) = 0; // pure virtual

//
// public members
//

public:

typedef long (Device::* DeviceMemberFunction)(void*, void*, long* );
typedef struct _DeviceCommandListEntry {
    DevCommand          cmd;
    DeviceMemberFunction fn;
    DevArgType          argin_type;
    DevArgType          argout_type;
    long                min_access;
}
    DeviceCommandListEntry;

typedef struct _DeviceCommandListEntry *DeviceCommandList;

virtual long State(void *vargin, void *vargout , long *error);
virtual long Status(void *vargin, void *vargout, long *error);

//
// class variables
//

char* class_name;
char dev_type[24];

char* name;

Device (DevString name, long *error);
~Device ();
virtual long Command ( long cmd,
    void *argin, long argin_type,
    void *argout, long argout_type,
    long *error);
long  Get_min_access_right(long,long *,long *);
void  Get_command_number(unsigned int *);
long  Command_Query(_dev_cmd_info *,long *);

//
// protected members - accessible only be derived classes
//

protected:

```

```
//
// the following virtual commands must exist in all new sub-classes
//
virtual long StateMachine( long cmd, long *error);

long state; // device state
long n_state; // convenience variable for storing next device state
long n_commands;
DeviceCommandList commands_list;

};
```

```
#define TYPE_DEFAULT "DevType_Default"
#define TYPE_INIT "DevType_"

#endif /* _DEVICE_H */
```

### 8.3.2 Device.cpp - source code file

The following points can be made about the Device class source code implementation :

- the Device constructor `Device::Device` (listed below) defines a command list containing two commands - `DevState` and `DevStatus`. This command list will normally be overridden by the derived device class but in the case that the device class defines no command list the derived class will have at least the two standard commands.
- the `ClassInitialise` method is called from the constructor via the static variable `class_inited`.

```
//+=====
//
// Function: Device::Device()
//
// Description: constructor to create an object of the base class Device
//
// Input: char *name - name (ascii identifier) of device to create
//
// Output: long *error - error code returned in the case of problems
//
//-----

Device::Device (char *devname, long *error)
{
    static DeviceCommandListEntry dev_cmd_list[] = {
        {DevState, &Device::State, D_VOID_TYPE, D_SHORT_TYPE},
        {DevStatus, &Device::Status, D_VOID_TYPE, D_STRING_TYPE},
    };

    static long no_commands = sizeof(dev_cmd_list)/
        sizeof(DeviceCommandListEntry);

    dev_printdebug(DBG_TRACE,"Device::Device() called, devname = %s\n",devname);
```

```

        *error = DS_OK;

//
// check if ClassInitialise() has been called
//

    if (Device::class_initd != 1)
    {
        if (Device::ClassInitialise(error) != DS_OK)
        {
            return;
        }
    }

//
// initialise class_name (this should be done here because class_name
// is NOT a static member of the device class for the case of device
// server with several embedded classes. Also initialises, device
// type
//

    this->class_name = "DeviceClass";
    sprintf(this->dev_type,TYPE_DEFAULT);

//
// initialise the device name
//

    this->name = (char*)malloc(strlen(devname)+1);
    sprintf(this->name,"%s",devname);

//
// initialise the commands list
//

    this->n_commands = no_commands;
    this->commands_list = dev_cmd_list;

    this->state = DEVON;
}

```

- one of the most important member methods of the Device class is the Command method which searches for the required command in the device class' command list, calls the state machine and then calls the command. One not so clean feature of this implementation is that the type checking is done by the method and not by the C++ compiler but this has so far proved unavoidable. Command is defined to virtual so that it can be overloaded by any of the sub-classes. This is necessary for the OICDevice class which needs to call the (old) OIC DevMethodCommandHandler. Normally no other classes need to overload the Command method.

```
long Device::Command (long cmd, void* argin, long argin_type,
```

```

        void* argout, long argout_type, long *error)
{
    int i;
    DeviceMemberFunction member_fn;

    printf("Device::Command() called, cmd = %d\n",cmd);

    // add code to execute a command here

    for (i = 0; i < this->n_commands; i++)
    {
        if (cmd == this->commands_list[i].cmd)
        {
            if (argin_type != this->commands_list[i].argin_type ||
                argout_type != this->commands_list[i].a
            {
                *error = DevErr_IncompatibleCmdArgumentTypes;
                return(DS_NOTOK);
            }

            // check state machine

            if (this->StateMachine(cmd,error) != DS_OK)
            {
                return(DS_NOTOK);
            }

            // now execute the command

            member_fn = this->commands_list[i].fn;

            if ((this->*member_fn)(argin,argout,error) != DS_OK)
            {
                return(DS_NOTOK);
            }
            else
            {
                return(DS_OK);
            }
        }
    }

    *error = DevErr_CommandNotImplemented;

    return(DS_NOTOK);
};

```

## 8.4 PowerSupply class - an example superclass

At the ESRF the functionalities of a standard powersupply class have been defined (cf. DSN/078) and are implemented in the superclass `PowerSupplyClass` in OIC. This section describes an equivalent C++ implementation which respects the ESRF

standard.

The following points can be made about this implementation :

- **PowerSupply** is defined as an **abstract class** (it has one **pure virtual** function (**StateMachine**)). This means it can only be used as a base class for other derived classes and cannot be instantiated,
- the exact same types for class member variables were used for the C++ implementation as for the previous OIC implementation, the only difference being that they were defined as **protected** which means that they are only visible to classes derived from the **PowerSupply** class,
- **CheckReadValue** is implemented as a protected method to be used only by classes derived from the **PowerSupply** class.

#### 8.4.1 PowerSupply.h - include file

```
class PowerSupply : public Device {

// private members

private :

    long ClassInitialise( long *error );
    long GetResources (char *res_name, long *error);

// protected members

protected:

    float set_val;
    float read_val;
    long channel;
    long n_ave;
    long fault_val;
    float cal_val;
    float conv_val;
    char *conv_unit;
    float set_offset;
    float read_offset;
    float set_u_limit;
    float set_l_limit;
    float idot_limit;
    long polarity;
    float delta_i;
    long time_const;
    long last_set_t;

    long CheckReadValue(DevBoolean *check, long *error);

    virtual long StateMachine( long cmd, long *error)=0; // pure virtual function

// public members

public:
```

```

    PowerSupply (char *name, long *error);
    ~PowerSupply ();

};

```

#### 8.4.2 PowerSupply.cpp - source code file

The implementation of `PowerSupply` class is very simple and does not do much. One interesting point however is the `GetResources` method which retrieves the `delta_i` and `time_constant` resources. Because this is called directly from the constructor during initialisation it is transparent to the sub-classes. This was not possible in the OIC and demonstrates the advantage of using C++. Here is the listing of the `GetResources` method :

```

long PowerSupply::GetResources (char *res_name, long *error)
{
    static db_resource res_powersupply[] = { {"delta_i", D_FLOAT_TYPE},
                                              {"time_constant", D_LONG_TYPE}, };
    static unsigned int res_powersupply_size = sizeof(res_powersupply)/
                                              sizeof(db_resource);

    register int ires;

    *error = DS_OK;

    //
    // setup the db_resource structure so that we can interrogate the database
    // for the two resources "delta_i" and "time_constant" which are needed
    // by all powersupplies to implement the read<>set check
    //

    ires = 0;
    res_powersupply[ires].resource_adr = &(this->delta_i); ires++;
    res_powersupply[ires].resource_adr = &(this->time_const); ires++;

    if (db_getresource(res_name, res_powersupply, res_powersupply_size, error)
        != DS_OK)
    {
        printf("PowerSupply::GetResources() db_getresource failed, error %d\n",
            *error);
        return(DS_NOTOK);
    }

    return(DS_OK);
}

```

### 8.5 AGPowerSupply class - an example derived class

`AGPowerSupply` is an example of a device class derived from the `PowerSupply` class, it simulates a real powersupply and is one of the simulators used by the application programmers to simulate the machine.

The class definition can be found in the public include file (`AGPowerSupply.h`). The following comments can be made on present implementation :

- the `State` command is inherited from base class `Device`,
- the `Status` command implemented in the `AGPowerSupply` derived class overrides the base class implementation.

### 8.5.1 `AGPowerSupply.h` - include file

```
class AGPowerSupply : public PowerSupply {

// private members

private :

    long ClassInitialise (long *error );
    long GetResources (char *res_name, long *error);

// protected members

protected:

// commands

    long Off (void *argin, void *argout, long *error);
    long On (void *argin, void *argout, long *error);
    long Status (void *argin, void *argout, long *error);
    long SetValue (void *argin, void *argout, long *error);
    long ReadValue (void *argin, void *argout, long *error);
    long Reset (void *argin, void *argout, long *error);
    long Error (void *argin, void *argout, long *error);
    long Local (void *argin, void *argout, long *error);
    long Remote (void *argin, void *argout, long *error);
    long Update (void *argin, void *argout, long *error);

    long StateMachine (long cmd, long *error);

// public members

public:

    AGPowerSupply (char *name, long *error);
    ~AGPowerSupply ();

};
```

### 8.5.2 `AGPowerSupply.cpp` - source code

Below are some examples taken from the `AGPowerSupply.cpp` source code which illustrates some of the details of the C++ implementation.

- the notion of **template** has been kept in the present C++ implementation. This is done in a somewhat unorthodox manner because of the fact that C++ does not implement this feature. A global pointer to a copy of an `AGPowerSupply` is defined in static address space. The pointer is initialised to point to a block of memory of size `sizeof(AGPowerSupply)` allocated in `ClassInitialise`. The individual fields of the template are then initialised to the class defaults



in `ClassInitialise`. The reason for this unorthodox approach is because it is not possible to address an object which does not exist (if you understand what I mean !)<sup>1</sup>

```
long AGPowerSupply::ClassInitialise (long *error)
{
    static AGPowerSupply *agps_template = (AGPowerSupply*)malloc(sizeof(AGPowerSupply));

    int iret=0;

    printf ("AGPowerSupply::ClassInitialise() called\n");

    // AGPowerSupplyClass is a subclass of PowerSupplyClass

    class_name = (char*)malloc(strlen("AGPowerSupplyClass")+1);
    sprintf(class_name,"AGPowerSupplyClass");

    class_inited = 1;

    // initialise the template powersupply so that DevMethodCreate has
    // default values for creating a powersupply, these values will be
    // overridden by the static database (if defined there).

    // default is to start with powersupply switched OFF; the state
    // variable gets (ab)used during initialisation to interpret the
    // initial state of the powersupply: 0==DEVOFF, 1==DEVON. this is
    // because the database doesn't support the normal state variables
    // like DEVON, DEVSTANDBY, DEVINSERTED, etc.

    agps_template->state = 0;
    agps_template->n_state = agps_template->state;
    agps_template->set_val = 0.0;
    agps_template->read_val = 0.0;
    agps_template->channel = 1;
    agps_template->n_ave = 1;
    agps_template->conv_unit = (char*)malloc(sizeof("AMP")+1);
    sprintf(agps_template->conv_unit,"AMP");
    agps_template->set_offset = 0.0;
    agps_template->read_offset = 0.0;
    agps_template->set_u_limit = AG_MAX_CUR;
    agps_template->set_l_limit = AG_MIN_CUR;
    agps_template->polarity = 1.0;

    // interrogate the static database for default values

    if(GetResources("CLASS/AGPS/DEFAULT",error))
    {
        printf("AGPowerSupply::ClassInitialise(): GetResources() failed, error %d\n",error);
        return(DS_NOTOK);
    }
}
```

---

<sup>1</sup>if `agps_template` was defined as a new `AGPowerSupply` the first time the constructor is called it will try to access `agps_template->something` but `agps_template` does not exist yet and will generate a bus error

```

    }

    agps_template->state = state;
    agps_template->set_val = set_val;
    agps_template->read_val = read_val;
    agps_template->channel = channel;
    agps_template->n_ave = n_ave;
    agps_template->conv_unit = (char*)malloc(sizeof(conv_unit)+1);
    sprintf(agps_template->conv_unit,conv_unit);
    agps_template->set_offset = set_offset;
    agps_template->read_offset = read_offset;
    agps_template->set_u_limit = set_u_limit;
    agps_template->set_l_limit = set_l_limit;
    agps_template->polarity = polarity;

    printf("returning from AGPowerSupply::ClassInitialise()\n");

    return(iret);
}

```

- the command list references the two commands `DevState` and `DevStatus` in the base class `Device`. Because they are defined in the base class as **virtual** C++ uses dynamic binding to resolve them and therefore at run-time `Device::DevState` and `AGPowerSupply::DevStatus` are executed respectively.

```

static Device::DeviceCommandListEntry commands_list[] = {
    {DevState, (DeviceMemberFunction)&Device::State, D_VOID_TYPE, D_SHORT_TYPE},
    {DevStatus, (DeviceMemberFunction)&Device::Status, D_VOID_TYPE, D_STRING_TYPE},
}

```

- simple commands (e.g. which don't take input or output parameters) have not changed much in their implementation e.g. `AGPowerSupply::Off` looks as follows :

```

long AGPowerSupply::Off (void *vargin, void *vargout, long *error)
{
    printf("AGPowerSupply::Off(%s) called\n",name);

    *error = DS_OK;

    read_val = 0.0;
    set_val = 0.0;
    state = DEVOFF;

    return (DS_OK);
}

```

- commands which take input or output parameters have to cast their parameters from void to pointers to the correct type. Here is an example of `AGPowerSupply::Update` which calls two other commands to return the state, set and read value :

```

long AGPowerSupply::Update ( void *vargin, void *vargout, long *error)

```

```

{
    DevStateFloatReadPoint *vargout_sfrp;
    DevShort darg_short;
    DevFloatReadPoint darg_frp;

    printf("AGPowerSupply::Update(%s) called\n",name);

    vargout_sfrp = (DevStateFloatReadPoint*)vargout;

    // update state

    State(NULL, &darg_short, error);
    vargout_sfrp->state = darg_short;

    // get latest set and read

    ReadValue(NULL, &darg_frp, error);
    vargout_sfrp->set = darg_frp.set;
    vargout_sfrp->read = darg_frp.read;

    return(DS_OK);
}

```

## 8.6 startup.cpp - an example startup file

Any device which has to be served by a Device Server has to be created and exported as usual in a startup procedure. Listed below is an example `startup()` for the `AGPowerSupply` class which reads a list of devices from the static database, instantiates them, executes a command on them (to see if they are alive) and then exports them.

```

#include <API.h>
#include <Device.h>
#include <DevServer.h>
#include <PowerSupply.h>
#include <AGPowerSupply.h>

#define MAX_DEVICES 1000

extern "C" long startup(char *svr_name, long *error);

unsigned int n_devices, i;
Device *device[MAX_DEVICES];

long startup(char *svr_name, long *error)
{
    char **dev_list;
    short state;
    long status;

    printf ("startup++() program to test dserver++ (server name = %s)\n",svr_name);

```

```

// get the list of device name to be served from the static database

if (db_getdevlist(svr_name,&dev_list,&n_devices,error))
{
    printf("startup(): db_getdevlist() failed, error %d\n",*error);
    return(-1);
}
printf("following devices found in static database: \n\n");
for (i=0;i<n_devices;i++)
{
    printf("\t%s\n",dev_list[i]);
}

// now loop round creating and exporting the devices

for (i=0; i<n_devices; i++)
{
    device[i] = new AGPowerSupply(dev_list[i],error);

    if ((device[i] == 0) || (*error != 0))
    {
        printf("Error when trying to create %s device\n",dev_list[i]);
        return(DS_NOTOK);
    }
    else
    {
// test calling Device::State via Device::Command method

        device[i]->Command(DevState, NULL, D_VOID_TYPE, (void*)&state, D_SHORT_TYPE, error);

// export the device onto the network

        status = dev_export((char*)device[i]->name,(Device*)device[i],(long*)error);

        printf("startup++() dev_export() returned %d (error = %d)\n",status,*error);
    }
}

return(DS_OK);
}

```

## 8.7 OICDevice wrapper class

Writing device servers in C++ is no problem for new classes which do not depend on any existing classes. However one of the main aims of object oriented programming is *code reuse*. Seeing as the majority of classes at the ESRF were written before C++ was available on OS9 they were written in C using the Objects In C (OIC) methodology. It is vital therefore that C++ classes can (re)use OIC classes. Two possibilities of including OIC classes in C++ considered were :

1. calling the OIC C functions `ds_create()`, `ds_method_finder()` directly from C++,

2. writing a C++ wrapper class for OIC which "wraps" the OIC DevServer objects as a C++ Device objects.

The first method (C++ calls OIC C directly) poses the problem of what happens when the programmer wants to export a mixture of C++ and C objects onto the network ? The device server `main()` routine assumes can manage a list of either all OIC DevServer's or all C++ Device's but not both. It was decided therefore to use the second method (C++ wrapper class) and write a class called `OICDevice`.

`OICDevice` is a C++ wrapper class for OIC classes. `OICDevice` is a generic class for creating objects of any OIC class, it is derived from the `Device` root class. The result is a C++ `OICDevice` object which has a pointer to the actual OIC object. Seen from the C++ programmer's point of view it appears as a C++ object. It has the same interface as all other C++ objects dervied from `Device`. Executing commands on the object will result in the OIC command method handler being called.

Some points to be aware of when wrapping your OIC objects with `OICDevice` :

- `OICDevice` implements basic versions of `DevState` and `DevStatus` which access the OIC device state,
- the actual state of the `OICDevice` object is stored in the OIC object, to access it use `(short)this->ds->devserver.state` (and NOT the `state` variable in the `Device` part of the object),
- to access the OIC object use the pointer stored in the `OICDevice` object part i.e. `this->ds` (use this for example to access any fields of the OIC object e.g. `((PowerSupply)this->ds)->powersupply.set_val`),
- to access the OIC object's class use the pointer stored in the `OICDevice` object part i.e. `this->ds_class`,

Note the `OICDevice` class is only a wrapper class for encapsulating OIC **objects** and not classes. Because of the differences between the OIC and C++ implementations it is not possible to *derive* new C++ classes from existing OIC classes as sub-classes. It is however possible to instantiate OIC classes in C++. If you want to use an existing OIC class as a super-class for C++ then you have to rewrite the OIC class in C++.

### 8.7.1 `OICDevice.h` - include file

Here is the source code of the `OICDevice.h` header file which defines the interface to the `OICDevice` class :

```
//static char RcsId[] = "$Header: /segfs/taco/doc/manual/cppdserver.tex,v 1.1 2000/07/24 C
//+*****
//
// File: OICDevice.h
//
// Project: Device Servers in C++
//
// Description: public include file containing definitions and declarations
// for implementing OICDevice class in C++. The OICDevice class
// wraps (old) OIC classes in C++ so that they can be used
// in C++ classes derived from the Device base class.
//
// Author(s): Andy Goetz
```

```

//
// Original: November 1996
//
// $Revision: 1.1 $
//
// $Date: 2000/07/24 09:42:46 $
//
// $Author: goetz $
//
// $Log: cppdserver.tex,v $
// Revision 1.1 2000/07/24 09:42:46 goetz
// Initial revision
//
//
//
//+*****
#ifndef _OICDEVICE_H
#define _OICDEVICE_H

class OICDevice : public Device {
//
// private members
//
private :
//
// private virtual functions which should be defined in each new sub-class
//
    static short class_inited;
    long ClassInitialise( long *error );
//
// not many OIC classes have this method
//
    long GetResources (char *res_name, long *error);
//
// public members
//
public:
    long State(void *vargin, void *vargout , long *error);
    long Status(void *vargin, void *vargout, long *error);
//
// class variables
//
    OICDevice (DevString devname, DevServerClass devclass, long *error);
    ~OICDevice ();
    long Command ( long cmd,
                  void *argin, long argin_type,
                  void *argout, long argout_type,
                  long *error);
    inline short get_state(void) {return(this->ds->devserver.state);}
    inline DevServer get_ds(void) {return(this->ds);}
    inline DevServerClass get_ds_class(void) {return(this->ds_class);}
//
// protected members - accessible only from derived classes
//

```

```
protected:
    long StateMachine( long cmd, long *error);
//
// OICDevice member fields
//
    DevServer ds; // pointer to the old OIC object
    DevServerClass ds_class; // pointer to the old OIC class
};
#endif /* _OICDEVICE_H */
```

### 8.7.2 startup.cpp - an example

Here is an example of a simple C++ startup function which creates a OIC AGPowerSupply object in C++ using the OICDevice wrapper class (note the syntax for the full C++ case commented out) :

```
static char RcsId[] = "$Header: /segfs/taco/doc/manual/cppdserver.tex,v 1.1 2000/07/24 09:
//+*****
//
// File: startup.cpp
//
// Project: Device Servers in C++
//
// Description: startup source code file for testing the OIC AGPowerSupply class
// in C++. AGPowerSupply class implements a simulated powersupply
// derived from the base classes PowerSupply and Device (root
// class).
//
//
// Author(s): Andy Goetz
//
// Original: November 1997
//
// $Revision: 1.1 $
//
// $Date: 2000/07/24 09:42:46 $
//
// $Author: goetz $
//
// $Log: cppdserver.tex,v $
// Revision 1.1 2000/07/24 09:42:46 goetz
// Initial revision
//
//+*****

#include <iostream.h>

#include <API.h>
#include <Device.H>
#include <DevServer.h>
#include <DevServerP.h>
#include <OICDevice.H>
#include <PowerSupply.h>
```

```

#include <PowerSupplyP.h>
#include <AGPowerSupply.h>
#include <AGPowerSupplyP.h>

#define MAX_DEVICES 1000

long startup(char *svr_name, long *error)
{
    char **dev_list;
    unsigned int n_devices, i;
    OICDevice *device[MAX_DEVICES];
    short state;
    long status;

    printf ("startup++() program to test dserver++ (server name = %s)\n",svr_name);
    //
    // get the list of device name to be served from the static database
    //
    if (db_getdevlist(svr_name,&dev_list,&n_devices,error))
    {
        printf("startup(): db_getdevlist() failed, error %d\n",*error);
        return(-1);
    }
    printf("following devices found in static database: \n\n");
    for (i=0;i<n_devices;i++)
    {
        printf("\t%s\n",dev_list[i]);
    }

    //
    // now loop round creating and exporting the devices
    //
    for (i=0; i<n_devices; i++)
    {
        //
        // DO NOT create AGPowerSupply (C++) objects
        //
        // device[i] = new AGPowerSupply(dev_list[i],error);
        //
        //
        // create old (OIC) AGPowerSupply objects
        //
        device[i] = new OICDevice(dev_list[i],(DevServerClass)aGPowerSupplyClass,error);
        //
        // test calling Device::State via Device::Command method
        //
        device[i]->Command(DevState, NULL, D_VOID_TYPE, (void*)&state, D_SHORT_TYPE, error);
        //
        // export the device onto the network
        //
        status = dev_export((char*)device[i]->name,(Device*)device[i],(long*)error);
        printf("startup++() dev_export() returned %d (error = %d)\n",status,*error);
    }
}

```



```

    return(DS_OK);
}

```

## 8.8 Implementation

In designing the present implementation the following requirements were considered :

1. to conserve as much as possible the investment made in the device server api and the existing classes,
2. to be compatible with any further developments made in the api,
3. clients should not have to be modified.

Based on these requirements it was decided to implement only the device classes in C++ and keep the api in C thereby satisfying all three requirements. It means that there is only one api implementation and it can be continued to be developed and the improvements/bug fixes will be visible to device servers written in C and C++ and to clients.

To implement device servers in C++ the following modifications were made:

1. the `svc_api.c` file (which implements the rpc stubs for the api functions) was modified so that (1) when compiled with the C compiler it uses the `OIC DevMethodCommandHandler` and (2) when compiled with C++ it calls the `Device::Command` method. Here is an example taken from the `dev_putget()` function :

```

#ifdef __cplusplus
/*
 * OIC version
 */
    client_data.status = (ds__method_finder (ds, DevMethodCommandHandler))
        (    ds,
          server_data->cmd,
          server_data->argin,
          server_data->argin_type,
          client_data.argout,
          client_data.argout_type,
          &client_data.error);
#else
/*
 * C++ version
 */
    client_data.status = device->Command(server_data->cmd,
                                        (void*)server_data->argin,
                                        server_data->argin_type,
                                        (void*)client_data.argout,
                                        client_data.argout_type,
                                        &client_data.error);
#endif /* __cplusplus */

```

2. `svc_api.c` was also modified so that it can deal with `Devices` and not `DevServers` anymore. In OIC a list of `DevServer` has to be managed, while in C++ a list of `Device` has to be managed i.e.

```

#ifndef __cplusplus
    DevServer      ds;

    ds      = (DevServer) ptr_ds;
#else
    Device      *device;

    device    = (Device*) ptr_ds;
#endif /* __cplusplus */

```

3. All include files had to be modified to declare external functions as C functions for the C++ compiler e.g.

```
extern "C" long dev_export PT_((char* dev_name, Device *ptr_dev, long *error))
```

## 8.9 Compilers

The first C++ implementation was done in 1995 (by AG) using the HP CC compiler on the HP 9000/700 series. This compiler is a 2.x C++ compiler and supports symbolic debugging. When compiling the following symbols have to be defined `_STDC_`, `unix`, and `HPUX_SOURCE`.

In 1996 this work was repeated (by ET) for the Kicker Powersupply at the ESRF using the Ultra-C++ compiler from Microware and the GNU g++ compiler on HP-UX.

For the future we propose that wherever possible the GNU g++ compiler must be used. Where it is not possible the best adapted native compiler should be used.

This is clearly the case for OS9 where the native Ultra-C++ compiler from Microware is the obvious choice. This is not so clear for HP-UX - the GNU g++ compiler does not support exceptions but is otherwise a good choice. For the present g++ is supported under HP-UX (i.e. the C++ libraries are compiled only with the g++ compiler<sup>2</sup>).

## 8.10 Template Class

In the absence of `xclassgen` supporting C++ we have written templates for a `Template` class. The templates were derived from the `KickerSupply` class but have never been compiled (i.e. we do not guarantee there are no bugs!). To use the templates simply copy them and modify them with a global editor replacing all occurrences of `Template` and `template` with `MyNewClassName` and `mynewclassname` (the name of your new class).

The templates can be found in `libra:/users/d/dserver/classes++/template`:

1. `include/Template.H` - template include file
2. `src/Template.cpp` - template source file
3. `src/startup.cpp` - template startup file
4. `src/Makefile` - template Makefile for HP-UX and Ultra C++

---

<sup>2</sup>note that because the GNU compiler uses a different algorithm for "name mangling" it is not possible to mix GNU object files with those compiled with a different compiler

## 8.11 C++ Programming Style

The following style conventions have been adopted :

- the suffixes `.H` and `.cpp` were used for C++ include files and source files respectively.
- the C++ commenting style which uses `//` at the beginning of each line has been used in order to distinguish it from the pure C style of `/* bla bla */`.
- extensive use of the `this` pointer has been made to make the code as explicit and readable as possible.
- no use has been made of `ref` types.

## 8.12 Advantages of C++

The following are some of the advantages of using C++ for writing device servers as opposed to OIC :

1. C++ is a real language with compiler support and symbolic debuggers,
2. C++ is well documented and has a large selection of literature (see the section on Suggested Reading),
3. because of the compiler support for C++ it is easier to program new classes, the programmer does not have to learn the many big and small letter conventions of OIC,
4. a new class can have more than one base class (**polymorphism**),
5. C++ is more compatible with new products for which only C++ bindings exist e.g. **Corba**, D00CS and **cdev**.

## 8.13 Disadvantages of C++

There are not many disadvantages of using C++ but here are some of them :

1. C++ with all its many concepts and possibilities has a steep learning curve,
2. extensive use of **operator overloading**, **function overloading** and **virtual functions** can very quickly make C++ totally unreadable,
3. C++ executables are big ( $\approx 500$  kilobytes on HP-UX) compared to OIC executables ( $\approx 150$  kilobytes on HP-UX).

## 8.14 Future developments

Some of the future directions to be considered are :

1. ports to other platforms e.g. Solaris, Linux, LynxOS, vxWorks, will be undertaken,
2. the class generator tool will be extended so that it can generate C++,
3. more use of **inline** functions should be made,

4. define and implement C++ bindings for the database API and the device server API (D00CS or cdev?).
5. if **templates** and **exception handling** become standard on all compilers then investigate how they can be best used,

## 8.15 Conclusion

OIC has served a useful purpose but the time has come to move to a real object oriented language and C++ seems to be the obvious choice. The present implementation shows that it is possible to implement Device Servers in C++ and still be backwards and forwards compatible with the device server api and the existing OIC classes.

## 8.16 Suggested Reading

A lot of literature exists on C++ (books, journals, conferences proceedings etc.) here is a short list of titles which can be recommended :

1. *C++ Primer* by Stanley B. Lippman,
2. *The C++ Programming Language* by Bjarne Stroustrup,
3. *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup (ANSI Base Document),
4. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs* by Scott Meyers,
5. *More Effective C++: 35 New Ways to Improve Your Programs and Designs* by Scott Meyers,



## Chapter 9

# DSAPI

*by J.Meyer and A.Götz*

### 9.1 Introduction

The DSAPI is the TACO Device Server Application Programmer's Interface for C and C++ programs. It is used by clients and servers to import, execute commands on, free and explore TACO devices. It uses the ONC-RPC (SUN remote procedure call) as underlying communication protocol. This document describes the latest version of the DSAPI V6.0.

This document is split into the following sections : "Getting Started" describes how to write a simple client which uses DSAPI, should be read by beginner's who want to get a quick start; "C library" is a reference guide to all DSAPI functions for clients and servers; "XDR Types" describes the XDR types supported by DSAPI; "Changes" describes what are the main changes in the different major releases; "Platforms Supported" lists the different platforms and compilers supported; "Interfaces to other Languages" contains a summary of DSAPI interfaces in other languages. Beginners should read "Getting Started" first, other programmers should read what changes have taken place in the latest version and use the reference guide.

### 9.2 What is DSAPI ?

DSAPI is a C (and C++) programmer's interface for accessing device in a TACO control system.

Devices in a TACO control system are network objects created and served by processes called device servers. A device is identified by its ASCII name :

`[//facility/]domain/family/member`

Each device understands a set of commands. The commands enable a remote client to execute actions on a device e.g. for a powersupply switch it on or off, read the state, read the current.

The DSAPI gives remote and local clients access to device commands.

Using DSAPI it is possible to execute any command on any device (assuming the client has the necessary permission) in a TACO control system. Data is passed from the client to the device via the input and output parameters of the DSAPI.

Devices are organised into classes. Each class implements and understands a fixed set of commands. The list of commands for a device class is documented in the Device Server User's Guide (DSUG). The set of C functions which implement the DSAPI are archived in static or shared libraries for all platforms supported.

## 9.3 Getting Started

This section will take you through the steps of writing a simple application using DSAPI. Two versions of a simple “*Hello World*” in C for sending a string to a “*hello world*” device will be presented. The first version demonstrates using the DSAPI to execute commands synchronously while the second version demonstrates asynchronous command execution. The user will be taken through the stages of compiling, linking, debugging and running. The section will terminate with tips on common pitfalls encountered by DSAPI beginner’s (and even old-timer’s sometimes).

### 9.3.1 “*Hello World*” (synchronous) example

This example will take you through the steps of writing a simple program to send a “Hello World” string to a device synchronously.

#### *Step 1*

The first step is to find out which commands the device understands. If you don’t know them off by heart then get hold of the user guide (DSUG) for that device class and read it. The DSUG will list all commands implemented for the device and their input and output arguments.

The command we will use in this example is DevHello.

#### *Step 2*

The next step is to write the program. This assumes we know what we have to control and how.

In the case of this example we want a program which sends a string to a device and reads one back.

The program is written in C and uses a simple ascii interface to interact with the user. The program listing can be found below (cf. section “*Code Example*”).

ALL device access is done using DSAPI (of course). The main statements to note are :

- `#include <API.h>` - include file required by all clients (and servers). Necessary to prototype all DSAPI functions, and to define symbols and types. API.h will include other include as necessary.
- `devserver hs` - variable which will contain the device handle. Every device has to have a device handle. It is passed as input parameter to every DSAPI call. It contains all information necessary to communicate with the device on the network (network address, protocol, security etc.) as well as various bookkeeping information (device name). The device handle is initialised on the first successful call to `dev_import()` (cf. below).
- `dev_import()` - initialise the device handle. This call takes as input the device name and permission level requested. It checks the database to see if the device is defined and if so it asks the database for the device’s network address. Then it tries to contact the device server. All this information is stored in the device handle and returned to the user. If the device is not defined in the database or the user does not have the necessary permission to use the device `dev_import()` will return an error and the device handle will be NULL. The import is stateless this means the routine will not fail if the device server is not running. The 2nd parameter is used for security (this is discussed in the C library reference).

- `dev_putget()` - execute a command on the device. This call is the workhorse of DSAPI. It is used to execute a command on a device synchronously i.e. the client sends her request to the device and then waits for the command to be executed and for the answer to be returned before continuing. For the asynchronous version see below. The client has to specify the input and output arguments and their types. This information is normally obtained from the DSUG but can be constructed dynamically (using `dev_cmd_query()`). All parameters are passed as pointers. If the output arguments contain any pointers in them the client can choose to allocate space for the result himself or let DSAPI allocate space. (by setting the pointer to NULL) In the latter case it is up to the client to free the space allocated by DSAPI. The question of when to allocate and when to free is a tricky one and is treated in more detail in the section “*Common Pitfalls*”.
- `dev_free()` - free the device handle. This call will try to inform device server that the client is not connected to this device anymore. If this is the client's last network connection to the device server it will free the socket connection to the device server. Finally it will free the device handle structure allocated by `dev_import()`.

#### Step 3

The next step is to compile and link the client. This is different depending whether you are using a Unix-like (HP-UX, Solaris, Linux, VxWorks), OS9 or a Windows-NT system.

##### Unix and OS9

To compile under Unix and OS9 you have to tell the compiler where to find the DSAPI include files and which libraries to link with.

Assuming the your program is called `helloworld`, `$DSHOME` is an environment variable which points to the root directory of your TACO installation and `$OS` the operating system type (`s700` for HP-UX 9.x, `hpux10.2` for HP-UX 10.2, `solaris` for Solaris, `linux` for Linux, `vxworks` for VxWorks, `os9` for OS9) then simply type :

```
$CC $CFLAGS -I$DSHOME/include -L$DSHOME/lib/$OS -ldsapi -ldbapi -ldsxdr
helloworld.c -o helloworld.
```

`$CC` and `$CFLAGS` have to be positioned for each platform (refer to the example Makefile). **Windows-NT**

To compile under Visual C++ 4.2 you need to set the following options using the graphical interface :

*to be filled in ...*

#### Step 4

The final step is to run your program. Make sure you are in a shell interpreter (e.g. `bash`, `ksh`, `tcsh`, `csh` for Unix and MSDOS for Windows-NT) and simple type the name of the client program plus the name of the device i.e. `helloworld exp/hello/world`. If you forget to provide a device name the program will prompt you for one.

##### Example code - `helloworld.c`

```
static char RcsId[] = "@(#)Header: /segfs/dserver/doc/notes/DSN101/RCS/DSN101.tex,v 2.1 1997
/*****
```

```
File      : helloworld.c
```



Project : Device Server

Description: A simple test client to test using the synchronous device server API.

Author(s) : Andy Goetz

Original : November 1997

\$Revision: 2.1 \$

\$Date: 1997/11/13 14:16:40 \$

\$Author: goetz \$

\$Log: DSN101.tex,v \$

Revision 2.1 1997/11/13 14:16:40 goetz

first release of DSAPI V6

Revision 1.5 1997/11/13 14:13:31 goetz

totally reworked doc; added "Hello World" examples; asynchronous call; xdr types

\*-\*\*\*\*\*/

```
#include <Admin.h>
```

```
#include <API.h>
```

```
main(argc,argv)
```

```
unsigned int argc;
```

```
char **argv;
```

```
{
```

```
    devserver hw;
```

```
    long access = WRITE_ACCESS, error, status;
```

```
    char *ch_ptr,helloworld[256], dev_name[256];
```

```
    switch (argc)      {
```

```
        case 1:
```

```
            printf("enter device name [\"exp/hello/world\"]? ");
```

```
            if(NULL==gets(dev_name) || '\0'==dev_name[0])
```

```
                strcpy(dev_name,"exp/hello/world");
```

```
            break;
```

```
        case 2:
```

```
            strcpy(dev_name,argv[1]);
```

```
            break;
```

```
        default:
```

```
            printf("usage: helloworld [device name]\n");
```

```
            exit(1);
```

```
    }
```

```
    status = dev_import(dev_name,access,&hw,&error);
```

```
    printf("dev_import(%) returned %d\n",dev_name,status);
```

```

    if (status != 0)
    {
        printf("%s", dev_error_str(error));
        exit(1);
    }

    sprintf(helloworld, "Hello World");
    ch_ptr = NULL;

    status = dev_putget(hw, DevHello,
                        &helloworld, D_STRING_TYPE,
                        &ch_ptr, D_STRING_TYPE,
                        &error);
    printf("\nDevHello dev_putget() returned %d\n", status);

    if (status == 0)
    {
        printf("device answered : %s\n", ch_ptr);
        dev_xdrfree(D_STRING_TYPE, &ch_ptr, &error);
    }
    else
    {
        dev_printerror_no(SEND, NULL, error);
    }

    dev_free(hw, &error);
    exit(0);
}

```

### 9.3.2 “Hello World” (asynchronous) example

This example is a repeat of the above but using the asynchronous version of DSAPI. Asynchronism in this case means the client requests a command to be executed but does not wait for the server to respond. Instead it continues on to the next statement immediately. The request is put into the server’s buffer of incoming requests. After the server has executed the command it returns an acknowledge plus any output arguments to the client asynchronously. The reply is buffered in the clients queue of incoming replies. When the client is ready it polls its input queue to see if there are any replies pending (using the `dev_synch()` call).

Asynchronous command execution is more difficult to program than synchronous. However it is more efficient and is particularly useful for windowing programs and for programs which want to start multiple commands on multiple devices executing simultaneously and don’t want to wait for the command to finish execution.

This example is identical to the above example excepting for the fact that DevHello command is executed asynchronously. A callback function specified. This makes the code longer and more slightly more complicated to read.

#### *Step 1*

Understanding the device - same as Step 1 above.

*Step 2*

Writing the program - in principal same as Step 2 above however this time round use the asynchronous version of DSAPI.

The new calls are :

- **callbacks** - functions to be called when client receives a reply. Every reply received by the client has to be signalled to the client and unpacked. The callback functions serve this purpose. One callback functions has been implemented for this example - `hello_callback()`. The client can pass its own data with every asynchronous call which can be used to identify each reply during the callback (`user_data` parameter).
- `dev_putget_asyn()` - execute a command asynchronously on a device. As explained above the client does not wait for the server to accept the request for the reply. The input arguments are the same as for `dev_putget()` (synchronous) plus three additional arguments. The additional arguments specify the callback function (to be triggered during a call to `dev_synch()`), a pointer to user data and an asynchronous id (returned by `dev_putget_asyn()`).
- `dev_synch()` - check to see if any asynchronous replies have been received. If so they are unpacked and the corresponding callback is triggered. `dev_synch()` takes as input the amount of time it should wait for pending replies before continuing.

*Step 3*

Compiling and linking - same as Step 3 above for Unix and OS9. The asynchronous calls are part of the standard library.

Not support under Windows-NT (yet).

*Step 4*

Running - same as Step 4 above for Unix and OS9.

Not supported under Windows-NT (yet).

**Example code - helloworld\_asyn.c**

```
static char RcsId[] = "@(#)Header: /segfs/dserver/doc/notes/DSN101/RCS/DSN101.tex,v 2.1 1
/*****
```

```
File      : helloworld_asyn.c
```

```
Project   : Asynchronous Device Server's
```

```
Description: A simple test client to test using the asynchronous
              device server API using callbacks.
```

```
Author(s) : Andy Goetz
```

```
Original  : January 1997
```

```
$Revision: 2.1 $
```

```
$Date: 1997/11/13 14:16:40 $
```

```
$Author: goetz $
```

```

$Log: DSN101.tex,v $
Revision 2.1 1997/11/13 14:16:40 goetz
first release of DSAPI V6

Revision 1.5 1997/11/13 14:13:31 goetz
totally reworked doc; added "Hello World" examples; asynchronous call; xdr types

*-*****-/

#include <API.h>
#include <DevStates.h>

/*****
Function      :   void hello_callback()

Description:   callback function to be called asynchronously after executing
               the DevHello commands

*****-/

void hello_callback(ds, user_data, cb_data)
devserver ds;
void *user_data;
DevCallbackData cb_data;
{
    long error;

    printf("hello_callback(%s): called with asynch_id=%d, status=%d (error=%d) user data
ds->device_name,cb_data.asynch_id, cb_data.status, cb_data.error, (char*)user_data);
    printf("hello_callback(%s): time executed by server = {%d s,%d us}\n",
           ds->device_name,cb_data.time.tv_sec,cb_data.time.tv_usec);

    if (cb_data.status == DS_OK)
    {
        printf("hello_callback(%s): device answered=%s\n",
               ds->device_name,*(DevString*)cb_data.argout);
        dev_xdrfree(D_STRING_TYPE, &cb_data.argout, &error);
    }
    else
    {
        dev_printererror_no(SEND,NULL,cb_data.error);
    }

    return;
}

/*****
Function      :   main()

Description:   main function to test asynchronous DSAPI.

*****-/

```

```

main(argc,argv)
unsigned int argc;
char **argv;
{

    devserver hw;
    long access = WRITE_ACCESS, error, status;
    char ch_ptr, helloworld[256], dev_name[256];
    struct timeval timeout_25s = {25,0};
    long asynch_id;
    char *user_data="my data";

    switch (argc)        {
        case 1:
            printf("enter device name [\"exp/hello/world\"]? ");
            if(NULL==gets(dev_name) || '\0'==dev_name[0])
                strcpy(dev_name,"exp/hello/world");
            break;
        case 2:
            strcpy(dev_name,argv[1]);
            break;

        default:
            printf("usage: helloworld_asyn [device name]\n");
            exit(1);
    }

    imported = dev_import(dev_name,access,&hw,&error);

    printf("dev_import(%s) returned %d\n",dev_name,imported);

    if (imported != 0)
    {
        printf("%s",dev_error_str(error));
        exit(1);
    }

    sprintf(helloworld, "Hello World");
    ch_ptr = NULL;

    status = dev_putget_asyn(hw,DevHello,
                            &helloworld,D_STRING_TYPE,
                            &ch_ptr,D_STRING_TYPE,
                            (DevCallbackFunction*)void_callback,
                            (void*)user_data, &asynch_id,
                            &error);

    printf("\nDevHello dev_putget_asynch(%d) returned %d\n",asynch_id, status);
    if (status < 0) dev_printerror_no(SEND,NULL,error);

    /*
     * wait for answer from client (waits for a max of 25 s)

```

```

*/
    status = dev_synch(&timeout_25s, &error);

    dev_free(hw,&error);
    exit(0);
}

```

### 9.3.3 Common Pitfalls

Using an API is easy once you know how. For beginner's this is not the case. This section will list the common pitfalls encountered by beginner's (and old-timers too!) when they start using DSAPI.

### 9.3.4 Nethost

Every TACO control system is managed by a NETHOST. The NETHOST is the name of the host where the TACO Manager has been started. It is referred to as the *facility* in the device name. The Manager is the entry point for all TACO clients and servers.

A common error when starting an application (e.g. `helloworld`) is to forget to specify the NETHOST environment variable.

In this case you will get an error similar to this :

```
Thu Nov  6 13:56:42 1997  environmental variable NETHOST not defined
```

The solution is to set the environment variable to the name of a host where a TACO control system Manager is running e.g. “`setenv NETHOST libra`” for `csh` or “`export NETHOST=libra`” for `ksh` or `bash`.

An alternative to specifying the NETHOST environment variable is to qualify the device name with the *facility* field which is the same as the NETHOST e.g. `//libra/exp/hello/world`.

If the Manager is not running you will get the following error :

```
Thu Nov  6 14:03:26 1997  no network manager available
```

If you don't know which host is your NETHOST then ask your TCO system administrator/guru. If you are supposed to be the guru then start the Manager. If you don't know how then send an email to the TACO help-line `taco@esrf.fr`

### 9.3.5 Shared Libraries

Another common error is not finding the DSAPI shared libraries.

If your application dies with the following message :

```
./helloworld: can't load library 'libdsapi.so'
```

You must add the DSAPI library directory for your platform to the shared library path searched by your system.

For Solaris and Linux use :

```
set $LD_LIBRARY_PATH:$DSHOME/lib/$OS for csh and tcsh,
```

```
export $LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$DSHOME/lib/$OS for ksh and bash.
```

For HP-UX use :

```
set $SHLIB_PATH:$DSHOME/lib/$OS for csh and tcsh,
```

```
export $SHLIB_PATH=$SHLIB_PATH:$DSHOME/lib/$OS for ksh and bash.
```

Where `$DSHOME` is an environment variable pointing to the TACO home directory and `$OS` the operating system flavour.

Shared libraries are not supported on OS9 and Windows/NT (yet).

### 9.3.6 Makefiles

Although the compile+link instructions listed above can be typed every time you want to recompile+relink it is much more efficient to write a makefile with the necessary instructions.

The TACO makefiles are multi-platform and make use of the conditional statements supported by GNU make (also known as gmake). gmake supports statements of the kind `ifdef $(symbol)`, `else` and `endif`. Most TACO conditional makefiles use the same symbols. These are :

- `_hp9000s700` - for HPPA 1.0 systems running HP-UX 9.x
- `_hpux10` - for HPPA 1.0 systems running HP-UX 10.2
- `_solaris` - for Solaris
- `linux` - for Linux
- `vw68k` - for Motorola 68k systems running VxWorks
- `vw86` - for Intel x86 systems running VxWorks
- `_UCC` - for OS9 systems using the Ultra C and C++ compiler
- `unix` - for HP-UX, Solaris, Linux and VxWorks platforms

A simple example Makefile for the helloworld program could look like this :

```
#
#
# Makefile for helloworld - a simple DSAPI client
#
#
# TACO home directory
#
DSHOME = $(LOCAL_DSHOME)
#
# library home directory - platform dependant
#
ifdef __hpux10
LIBHOME = $(DSHOME)/lib/hpux10.2
endif # __hpux10
ifdef _solaris
LIBHOME = $(DSHOME)/lib/solaris
endif # _solaris
ifdef linux
LIBHOME = $(DSHOME)/lib/linux
endif # linux
ifdef _UCC
LIBHOME = $(DSHOME)/lib/os9
endif # _UCC
ifdef vw68k
LIBHOME = $(DSHOME)/lib/vw68k
```

```

endif # vw68k
ifdef vwx86
LIBHOME = $(DSHOME)/lib/vwx86
endif # vwx86
#
# include files home directory
#
INCLDIRS = -I$(DSHOME)/include \
           -I$(DSHOME)/include/private
#
# compiler flags - platform dependant
#
ifdef __hpux10
CC = /bin/cc
CFLAGS = -Aa -g -DEBUG -Dunix -D_HPUX_SOURCE -D__hpux10 -DBSD=199704 \
        -c $(INCLDIRS)
endif # __hpux10
ifdef _solaris
CC = /opt/SUNWspro/SC4.0/bin/cc
CFLAGS = -Xa -g -Dsolaris -DEBUG -c $(INCLDIRS)
endif # _solaris
ifdef linux
CC = gcc
CFLAGS = $(INCLDIRS) -Dlinux -Dunix -ansi -DEBUG -g -c
endif # linux
ifdef _UCC
CC = xcc
CFLAGS = -mode=c89 -g -D EBUG -to osk -tp 020 -x il -e as=. $(INCLDIRS)
endif # _UCC
ifdef vw68k
CC = cc68k
CFLAGS = -Dvxworks -Dunix -DCPU=MC68020 -ansi -m68030 \
        -msoft-float -DEBUG -e $(INCLDIRS) -g
endif # vw68k
ifdef vwx86
CC = cc386
CFLAGS = -v -c -Dvxworks -Dunix -DCPU=I80386 -ansi \
        -DEBUG $(INCLDIRS) -g
endif # vwx86
#
# library flags
#
ifdef __hpux10
LFLAGS = -L$(LIBHOME) -ldsapi -ldsxdr -ldbapi -lm
endif # __hpux10
ifdef _solaris
LFLAGS = -L$(LIBHOME) -ldsapi -ldsxdr -ldbapi -lnsl -lsocket -lm
endif # _solaris
ifdef linux
LFLAGS = -L$(LIBHOME) -ldsapi -ldsxdr -ldbapi -lm
endif # linux
ifdef _UCC
LFLAGS = -L$(LIBHOME) -l dsapi -l dsxdr -l dbapi -l rpclib -l netdb_small \
        -l socklib.1 -l sys_clib.1 -l unix.1

```



```

endif # _UCC
#
#-----main-target-to-make-----
#
all : helloworld

helloworld : helloworld.c
$(CC) $(CFLAGS) helloworld.c -o helloworld $(LFLAGS)

```

*NOTE: don't forget to start all rules with a tabulation mark !*

Although even this simple example looks complicated keeping all platform dependencies in one file can prove to be a time saver when developing on multiple platforms.

### 9.3.7 Memory Allocation

Probably the trickiest part for beginners to DSAPI is memory allocation. DSAPI uses the memory allocation of the XDR library of the ONC-RPC. The difficulties come from the fact that all procedure calls are to remote servers and pointers to memory areas have to be copied to the (remote) server and vice versa.

The rules for memory allocation in DSAPI can be summarised as follows :

1. arguments are either outgoing (input) or incoming (output) from the client to the server,
2. all input and output arguments are passed via pointers,
3. memory for input arguments have to allocated by the client (of course !),
4. memory for pointers in output arguments can be allocated either by the client or by the DSAPI (actually the XDR layer),
5. if memory in output arguments is to be allocated by DSAPI then initialise pointers in output arguments to NULL,
6. if pointers to memory in output arguments are NOT initialised to NULL DSAPI assumes the client has allocated the necessary memory and will try to use it (with catastrophic consequences if this is not the case !),
7. any memory allocated by DSAPI has to be freed by the client using `dev_xdrfree()`.
8. in order to avoid nasty bugs or strange core dumps therefore clients MUST initialise all incoming pointers to NULL or to locally allocated memory.

If you understand the above rules and follow them you should not have any problems. The problems come from not understanding and following these rules. The XDR types supported by DSAPI are covered in the section on “XDR Types”.

To illustrate the above rules here are some examples :

- **simple C types**

```

devserver ps;
long status, error;
float readvalue;
.
.
.
status = dev_putget(ps, DevReadCurrent, NULL, D_VOID_TYPE,

```

```

        &readvalue, D_FLOAT_TYPE, &error);
printf("current %6.3f\n", readvalue);
.
.
.

```

This is a simple example of using a simple C type to receive output from the server. Simply pass the pointer to the simple type to DSAPI.

*NOTE : DSAPI cannot allocated memory for simple types because it expects a pointer to a value and not a pointer to a pointer to a value and it therefore has no way of distinguishing between a pointer to the value ZERO and a pointer to NULL (if you know what I mean ...)*

- **output arguments - memory allocated by client**

```

devserver ps;
long status, error, i;
float readvalues[MAX_READVALUES];
DevFloatVarArray float_vararr;
.
.
.
float_vararr.length = MAX_READVALUES;
float_vararr.sequence = readvalues;
status = dev_putget(ps, DevReadAll, NULL, D_VOID_TYPE,
        &float_vararr, D_VAR_FLOATARR, &error);
printf("read %d value\n\n", float_vararr.sequence);
for (i=0; i<float_vararr.sequence; i++)
{
    printf(" current[%d] %6.3f\n", i, readvalues[i]);
}
.
.
.

```

In this example the client receives a variable length array of floats. The client has allocated memory for the array of floats itself. It is the responsibility of the client to ensure that sufficient memory is allocated for the return arguments and that the server does not send more values than the client expects.

- **output arguments - memory allocated by DSAPI**

```

devserver ps;
long status, error, i;
DevFloatVarArray float_vararr;
.
.
.
float_vararr.length = 0;
float_vararr.sequence = NULL;
status = dev_putget(ps, DevReadAll, NULL, D_VOID_TYPE,
        &float_vararr, D_VAR_FLOATARR, &error);
printf("read %d value\n\n", float_vararr.sequence);
for (i=0; i<float_vararr.sequence; i++)

```

```

{
    printf(" current[%d] %6.3f\n", i, float_vararr.sequence[i]);
}
dev_xdrfree(D_VAR_FLOATARR, &float_vararr, &error);
.
.
.

```

In this example the client sets the `sequence` to `NULL` and lets DSAPI allocate memory for the output arguments. The client has to free the allocated memory.

### 9.3.8 Advanced Features

Before leaving the “*Getting Started*” section we would like to mention some advanced features of the DSAPI which are very useful.

### 9.3.9 Timeouts

The DSAPI is managed by timeouts. Both synchronous and asynchronous calls have a timeout. A client will receive a timeout error (`DevErr_RPCTimedOut`) if the server has not sent an answer within the timeout period.

The default timeout for synchronous calls is **3 seconds**. The default timeout for asynchronous calls is **25 seconds**.

The client can modify the timeout per device using the `dev_rpc_timeout()` call (cf. the C library reference). This can be necessary if the request is known to take longer than the default timeout to execute.

If a client gets lots of timeouts there could be a network problem i.e. lots of network traffic. This can be fixed by simply changing from UDP to TCP protocol (see next section).

### 9.3.10 Protocol

The DSAPI is based on the ONC-RPC and makes use of UDP and TCP (the two main IP protocols). The difference between the two protocols is :

- UDP is a connectionless unreliable protocol. UDP has the advantage that it does not require a dedicated file descriptor per client-server connection and it is (sometimes) faster than TCP. It has the disadvantage that it does not retry if a request fails and it is limited in maximum packet size to 8 kilobytes. All device imports are done using UDP. UDP is the default protocol
- TCP is a connection-oriented reliable protocol. It has the advantage that it is reliable i.e. it will retry if a request fails to be acknowledge, and can transfer unlimited packet sizes (in reality limited by the receiving computer to a few megabytes). It has the disadvantage that it requires a file descriptor per client-server connection and it is a more complicated protocol to implement.

To change from UDP to TCP or vice-versa use the `dev_rpc_protocol()` call (cf. below).

## 9.4 C Library

Below you will find all the DSAPI calls in the C library in alphabetical order.

### 9.4.1 Synchronous Client API

These calls are used by DSAPI clients to send a synchronous request to a device server. The notion of client-server refers to sender and receiver of each DSAPI call. This means a device server itself can become a DSAPI client if it accesses a device.

#### **dev\_cmd\_query()**

```
typedef struct {
    u_int          length;
    DevCmdInfo     *sequence;
} DevVarCmdArray;

typedef struct {
    long    cmd;           /* command */
    char    cmd_name [20]; /* command name as ASCII string */
    char    *in_name;      /* description of input arguments */
    long    in_type;       /* type of input arguments */
    char    *out_name;     /* description of output arguments */
    long    out_type;      /* type of output arguments */
} DevCmdInfo;

long dev_cmd_query (ds, varcmdarr, error)
    devserver    ds;           /* client handle */
    DevVarCmdArray *varcmdarr; /* results of query */
    long          *error;      /* error */
```

Dev\_cmd\_query() returns a sequence of DevCmdInfo structures containing all available commands, their names, their input and output data types, and type descriptions for one device. Commands and data types are read from the command list in the device server. Command names are read from the CMDS table of the resource data base. Data type descriptions for input and output arguments for a command function have to be specified in the resource database in the CLASS table as:

```
CLASS/class_name/cmd_name/IN_TYPE: "Current in mA"
CLASS/class_name/cmd_name/OUT_TYPE: "Power in MW"

class_name : Name of the device class. Retrieved from
             the device server.
cmd_name   : Name of the command. Retrieved from the
             CMDS table in the resource data base.
```

#### **dev\_free()**

```
long dev_free (ds,error)
    devserver    ds;           /* client handle */
    long          *error;      /* error */
```

Dev\_free() closes the connection to a device associated with the passed client handle.

**dev\_import()**

```

long dev_import (dev_name,access,ds_ptr,error)
    DevString      dev_name;          /* device name */
    long           access;             /* requested access level */
    devserver      *ds_ptr;           /* returned pointer to the client
                                      handle */
    long           *error;             /* error */

```

Opens a connection to a device and returns a client handle for the connection. Dev\_import can distinguish between local and remote devices.

If the control system is running with security on then the **access** parameter determines what level of access permission the client wants on the device. The following levels are supported :

1. READ\_ACCESS for read-only access
2. WRITE\_ACCESS for read and write access (**default**)
3. SI\_WRITE\_ACCESS for single user write access
4. SU\_ACCESS for super-user access
5. SI\_SU\_ACCESS for single user super-user access
6. ADMIN\_ACCESS for administrator access

The default access is WRITE\_ACCESS and corresponds to access=0. If the TACO control system is running with security the client has to have the necessary permission in the security database for the (UID,GID,HOST,NETWORK) quadrupole.

For more information on security refer to “Access Control and Security for the ESRF Control System” by J.Meyer (DSN/102).

**dev\_inform()**

```

typedef struct {
    char device_name[80];
    char device_class[32];
    char device_type[32];
    char server_name[80];
    char server_host[32];
} DevInfo;

long dev_inform (clnt_handles, num_devices, dev_info, error)
    devserver      *clnt_handles;      /* list of client handles */
    long           num_devices;         /* number of client handles */
    DevInfo        **dev_info;         /* returned list of
                                      information structures */
    long           *error;              /* error */

```

Dev\_Inform() returns to the user a structure containing device information for every specified device client handle. The information structure contains:

- the name of the device,
- the class name,
- the device type,

- the device server name,
- the host name of the device server

The returned information structures are allocated by `dev_inform()` with `malloc(3C)`. The can be freed by using `free(3C)`.

### `dev_put()`

```
long dev_put (ds,cmd,argin,argin_type,error)
    devserver    ds;           /* client handle */
    long         cmd;          /* command */
    DevArgument   argin;       /* pointer to input arguments */
    DevType       argin_type;   /* type of input arguments */
    long         *error;       /* error */
```

`Dev_put()` executes a command on the device associated with the passed client handle, without returning any output data. The device might be remote or local. Input data types must correspond to the types specified for this command in the device server's command list. Otherwise an error code will be returned. The output data type in the device server's command list must be set to `D_VOID_TYPE`. All arguments have to be passed as pointers.

### `dev_put_async()`

```
long dev_put_async (ds,cmd,argin,argin_type,error)
    devserver    ds;           /* client handle */
    long         cmd;          /* command */
    DevArgument   argin;       /* pointer to input arguments */
    DevType       argin_type;   /* type of input arguments */
    long         *error;       /* error */
```

The function `dev_put_async()` is similar to `dev_put()`. The only difference is, that `dev_put_async()` sends a request to execute a command to a device server and returns immediately when the command was received. The only errors which can be returned by `dev_put_async()` are errors during the sending of the command. A correct return status only indicates that the command execution was started.

**No failures during command execution can be reported back to the client.**

### `dev_putget()`

```
long dev_putget (ds,cmd,argin,argin_type,argout,argout_type,error)
    devserver    ds;           /* client handle */
    long         cmd;          /* command */
    DevArgument   argin;       /* pointer to input arguments */
    DevType       argin_type;   /* type of input arguments */
    DevArgument   argout;      /* pointer to output arguments */
    DevType       argout_type;  /* type of output arguments */
    long         *error;       /* error */
```

`Dev_putget()` executes a command synchronously on the device associated with the passed client handle. The device might be remote or local. Input and output data types must correspond to the types specified for this command in the device server's command list. Otherwise an error code will be returned. All arguments have to be passed as pointers.

Memory for outgoing arguments will be automatically allocated by XDR, if pointers are initialised to `NULL`. To free the memory allocated by XDR afterwards, the function `dev_xdrfree()` must be used.

**dev\_putget\_raw()**

```

typedef struct {
    u_int    length;
    char     *sequence;
} DevOpaque;

long dev_putget_raw (ds, cmd, argin, argin_type, argout, argout_type, error)
    devserver    ds;           /* client handle */
    long         cmd;          /* command */
    DevArgument   argin;       /* pointer to input arguments */
    DevType       argin_type;  /* type of input arguments */
    DevOpaque     *argout;     /* pointer to opaque data */
    DevType       argout_type; /* type of output arguments,
                                returned by the command */
    long         *error;       /* error */

```

`Dev_putget_raw()` executes a command on the device associated with the passed client handle and returns the outgoing arguments as a block of opaque data in XDR format. All arguments have to be passed as pointers. Memory for the opaque block will be allocated by the RPC if the sequence pointer is initialised to NULL. The allocated memory can be freed with `dev_xdrfree()` and the type identifier `D_OPAQUE_TYPE`.

**dev\_rpc\_protocol()**

```

long dev_rpc_protocol (ds, protocol, error)
    devserver    ds;           /* client handle */
    int          protocol;     /* transport protocol */
    long         *error;       /* error */

```

By calling `dev_rpc_protocol()` with one of the two defined protocol parameters `D_UDP` and `D_TCP` (`API.h`), the transport protocol for an open RPC connection will be set to the chosen protocol. Before switching the protocol, an RPC connection to a device server has to be opened by a `dev_import()` call.

All devices implemented in the same server and imported by the client use the same RPC connection. Changing the protocol of a RPC connection with `dev_rpc_protocol` means changing the protocol for all devices of the same server.

- `D_UDP`  
UDP protocol with maximal 8kbyte data transfer.
- `D_TCP`  
TCP protocol. TCP point to point connection with no transfer limitations.

**dev\_rpc\_timeout()**

```

long dev_rpc_timeout (ds, request, dev_timeout, error)
    devserver    ds;           /* client handle */
    int          request;      /* CLSET_TIMEOUT or CLGET_TIMEOUT */
    struct timeval *dev_timeout; /* timeout value */
    long         *error;       /* error */

```

Sets or reads the timeout for a RPC connection with UDP protocol. A request to set the timeout has to be asked with `CLSET_TIMEOUT` as request parameter and the timeout specified by the timeval structure `dev_timeout`. The timeout will

be set without any retry. A request to read the timeout has to be asked with `CLGET_TIMEOUT`, and the current timeout will be returned in `dev_timeout`. All devices implemented in the same server and imported by the client use the same RPC connection. Changing the timeout of a RPC connection with `dev_rpc_timeout` means changing the timeout value for all devices of the same server.

#### **dev\_xdrfree()**

```
long dev_xdrfree (type, objptr, error)
    DevType      type;           /* type of arguments */
    DevArgument   objptr;        /* pointer to arguments */
    long          *error;        /* error */
```

`Dev_xdrfree` frees the memory for device server data allocated by XDR. An example for the use of `dev_xdrfree()` is the freeing of a `D_VAR_FLOATARR` data type. Using `dev_xdrfree()` you don't have to care about the length of the internal sequence of float values. Just pass a pointer to a `D_VAR_FLOATARR` structure and the allocated memory for the sequence will be freed, according to the length specified in the structure.

### **9.4.2 ASynchronous Client API**

These calls are used by DSAPI clients to send and receive asynchronous requests to a device server. The notion of client-server refers to sender and receiver of each DSAPI call. This means a device server itself can become a DSAPI client if it accesses a device.

#### **dev\_asynch\_timeout**

```
long dev_asynch_timeout ( devserver ds, long request,
                          struct timeval *tout, long *error)
```

Call to set/get the timeout for an asynchronous call to the device `ds`. Get/Set operation is determined by `request = CLSET_TIMEOUT` or `CLGET_TIMEOUT`. The timeout is returned/specified in `tout`. If an error occurs the call returns `DS_NOTOK` and an appropriate error code in `error`.

#### **dev\_pending**

```
long dev_pending ( devserver ds)
```

Call to return the number of asynchronous requests still pending replies for device `ds`. If `ds = NULL` then return the total number of pending calls.

#### **dev\_putget\_asyn()**

```
struct _DevCallbackData {
    long asynch_id;           /* id of asynchronous call */
    DevArgument argout;       /* pointer to output argument */
    DevType argout_type;      /* argout type */
    long status;              /* status of command execution */
    long error;               /* error code after command execution */
    struct timeval time;      /* time at server when command was executed */
} DevCallbackData;
```

```
void callback (devserver ds, void *user_data, DevCallbackData cb_data);
```



```

long dev_putget_asyn (ds,cmd,argin,argin_type,argout,argout_type,
                    callback, user_data, asynch_id, error)
    devserver      ds;                /* client handle */
    long           cmd;                /* command */
    DevArgument    argin;              /* pointer to input arguments */
    DevType        argin_type;         /* type of input arguments */
    DevArgument    argout;             /* pointer to output arguments */
    DevType        argout_type;        /* type of output arguments */
    DevCallbackFunction *callback;     /* pointer to callback function */
    void           *user_data;         /* pointer to user data to pass to callback */
    long           *asynch_id;         /* asynchronous id returned by call */
    long           *error;             /* error */

```

`Dev_putget_asyn()` executes a command asynchronously on the device associated with the passed client handle. The device must be remote and compiled with V6. Input and output data types must correspond to the types specified for this command in the device server's command list. Otherwise an error code will be returned. All arguments have to be passed as pointers.

Memory for outgoing arguments will be automatically allocated by XDR, if pointers are initialised to **NULL**. To free the memory allocated by XDR afterwards, the function `dev_xdrfree()` must be used.

The client continues immediately and does not wait for the server to execute the request. The callback function has to be specified otherwise an error will be returned. The callback function is triggered by making a call to `dev_synch()`. The client can pass data to the callback function via `user_data`. The callback function receives the device server handle, user data and a `DevCallbackData` structure as input. The function returns a (unique) id in `asynch_id` for each call.

### `dev_synch()`

```

long dev_synch (struct timeval *timeout, long *error);

```

This calls checks to see if any asynchronous replies are pending. If so it triggers the associated callback routines. The call will wait for a maximum of `timeout` time before returning if no replies are received otherwise it returns immediately after unpacking all received replies. A timeout of zero means check to see if any replies are pending otherwise returning immediately.

## 9.4.3 Server

### `dev_cmd()`

```

long dev_cmd (ds, cmd, argin, argin_type, argout, argout_type, error)
    DevServer      ds;                /* object pointer */
    long           cmd;                /* command */
    DevArgument    argin;              /* pointer to input arguments */
    long           argin_type;         /* type of input arguments */
    DevArgument    argout;             /* pointer to output arguments */
    long           argout_type;        /* type of output arguments */
    long           *error;             /* error */

```

`Dev_cmd` executes a command on a given object locally in a device server. Memory freeing must be done with `free()` and not with `dev_xdrfree()`.

With the extended functionality of `dev_putget` and `dev_put` the function should be used only to access objects which are not exported.

To access internal exported devices the unified interface must be used, to avoid access and security problems in the coming releases.

#### **ds\_\_create()**

```
long ds__create (name, ds_class, ds_ptr, error)
    char          *name;           /* device name */
    DevServerClass ds_class;       /* class of the object */
    DevServer      *ds_ptr;        /* returned pointer to the object */
    long           *error;         /* error */
```

Ds\_\_create() creates a new device server object of the class ds\_class and will return a pointer on the object. Before creating the object (DevMethodCreate : obj\_create(3x)) the class and all its superclasses are checked to see if they have been initialised. If not, then the DevMethodClassInitialise (class\_init(3x)) is called for each uninitialised class.

#### **ds\_\_destroy()**

```
long ds__destroy (ds, error)
    DevServer      ds;             /* object pointer */
    long           *error;         /* error */
```

Ds\_\_destroy() searches for a destroy method (**DevMethodDestroy**) in the object class. If no destroy method is implemented in the object class, its superclasses are searched. Arriving at the end of the class tree, the destroy method of the general device server class will be executed.

The general destroy method will free the object correctly only, if no memory allocation was done for object fields outside the **DevServerPart** structure of the object. The device name, as a field of DevServerPart will be freed correctly by the general device server class destroy method.

Also exported objects can be destroyed. They will be deleted from the list of exported devices and all client accesses will be stopped.

#### **dev\_export()**

```
long dev_export (name, ds, error)
    char          *name;           /* device name */
    DevServer      ds;             /* object pointer */
    long           *error;         /* error */
```

Dev\_export makes devices visible for device server clients. All necessary connection information for a dev\_import() call will be stored in a database table. Moreover the exported devices are added to the device server's global list of exported devices. Dev\_export is installed as a method in the DeviceServerClass and accessible by the name **DevMethodDevExport**.

#### **ds\_\_method\_finder()**

```
DevMethodFunction ds__method_finder (ds, method)
    DevServer ds;                 /* */
    DevMethod method;             /* */
```

Ds\_\_method\_finder() searches for a method in the class hierarchy of the object ds and returns a pointer to the method function. If the method was not found in the

object's class, the search continues in all its superclasses up to the general device server class.

If the method is not implemented the method finder takes **DRASTIC** action and exits. This has been included in the specification to guarantee that on returning from the method finder the method can be directly executed.

#### **ds\_\_method\_search()**

```
long ds__method_search (ds_class, method, function_ptr)
    DevServerClass      ds_class;          /* class pointer */
    DevMethod           method;           /* method to search for */
    DevMethodFunction    *function_ptr;    /* returned pointer to the
                                           method function */
```

Ds\_\_method\_search() searches for a method in the class specified. It returns the pointer to the method function if the requested method was found in the class. If no such method was specified the status DS\_NOTOK is returned.

#### **ds\_\_svcrun()**

```
long ds__svcrun (error)
    long *error;          /* error */
```

Ds\_\_svcrun() supports the checking of pending RPC requests to the device server on all open sockets. If requests are available on file descriptors (sockets), the next pending request for every descriptor will be executed and ds\_\_svcrun() will return afterwards. If no commands are pending on any descriptor ds\_\_svcrun() should return after 10ms.

### **9.4.4 General Purpose Functions**

#### **dev\_\_printererror\_no()**

```
void dev__printererror_no (mode, comment, dev_errno)
    DevShort      mode;          /* indicates, how to handle the
                                error message buffer*/
    char          *comment;      /* comment on error */
    long          dev_errno;     /* error */
```

If a message service is imported, all error messages are sent to an error file, on the NETHOST, called :

```
      NETHOST:/DSHOME/api/error/hostname_program-number
NETHOST      = device server system host.
DSHOME       = device server system directory on NETHOST.
hostname     = name of the host where the service is installed.
prog_number  = program number of the registered service.
```

If no message service is imported, all error messages are sent to **stderr** and printed on the terminal.

The **mode** parameter indicates, how to handle the error message buffer. Single messages can only be 256 characters long. To printout longer messages, short strings can be buffered and printed later as a text.

- **WRITE**: Writes error message to buffer.
- **SEND**: Adds the last error message to the buffer, sends the buffer contents to an output device and clears the buffer.

- CLEAR: Clears the message buffer from all stored messages.

**dev\_error\_str()**

```
char *dev_error_str (dev_errno)
    long            dev_errno;      /* error */
```

Dev\_error\_str() returns the error string for a given error number. It first checks to see if the error is negative. If so it returns an standard error message (negative errors are not supported). Then it checks if the error is one of the kernel errors (e.g. NETHOST not defined, RPC timeout etc.) and returns a corresponding error message. Then it checks to see if a dynamic error message was returned by the last dev\_put\_get(), dev\_put() or dev\_putget\_asyn() call, if so it returns this error message. If none of the above are true it searches the TACO database for the (static) error string. If an appropriate error string cannot be found in the data base, dev\_error\_str() returns a string, indicating the failure. **dev\_error\_str() allocates memory for the returned error string everytime using malloc(), it is the client's responsibility to free this memory using free()**<sup>1</sup>.

**dev\_error\_push()**

```
void dev_error_push (char *error_string);
```

Dev\_error\_push is a server side call for generating dynamic error strings. If called by the server while executing a dev\_putget() it will make a copy of the error string and transmit it back to the client. The client can recover the error string by calling dev\_error\_str() immediately after the return of the dev\_putget() call in question. Note if a new call to dev\_putget() is made the error string returned by the previous call(s) is lost. Dev\_error\_push() can be called multiple times to stack errors if necessary e.g. to return errors from multiple nested calls.

Dev\_error\_push() is available only from DSAPI version V8.18 and onwards.

**dev\_printdebug()**

```
void dev_printdebug (debug_bits, fmt, [a0], [a1], ....)
    long    debug_bits;    /* debug flags */
    char    *fmt;          /* A printf(3S) like format string */
    double  a0, a1, ...;   /* variables to be printed */
```

Dev\_printdebug sends the debug information if the specified **debug\_bits** are set. Possible debug\_bits (debug flags) are:

#define DBG_TRACE	0x1
#define DBG_ERROR	0x2
#define DBG_INTERRUPT	0x4
#define DBG_TIME	0x8
#define DBG_WAIT	0x10
#define DBG_EXCEPT	0x20
#define DBG_SYNC	0x40
#define DBG_HARDWARE	0x80
#define DBG_STARTUP	0x100
#define DBG_DEV_SVR_CLASS	0x200
#define DBG_API	0x400

---

<sup>1</sup>this is a common source of memory leaks in TACO clients

```
#define DBG_COMMANDS      0x800
#define DBG_METHODS      0x1000
#define DBG_STARTUP      0x100
#define DBG_DEV_SVR_CLASS 0x200
#define DBG_API           0x400
#define DBG_COMMANDS      0x800
#define DBG_METHODS      0x1000
#define DBG_SEC           0x2000
#define DBG_ASYNC        0x4000
```

If a message service is imported, debug messages are sent to a named pipe, on the NETHOST, called :

```
      NETHOST:/DSHOME/api/pipe/hostname_program-number
NETHOST      = device server system host.
DSHOME       = device server system directory on NETHOST.
hostname     = name of the host where the service is installed.
prog_number  = program number of the registered service.
```

If no message service is imported, debug messages are sent to **stdout** and printed on the terminal.

## 9.5 XDR types

All DSAPI types are implemented as XDR types. In order to prevent having to implement too many XDR types (a problem for generic programs e.g. `xdevmenu`) a set of kernel types has been defined.<sup>2</sup> Servers should use ONLY these types as input and output arguments.

### 9.5.1 Kernel Types

The DSAPI kernel XDR types are described below. They include all simple C types, variable length arrays of simple C types and a few DSAPI specific types. Each type is characterised by a defined symbol (needed by `dev_putget()` and `dev_xdrfree()`), a C type and an XDR routine.

### 9.5.2 Simple C Types

The following simple C types are implemented as part of the DSAPI kernel :

1. `D_VOID_TYPE`  

```
typedef void DevVoid
```
2. `D_CHAR_TYPE`  

```
typedef char DevChar
```
3. `D_BOOLEAN_TYPE`  

```
typedef char DevBoolean
```
4. `D_USHORT_TYPE`  

```
typedef u_short DevUShort
```
5. `D_SHORT_TYPE`  

```
typedef short DevShort
```
6. `D_USLONG_TYPE`  

```
typedef u_long DevULong
```
7. `D_LONG_TYPE`  

```
typedef long DevLong
```
8. `D_FLOAT_TYPE`  

```
typedef float DevFloat
```
9. `D_DOUBLE_TYPE`  

```
typedef double DevDouble
```
10. `D_STRING_TYPE`  

```
typedef char* DevString
```

---

<sup>2</sup>in the past new types were added by device server programmer's as they needed them; this led to a proliferation of exotic types which was difficult to maintain and which needed to be implemented by clients

### 9.5.3 Combinations of Simple Types

A number of combinations of simple C types are supported as part of the DSAPI kernel types :

1. `D_INT_FLOAT_TYPE`

```
typedef struct {
    long state;
    float value;
} DevIntFloat;
```

2. `D_FLOAT_READPOINT`

```
typedef struct {
    float set;
    float read;
} DevFloatReadPoint;
```

3. `D_STATE_FLOAT_READPOINT`

```
typedef struct {
    short state;
    float set;
    float;
} DevStateFloatReadPoint;
```

4. `D_LONG_READPOINT`

```
typedef struct {
    long set;
    long read;
} DevLongReadPoint;
```

5. `D_DOUBLE_READPOINT`

```
typedef struct {
    double set;
    double read;
} DevDoubleReadPoint;
```

### 9.5.4 Variable Length Arrays

The second major set of XDR types implemented by DSAPI are the so-called variable length arrays. Variable length arrays are arrays which have a length field specifying the number of elements in the array. They are described by a C structure consisting of two fields - an unsigned integer length field and a sequence field. The sequence is a pointer to an array of elements of the required type. The C definition is of variable length arrays is :

```
struct { u_int length; <Type> *sequence} Dev<Type>VarArr;
```

where `<Type>` is the required type.

The following variable length arrays are implemented as part of the DSAPI kernel types :

## 1. D\_VAR\_CHARARR

```
typedef struct {
    u_int length;
    char *sequence;
} DevVarCharArray;
```

## 2. D\_VAR\_STRINGARR

```
typedef struct {
    u_int length;
    DevString *sequence;
} DevVarStringArray;
```

## 3. D\_VAR\_USHORTARR

```
typedef struct {
    u_int length;
    u_short *sequence;
} DevVarUShortArray;
```

## 4. D\_VAR\_SHORTARR

```
typedef struct {
    u_int length;
    short *sequence;
} DevVarShortArray;
```

## 5. D\_VAR\_ULONGARR

```
typedef struct {
    u_int length;
    u_long *sequence;
} DevVarULongArray;
```

## 6. D\_VAR\_LONGARR

```
typedef struct {
    u_int length;
    long *sequence;
} DevVarLongArray;
```

## 7. D\_VAR\_FLOATARR

```
typedef struct {
    u_int length;
    float *sequence;
} DevVarFloatArray;
```

## 8. D\_VAR\_DOUBLEARR

```
typedef struct {
    u_int length;
    double *sequence;
} DevVarDoubleArray;
```



## 9. D\_VAR\_FRPARR

```
typedef struct {
    u_int length;
    DevFloatReadPoint *sequence;
} DevVarFloatReadPointArray;
```

## 10. D\_VAR\_SFRPARR

```
typedef struct {
    u_int length;
    DevStateFloatReadPoint *sequence;
} DevVarStateFloatReadPointArray;
```

## 11. D\_VAR\_LRPARR

```
typedef struct {
    u_int length;
    DevLongReadPoint *sequence;
} DevVarLongReadPointArray;
```

### 9.5.5 Exotic Types

All other XDR types which are supported by the DSAPI are considered as “*exotic*” types and the programmer must refer to the relevant Device Server User Guide and/or xdr include files. In the future device server programmer’s are urged to stick to the kernel types and where possible provide equivalent functions for old classes which use standard kernel types (e.g. using command overloading).

## 9.6 Changes

### 9.6.1 Version 8.0

Version 8 introduces support for TANGO. TANGO<sup>3</sup> is the new version of TACO based on CORBA (instead of RPC) and with support for C++ and Java. The TANGO interface allows TACO clients to do a `dev_putget()` call on a TANGO device in a transparent manner - simply add "tango:" in front of the device name to switch protocol from RPC to CORBA. To use the TACO-TANGO interface link your C or C++ program with the C++ linker and the `libdsapi++` library (or `libdsapig++` if you are using GNU).

### 9.6.2 Version 7.0

Version 7 introduces events. Events use the same mechanism as the asynchronous call for dispatching. They allow servers to be programmed to generate true asynchronous events to clients.

### 9.6.3 Version 6.0

The main changes in the new version are the inclusion of true asynchronous `dev_putget()` calls - `dev_putget_asyn()` and related calls (cf. "*C library reference*" above).

### 9.6.4 Version 5.1

The main changes to this version were the following - support for multi-nethost, ports to Windows (95 and NT), Linux and VxWorks.

### 9.6.5 Version 4.1

The main changes here were security was implemented, and port to Ultra C for OS9.

### 9.6.6 Version 3.37

#### An Asynchronous `dev_put()`

The new function `dev_put_asyn()` is similar to the ancient `dev_put()`. The only difference is, that `dev_put_asyn()` sends a request to execute a command to a device server and returns immediately when the command was received. The only errors which can be returned by `dev_put_asyn()` are errors during the sending of the command. A correct return status only indicates that the command execution was started. **No failures during command execution can be reported back to the client.**

```
long dev_put_asyn (ds, cmd, argin, argin_type, error)
    devserver      ds;                /* client handle to the device */
    long           cmd;                /* command to execute */
    DevArgument     argin;             /* pointer to input arguments */
    DevType         argin_type;        /* input argument data type */
    long           *error;             /* error */
```

---

<sup>3</sup>cf. <http://www.esrf.fr/tango>

### Destroying Objects

With the function `ds__destroy()` a proper interface was created to destroy objects in a device server. `Ds__destroy()` searches for a destroy method (**DevMethodDestroy**) in the object class. If no destroy method is implemented in the object class, its superclasses are searched. Arriving at the end of the class tree, the destroy method of the general device server class will be executed.

The general destroy method will free the object correctly only, if no memory allocation was done for object fields outside the **DevServerPart** structure of the object. The device name, as a field of **DevServerPart** will be freed correctly by the general device server class destroy method.

Also exported objects can be destroyed. They will be deleted from the list of exported devices and all client accesses will be stopped.

```
long ds__destroy (ds, error)
    DevServer      ds;           /* Pointer to the object */
    long           *error;       /* error */
```

Attention:

To destroy an exported object, `ds__destroy()` must be used. Executing only the destroy method will not delete the device from the list of exported devices. With the next client access a nice core will be generated.

### Accessing Process Internal Devices

Until version 3.37 the only possibility to access devices internally was the function `dev_cmd()`. That was not enough to handle the coming security features. Out of this reason the functionality of the functions:

```
dev_import()
dev_putget()
dev_put()
dev_free()
```

was enlarged. They can be used now on all exported devices, remote via RPCs or internally just via function calls. `Dev_import()` will detect automatically whether a device is internal and will avoid all overhang of the remote access on the client handle. Also memory treatment was unified. All outgoing arguments (remote or intern) are allocated by XDR. `Dev_xdrfree()` must be used to free the memory.

Attention:

This unified interface for device access works on all **exported** devices. Objects which are not exported, can be accessed only by `dev_cmd()`.

To access process internal devices the unified interface must be used to avoid access and security problems in the coming releases.

### Dynamic Memory Allocation

The general structures handling exported devices and client connections to the devices

```
typedef struct _DevServerSec {
    long      security_key;
    long      access_right;
    long      single_user_flag;
} DevServerSec;
```

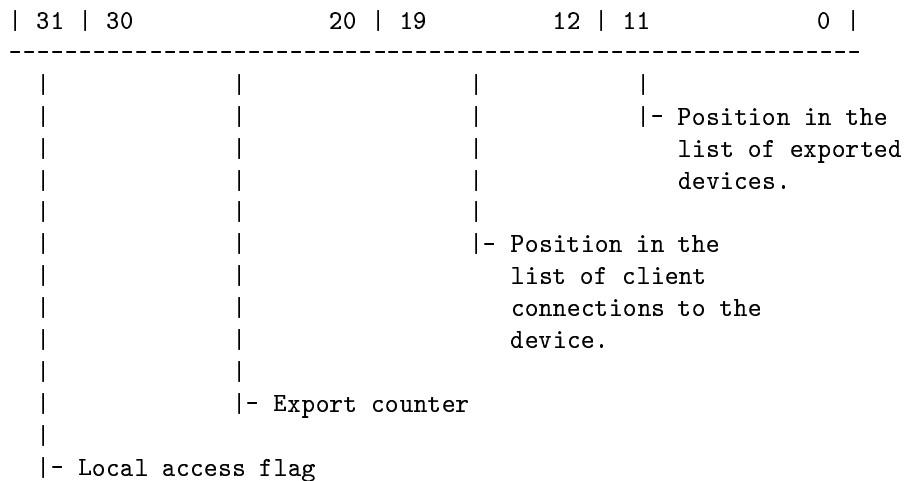
```
typedef struct _DevServerAccess {
    DevServer      ds;
    char           export_name[80];
    long           export_status;
    long           export_counter;
    long           single_user_flag;
    long           max_no_of_clients;
    DevServerSec   *client_access;
} DevServerDevices;
```

```
DevServerDevices      *devices /* Exported devices; in DevServer.c */
```

are no longer static arrays. The are allocated dynamically in data blocks. The BLOCK\_SIZE is defined in ApiP.h and set to 5 structures per data block. To avoid the growth of a device server, all client connections should be freed correctly.

### The Device ID

Every device in a server is referenced by a device ID. The ID is send with every client call to identify the device and is hidden to the user in the client handle to the device. Up to version 3.37 the device ID was a simple number. Indicating the position of the device in the list of exported devices. Now the device ID was split up into several information fields.



The **export counter** field becomes interesting only if you destroy an exported object and reexport another or the same object again. In the case of a destroyed object, the export counter is increased and all client connections on the old value are no longer valid. A newly exported device might take the place in the list of exported devices afterwards.

The **Local access flag** is set if the dev\_import() detects a local device.

The split up of the device ID limits a device server to the following values:

```
Maximum number of exported devices      = 4096
Maximum number of client connections per device = 256
```



## Chapter 10

# Database guide - ndbm

*by E. Taurel*

### 10.1 Introduction

The TACO static database is used to keep three kinds of information about device servers:

1. Device server configuration data called **resources**.
2. Device and pseudo device information (location, type...).
3. Security data.

Resources are used to configure device server without recompilation. Device information allows application software to build network connections with devices through the device server API. Pseudo device information allow easier debugging session. Security data are used by the device server API to check if a device request is authorized. The database is filled up with a graphical interface called **greta** or with the contents of **resource file**. A C library allows software to get/store data from/into this database. A large set of utilities allows a simple management of this database.

The database itself is the ndbm package which is part of the UNIX operating system. It is a file oriented database.

TACO is a distributed control system. This is also true for the static database. The C library get/store data from/into the database through a database server across the network with RPC's. This is hidden to the user and implemented in the C library functions.

### 10.2 Device and resource definition

#### 10.2.1 The devices list

Within a TACO control system, every device must have a name build with the following syntax:

**DOMAIN/FAMILY/MEMBER**

For example, the first attenuator device name on the ESRF beam line behind insertion device 12 must be ID12/att/1 because the device domain is ID12, the device

family is att and the member is 1. **A device name must be unique in a TACO control system.**

To identify every device server instance, a device server is started with a **personal name** which is different for each instance. For example, a device server for PerkinElmer vacuum pump called Perkin will be started with the personal name ID16 when it will drive pump installed on ESRF beam line ID16 and will be started with the personal name ID11 when it will drive pumps on the ESRF beam line ID11. The device list must be entered with the following format :

```
device server process name/personal name/device:  device names list
```

device is a key word allowing the software to know that it is a device list. Example:

```
BlValves/ID10/device:          ID10/rv/1, ID10/rv/2 \
                                ID10/rv/3
```

In this case, the device server process name is *BlValves*, the personal name is *ID10* and it drives three devices. The device server must be started on the command line as *BlValves ID10*.

In the device list, each device name must be separated by a comma. If the list continue on the next line, use the `\` character at the end of the line. All devices driven by the same device server must be defined in only one device list.

A device name must not have more than 23 characters with a family and member name limited to 19 characters. A device server process name is limited to 23 characters and the personal name to 11 characters.

### 10.2.2 Resource definition

A resource is defined with the following syntax:

```
device name/resource name:    resource value
```

Example

```
sy/ps-b/1/fbus_channel:      2
sy/ps-b/1/upper_limit:      456.5
sy/ps-b/1/fbus_desc:        fb0
sy/ps-b/1/error_str:        "G64 crate out of order"
sy/ps-b/1/linear_coeff:      8.123, 9.18, 10.78 \
                             7.32, 101.78, 27.2
```

Resource name must not exceed 23 characters. Resource value are stored in the database as ASCII characters and converted to the requested type when they are returned to the caller. The available types are :

- D\_BOOLEAN\_TYPE
- D\_SHORT\_TYPE
- D\_LONG\_TYPE
- D\_FLOAT\_TYPE
- D\_DOUBLE\_TYPE
- D\_STRING\_TYPE
- D\_VAR\_CHARARR
- D\_VAR\_SHORTARR

- D\_VAR\_LONGARR
- D\_VAR\_FLOATARR
- D\_VAR\_STRINGARR

For the D\_BOOLEAN\_TYPE, a resource value can be set in the resource file to 0, 1, False, True, Off, On. It is possible to define resources which are arrays (resource linear\_coeff in the previous example). In this case, each array element are separated by the , character. To continue the array on the next line, use the \ character at the end of the line. It is also possible to give a resource value as a hexadecimal number if the resource value begins with the 0x characters (C syntax) and if it is converted to a numerical type. If the resource is a string with spaces, the string must be enclosed with the " characters.

It is also possible to define resources for non physical devices and to use them to configure any software. A resource definition can look like

```
class/tutu/titi/tata:          "When will we eat?"
```

and be retrieved by a C program. In this case, the second and third fields length is limited to 19 characters.

To delete resources from a resource file, init the resource value with the character %.

```
ID10/att/1/upper_limit:      %
```

will erase the resource upper\_limit for the device ID10/att/1 from the database.

### 10.2.3 Domain names and NDBM files

The domain name is the device or resource name first field. In a TACO control system, domain names are free. Nevertheless, data for each domain are stored in two different files and the database server needs to know all the domain names involved in a control system. This is done by the **DBTABLES** environment variable. This variable is a list (comma separated) of all the domain used in the control system.. It is recommended to have the **CLASS**, **CMDS**, **ERROR**, **SYS** and **SEC** domains to get all the device server features running correctly. A **NAMES** and **PS\_NAMES** pseudo domain names are automatically added to the list of the user defined domain names.

The SEC domain is reserved for the security aspect of the device server model. All the update, insert, delete from this domain are protected by a password.

The SYS domain is a generic domain for resources and devices which are part of the beam line control system itself (data collector resources...)

The CMDS and ERROR domain are used to store error messages and commands strings.

Files used by the NDBM software to keep data (two files per domain) are stored in a directory pointed to by the DBM\_DIR environment variable software also needed by the database server.

## 10.3 Greta

Greta (**G**raphical **R**esource **E**ditor for **T**AcO) is the graphical interface to the TACO static database. This tool allows the user to retrieve, add, delete or update resources, to add, delete update device list for a device server, to save/load data to/from a file, to get device, server or database informations. For greta, all the informations stored into the database are splitted into three parts which are :



1. The device list : All the entities defined as served by a device server
2. The server list : List of all device server defined in the database
3. The resource list : All the resources defined in the database including resources which don't belong to any device

### 10.3.1 The device window

To open a device window, click on File-Open device. A database device browsing window is popped-up. Once a device is selected (by double click on the field name or by pressing the filter button), pressing the open button or a double click on the Member field will popped-up a device window.

The Informations part of the device window contains device information like device server host, device server PID, device class... This sub-window is not editable. The Resources sub-window displays all the resources defined for the selected device and is editable. It is possible to update, delete, add device resource(s) in this sub-window. The five window main buttons are :

- **Update** to update the database with the contents of the above sub-window. A confirmation window is popped-up
- **Cancel** to close the window without any database change
- **Delete** to delete the device from database. A window is popped up in order to give the user the choice to delete device with or without its resources.
- **Ping** to ping the device. The device answers to such request only if the device server is linked with DSAPI release 5.11 and above.
- **(Re)start** to start or restart the device server in charge of the selected device. This feature is available only for device served by a device server linked with database software release 5.0 and above and also if the "starter" device server release 2.0 or above is running on the host where the device server is running. If it is not the case, an alarm window is popped-up. In all cases, a confirmation window is popped up.

Under the window File button, it is possible to :

- Print window content
- Save window content to a file
- Close the window

Under the Edit button, the user will find the classical edit features plus the "insert device resource" button. If some device resources are device name, by selecting this device name and clicking in Edit-insert device resource, all the resources belonging to the newly selected device will be added at the bottom of the Resources sub-window. This feature is also possible by a click on the right mouse button when the device name is selected.

It is possible to open up to 10 different device windows. The device name is displayed in the window title.



Figure 10.1: Greta device window

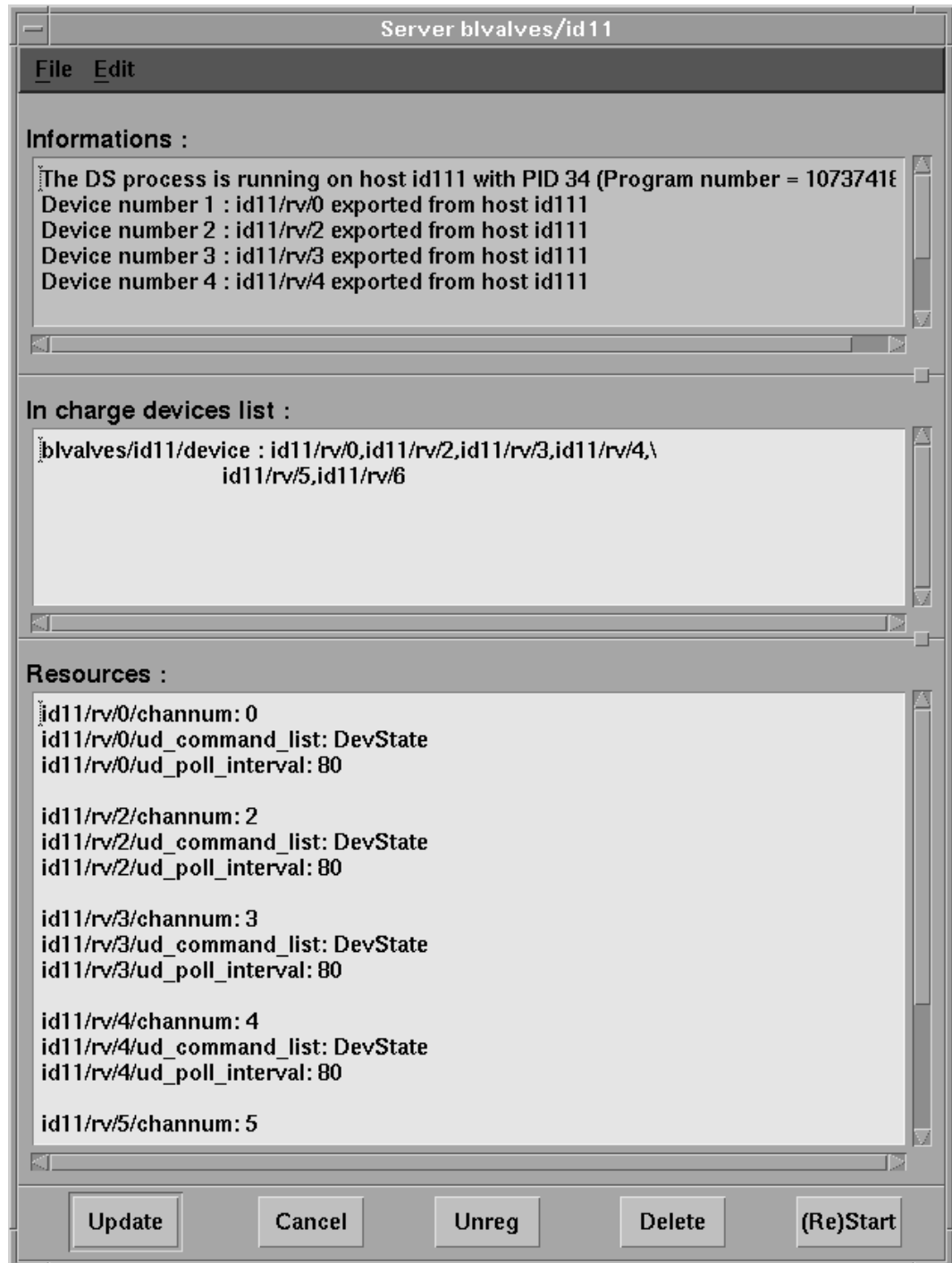


Figure 10.2: Greta server window

### 10.3.2 The server window

To open a server window, click on File–Open server. A database server browsing window is popped-up. Once a server is selected (by double click on the field name or by pressing the filter button), pressing the open button or a double click on the Personal name field will popped-up a server window.

The Informations part of the device window contains server informations like devices number defined for this server, device name... This sub-window is not editable. The "In charge device list" sub-window displays the list of device(s) defined for this server. This list follows the syntax described in the device list chapter. This sub-window is editable and the device list can be modified. The Resources sub-window displays all the resources belonging to each server device and is editable. It is possible to update, delete, add device resource(s) in this sub-window. The five window main buttons are :

- **Update** to update the database with the contents of the two editable sub-windows. A confirmation window is popped-up
- **Cancel** to close the server window without any database change
- **Unreg** to unregister the server from the database. To unregister a server from the database means to mark all its devices as non-exported (unable to answer to network request). A confirmation window is popped-up.
- **Delete** to delete the server from database. A window is popped up in order to give the user the choice to delete the server with or without all its devices resources.
- **(Re)start** to restart the device server. This feature is available only for device server linked with database software release 5.0 and above and also if the "starter" device server release 2.0 or above is running on the host where the selected device server is running. If it is not the case, an alarm window is popped-up. In all cases, a confirmation window is popped up.

Under the window File button, it is possible to :

- Print window content
- Save window content to a file
- Close the window

Under the Edit button, the user will find the classical edit features plus the "insert device resource" button. If some device resources are device name, by selecting this device name and clicking in Edit–insert device resource, all the resources belonging to the newly selected device will be added at the bottom of the Resources sub-window. This feature is also possible by a click on the right mouse button when the device name is selected.

It is possible to open up to 10 different server windows. The server name is displayed in the window title.

### 10.3.3 The resource window

To open a resource window, click on File–Open resources. A database resource browsing window is popped-up. Once a resource is selected (by double click on the field name or by pressing the filter button), pressing the open button or a double click on the Name field will popped-up a server window. It is always proposed by greta to use the wildcard \* as Member and/or Name field.

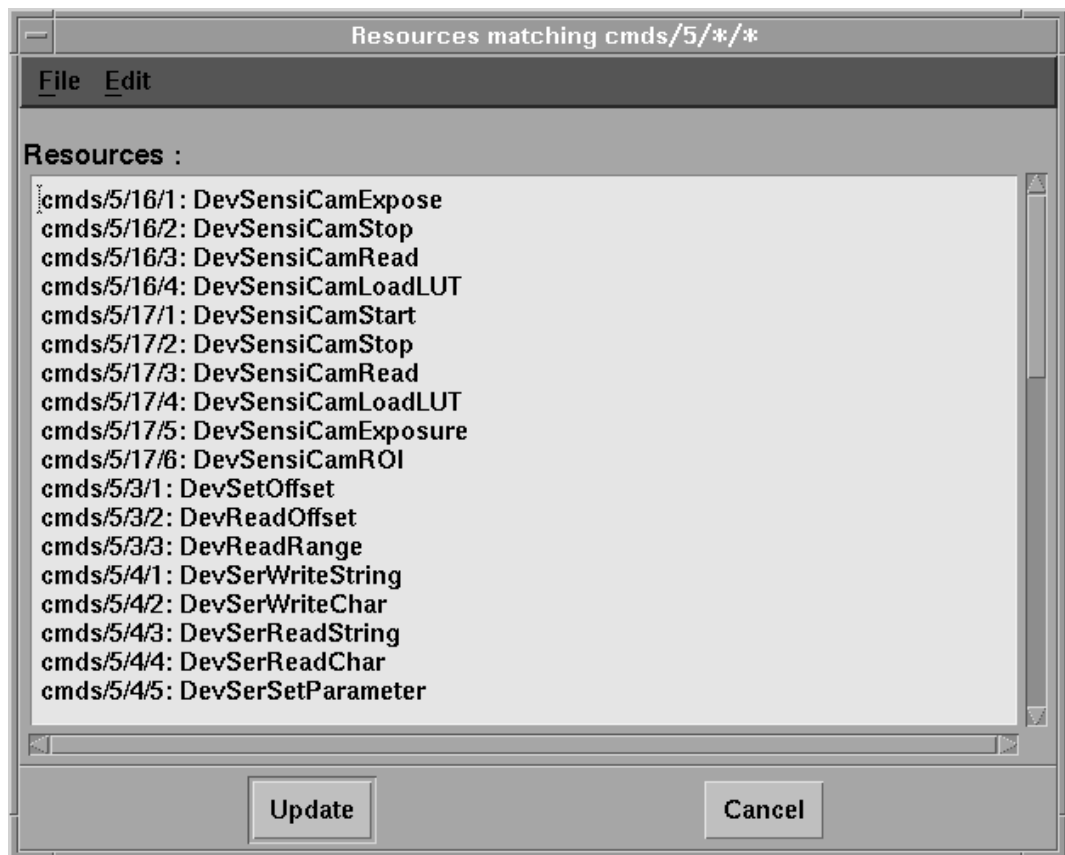


Figure 10.3: Greta resource window

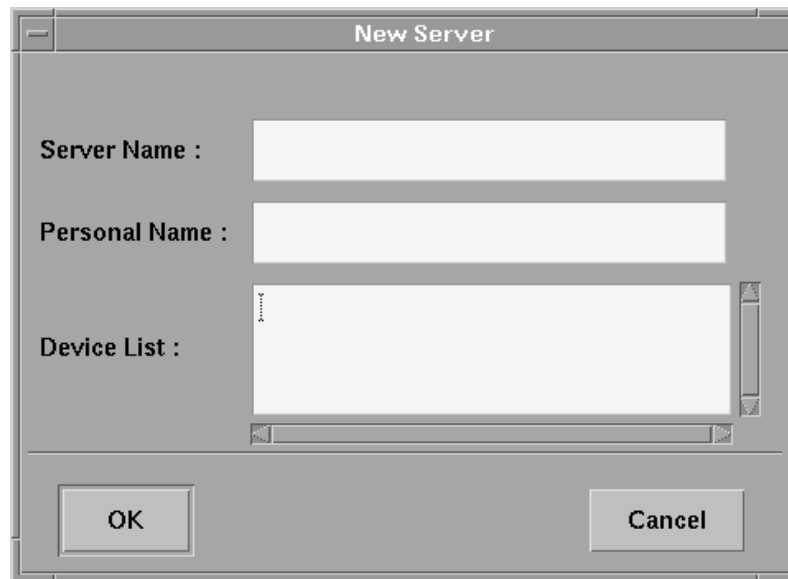


Figure 10.4: Greta new server window

The Resources sub-window displays all the resources selected. This sub-window is editable. It is possible to update, delete, add device resource(s) in this sub-window. The two window main buttons are :

- **Update** to update the database with the contents of the above sub-window.  
A confirmation window is popped-up
- **Cancel** to close the window without any database change

Under the window File button, it is possible to :

- Print window content
- Save window content to a file
- Close the window

Under the Edit button, the user will find the classical edit features.  
It is possible to open up to 10 different resources windows.

#### 10.3.4 The new server window

The new server window allows a user to create new device server within the database. This window is popped-up after a click on File-New server. The user must fill in the server name field with the device server name and the personal name field with the argument used to start the device server. The device list must also be filled in as described in the device list chapter of this documentation. When these three fields are filled in, clicking on OK will register the server in the database. To define server device(s) resources, open a server window as explained earlier.

#### 10.3.5 The load file window

Once a file has been selected in the file selection window, the file contents is displayed in a separate window. This window is not editable. The two window main buttons are :

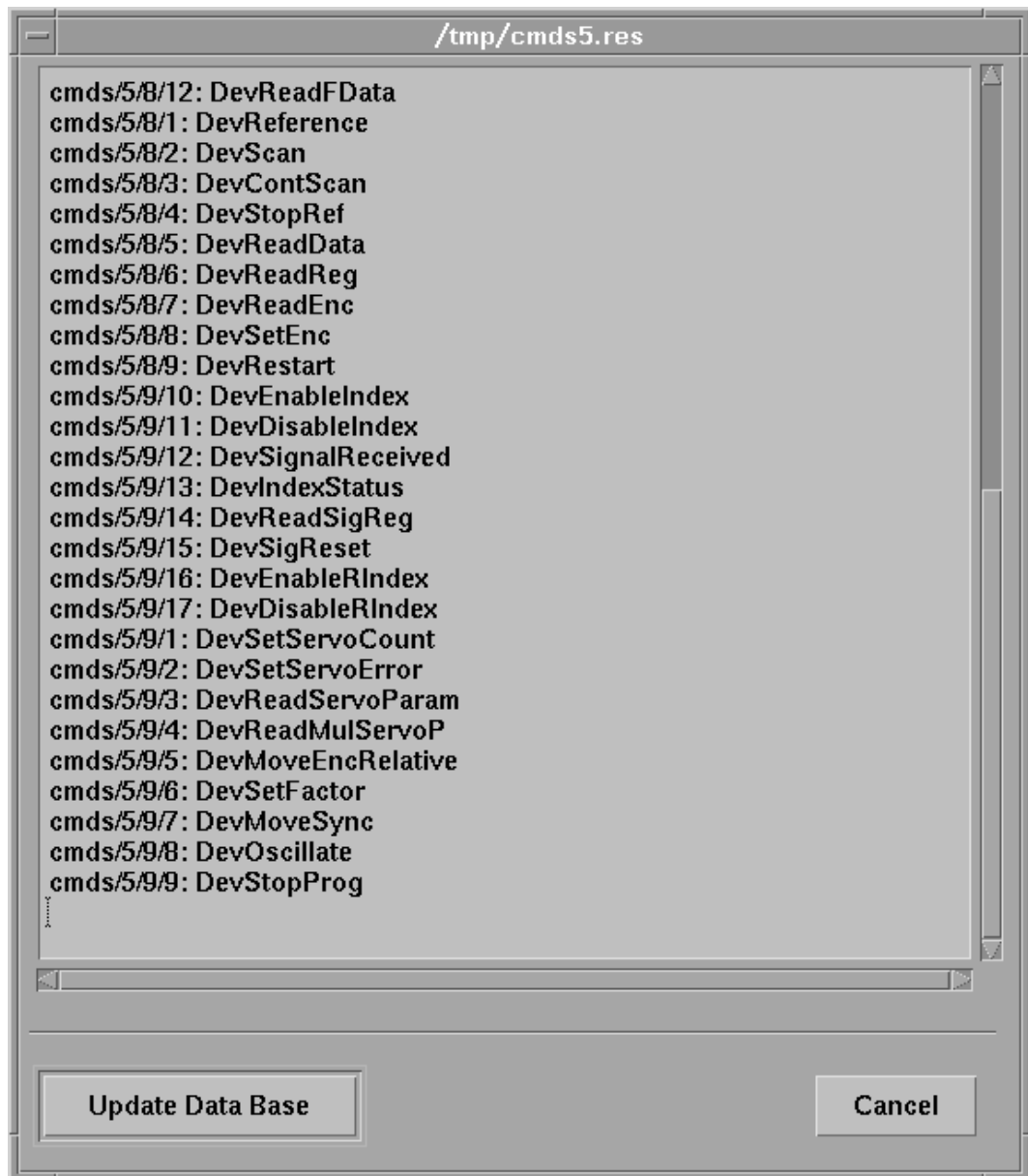


Figure 10.5: Greta file window

- **Update** to update the database with the contents of the above sub-window.
- **Cancel** to close the window without any database change

### 10.3.6 The Option menu

Four options are implemented. These options are :

- **Server displayed with class resources.** This option deals only with server window. When this option is chosen, class resources are also displayed in the server window Resources part. Class resources are all the resources with the following syntax :
  - class/server\_name/\*/\*
  - class/device\_class/\*/\*
- **Display all embedded server in a process.** This option is usefull when several device servers are embedded in one process. If such a process is selected in the server selection window, device list and device resources for all the server embedded in the process will be displayed in the server window.
- **Display device data collector info.** If this option is set, a forth part is added to the device window. This sub-window (not editable) is entitled "DC/HDB informations". It displays data related to the device and the TACO data collector. If the device is registered in the data collector, the command used for polling is displayed as well as the time needed to execute the last command. The polling period is also displayed and the time spent since the last command result update. Some informations about the poller process in charge of the device are also displayed (host where the poller is running, its PID...)
- **Display device history database info.** If this option is set, a forth part is added to the device window. This sub-window is entitled "DC/HDB informations". It displays the storage mode chosen to store device data into HDB (History DataBase) and the last nine records value with their record dates.

If the last two options are selected, DC and HDB informations are displayed in the same sub-window of the device window.

### 10.3.7 Other features

Some miscellaneous features are also incorporated into greta.

**Global-Information :** Display in greta main window general database informations. These informations are the number of devices defined in the database, the number of exported devices for each device's domain, the pseudo-devices number and the number of resources for each domain.

**Help-On version :** Display a window with the greta software release number

**File-Print :** Print the greta main window

**File-Exit :** Exit the application

## 10.4 Resource file

A resource file is the way to store resource and device information into the static database. The user writes its resource file and updates the database with one of the database utilities called **db\_update**. Then a C program (a device server or any other C program) is able to retrieve these resources with a library call and in the



case of a device server, it is also able to mark its devices as exported to the rest of the world (ready to accept requests).

A resource file is divided in two parts which are:

- The list of devices driven by a particular instance of a device server. The same device server can run on several computers. This list allows the system to know that the this particular instance of the device server drive this list of devices.
- Resources definition

A resource file must have a suffix **.res**. Any line beginning with the **#** character will be considered as a comment line. It is not allowed to begin comment at the middle of a line. Blank lines are allowed. All the resource files must be stored in directory and sub-directories under a defined path which is known to the static database utilities by the **RES\_BASE\_DIR** environment variable. On most of the ESRF beam line control system, the resource files base directory is **dserver login directory/dbase/res**. For test purpose, another resource database is running on **margaux**.

## 10.5 Utilities

These utilities are commands run from the UNIX command line. They can be grouped in three different parts which are:

- Database administration commands
- Database user commands
- Security commands

These utilities are briefly describe below. Man pages are available to get complete information.

## 10.6 Database administration commands

### 10.6.1 db\_fillup

**db\_fillup** <data\_source>

This command creates the database into memory and load it with resource files contents or with a database backup file according to the **data\_source** parameter. This command directly access the **ndbm** files (not via the server) and therefore needs the **DBM\_DIR** and **DBTABLES** environment variables. To hide these environment variables, this command is a little script which set these environment variable and then, call the real command with the argument given by the user. The setting of these environment variables is done by a file called **dbm.env**. Example :

```
db_fillup 0
```

### 10.6.2 db\_info

**db\_info**

This command displays the total number of devices and resources defined in the database as well as the number of devices and resources for each domain. Example :

```
$db_info
```

#### DEVICE STATISTICS

```
90 devices are defined in database
84 of the defined devices are actually exported:
  0 for the CLASS domain
  6 for the SYS domain
  0 for the ERROR domain
  0 for the CMDS domain
  0 for the SEC domain
  78 for the ID16 domain
12 pseudo devices are defined in database
```

#### RESOURCE STATISTICS

```
4126 resources are defined in database:
  42 resources for the CLASS domain
  28 resources for the SYS domain
  348 resources for the ERROR domain
  651 resources for the CMDS domain
  0 resources for the SEC domain
  3057 resources for the ID16 domain
```

### 10.6.3 db\_read

```
db_read <domain name>
```

This function displays all the data recorded in the database for a specific domain. This command directly access the ndbm files (not via the server) and therefore needs the DBM\_DIR and DBTABLES environment variables. To hide these environment variables, this command is a little script which set these environment variable and then, call the real command with the argument given by the user. The setting of these environment variables is done by a file called **dbm\_env**. Example :

```
$db_read class
CLASS: relayserver|id16|unitttype|1|: icv196
CLASS: dc|1|host|1|: inel1
CLASS: dc|1|max_call|1|: 1000
CLASS: dc|1|36_default|1|: inel1
CLASS: dc|inel1|dev_number|1|: 100
CLASS: dc|inel1|cellar_number|1|: 50
CLASS: dc|inel1|path|1|: /users/b/dserver/system
CLASS: dc|inel1|login|1|: dserver
CLASS: dc|server_nb|inel1_rd|1|: 2
```

## 10.7 Database user commands

### 10.7.1 db\_update

```
db_update <file>
```

This command allows a user to load into the database all the resources and devices list defined a resource file. It will insert new resources or update already existing ones. It will also updates or insert device information. Example :

```
db_update FluoScreen_ID16.res
```

### 10.7.2 db\_devres

`db_devres <device_name>`

`db_devres` displays all the resources belonging to a device. Example :

```
$ db_devres id16/att/1
block1 : ID16/att1_b/1
number_of_blocks : 3
block3 : ID16/att1_b/3
unitnumber : 1
block2 : ID16/att1_b/2
fluorscreen : NO
attenuatornum : 1
```

### 10.7.3 db\_devinfo

`db_devinfo <device_name>`

`db_devinfo` displays device (or pseudo device) information. For device, these information are the host name where the device server in charge of the device is running, the device server process identifier and the device server name. For pseudo device, it is just the PID and the host of the process which created the pseudo device. Example (for a real device) :

```
$ db_devinfo id16/att/1
Device id16/att/1 belongs to class : attenuatorClass
It is monitored by the server : attenuator/id16 version 1
The device server process name is : attenuator
This process is running on the computer : id161 with process ID : 117
```

Example (for a pseudo device) :

```
$ db_devinfo id16/bidon/1
Device id16/bidon/1 is a pseudo device
It is created by a process with PID : 234 running on host : inel1
```

### 10.7.4 db\_servinfo

`db_servinfo <full device server name>`

This command displays the device list for a specific device server. The device server is specified by its full device server name which is the device server process name/personal name. For device server with several embedded classes, device belonging to each class will be displayed. Example :

```
$ db_servinfo attenuator/id16
Device number 1 : id16/att/1 exported from host id161
The device server is part of the process : attenuator with PID : 45
```

### 10.7.5 db\_devdel

`db_devdel [-r] <device_name>`

This command delete a device (or a pseudo device) and all its resources from the database. The `-r` option prevents the command to also remove all the device resources. Example :

```
$ db_devdel id12/att/1
```

### 10.7.6 db\_resdel

`db_resdel <device name/resource name>`

This command deletes a resource from the database. Example :

```
$ db_resdel fe/id/10/io_word
```

### 10.7.7 db\_servdel

`db_servdel [-r] <full device server name>`

This command deletes all the device(s) belonging to a device server from the database. It also deletes all the resources belonging to these devices. The -r option prevents the command to delete resources. Example :

```
$ db_servdel attenuator/id16
```

### 10.7.8 db\_servunreg

`db_servunreg <full device server name>`

This command unregisters all the device(s) belonging to a device server from the database. After this command, all the devices are not exported anymore. Example :

```
$ db_servunreg attenuator/id16
```

## 10.8 Security commands

### 10.8.1 dbm\_sec\_passwd

`dbm_sec_passwd`

It is possible to protect security data (in the SEC domain) with a password. This password will be asked for each insert/update into the SEC domain. `dbm_sec_passwd` is the command which allows to define or change the password.

### 10.8.2 dbm\_sec\_objinfo

`dbm_sec_objinfo <obj_name>`

`dbm_sec_objinfo` displays security data for a given object. A object can be a domain, a family or a device.

### 10.8.3 dbm\_sec\_userinfo

`dbm_sec_userinfo [-u user_name] [-g group_name]`

`sec_userinfo` returns all accesses specified for a user and (or) for a group.

## 10.9 The C library

A C library with 39 calls has been written which allows a C program to

- retrieve, update, insert, delete resources.
- retrieve device list, mark device as exported, return device information.
- retrieve all or part of the exported devices.
- register and unregister pseudo devices
- browse the database
- retrieve command code from command name

These calls are briefly described here. Man pages are available for all of them to get complete information. The library (client part of RPC calls) is available for HP-UX, Solaris, OS-9 and Linux.

## 10.10 Resource oriented calls

All the following calls are linked to resources

### 10.10.1 db\_getresource()

```
int db_getresource (dev_name, res, res_num, error)
    char          *dev_name;    /* The device name */
    Db_resource    res;          /* Array of res. name, type and pointer to store
                                resource value */
    unsigned int   res_num;      /* Resource number */
    long           *error;       /* Error */
```

This function retrieve resources from the database, convert them to the desired type and store them at the right place.

### 10.10.2 db\_putresource()

```
int db_putresource (dev_name, res, res_num, error)
    char          *dev_name;    /* The device name */
    db_resource    *res;         /* Array of res. name, type and pointer to
                                resource value */
    unsigned int   res_num;      /* Resource number */
    long           *error;       /* Error */
```

This function update already defined resource(s) or add new resource(s) if it (they) does not exist. Resource files are not updated by this function. It is not possible to update/insert resource belonging to the SEC domain.

### 10.10.3 db\_delresource()

```
int db_delresource (dev_name, res_name, res_num, error)
    char          *dev_name;    /* The device name */
    char          **res_name;    /* Resource name(s) to be deleted. */
    unsigned int   res_num;      /* Resource number */
    long           *error;       /* Error */
```

db\_delresource allows a user to remove resources from the database. The resource file where the resource was initially defined is not updated. It is not possible to delete resource(s) from the SEC domain with this function.

## 10.11 Exported device list oriented calls

The two following calls are used to get information on which devices are available for request in the control system.

### 10.11.1 db\_getdevexp()

```
int db_getdevexp (filter, tab, dev_num, error)
    char          *filter;    /* The filter to select exported devices */
    char          ***tab;     /* Exported devices name */
    unsigned int  *dev_num;   /* Exported devices number */
    long          *error;     /* Error */
```

This function allows a user to get the name of exported (and then ready to accept command) devices. With the filter parameter, it is possible to limit the devices name returned by the function. This function is not available for OS-9 client.

### 10.11.2 db\_freedevevp()

```
int db_freedevevp (ptr)
    char          **ptr;     /* Exported devices name array*/
```

The previous function can return a lot of device names and allocate memory to store them. This call is a local call and frees all the memory allocated by the db\_getdevexp function.

## 10.12 Device oriented calls

The following functions are device oriented.

### 10.12.1 db\_getdevlist()

```
int db_getdevlist (ds_full_name, dev_tab, dev_num, error)
    char          *ds_full_name; /* Full device server name (device server
                                process name/personal name) */
    char          ***dev_tab;    /* Device name(s) array */
    unsigned int  *dev_num;     /* Device number */
    long          *error;       /* Error */
```

db\_getdevlist returns to the caller the devices list for the device server with the full device server name ds\_full\_name.

### 10.12.2 db\_dev\_import()

```
int db_dev_import (name, tab, dev_num, error)
    char          **name;      /* Device(s) name to be imported */
    Db_devinfo_imp *tab;       /* RPC device(s) parameters array */
    unsigned int  *dev_num;    /* Device number */
    long          *error;      /* Error */
```

This function returns all the necessary parameters to build RPC connection between a client and the device server in charge of a device. It allows to retrieve these RPC's information for several devices at the same time.

### 10.12.3 db\_dev\_export()

```
int db_dev_export (devexp, dev_num, error)
    Db_devinfo      *tab;           /* RPC device(s) parameters array */
    unsigned int     *dev_num;      /* Device number */
    long             *error;        /* Error */
```

This function stores into the database the network parameters for a device or a group of devices. The network parameters are all the information needed by RPC to build a connection between a client and the device server in charge of a device.

### 10.12.4 db\_deviceinfo()

```
long db_deviceinfo (dev_name, devinfo, error)
    char             *dev_name;     /* Device name */
    db_devinfo_call   *devinfo;     /* Device informations */
    long             *error;        /* Error */
```

This function returns to the caller a structure with many device informations. These informations are the name of the server in charge of the device, the host where it is running, the device server program number, the device class...

### db\_deviceres()

```
long db_deviceres (dev_nb, dev_name_list, res_nb, res_list, error)
    long             dev_nb;        /* Number of device */
    char             **dev_name_list; /* Device name list */
    long             res_nb;        /* Number of resource(s) */
    char             ***res_list;    /* Resource(s) list */
    long             *error;        /* Error */
```

This function returns to the caller the list of all resources for a list of devices. The resources are returned as string(s) with the following syntax : "device name/resource name : resource value".

### 10.12.5 db\_devicedelete()

```
long db_devicedelete (dev_name, error)
    char             *dev_name;     /* Device name */
    long             *error;        /* Error */
```

This function deletes a device from the list of device registered in the database.

### 10.12.6 db\_devicedeletereres()

```
long db_devicedeletereres (dev_nb, dev_name_list, error)
    long             dev_nb;        /* Number of device */
    char             **dev_name_list; /* Device name list */
    db_error         *error;        /* Error */
```

This function deletes all the resources belonging to a list of devices from the database.

### 10.12.7 db\_getpoller()

```
long db_getpoller (dev_name, poll, error)
    char          *dev_name;      /* Device name */
    db_poller     *poll;          /* Device poller info */
    db_error       *error;         /* Error */
```

This function returns to the caller information about the device poller in charge of a device. A poller is a process in charge of "polling" the device in order to store device command result into the TACO data collector. The poller informations are the poller name, the host where it is running,....

## 10.13 Server oriented calls

The following functions deals with device server.

### 10.13.1 db\_svc\_unreg()

```
int db_svc_unreg (ds_full_name, error)
    char          *ds_full_name;  /* Full device server name (dev. server process
                                   name/personal name) */
    long          *error;         /* Error */
```

db\_svc\_unreg mark all the devices driven by the device server with a full name ds\_full\_name as not exported devices.

### 10.13.2 db\_svc\_check()

```
int db_svc_check (ds_full_name, h_name, p_num, v_num, error)
    char          **ds_full_name; /* Full device server name (dev. server
                                   process name/personal name) */
    char          *h_name;        /* Device server host name */
    unsigned int   *p_num;        /* Device server program number */
    unsigned int   *v_num;        /* Device server version number */
    long          *error;         /* Error */
```

This function returns host name, program number and version number of the first device found in the database for the device server with the full name ds\_full\_name.

### 10.13.3 db\_servinfo()

```
long db_servinfo (ds_name, pers_name, s_info, error)
    char          *ds_name;       /* Device server name */
    char          *pers_name;     /* Device server personal name */
    db_svcinfo_call *s_info;      /* Server information */
    long          *error;         /* Error */
```

This function returns miscellaneous informations for a device server started with a personal name. These informations are the number and name of device served by the server, the device server process name....



### 10.13.4 db\_servdelete()

```
long db_servdelete (ds_name, pers_name, delres_flag, error)
char          *ds_name;          /* Device server name */
char          *pers_name;        /* Device server personal name */
long          delres_flag;       /* Delete device(s) resource flag */
long          *error;            /* Error */
```

This function deletes a device server from the database and if needed, all the server device resources.

### 10.13.5 db\_servunreg()

```
long db_servunreg (ds_name, pers_name, error)
char          *ds_name;          /* Device server name */
char          *pers_name;        /* Device server personal name */
long          *error;            /* Error */
```

This function unregisters (mark device(s) as not exported) for all the device(s) served by the device server ds\_name started with the personal name pers\_name.

## 10.14 Database browsing oriented calls

All the following 11 calls allows database browsing

### 10.14.1 db\_getdevdomainlist()

```
long db_getdevdomainlist(domain_nb, domain_list, error)
long          *domain_nb;        /* The number of domain */
char          ***domain_list;    /* Domain name list */
long          *error;            /* Error */
```

This function returns to the caller a list of domain used for all devices defined in the database.

### 10.14.2 db\_getdevfamilylist()

```
long db_getdevfamilylist(domain, family_nb, family_list, error)
char          *domain;           /* The domain name */
long          *family_nb;        /* The number of families */
char          ***family_list;    /* Family name list */
long          *error;            /* Error */
```

This function returns to the caller a list of families for all devices defined in the database with the first field set to a given domain name.

### 10.14.3 db\_getdevmemberlist()

```
long db_getdevmemberlist(domain, family, member_nb, member_list, error)
char          *domain;           /* The domain name */
char          *family;           /* The family name */
long          *member_nb;        /* The number of members */
char          ***member_list;    /* Member name list */
long          *error;            /* Error */
```

This function returns to the caller a list of members for all devices defined in the database with the first field name set to a given domain and the second field name set to a given family.

#### 10.14.4 db\_getresdomainlist()

```
long db_getresdomainlist(domain_nb, domain_list, error)
    long      *domain_nb;      /* The number of domain */
    char      ***domain_list; /* Domain name list */
    long      *error;          /* Error */
```

This function returns to the caller a list of domain used for all resources defined in the database.

#### 10.14.5 db\_getresfamilylist()

```
long db_getresfamilylist(domain, family_nb, family_list, error)
    char      *domain;        /* The domain name */
    long      *family_nb;     /* The number of families */
    char      ***family_list; /* Family name list */
    long      *error;         /* Error */
```

This function returns to the caller a list of families for all resources defined in the database with the first field name set to a given domain name.

#### 10.14.6 db\_getresmemberlist()

```
long db_getresmemberlist(domain, family, member_nb, member_list, error)
    char      *domain;        /* The domain name */
    char      *family;        /* The famiy name */
    long      *member_nb;     /* The number of members */
    char      ***member_list; /* Member name list */
    long      *error;         /* Error */
```

This function returns to the caller a list of members for all resources defined in the database with the first field name set to a given domain and the second field name set to a given family.

#### 10.14.7 db\_getresresolist()

```
long db_getresresolist(domain, family, member, resource_nb, resource_list, error)
    char      *domain;        /* The domain name */
    char      *family;        /* The famiy name */
    char      *member;        /* The member name */
    long      *resource_nb;   /* The number of members */
    char      ***resource_list; /* Resource name list */
    long      *error;         /* Error */
```

This function returns to the caller a list of resource name for all resources defined in the database for a device with a specified domain family and member field name.

### 10.14.8 db\_getresresoal()

```
long db_getresresoal(domain, family, member, resource, resval_nb, resource_list, error)
char      *domain;           /* The domain name */
char      *family;           /* The famiy name */
char      *member;           /* The member name */
char      *resource;         /* The resource name */
long      *resval_nb;        /* The number of resource values */
char      ***resource_list;  /* Resource value list */
long      *error;            /* Error */
```

This function returns to the caller a list of resource values for all the resource with a domain, family, member and name specified in the first four function parameters. Member and resource field name can be set to wild card (\*).

### 10.14.9 db\_getdsserverlist()

```
long db_getdsserverlist(server_nb, server_list, error)
long      *server_nb;        /* The number of device server */
char      ***server_list;    /* Server name list */
long      *error;            /* Error */
```

This function returns to the caller a list of device server executable name.

### 10.14.10 db\_getdspersnamelist()

```
long db_getdspersnamelist(server, persname_nb, persname_list, error)
char      *server;           /* The device server executable name */
long      *persname_nb;      /* The number of personal name */
char      ***persname_list;  /* Personal name list */
long      *error;            /* Error */
```

This function returns to the caller a list of device server personal name list for device server with a given executable name.

### 10.14.11 db\_gethostlist()

```
long db_gethostlist(host_nb, host_list, error)
long      *host_nb;          /* The number of host name */
char      ***host_list;     /* Host name list */
long      *error;            /* Error */
```

This function returns to the caller a list of hosts name where device server should run.

## 10.15 Pseudo device oriented calls

### 10.15.1 db\_psdev\_register()

```
int db_psdev_register (psdev, num_psdev, error)
db_psdev_info *psdev;        /* Pseudo device parameters array */
long          num_psdev;      /* Pseudo devices number */
db_error      *error;         /* Error */
```

This function is used to register pseudo devices into the database. This feature has been implemented only for control system debug purpose. It helps the debugger to know which process has created pseudo devices and on which computer they are running.

### 10.15.2 db\_psdev\_unregister()

```
int db_psdev_unregister (psdev_list, num_psdev, error)
    char          **psdev_list;      /* Pseudo device(s) names list */
    long          num_psdev;         /* Pseudo devices number */
    db_error      *error;            /* Error */
```

This function is used to unregister pseudo devices from the database.

## 10.16 Database update calls

### 10.16.1 db\_analyse\_data()

```
long db_analyse_data (in_type, buffer, nb_devdef, devdef, nb_resdef, resdef,
                    error_line, error)
    long          in_type            /* Buffer type (buffer or file) */
    char          *buffer;           /* Buffer */
    long          *nb_devdef;        /* Number of device definition list */
    char          ***devdef;         /* Device definition list */
    long          *nb_resdef;        /* Number of resource definition list */
    char          ***resdef;         /* Database definition list */
    long          *error_line;       /* Buffer line number with error */
    long          *error;            /* Error */
```

This function analyses a buffer (file or buffer) assuming that this buffer is used to update the database and returns device definition list and resource definition list.

### 10.16.2 db\_upddev()

```
long db_upddev ( nb_devdef, devdef, deferr_nb, error)
    long          nb_devdef;         /* Number of device definition list */
    char          **devdef;          /* Device definition list */
    long          *deferr_nb;        /* Device def. list number with error */
    long          *error;            /* Error */
```

This function updates the database with the new device definition defined in the device definition list.

### 10.16.3 db\_updres()

```
long db_updres ( nb_resdef, resdef, deferr_nb, error)
    long          nb_resdef;         /* Number of resource definition */
    char          **resdef;          /* Resource definition list */
    long          *deferr_nb;        /* Resource def. number with error */
    long          *error;            /* Error */
```

This function updates the database with the new resource definition contained in the resource definition list.

## 10.17 Miscellaneous calls

### db\_stat()

```
long db_stat (info, error)
    db_stat_call *info;          /* Database information */
    long          *error;         /* Error */
```

This functions returns database global informations as the number of exported devices defined in the database, the number of resources defined for each device domain...

### db\_secpass()

```
long db_secpass (pass, error)
    char          **pass;         /* Database security password */
    long          *error;         /* Error */
```

The static database is also used to store security resources. A very simple system protects security resources from being updated by a user if the administrator choose to protect them. This function returns database protection data to the caller allowing an application to ask its user for security resources password.

### db\_cmd\_query()

```
int db_cmd_query (cmd_name, cmd_code, error)
    char          *cmd_name;      /* Command name */
    unsigned int  *cmd_code;      /* Command code */
    long          *error;         /* Error */
```

The static database is also used to store (as resources) command name associated to command code (in the CMDS domain). db\_cmd\_query returns the command code associated to a command name.

### db\_svc\_close()

```
int db_svc_close ( error)
    long          *error;         /* Error */
```

This function asks the database server to close all the files needed to store database data (the ndbm files) allowing another process to open these files. When this function is called, no further call to database server will work until the db\_svc\_reopen function will be executed.

### db\_svc\_reopen()

```
int db_svc_close ( error)
    long          *error;         /* Error */
```

This function asks the database server to reopen database files.

## 10.18 Multi TACO control system access

With release 5.5 and above of database software, the *db\_getresource* and *db\_dev\_import* calls of the C library have been modified in order to allow access to multiple TACO control system. To specify which TACO control system should be used, a forth field

must be added to the device name. This forth field is the name of the computer where the TACO anchor process is running (The process called **Manager**). In this case, the naming syntax is :

**//FACILITY/DOMAIN/FAMILY/MEMBER**

The facility name is also known as **NETHOST**. Example of device name which specify the machine control system : //ARIES/SR/D\_CT/1. Another example for a device sy/ps-b/1 defined in a control system where the nethost is libra: //LIBRA/SY/PS-B/1. For device where the nethost is not specifed, the NETHOST environment variable is used.

The db\_dev\_import enables a user to retrieve necessary parameters to build RPC connections between clients and server for several devices with the same call. The TACO control system defined by the first device of the list will be used.



# Chapter 11

## Events

*by A. Götz*

### 11.1 Introduction

The TACO control system was originally based on synchronous remote procedure calls (RPCs) and the client-server model. Clients and servers which required asynchronism made use of the data collector (a distributed online buffer of device command results) or the servers implemented their own mini-buffers locally and the clients polled the server. This is not always efficient in terms of time, network bandwidth and CPU usage. Therefore an asynchronous call was added and has been available for over a year now. The asynchronous call implements the mechanisms necessary to add events without much effort. It was logical therefore with the recent move towards Linux on frontends to take advantage of the excellent TCP/IP stack implementation on Linux to offer programmers and clients events.

The present implementation offers a simple model for user events which will permit device server programmers to add their own events (user events) to their code thereby providing adding value to their device servers. The present implementation is ideal for device servers which have a small number of clients. A full implementation with sophisticated system and user events which provides efficient mechanisms for distributing events to large numbers of clients will be implemented in TANGO (next generation TACO). The present implementation in TACO is simply an avant-gout of TANGO events and allows TACO programmers to gain experience using events.

This chapter presents the user event api, examples of how to program them and a discussion on performance and problems which can arise.

### 11.2 Events

Events are short messages which are sent to clients asynchronously. The origin of the messages is a device server. Clients only receive messages if they have solicited them. Events are classified according to type. Event types are specific to the device server and should be defined as unique long integers. The most obvious way to do so is to use the device class unique base as offset and number events starting from 1 e.g. :

1. `#define D_EVENT_AGPS_STATE    DevAgpsBase + 1`
2. `#define D_EVENT_OMS_STATE_CHANGE DevOmsBase + 1`





```

    long event_type - event type to dispatch
    DevArgument argout - pointer to argout to dispatch with event
    DevType argout_type - argout type
    long event_status - status of event to dispatch to client
    long event_error - error code of event to dispatch to client (if status != DS_OK)
- C++ using the Device class :

    long dev_event_fire (Device *device, long event_type,
                        DevArgument argout, DevType argout_type,
                        long event_status, long event_error)

    long event_type - event type to dispatch
    DevArgument argout - pointer to argout to dispatch with event
    DevType argout_type - argout type
    long event_status - status of event to dispatch to client
    long event_error - error code of event to dispatch to client (if status != DS_OK)

```

## 11.4 Implementation

User events have been implemented in TACO DSAPI v7.0. They have been tested on Linux/x86, Linux/m68k, HP-UX and Solaris 2.5. They should work in principle on OS-9 but because of its flaky TCP/IP stack implementation programmers are urged to port their device servers to one of the Unix derivatives e.g. Linux, where they will not be plagued by sockets closing when they shouldn't or not closing when they should ! No port has been undertaken so far for Windows or VxWorks.

## 11.5 Timeouts

Events depend on detecting the server or client going down in order to work correctly. This is treated as a timeout in the client. If the client does not receive any events during a period exceeding the asynchronous timeout value (set using `dev_async.timeout()`) it will ping the server to see if it is still alive. If not it will trigger the event callback with status = `DS_NOTOK` and error = `DevErr_RPCTimedout`. The event will be unregistered on the client side. If the server detects a client is not there anymore it will silently remove it from the list of registered clients.

## 11.6 Examples

How best to generate events in a device server ? The most obvious way is to create an event thread whose job it is to poll a variable (state or value) to detect the event. Once the event is detected the event thread calls `dev_event_fire()` to dispatch the event. Here is a simple example to generate a periodic event using Posix threads :

```

void * events_thread(void * arg)
{
    long event = 1;
    long counter=0;
    struct timespec t100ms;

    fprintf(stderr, "\nfire_events(): starting thread %s\n",

```

```

        (char *) arg);

    for (;;)
    {

        dev_event_fire(ds, event,&counter,D_LONG_TYPE,DS_OK,0);
        counter++;

/*
 * sleep for 90 ms
 */
        t100ms.tv_sec = 0;
        t100ms.tv_nsec = 90000000;
        nanosleep(&t100ms, NULL);
    }
    return NULL;
}

int event_thread_start()
{
    int retcode;
    pthread_t th_a, th_b;
    void * retval;

#ifdef linux || defined(solaris)
    retcode = pthread_create(&th_a, NULL, fire_events, "a");
#else
    retcode = pthread_create(&th_a, pthread_attr_default,
                            (pthread_startroutine_t)fire_events,
                            (pthread_addr_t)"a");
#endif /* linux || solaris */
    if (retcode != 0) fprintf(stderr, "create a failed %d\n",
                               retcode);
}

```

The function `event_thread_start()` has to be called at an appropriate point in the device server e.g. during `class_initialise()` or `object_create()`.

## 11.7 Performance

The performance of events depends naturally on what type of system the device server is running on. Tests have been carried out on Linux/x86, Linux/68k, HP-UX and Solaris running on Pentiums, 68030s, s700s and SPARC CPUs. They all showed similar performance with variations due to the scheduler. Firing of events uses the one-way ONC RPC mechanism which means it is immediately copied to the system buffer without waiting. This means there is very little overhead introduced in the device server. Generating events at maximum speed shows that the minimum time between events is about 25 microseconds with an average of 500 microseconds over a long (seconds) time scale. This is due to scheduler stopping the device server at regular intervals (presumably to dispatch the events).

Using the example code above a number of tests were done on different platforms. The results were all roughly the same i.e. the server could generate events at regular time intervals of 100 milliseconds with a jitter of less than 10 microseconds. The jitter goes up as a function of the number of clients e.g. jitter of 25 microseconds for 10 clients on Linux/m68k. Here is an example output log from a client (Linux/x86

+ Pentium) which accepts the events from a device server running on a tacobox (Linux/x86 + Pentium) and prints out their times :

```

counter =      3362 , server time = {924772119 s,342170 us} delta time = 99974 us
counter =      3363 , server time = {924772119 s,442169 us} delta time = 99999 us
counter =      3364 , server time = {924772119 s,542169 us} delta time = 100000 us
counter =      3365 , server time = {924772119 s,642169 us} delta time = 100000 us
counter =      3366 , server time = {924772119 s,742169 us} delta time = 100000 us
counter =      3367 , server time = {924772119 s,842169 us} delta time = 100000 us
counter =      3368 , server time = {924772119 s,942169 us} delta time = 100000 us
counter =      3369 , server time = {924772120 s,042173 us} delta time = 100004 us
counter =      3370 , server time = {924772120 s,142169 us} delta time = 99996 us
counter =      3371 , server time = {924772120 s,242169 us} delta time = 100000 us
counter =      3372 , server time = {924772120 s,342169 us} delta time = 100000 us
counter =      3373 , server time = {924772120 s,442169 us} delta time = 100000 us
counter =      3374 , server time = {924772120 s,542169 us} delta time = 100000 us

```

## 11.8 Known problems

Known problems so far are that when the server or client die then HP-UX and Solaris servers and clients have difficult to detect this due to the way sockets are handled. The next release will fix this by implementing an event heartbeat which will reactivate the event channel. Failure to do so will result in the event timing out and the client being removed from the list of registered clients in the server.



## Chapter 12

# The Signal Interface

*by J.Meyer and J-L.Pons*

### 12.1 Introduction

The device server signal interface is based on the use of the device server signal and multi signal classes. They define a signal object for a value with a set of standard properties and functionality. The implementation of three commands with standard behaviour in the device class offers a standard interface to clients. Following this conventions, generic monitoring applications and the history database can be easily used, without coding, on the devices of a class.

### 12.2 Conventions on Signals

The signal class allows the creation of signal objects with a naming convention as:

DOMAIN/FAMILY/MEMBER/SIGNAL

The signal name is an extension to the device name used in the ESRF control system. To create a signal object a name with four fields must be used. This corresponds to signal naming as it is used in the history database and in general data display applications.

- A signal represents a simple data value.
- All signals of a class must be of the same data type.
- The data type might be float values or double values.

A special problem is the relation between read and set values. To identify all signals which can be set clearly the following naming convention must be respected. A set-point signal name must be preceded by the identifier "set-".

Example: SR/RF-FOC/TRA3-1/set-Voltage

A set-point signal can be modified and its actual value can be read.

In the case of a readable set-point value and a separate read value (as on most of the power supplies) the read values must keep the same signal name without the preceding identifier "set-".

Example: SR/RF-FOC/TRA3-1/Voltage

With this convention all signals which can be modified can be easily identified. Also the relation between separate read and set signals can be automatically established.

## 12.3 The Signal Properties

A set of signal properties is defined in the signal class. The properties must be defined for a device class. They are used for signal identification and the automatic configuration of monitoring and tuning applications and the history database configuration tool.

The properties of a signal object are:

1. **Name** - The full signal name.
2. **Label** - A label for the signal value, which can be used in applications.
3. **Unit** - The unit of the signal value.
4. **Format** - The format in which the data should be displayed (printf() format).
5. **Description** - A text describing the signal.
6. **Max** - A maximum value. Can be used for data display or to check limits of set values.
7. **Min** - A minimum value. Can be used for data display or to check limits of set values.
8. **AlHigh** - Above this limit an alarm will be indicated.
9. **AlLow** - Under this limit an alarm will be indicated.
10. **Delta** - If the nominal value and the read value of the signal differ by +/- delta during the number of seconds specified by "Delta.t" , an alarm will be raised.
11. **Delta.t** If the nominal value differs from the read value for a longer time than Dta.t seconds, an alarm will be raised.
12. **Standard Unit** - A multiplier factor to convert the given signal unit into a standard unit (V, A, W, bar ...).

## 12.4 The Server Side

### 12.4.1 The Commands to Access Signals

Four commands must be defined in a device class to access signals. One to read an array of signal values, one to identify and to describe each signal value, one to update changed signal properties and one to set a signal value.

#### DevReadSigValues

The command reads an array of signal values. The array should contain all signals for this class. The data type for all signals of a class must be the same. Possible data types are float values or double values. The command must always return an array, even if only one signal value is defined.

To avoid the polling of several commands in the data collector, the state of a device should be also treated as a signal and should be returned as the signal "DOMAIN/FAMILY/MEMBER/State" by this command.

Command list entry:

```
DevReadSigValues, read_signal_values, D_VOID_TYPE, D_VAR_FLOATARR, READ_ACCESS
```

Command function definition:

```
long read_signal_values (xxx ds, DevVoid *argin, DevVarFloat Array *argout, long *error)
```

Description: Returns the signal values of a device.

Arg(s) In: None

Arg(s) Out: DevVarFloatArray signal\_values - Array of signal values.

long \*error - Pointer to error code, in case routine fails.

### **DevGetSigConfig**

The command reads the properties of all signals returned by DevReadSigValues. The order of the signals must be the same for the two commands. The first value returned by DevReadSigValues must correspond to the first set of properties returned by DevReadSigConfig.

The properties of all signals of a class are returned as a string array. The first string (element [0]) must indicate the number of properties per signal, to have the flexibility to add new properties. The number of elements in the string array will be:

length = number of properties \* number of signals + 1

The properties of the signals must be added to the string array by using the result of the method DevMethodReadProperties on the signal or multi signal object (see: the user guides of the two classes).

Command list entry:

```
DevReadSigConfig, read_signal_config, D_VOID_TYPE, D_VAR_STRINGARR, READ_ACCESS
```

Command function definition:

```
long read_signal_config (xxx ds, DevVoid *argin, DevVarStringArray *argout, long *error)
```

Description: Returns the signal properties of all signals of a device.

Arg(s) In: None

Arg(s) Out: DevVarStringArray signal\_values - Array of signal properties.

long \*error - Pointer to error code, in case routine fails.

### **DevUpdateSigConfig**

The command reinitialises all signal properties of all signals of a device. After an update of the resource database calling this command reinitialises all signal properties dynamically with their actual resource values. The goal is an interactive resource editor with a direct update of the device configuration.

The method DevMethodSignalsReset must be used on the signal or multi signal object (see: the user guides of the two classes)

Command list entry:

```
DevUpdatedSigConfig, update_signal_config, D_VOID_TYPE, D_VOID_TYPE, WRITE_ACCESS
```

Command function definition:



```
long update_signal_config (xxx ds, DevVoid *argin, DevVoid *argout, long *error)
```

Description: Reinitialises all signal properties of all signals of a device with the actual resource values.

Arg(s) In: None

Arg(s) Out: None

### DevSetSigValue

Receives a new value for a set-point signal (with "set-" identifier). Verifies the validity of the given signal name and that the value doesn't exceed the specified range for the signal by using the method DevMethodCheckLimits on the signal or multi signal object (see: the user guides of the two classes). Applies the new set-point.

Command list entry:

```
DevSetSigValue, set_signal_value, D_STRINGDOUBLE_TYPE, D_VOID_TYPE, WRITE_ACCESS
```

Command function definition:

```
long set_signal_value (xxx ds, DevStringDouble *argin, DevVoid *argout, long *error)
```

Description: Receives a new value for a signal. Verifies that the value doesn't exceed the specified range for the signal.  
Applies the new set-point.

Arg(s) In: DevStringDouble \*argin - Structure containing the name of the signal to modify as a string and the value to be applied as double.

Arg(s) Out: None

### 12.4.2 Coding Example using a Multi Signal Object

This example is for a device server written in "C". For the use in a "C++" device server the multi signal object must be created via the OIC interface, but can be used with the same functionality.

To use a multi signal object it must be created and initialised in the object\_initialise() method:

```
#include <MDSSignalP.h>
#include <MDSSignal.h>

/*
 * Create the signal objects specified for this class
 */

    if (ds__create (ds->devserver.name, mDSSignalClass,
                  &ds->focus.msignal_obj, error) == DS_NOTOK)
    {
        return(DS_NOTOK);
    }

    if (ds__method_finder (ds->focus.msignal_obj, DevMethodInitialise)
```

```

        (ds->focus.msignal_obj, focusClass->devserver_class.class_name,
         error) == DS_NOTOK)
    {
        return(DS_NOTOK);
    }

```

Afterwards two commands can be implemented using the multi signal object:

=====

Function:        static long read\_signal\_config()

Description:    Read the properties of all signals specified  
                 for the focus power supply.

Arg(s) In:       Focus ds        - pointer to object  
                 void \*argin    - no input arguments

Arg(s) Out:      DevVarStringArray \*argout - Array of signal properties  
                 long \*error    - pointer to error code, in case routine fails

=====

```

static long read_signal_config (Focus ds, DevVoid *argin,
                               DevVarStringArray *argout, long *error)
{
    *error = 0;
    if (ds__method_finder (ds->focus.msignal_obj,
                          DevMethodReadProperties)
        (ds->focus.msignal_obj, argout, error) == DS_NOTOK)
    {
        return(DS_NOTOK);
    }
    return (DS_OK);
}

```

=====

Function:        static long update\_signal\_config()

Description:    Reinitialises all specified signal properties with  
                 their actual resource values..

Arg(s) In:       Focus ds        - pointer to object  
                 void \*argin    - no input arguments

Arg(s) Out:      void \*argout   - no outgoing arguments  
                 long \*error    - pointer to error code, in case routine fails

=====

```

static long    update_signal_config (Focus ds, DevVoid *argin,
                                    DevVoid *argout, long *error)
{
    *error=0;

```

```

        if (ds__method_finder (ds->focus.msignal_obj, DevMethodSignalsReset)
            (ds->focus.msignal_obj, error) == DS_NOTOK)
        {
            return(DS_NOTOK);
        }
        return(DS_OK);
}

```

The third command just has to return an array of values which must be ordered as the signal properties!

=====

Function:        static long read\_signal\_values()

Description:    Read the measurement and setpoint values  
                 for this device.

                 [0] : current setpoint  
                 [1] : voltage  
                 [2] : current

Arg(s) In:      Focus ds        - pointer to object  
                 void \*argin    - no input arguments

Arg(s) Out:     DevVarFloatArray \*argout - Array of signal values..  
                 long \*error    - pointer to error code, in case routine fails

=====

```

static long read_signal_values (Focus ds, DevVoid *argin,
                               DevVarFloatArray *argout, long *error)
{
    static float      values[3];
    *error = 0;

    .....

    -> Read the signal values here!

    .....

    argout->length     = 3;
    argout->sequence = &values[0];
    return (DS_OK);
}

```

The fourth command must treat all available set-points, which are identified by their name.

=====

Function:        static long set\_signal\_value()

Description:    Receives a new value for a signal. Verifies that the value

doesn't exceed the specified range for the signal. Applies the new set-point.

Arg(s) In:       Focus ds - pointer to object  
               DevStringDouble \*argin - Structure containing the name  
               of the signal to modify as a string and  
               the value to be applied as double.

Arg(s) Out:      void \*argout - no output arguments.  
                   long \*error - pointer to error code, in case routine fails

```
=====

static long set_signal_value (Focus ds, DevStringDouble *argin,
                             void *argout, long *error)
{
    long limit_state;
    char *sig_name;

    *error = 0;

    /*
     * Check whether the signal name is a valid set-point signal and
     * whether its values are in the specified range.
     */

    if (ds__method_finder (ds->focus.msignal_obj, DevMethodCheckLimits)
        (ds->focus.msignal_obj, argin, &limit_state, error)
        ==DS_NOTOK)
    {
        return(DS_NOTOK);
    }

    if ( limit_state != DEVRUN )
    {
        *error = DevErr_ValueOutOfBounds;
        return (DS_NOTOK);
    }

    /*
     * Find the set-point signal amongst all available set-points and
     * apply the new set value.
     */

    sig_name = strrchr (argin->name, '/');
    sig_name++;

    if ( strcmp (sig_name, "set-Voltage") == 0 )
    {
        .....
        -> Set the value here!
        .....
    }
}
```

```

        if ( strcmp (sig_name, "set-Current") == 0 )
        {
            .....
            -> Set the value here!
            .....
        }

        return (DS_OK);
    }
}

```

The multi signal object is also used to handle alarms on signals which change the state of a device. The method used in the DevState command is DevMethodCheckAlarms and the method used in the DevStatus command is DevMethodReadAlarms. See the Multi Signal Class Users Guide for more information.

## 12.5 Reading the Signal Properties without Accessing the Device

A second way to extract the signal names and properties of a device was developed. They are read directly from the resource database without a connection to the device. This interface is used in applications like fsgmon, devsel, hdb\_config and the hdb\_filler which can read data only from the data collector without having access to a device server running on a VME crate.

To use this functionality your client must be linked with the shared library: libdssig.sl The functions were not integrated to the TACO API-library, because it uses internally the signal and multi signal classes. This would cross reference the API-library with the class library. Linking problems and Makefile changes would be the result. Available functions are:

### 12.5.1 dev\_get\_sig\_config()

```

long dev_get_sig_config (char *device_name, DevVarStringArray *sig_config,
                        long *error)

```

**Description:** Extract the signal configuration for a device from the resource database. The result is the same as calling the command DevGetSigConfig on the device. The returned data must not be freed. Data will be freed with the next call to the function.

**Arg(s) In:** char \*device\_name - Name of the device.

**Arg(s) Out:** DevVarStringArray \*sig\_config - Array containing the configuration of all signals known for this device.

long \*error - pointer to error code, in case routine fails.

### 12.5.2 dev\_get\_sig\_config\_from\_name()

```

long dev_get_sig_config_from_name (char *signal_name,
                                   DevVarStringArray *sig_config,
                                   long *error)

```

Description: Extract the signal configuration for one signal of a device from the resource database. The returned data must not be freed. Data will be freed with the next call to the function.

Arg(s) In: char \*device\_name - Name of the device.  
char \*signal\_name - Name of the signal.

Arg(s) Out: DevVarStringArray \*sig\_config - Array containing the configuration of the signal for this device.

long \*error - pointer to error code, in case routine fails.

### 12.5.3 dev\_get\_sig\_list()

```
long dev_get_sig_list (char *device_name, DevVarStringArray *sig_list,
                      long *error)
```

Description: Extract all signal names defined for a device.

Arg(s) In: char \*device\_name - Name of the device.

Arg(s) Out: DevVarStringArray \*argout - Array containing the list of signals defined for the device.

long \*error - pointer to error code, in case routine fails.

### 12.5.4 dev\_get\_sig\_set\_list()

```
long dev_get_sig_set_list (char *device_name, DevVarStringArray *argout,
                          long *error)
```

Description: Extract all signal names for set-points defined for a device. Signal names for set-points are pre-ceeded by the by the identifier "set-".

Arg(s) In: char \*device\_name - Name of the device.

Arg(s) Out: DevVarStringArray \*sig\_list - Array containing the list of signals for set-points defined for the device.

long \*error - pointer to error code, in case routine fails.

### 12.5.5 dev\_get\_sig\_setread\_from\_name()

```
long dev_get_sig_setread (char *signal_name, DevLongString *set_signal,
                        DevLongString *read_signal, long *error)
```

Description: Returns for a given signal of a device the corresponding set-point signal and read-point signal names together with their index in the signal list of the device.

The signal name entered can be either the set-point signal

or the read-point signal name. If a set-point doesn't exist for a entered signal name, a NULL pointer is returned for the signal name and the index is initialised to "-1". The same is true for a set-point signal which has no separate read-signal defined.

Signal names for read-points and set-points are the same, only the set-point signal name is preceded by the identifier "set-".

Arg(s) In:        char \*device\_name - Name of the device.  
                    char \*signal\_name - Name of the signal.

Arg(s) Out:       DevLongString \*set\_signal - The name and the index,  
                     in the signal list, of the set-point signal.

                   DevLongString \*read\_signal - The name and the index,  
                     in the signal list, of the read-point signal.

                   long \*error - pointer to error code, in case routine fails.

## 12.6 The Client Side

With the described commands, signals can be displayed in a generic way on the client side.

1. To find out the data type used by the command DevReadSigValues, the function dev\_cmd\_query() of the API-library can be used. Filtering for the command indicates the data type of the outgoing arguments.
2. By executing the command DevReadSigConfig the place of a signal in the array can be determined by its name. All other properties needed for a signal display are following the signal name in the described order (see "The Signal Properties" on page2).
3. DevReadSigValues returns the signal values in the same order as indicated by DevReadSigConfig.

An example shows how DevReadSigConfig and DevReadSigValues can be used to display signals in a device server menu. The data type in this case is known and dev\_cmd\_query() is not used.

```
devserver          device;
DevVarStringArray  sig_config;
DevVarFloatArray   param_array;
long               nu_of_properties;
long               nu_of_signals;
long               i, k;

case (3) :

    /*
     * Read the device signal values.
     */
```

```

    param_array.length = 0;
    param_array.sequence = NULL;

    if (dev_putget (device, DevReadSigValues, NULL, D_VOID_TYPE,
                   &param_array, D_VAR_FLOATARR, &error) < 0)
    {
        dev_printerror_no (SEND, "DevReadSigValues", error);
        break;
    }

    /*
     * Read the signal properties to display the values.
     */

    sig_config.length = 0;
    sig_config.sequence = NULL;

    if (dev_putget (device, DevGetSigConfig, NULL, D_VOID_TYPE,
                   &sig_config, D_VAR_STRINGARR, &error) < 0)
    {
        dev_printerror_no (SEND, "DevGetSigConfig", error);
        break;
    }

    /*
     * Find the label format and unit for the signal values.
     */

    nu_of_properties = atol (sig_config.sequence[0]);
    nu_of_signals = (sig_config.length - 1) / nu_of_properties;

    printf ("Device parameters:\n");
    for (i=0; i<nu_of_signals; i++)
    {
        sprintf (format, "%24s [%2s] : %s\n",
                 sig_config.sequence[(i*nu_of_properties) + 2],
                 sig_config.sequence[(i*nu_of_properties) + 3],
                 sig_config.sequence[(i*nu_of_properties) + 4]);
        printf (format, param_array.sequence[i]);
    }

    /*
     * Free the allocated arrays.
     */

    if ( dev_xdrfree (D_VAR_FLOATARR, &param_array, &error) < 0 )
    {
        dev_printerror_no (SEND, "dev_xdrfree", error);
    }

```



```

        if ( dev_xdrfree (D_VAR_STRINGARR, &sig_config, &error) < 0 )
        {
            dev_printerror_no (SEND, "dev_xdrfree", error);
        }
        break;

```

## 12.7 The Signal Interface to HDB

An entry point to the HDB signal library was developed to allow signal configuration in HDB with the same names as they are known in a device class. Using `dev_get_sig_config()` in the HDB signal library and storing the result of the command `DevReadSigValues` in the data collector, all signals configured for a device class (in the device server) are dynamically available in HDB with the same names and descriptions.

But, today the HDB signal library still needs for dynamic loading one module for each device class. It is just a question of copy and paste to install such a module for a device class using the signal interface, but it implies recompilation of the HDB signal library. Studies are going on to change this to avoid recompilation and reinstallation of the HDB signal library in the future.

Here is an example module for the HDB signal library. This can be copied, but the function names must be changed to the class name the new module will be used for.

```

#include <API.h>
#include <siggen.h>

/*
 * function prototypes
 */

long RF_FOCUS_load_type (long *error);
long RF_FOCUS_signal_list_init (char *device_name,
                                SigDefEntry **signal_list_ptr,
                                long *n_signal,
                                long *error);

extern long signal_list_init (char *device_name,
                              SigDefEntry **signal_list_ptr,
                              long *n_signal,
                              long *error);

/*
 * The load type function
 */

long RF_FOCUS_load_type (long *error)
{
    return (DS_OK);
}

/*
 * Dynamic signal initialisation function.
 * Uses signals defined on the device server level.

```

```
*/

long RF_FOCUS_signal_list_init (char *device_name,
                                SigDefEntry **signal_list_ptr,
                                long *n_signal,
                                long *error)
{
    /*
     * calls the general signal init function, which is
     * used for all classes which implement signals on
     * the device server level.
     */

    if ( signal_list_init (device_name, signal_list_ptr,
                          n_signal, error) == DS_NOTOK )
    {
        return (DS_NOTOK);
    }
    return (DS_OK);
}
```

## 12.8 Conclusion

The device server signal interface was developed for the SRRF project and was adapted mainly to the project needs. But, I see it as a useful extension to other device server classes. The advantage of using signals is that you can immediately profit from generic plotting and display programs like `fsigmon` and `xtuning`. Contact `meyer@esrf.fr` or `pons@esrf.fr` for more information on these programs.



## Chapter 13

# Access Control and Security

*by J.Meyer*

### 13.1 Introduction

In TACO an object can be a physical piece of hardware, an ensemble of hardware, a logical device or a combination of all these [1]. Objects (**devices**) are created, exported and stored in a process called a **device server**. Every device is exported with a unique three field name consisting of **DOMAIN/FAMILY/MEMBER** and understands a set of commands which are specific for a class of objects in the device server. Every exported object can be accessed via the Remote Procedure Call (**RPC**) interface of the device server.

A device server client uses the Application Programmers Interface (API) to access devices. The API is based on the file paradigm which consists of **opening** the file, **reading/writing** to the file and then **closing** the file. In the device server API paradigm these actions are **importing**, **accessing** and **freeing** the device connection [1].

### 13.2 The Problem

One problem of TACO was the open access to devices from all over the network and by all users on the network. Access restrictions were only possible by system administration means, like restricted network access.

It was not possible to protect sensitive actions on devices because, once a device was imported, all commands could be executed. Also no possibility was given to block a device in a kind of single user mode to do some action which required exclusive access for a user (e.g. tuning or calibration of hardware).

To solve the above mentioned problems, a database supported security system was needed. Sufficient control over users and groups of users, which are allowed to access devices in the control system, had to be given. In order not to be dependent on machines where the control system is running, access control for networks and hosts had to be added. A list of hierarchical rights was established to specify access modes to devices. Combining a minimal access right with a command of a device, allows a protection for critical actions. A single user mode was added to give clients the possibility to be sure, that a sequence of commands on a device is not interrupted by other clients.

The solution described has been modelled on the **Amoeba** distributed operating system [3] **capability** lists and the UNIX **access control lists**. Development effort

has gone into making the system as flexible as possible, with reconfigurable access rights at runtime and fast access verification for received RPC calls in a device server.

## 13.3 The Model

### 13.3.1 Users, Groups and Networks

To guarantee sufficient access control the following points have to be verified with the reference data in the security database:

- If a user is explicitly specified in the database, the user name **and** the user ID must be correct. This avoids problems with badly configured user ID's.
- If no user data is available, the actual group name **and** group ID must be correct.
- If the user or his group are verified, the IP-address of the host, where the client was started, has to be compared with the specified network access for the user or his group.
- If neither user data nor group data is available, only the specified minimal default access to the control system can be given. Also for no network access specifications, a minimal default access can be granted.

Figure 1 shows an example of possible access security database specifications.

Entry	Name	ID	Network Access
user	meyer	215	
user	taurel	261	160.103.10 160.103.5.68
user	operator	226	160.103.10 160.103.11 160.103.12
group	comp	101	160.103
group	machine	102	160.103.10 160.103.11 160.103.12
default			160.103.10

Figure 13.1: The control system access table

### 13.3.2 Access Rights

Access rights on devices are requested by clients, when opening the connection (**importing**) to a device. All predefined rights are hierarchical. A requested access is limited by the highest possible right for a user or a group in the security database. Possible rights are:

- NO\_ACCESS : No access to the device at all.
- READ\_ACCESS : Commands which only **read** values from the device require the minimum access right READ\_ACCESS.

- **WRITE\_ACCESS** : All commands which **read and write** values require the minimum access right **WRITE\_ACCESS**.
- **SI\_WRITE\_ACCESS** : If this access right is requested, the device will be set into single user mode and all commands which require **WRITE\_ACCESS** can be executed. At the same time other clients can execute **read** commands.
- **SU\_ACCESS** : All commands which are classified as critical actions require super user (**SU\_ACCESS**) right to be executed. All read and write commands can also be executed.
- **SISU\_ACCESS** : If this access right is requested, the device will be set into single user mode and all commands which require **SU\_ACCESS** can be executed. At the same time other clients can execute **read** commands.
- **ADMIN\_ACCESS** : The **ADMIN\_ACCESS** is the highest access right. It will set the device into the single user mode and will cancel another single user session with lower access right. Even **read** commands from other clients are **blocked**.

To change the access right to a device, the device connection must be freed and afterwards reestablished with the new right.

### 13.3.3 Domain, Family or Member

Access rights on devices for users or groups have to be specified in the security database. To avoid entries for every device, the TACO device naming scheme **DOMAIN/FAMILY/MEMBER** is used to enter wide range access specifications for users or groups. Device access right entries in the security database are possible for

- **DOMAIN** = a whole area of the ESRF,
- **DOMAIN/FAMILY** = a class of devices inside a domain,
- **DOMAIN/FAMILY/MEMBER** = a single device.

Figure 2 shows an example of possible device access specifications for the device, its family or its domain.

Entry	Domaine/Family/Membre	Name	Access
user	SR/V-RV/C1-3	meyer	SU_ACCESS
user	SR/V-RV	meyer taurel	SI_WRITE_ACCESS WRITE_ACCESS
user	SR	operator meyer	WRITE_ACCESS WRITE_ACCESS
group	SR/V-RV/C1-3	dserver	ADMIN_ACCESS
group	SR/V-RV	vacuum	SISU_ACCESS
group	SR	dserver operator	WRITE_ACCESS WRITE_ACCESS
default			READ_ACCESS

Figure 13.2: The device access table

The access control system uses the following hierarchy to find the maximal access right, for a requesting client, in the database. The device can only be imported, if the requested access is lower or equal the maximal access right.

1. Verify the user entry on the device (DOMAIN/FAMILY/MEMBER).
2. If nothing was specified, verify the user entry of the device class (DOMAIN/FAMILY)
3. If nothing was specified, verify the user entry for the domain.
4. If nothing was specified, verify the group entry in three steps as mentioned in the last three points.
5. If no maximal access right was found in the user or group entries, a default value will be applied.

### 13.3.4 Verification Speed and Reliability

In contrast to the design document of the security system, the final implementation is based more on a good integration to the system than on a maximised verification speed. Experience with the first version has shown that reliability and adaptation to the general system design are more important than the highest possible verification speed. In the first version it was tried to add to a connectionless (UDP) device server, information on client connections. This kind of connection information is very hard to verify and impossible to guarantee as valid information. Out of this reason, the design had to be changed. Only the information on a single user connection was left in a device server. To make a single user connection reliable, it is always a TCP connection. A dead single user client can be detected and deadlocks avoided.

Client authentication happens only once during the import of the first device. For all other new connections only the device access must be verified. That requires one or two database requests. A security key is created on the client side after the import of a device. By verifying this key all parameters for the open client connection to a device can be guaranteed unchanged. Nothing can be modified on the connection. Parameters necessary to check the device and command access are sent to the server with every access. The parameters are checked on the server side. Sending parameters and verifying for every server access slow down the system, but is better adapted to a connectionless system and runs more reliable. Figure 3 and figure 4 show how the security key is created and how parameters are transferred.

## 13.4 Integration into TACO

The security system is created as an optional part of TACO. At startup time a resource of the central control system process (**Network Manager**) allows to suppress or add the security system. This flexibility is necessary because the security system will be implied for the machine control, but it is up to every beam line responsible to use it in the beam line control systems.

To make database access as general as possible, the resource database was reused for security data. A specially protected table (**SEC domain**) was added to avoid any overwriting of data by unauthorised persons. With this solution all available database access functions of the control system could be reused. This might be not the fastest solution. One can imagine to suppress one or two database accesses by creating a new security database and security service. But a major advantage of the current solution was the very easy maintenance of a well defined interface.

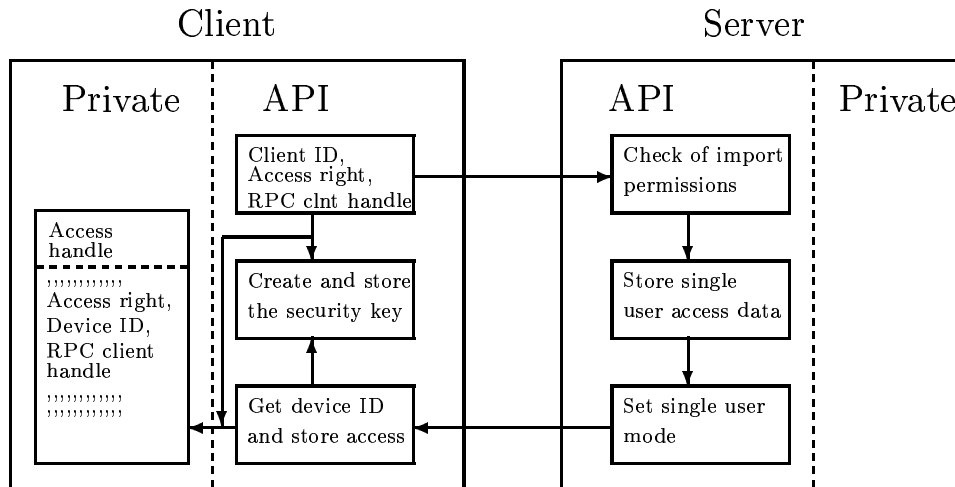


Figure 13.3: The security key creation

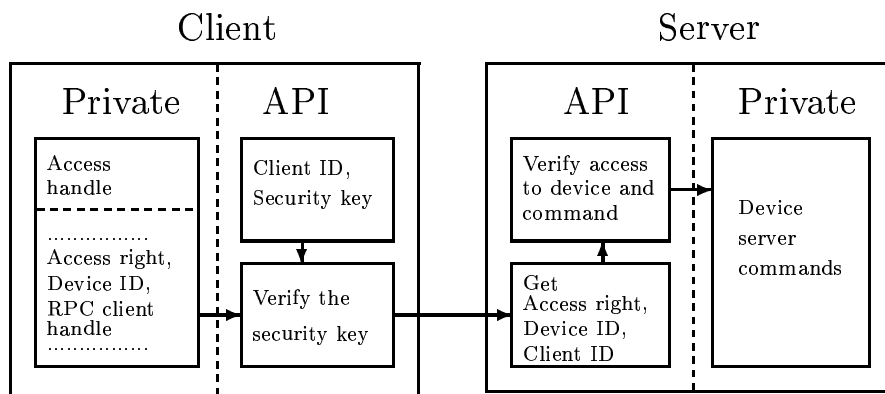


Figure 13.4: Access control with the Security Key

The main part of the security system is part of the API library, added to the import, access and free functions. Figure 5 shows the security aspects added to the API library.

## 13.5 Complex Access Handling

The device server model (ref. [2]) of TACO allows two major ways for a device server to communicate with other devices.

1. The server - server connection (figure 6)  
Device servers can communicate with devices, served by any server in the control system, via the RPC based API library functions.
2. The internal communication (figure 7)  
The device server model also allows device classes to be linked into one server process. Devices of the different classes can be exported and accessible by clients via the network. Also a fast way of internal communication exists. It



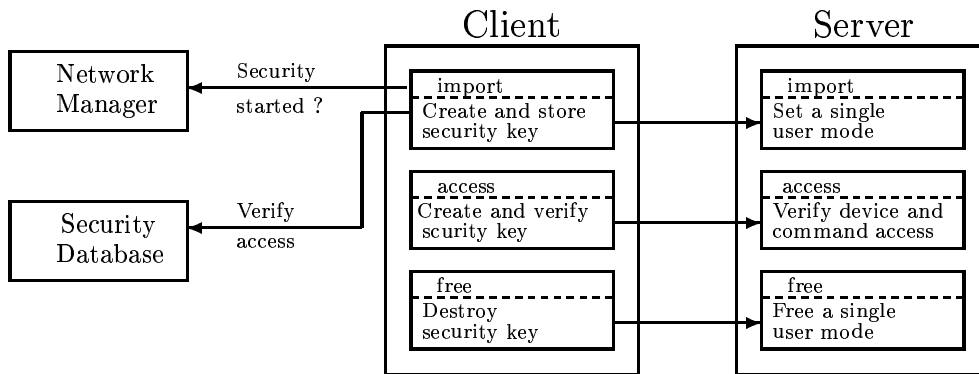


Figure 13.5: The security system integration to the API

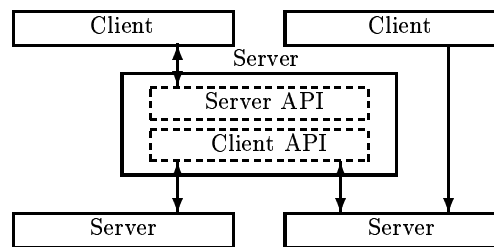


Figure 13.6: Server - server connections

uses the same import, access and free functions for internal communication without RPCs (see DSN101). Offering the same functionality as the external API. Proper access control, in the case both interfaces are open for device access, can be guaranteed in a transparent way for the user.

With the two above mentioned communication schemes access control and security are guaranteed. Only the user/group ID of a device server process must have the necessary access rights in the security database. This protects against the starting of critical device servers by unauthorised persons.

One problem remains and can only be solved by the device server programmer himself. For example:

What does a single user mode mean for a device which itself accesses two underlying devices in other servers? Do these low level devices also have to be set in single user mode or would this disturb other clients using the same low level devices? This kind of access control over hierarchical levels can not be given automatically. Needs might be different from case to case and requirements are only known to the device server programmer. The access control system can only give the tools to handle complex access hierarchies.

## 13.6 Conclusion

Access control and security in a distributed control system has been presented. Three points should be mentioned again:

(1) With the TACO device naming convention a wide range access could be implemented very easy. (2) The reuse of the resource database and its services offers a

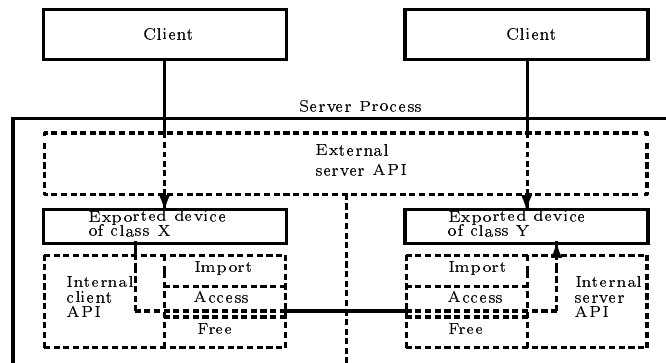


Figure 13.7: Internal and external API

well defined interface and easy maintenance of the security database. (3) Via the internal and external API, hierarchically structured access levels can be controlled. The main problem for TACO security is the **OS9** operating system which, in the currently used version, still requires **super-user rights** to execute **RPCs**. Effort still has to go into a so-called device black box. A record should be kept of the last  $n$  commands executed on a device. This record can be dumped or stored in a database for offline analysis. It enables diagnostics to be carried out in the event of device failure or crash.

### 13.6.1 The Current Implementation

Security for a control system is used if the Network Manger was started with the security option:

Manager -security

As default the security system is switched off.

If a device server exited and comes back to action, all clients which had open connections will be reconnected automatically with the device accesses they had before. During the reconnection the security database is read again and changes are applied.

To achieve proper access control in a device server, the functions **dev\_import()**, **dev\_putget()**, **dev\_put()** and **dev\_free()** must be used for internal communication as described in **DSN101**.

A single user connection is always a TCP connection. A died single user or administrator client will be detected on the next access to the server and the single user lock will be freed.

It is not possible to change the RPC protocol for a connection if a single user mode is active. When freeing a single user mode, the protocol on the connection will be set back to the initial protocol.

Tools are now available to handle security resources easily.

- To protect the SEC table in the resource database a password can be set, which will be requested on every update of the database.

sec\_passwd database\_name

No password is set on libra, to give you the chance to modify and test everything.

- To read all accesses specified for a user or a group in the security table.

```
sec_userinfo [-u user_name] [-g group_name]
```

If no user name or group name is specified, the actual login name and group accesses are listed.

- To list all users and groups which have a specified access right on a domain, a family or a member.

```
sec_objinfo domain[/family][member]
```

Attention: A list of accesses on a family will not list users or groups with the right to access the whole domain!

### 13.6.2 How to get started?

To install a device server and his clients with configured access control, three steps are necessary:

1. The minimum access right for every command of the device server has to be added to the extended command list.

```
static DevCommandListEntry commands_list[] = {
    {DevState,      dev_read_state,  D_VOID_TYPE, D_LONG_TYPE,  READ_ACCESS},
    {DevStatus,     dev_read_status, D_VOID_TYPE, D_STRING_TYPE, READ_ACCESS},
    {DevOpen,       dev_open_valve,  D_VOID_TYPE, D_VOID_TYPE,  WRITE_ACCESS},
    {DevClose,      dev_close_valve, D_VOID_TYPE, D_VOID_TYPE,  WRITE_ACCESS},
    {DevSetCalib,   dev_set_calib,   D_VAR_LONGARR, D_VOID_TYPE, SU_ACCESS},
};
```

Dangerous commands can be protected and only be executed by a client with super user rights or an administrator.

Remember:

- A device is locked in single user mode. Other clients than the single user can only access commands with the minimum access right `READ_ACCESS`.
  - Recompiling an old device server with unchanged command list will set the minimum access right for all commands to `WRITE_ACCESS`.
2. As a second step, the access control and security resources for users and groups using the device server must be set up.

```
#
# default access right, if no user or group entry can be
# found.
#
SYS/MINIMAL/ACC_RIGHT/default:  READ_ACCESS,      160.103.5,  \
                                160.103.2.132
#
# user resources for the SY domain
#
SYS/USER/ACC_RIGHT/sy:          meyer,             READ_ACCESS,  \
```

```

                                taurel,          WRITE_ACCESS
#
# user resources for device families in the SY domain
#
SYS/USER/ACC_RIGHT/sy|v-rv:      meyer,          SU_ACCESS, \
                                os9,            WRITE_ACCESS

#
# user resources for devices in the SY domain
#
SYS/USER/ACC_RIGHT/sy|v-rv|s9:  meyer,          ADMIN_ACCESS

SYS/USER/ACC_RIGHT/sy|v-rv|s2:  meyer,          ADMIN_ACCESS
#
#####
#
#
# group resources for the SY domain
#
SYS/GROUP/ACC_RIGHT/sy:         dserver,        WRITE_ACCESS, \
os9, READ_ACCESS
#
# group resources for device families in the SY domain
#
SYS/GROUP/ACC_RIGHT/sy|v-rv:     vacuum,         SU_ACCESS
#
# group resources for devices in the SY domain
#
SYS/GROUP/ACC_RIGHT/sy|v-rv|s1: dserver,        ADMIN_ACCESS
#
#####
#
# user identification information
#
SYS/USER/IDENT/meyer:           215,             160.103.5.54, \
                                160.103.2.132

SYS/USER/IDENT/taurel:          261,             160.103.2, \
                                160.103.5.68

#
# group identification information
#
SYS/GROUP/IDENT/dserver:        101,             160.103

SYS/GROUP/IDENT/vacuum:         310,             160.103.4.29

SYS/GROUP/IDENT/os9:            0,               160.103.4.218
#

```

The resources must be stored in the SEC table of the resource database. The SEC table on libra is not protected. Everybody can try and set up some resources. To avoid the total chaos when redefining the default access or some global access on a whole domain, please put your resource files in the directory:

```
libra:/users/d/dserver/dbase/res/SEC
```

Use the database tools find out the actual database contents and why an access was denied.

Specifying access control and security resources for **OS9** clients, use as pre-defined user and group name **os9** with the uid = 0 and the gid = 0. Other names are not possible, because any OS9 user must have the uid = 0 and super user rights on a crate to run a device server. The name was changed from root to **os9** to avoid conflicts with the UNIX user root.

3. The client has to request how he wants to access a device, when importing the device.

```
#include DevSec.h

char          *dev_name = "SY/V-RV/S1";
long          readwrite = WRITE_ACCESS;
devserver     pv;
long          error     = 0;

/*
 * import the device
 */
if ( dev_import (dev_name, readwrite, &pv, &error) == DS_NOTOK )
{
    return (DS_NOTOK);
}
```

For Example, the requested WRITE\_ACCESS was verified in the security database and granted. The client can execute all commands on the device which are specified with READ\_ACCESS or WRITE\_ACCESS in the command list of the device server. A command specified with SU\_ACCESS cannot be executed.

Remember:

- The access rights SL\_WRITE\_ACCESS and SL\_SU\_ACCESS will set the device into single user mode.
- Trying to import a device with SL\_WRITE\_ACCESS or SL\_SU\_ACCESS if another single user is already logged in, will return an error.
- Importing a device with ADMIN\_ACCESS if another single user is already logged in, will cancel the old single user session and set the device into administration mode.
- Importing a device with any other access right will work, but only commands which are specified in the command list for READ\_ACCESS can be executed. All other commands are locked for the time the single user is logged in.
- In **DevSec.h** a list is defined, combining the defined access rights and the rights as a string. This can be used to handle interactive input of access rights.

```
typedef struct _DevSecListEntry {
    char  *access_name;
```

```

        long  access_right;
    } DevSecListEntry;

    static DevSecListEntry DevSec_List[] = {
        {"NO_ACCESS",          NO_ACCESS},
        {"READ_ACCESS",       READ_ACCESS},
        {"WRITE_ACCESS",      WRITE_ACCESS},
        {"SI_WRITE_ACCESS",   SI_WRITE_ACCESS},
        {"SU_ACCESS",         SU_ACCESS},
        {"SI_SU_ACCESS",      SI_SU_ACCESS},
        {"ADMIN_ACCESS",      ADMIN_ACCESS},
    };
    #define SEC_LIST_LENGTH    (sizeof(DevSec_List)/sizeof(DevSecListEntry))

```

### 13.6.3 Pending Problems

Here is a list of pending problems, which will be solved in the coming releases.

- The search in the command list of a device server, for the minimum access right of a command and the command function, is not yet optimised. The command list is searched twice, because the command handler interface could not be changed for compatibility reasons.



## Chapter 14

# Standard Makefiles using GNU make (gmake)

*by A. Götz*

### 14.1 Introduction

The TACO device servers have until recently used conditional Makefiles which required processing by a program based on a mixture of lex and yacc and cpp before calling make. Although this method was well-adapted to writing Makefiles which supported multiple platforms it was non-standard and always posed a problem when moving to a new platform because it often involved porting lex and yacc as well. During the port of TACO to Linux it was decided to move to a more standard method for conditional Makefiles and adopt the GNU make tool. GNU make (sometimes called gmake) offers a wide range of facilities including conditional statements, it has been ported to a wide variety of platforms and is well-documented. This chapter describes the standard way to write GNU Makefiles for building TACO source code in general and device servers in particular.

### 14.2 Philosophy

The philosophy adopted for TACO Makefiles is to have **one** Makefile per project which supports multiple platforms as opposed to one Makefile per platform per project.

Once this philosophy is accepted there is still the choice to be made between a so-called master Makefile from which platform dependant Makefiles can be generated (using a tool like imake) or a single Makefile with conditional statements (as supported by GNU make for example) for handling platform dependencies at make time. The latter approach is the one adopted for TACO and described in this chapter.

### 14.3 GNU Make Commands

GNU make extends the standard Unix make with a number of commands. The most important of these are :

1. `ifdef variable-name [else] endif` - conditional statement which can be used to detect the presence of variable to determine which branch of the



if statement will be executed. TACO uses the conditional statement to distinguish between different platforms e.g.

```
ifdef linux
CC = gcc
endif
```

2. `ifndef variable-name [else] endif` - conditional statement which can be used to detect the absence of a variable
3. `ifeq (arg1,arg2) [ else ] endif` - test if *arg1* and *arg2* are identical (*arg1* and *arg2* are variable references)
4. `ifneq (arg1,arg2) [ else ] endif` - test if *arg1* and *arg2* are different (*arg1* and *arg2* are variable references)

In addition there are a host of string substitution and analysis functions e.g. `subst`, `strip`, `findstring`, `filter`, `sort`, as well as built-in expansion functions e.g. `dir`, `suffix`, `basename`, `join`, `wildcard` which can be used to define *arg1* and *arg2*. Refer to chapter 8 of the manual.

## 14.4 Standard Symbols

The following standard symbols should be used to identify the presence of a platform :

1. `unix` - Unix like platform (HPUX, Solaris, SUN, Linux, LynxOS)
2. `__unix__` - Unix like platform (HPUX, Solaris, SUN, Linux, LynxOS)
3. `__hpux` - HPUX running on any architecture
4. `__hpux9000s700` - HPUX running on PA-RISC1.1
5. `__hp9000s700` - HPUX running on PA-RISC1.1
6. `__solaris` - Solaris running on SPARC
7. `__solaris__` - Solaris running on SPARC
8. `linux` - Linux running on Intel 80x86
9. `lynxos` - LynxOS running on Motorola 68040
10. `__UCC` - (new) Ultra C/C++ compiler for OS9
11. `sun` - SunOS running on SPARC
12. `OSK` - (old) Unibridge compiler for OS9

## 14.5 Standard Targets

Each Makefile must have the following standard targets (generic scripts depend on them existing) :

1. `all` - make all binary targets (should be first target in Makefile so that it is taken as default)

2. `icode` - make icode versions of object files for Ultra C++/C
3. `install` - copy binaries to a common directory and update object files in library (if one exists)
4. `clean` - clean up so that a call to make will regenerate binaries
5. `clobber` - remove all binaries and make clean
6. `lock` - check out all source files (under RCS control )with lock
7. `co` - check out all source files (under RCS control ) without lock
8. `ci` - check in all source files (under RCS control ) with lock message indicating why they are being checked in (LOCKMSG="my message")

## 14.6 Scripts

To make life easier for TACO programmers a set of one-liner scripts have been defined for each platform which call gmake with the appropriate variables defined :

1. `hpuxmake` - calls gmake with `unix=1 __unix__=1 _hpux=1 _hp9000s700=1 _hpux9000s700=1`
2. `solmake` - calls gmake with `__unix__=1 unix=1 __solaris__=1 _solaris=1`
3. `ultracmake` - calls gmake with `_UCC=1`
4. `linuxmake` - call gmake with `linux=1 unix=1 __unix__=1`
5. `sunmake` - calls gmake with `__unix__=1 unix=1 sun=1`

These scripts can be found in `/users/d/dserver/make/bin` on the file server(s). `gmake` is also available as binary for all supported platforms and can be found in `/users/d/dserver/make/bin/$OS` where `$OS` stands for the operating system e.g. `s700`, `solaris`, `sun4` (`gmake` is the standard make on Linux).

For those sites running TACO who support only one platforms it would be advisable to simply define the appropriate variables for that platform in the Makefile and then call `gmake` without any arguments.

## 14.7 Example Makefile

Here is a full example of a typical Makefile to make device servers using GNU make (cf. `classes/template/simple/src/Makefile`) :

```
#
# RcsID = " $Header: /libra/users/d/dserver/doc/notes/DSN122/RCS/DSN122.tex,v 1.1 1997/01/15
#
#*****
#
# File:           Makefile
#
# Project:        <PROJECT>
#
# Description:    GNU Makefile for Template device server
#
# Author(s):      <AUTHOR>
```

```

#
# Original:          <DATE>
#
# $Log: DSN122.tex,v $
# Revision 1.1  1997/01/15 06:18:54  goetz
# Initial revision
#
#
# Copyright (c) 1996 by European Synchrotron Radiation Facility,
#                       Grenoble, France
#
#*****
#      GNU Makefile Generated by the Automatic Class Generation Tool, <REVISION>
#                               <GENERATIONDATE>.
#
#-----
#      This Makefile works with the GNU make (sometimes called gmake)
#      It makes use of the GNU make conditional statements to support
#      multiple platforms. To use this makefile for a particular platform
#      call GNU make with the appropriate symbol for that platform
#      defined e.g. "gmake __hp9000s700=1 unix=1 all". The following symbols
#      are used to identify the following platforms :
#
#      __hp9000s700      =      HPUX 9000 series 700
#      _solaris          =      Solaris
#      sun               =      SunOS
#      _UCC              =      OS9 Fastrak Ultra-C Compiler
#      unix              =      various unix flavours (Solaris, HPUX, Lynx, Linux)
#      lynx              =      LynxOS
#      Linux             =      Linux
#
#-----
#
#      The variables DSHOME is passed to the Makefile
#      as input argument or via the environment.
#
#      For UltraC use the settings for the environment variables:
#      MWOS              =      /usr/local/MWOS
#      PATH              =      $PATH:$MWOS/UNIX/bin/hp97k
#      CDEF              =      $MWOS/OS9/SRC/DEFS
#      CDEFESRF          =      /usr/local/os9/dd/DEFS
#      CLIB              =      $MWOS/OS9/LIB
#      CLIBESRF          =      /usr/local/os9/dd/LIB
#
#-----
#
#ifdef _UCC
LIB_HOME      = $(DSHOME)/lib/os9/ucc
OBS_HOME      = $(DSHOME)/lib/os9/ucc/objs
INSTALL_HOME  = $(DSHOME)/bin/os9/ucc
endif
#ifdef lynx
LIB_HOME      = $(DSHOME)/lib/lynxos

```

```

INSTALL_HOME = $(DSHOME)/bin/lynxos
endif
ifdef __hp9000s700
LIB_HOME      = $(DSHOME)/lib/s700
INSTALL_HOME = $(DSHOME)/bin/s700
endif
ifdef sun
LIB_HOME      = $(DSHOME)/lib/sun4
INSTALL_HOME = $(DSHOME)/bin/sun4
endif
ifdef _solaris
LIB_HOME      = $(DSHOME)/lib/solaris
INSTALL_HOME = $(DSHOME)/bin/solaris
endif
ifdef linux
LIB_HOME      = $(DSHOME)/lib/linux
INSTALL_HOME = $(DSHOME)/bin/linux
endif

```

```

#-----
# All include file and standard library pathes
#
#           make sure to get always the new include files
#           under ../include
#

```

```

INCLDIRS =      -I ../include \
                 -I $(DSHOME)/include \
                 -I $(DSHOME)/include/private

```

```

#-----
# All necessary compiler flags for UNIX and OS9
#
ifdef _UCC
#           The C Compiler for OS9
CC =         /usr/local/MWOS/UNIX/bin/hp97k/xcc

```

```

#           Libraries
LIBDIRS =      -L $(LIB_HOME) -L $(CLIB)

```

```

LFLAGS =       $(LIBDIRS) \
               -l dsclass \
               -l dsapi \
               -l dsxdr \
               -l dbapi \
               -l dcapi \
               -l rpclib.1 \
               -l netdb_small.1 \
               -l socklib.1 \
               -l sys_clib.1 \
               -l unix.1

```

```

ICODE_LFLAGS = $(LIBDIRS) \

```

```

-Wi,-l=$(LIB_HOME)/libdsapi.il \
-Wi,-l=$(LIB_HOME)/libdsxdr.il \
-Wi,-l=$(LIB_HOME)/libdbapi.il \
-Wi,-l=$(LIB_HOME)/libdcapi.il \
-l dsapi \
-l rpclib.l \
-l netdb.l \
-l socklib.l \
-l sys_clib.l

#                               Compiler Flags with ANSI standart for OS9
CFLAGS = -mode=c89 -i -to osk -tp 020 $(INCLDIRS)
ICODE_CFLAGS = -mode=c89 -i -j -0 7 -to osk -tp 020 $(INCLDIRS)
NAME = -o $$
endif

ifdef unix
#                               The C Compilers for UNIX
ifdef sun
CC = /usr/lang/acc
endif
ifdef _solaris
CC = /opt/SUNWspro/SC4.0/bin/cc
endif
ifdef lynx
CC = gcc
endif
ifdef __hpux
CC = /bin/cc
endif
ifdef linux
CC = gcc
endif

#                               Libraries
LIBDIRS = -L $(LIB_HOME)
ifdef _solaris
LFLAGS = $(LIBDIRS) -ldsclass -ldsapi -ldbapi -ldsxdr -ldcapi -lnsl -lsocket
else
LFLAGS = $(LIBDIRS) -ldsclass -ldsapi -ldbapi -ldsxdr -ldcapi -lm
endif

NAME = -o
endif #unix

#                               Compiler flags with ANSI standart for UNIX
ifdef __hpux
CFLAGS = -Aa -D_HPUX_SOURCE $(INCLDIRS)
endif
ifdef sun
CFLAGS = -Aa $(INCLDIRS)
endif
ifdef _solaris
CFLAGS = -Xa $(INCLDIRS)

```

```

endif
ifdef lynx
CFLAGS =      -ansi -Dlynx -Dunix -X $(INCLDIRS)
endif
ifdef linux
CFLAGS =      -ansi -Dlinux -Dunix $(INCLDIRS)
endif

```

```

#-----
# RCS options to lock and check out a version.
# Or to check in a new version.
#
#           RCS lock options
RCSLOCK =    co -l -r$(VERSION)
#           RCS check out options
RCSC0      =    co -r$(VERSION)
#           RCS check in options
RCSCI      =    ci -u -f -s"Rel" -r$(VERSION) -m"$(LOCKMSG)"

#-----
# Class library
# The object file representing the class has
# to be added to the class library.
#
CLASS_LIB  = libdsclass.a
CLASS_OBJS = Template.o

#
#-----
# All Files needed for the Server and the client
#
#           all include files
INCL       =    TemplateP.h \
                Template.h

#           source files
SRC        =    Template.c \
                startup.c \
                ps_menu.c

#           object files
SVC_OBJS   =    Template.o \
                startup.o

SVC_ICODE  =    Template.ic \
                startup.ic

CLN_OBJS   =    ps_menu.o

#-----
# What has to be made
#
#           Names of executables in the home directory

```

```

SERVER      =      Templateds
CLIENT      =      template_menu

#           Names of executables
#           and include files in the installation directories
SVC_INST    =      $(SERVER)
CLN_INST    =      $(CLIENT)
INCL_INST   =      Template.h
INCLP_INST  =      TemplateP.h

#-----
# build server and client
#

ifdef _UCC
#           Rule for making OS-9 relocatable files
.SUFFIXES: .ic .o .c
.c.ic:
$(CC) $(CFLAGS) -efe $<
.c.o:
$(CC) $(CFLAGS) -c $<

all:
$(SERVER) $(CLIENT)

$(SERVER):
$(SVC_OBJS)
$(CC) $(CFLAGS) $(NAME) $(SVC_OBJS) $(LFLAGS)

$(CLIENT):
$(CLN_OBJS)
$(CC) $(CFLAGS) $(NAME) $(CLN_OBJS) $(LFLAGS)

icode:
$(SVC_ICODE)
echo Linking with icode libraries!
$(CC) $(ICODE_CFLAGS) -o $(SERVER) $(SVC_ICODE) $(ICODE_LFLAGS)

endif

ifdef unix
all:
$(SERVER) $(CLIENT)

$(SERVER):
$(SVC_OBJS)
$(CC) $(CFLAGS) $(NAME) $@ $(SVC_OBJS) $(LFLAGS)
$(CLIENT):
$(CLN_OBJS)
$(CC) $(CFLAGS) $(NAME) $@ $(CLN_OBJS) $(LFLAGS)

endif

#           Add object file representing the class
#           to the class library.
#
$(CLASS_LIB):
$(CLASS_OBJS)
ifdef _UCC
#           For os9 all object files are kept are

```

```

#           kept in a special directory, because
#           the library has to be built by a cat
#           of all object files.
#
cp $(CLASS_OBJS) $(OBJS_HOME)
libgen -c $(OBJS_HOME)/?*.o -o=$(OBJS_HOME)/$(CLASS_LIB)
cp $(OBJS_HOME)/$(CLASS_LIB) $(LIB_HOME)
rm -rf $(OBJS_HOME)/$(CLASS_LIB)

endif
ifndef unix
ar rv $(LIB_HOME)/$(CLASS_LIB) $(CLASS_OBJS)
endif

#
#           install executables
#
ifndef _UCC
install: $(SERVER) $(CLIENT) $(CLASS_LIB)
cp $(SERVER) $(INSTALL_HOME)/$(SVC_INST)
cp $(CLIENT) $(INSTALL_HOME)/$(CLN_INST)
endif
ifndef unix
install: $(SERVER) $(CLIENT)
cp $(SERVER) $(INSTALL_HOME)/$(SVC_INST)
cp $(CLIENT) $(INSTALL_HOME)/$(CLN_INST)
endif
#
#           install include files
#
rm -f $(DSHOME)/include/$(INCL_INST)
cp ../include/$(INCL_INST) $(DSHOME)/include
chmod 664 $(DSHOME)/include/$(INCL_INST)
rm -f $(DSHOME)/include/private/$(INCLP_INST)
cp ../include/$(INCLP_INST) $(DSHOME)/include/private
chmod 664 $(DSHOME)/include/private/$(INCLP_INST)

clean:
-rm -f $(SVC_OBJS)
-rm -f $(CLN_OBJS)
-rm -f $(SVC_ICODE)
-rm -f *.i

clobber: clean
-rm -f $(SERVER)
-rm -f $(CLIENT)

lock:
$(RCSLOCK) $(SRC)
cd ../include; $(RCSLOCK) $(INCL); cd ../src

```



```
co:
    $(RCSC0) $(SRC)
    cd ../include; $(RCSC0) $(INCL); cd ../src

ci:
    $(RCSCI) $(SRC)
    cd ../include; $(RCSCI) $(INCL); cd ../src
```

## 14.8 Further Reading

1. *GNU Make* by Richard M. Stallman and Roland McGrath

## Chapter 15

# Basic steps to install and configure a device server

*by A.Götz*

1. Write your new class (e.g. `NewClass`).
2. Write the startup for the new class (`start.C`).
3. Compile and link the device server (e.g. `Newds`).
4. Create a resource file containing a list of devices to be created for a copy of the device server.<sup>1</sup> The resource file must contain at least one line which consists of the device server name followed by the keyword `device` colon and at least one device for a valid domain (e.g. TL1, SY, TL2, SR, ID, EXP at the ESRF). An example for the New class would be :

```
newds/test/device: id/new/1
```

The resource file can contain other resources which are device specific. The resource file must be stored in the resource base directory (e.g. `/users/d/dserver/dbase/res` on libra for the test control system used at the ESRF).

5. If your device server defines new commands and/or errors (cf. DSN/096) then define a class base number (e.g. `DevNewBase`) and define the commands in the resource file e.g.

```
#
# test device for the Newds device server
#
newds/test/device: id/new/1
#
# private commands
#
cmds/4/6/1: "DevNewCmd1"
```

This is all explained in the section on "Adding Private Commands".

---

<sup>1</sup>each copy of a device server has its own so-called "personal name" which is used to identify, the full server name is therefore the name of the executable followed by the personal name e.g. `Newds/test`

6. Update the resource file in the static database using the command `db_update file` (where `file` is the resource file name w.r.t to the resource base directory) or `greta` (the graphical resource editor).
7. Start the device server with the personal name specified in the resource file and the option `-m` (e.g. `Newds test -m`), make sure the environment variable `$NETHOST` is pointing to a valid control system nethost (e.g. `libra` at the ESRF).

## Chapter 16

# A tool to test a TACO control system

*by E. Taurel*

### 16.1 Introduction

**testcs** is a TACO tool built to test a control system. It is able to test from a single device server to a complete TACO control system. Testing a device server is done by sending a network request to it and waiting for the answer. It does not test the device served by the device server but only the device server ability to answer to network request. The tool takes its input directly from the TACO device server database and must run on the same computer than the database. It is available for HP-UX, SunOS and Solaris.

### 16.2 Usage

Five option are available :

- **-k** to test a TACO control system kernel servers. The kernel servers are :
  - The manager
  - The database server
  - The data collector server(s) if the control system is running with a data collector
- **-d** to test a device server. The full device server name must be specified (device server executable name/personal name)
- **-h** to test all the device server running on a specific host. The host name must be specified.
- **-a** to test a complete control system. In this case, the tool will test the kernel servers and all the device servers running on all the hosts used in the control system.
- The last option **-v** is a verbose option. This option has a meaning only with the **-k**, **-h** and **-a** options. In verbose mode, the tool displays the answer of all the tested device server. In non verbose mode, only the faulty device server are reported to the user.

If the option `-a` is used, the tool will inform you of :

- All the missing device servers which have not been started.
- All the started but dead device servers.

A manual page is available under UNIX.

### 16.3 Usage example

### 16.4 Testing a device server

Test of a running device server called PneumValves started with the personal name `sr_c02`.

```
$testcs -d pneumvalves/sr_c02
DS pneumvalves/sr_c02 : UDP version 1 ==> OK
DS pneumvalves/sr_c02 : TCP version 1 ==> OK
DS pneumvalves/sr_c02 : UDP version 4 ==> OK
DS pneumvalves/sr_c02 : TCP version 4 ==> OK
$
```

If the device server is badly killed (with a `kill -9` under UNIX or if the device server has crashed).

```
$testcs -d pneumvalves/sr_c02
DS pneumvalves/sr_c02 : UDP version 1 ==> NOK, leaving test
DS process PID found in database : 17185
$
```

If the device server is nicely killed.

```
$testcs -d pneumvalves/sr_c02
DS pneumvalves/sr_c02 defined in database on host libra but not started
$
```

If the device server is unregistered from the database (`dbset_servunreg` or `dbm_servunreg` command) or has never been started.

```
$testcs -d pneumvalves/sr_c02
Device server is not running (PN in db = 0)
$
```

If the device server is deleted from the database (`dbset_servdel` or `dbm_servdel` command)

```
$testcs -d pneumvalves/sr_c02
Device server not defined in database
$
```

### 16.5 Testing control system kernel servers

Example of the `testcs` answer started with option `-k` and `-v` on the ESRF machine control system

```
$ testcs -k -v
Manager : UDP version 1 ==> OK
Manager : UDP version 4 ==> OK
Database server : UDP version 1 ==> OK
Database server : UDP version 2 ==> OK
Database server : UDP version 3 ==> OK
Database server : TCP version 1 ==> OK
Database server : TCP version 2 ==> OK
Database server : TCP version 3 ==> OK
Data collector read server 1 on gemini : TCP version 1 ==> OK
Data collector read server 1 on gemini : UDP version 1 ==> OK
Data collector read server 2 on gemini : TCP version 1 ==> OK
Data collector read server 2 on gemini : UDP version 1 ==> OK
Data collector read server 3 on gemini : TCP version 1 ==> OK
Data collector read server 3 on gemini : UDP version 1 ==> OK
Data collector read server 4 on gemini : TCP version 1 ==> OK
Data collector read server 4 on gemini : UDP version 1 ==> OK
Data collector read server 5 on gemini : TCP version 1 ==> OK
Data collector read server 5 on gemini : UDP version 1 ==> OK
Data collector write server 1 on gemini : TCP version 1 ==> OK
Data collector write server 1 on gemini : UDP version 1 ==> OK
Data collector write server 2 on gemini : TCP version 1 ==> OK
Data collector write server 2 on gemini : UDP version 1 ==> OK
Data collector write server 3 on gemini : TCP version 1 ==> OK
Data collector write server 3 on gemini : UDP version 1 ==> OK
Data collector write server 4 on gemini : TCP version 1 ==> OK
Data collector write server 4 on gemini : UDP version 1 ==> OK
Data collector read server 1 on aries : TCP version 1 ==> OK
Data collector read server 1 on aries : UDP version 1 ==> OK
Data collector read server 2 on aries : TCP version 1 ==> OK
Data collector read server 2 on aries : UDP version 1 ==> OK
Data collector read server 3 on aries : TCP version 1 ==> OK
Data collector read server 3 on aries : UDP version 1 ==> OK
Data collector read server 4 on aries : TCP version 1 ==> OK
Data collector read server 4 on aries : UDP version 1 ==> OK
Data collector read server 5 on aries : TCP version 1 ==> OK
Data collector read server 5 on aries : UDP version 1 ==> OK
Data collector write server 1 on aries : TCP version 1 ==> OK
Data collector write server 1 on aries : UDP version 1 ==> OK
Data collector write server 2 on aries : TCP version 1 ==> OK
Data collector write server 2 on aries : UDP version 1 ==> OK
Data collector write server 3 on aries : TCP version 1 ==> OK
Data collector write server 3 on aries : UDP version 1 ==> OK
Data collector write server 4 on aries : TCP version 1 ==> OK
Data collector write server 4 on aries : UDP version 1 ==> OK
$
```

## 16.6 Testing all the device server running on a host

This is a copy of the output of testcs started with the -h and -v option for one of the ESRF machine control system VME

```

$ testcs -h vme006 -v
Test host : vme006
DS plc/sy_s678 and pneumvalves/sy_s678 : UDP version 1 ==> OK
DS plc/sy_s678 and pneumvalves/sy_s678 : TCP version 1 ==> OK
DS plc/sy_s678 and pneumvalves/sy_s678 : UDP version 4 ==> OK
DS plc/sy_s678 and pneumvalves/sy_s678 : TCP version 4 ==> OK
DS ripc/sy_s678 and ripc-channel/sy-s678 : UDP version 1 ==> OK
DS ripc/sy_s678 and ripc-channel/sy-s678 : TCP version 1 ==> OK
DS arun/sy_s678 and pg_arun/sy_s678 : UDP version 1 ==> OK
DS arun/sy_s678 and pg_arun/sy_s678 : TCP version 1 ==> OK
DS arun/sy_s678 and pg_arun/sy_s678 : UDP version 4 ==> OK
DS arun/sy_s678 and pg_arun/sy_s678 : TCP version 4 ==> OK
DS magvaccoolingilds/sy and cellmagil/sy : UDP version 1 ==> OK
DS magvaccoolingilds/sy and cellmagil/sy : TCP version 1 ==> OK
DS thctrl/sy and srthc/sy : UDP version 1 ==> OK
DS thctrl/sy and srthc/sy : TCP version 1 ==> OK
DS thctrl/sy and srthc/sy : UDP version 4 ==> OK
DS thctrl/sy and srthc/sy : TCP version 4 ==> OK
$

```

On this output, you can remark that device server with several embedded classes are tested as one server (plc/sy\_s678 and pneumvalves/sy\_s678 are part of the same device server process). It is also possible to detect old device server which are registered in the RPC layers with version 1 only (ripc/sy\_s678 and magvaccoolingilds servers).

## 16.7 Testing a complete control system

The following is a result of testcs started on a ESRF beam line control system with the -a option

```

$ testcs -a
Testing control system kernel components
Getting information from the whole control system
On large control system, this may needs time !
Getting information for : id101
Getting information for : id102
Getting information for : id106
Getting information for : tina
Control system with 34 server process(s) distributed on 4 host(s)
Testing device server(s) running on id101
Testing device server(s) running on id102
DS gpib/dummy and mcamb/id10 : UDP version1 ==> NOK !!!!!
DS process PID found in database : 66
DS wxbpm/mcd defined in database on host id102 but not started
Testing device server(s) running on id106
Testing device server(s) running on tina
DS ud_daemon/ud_atte defined in database on host tina but not started
$

```

This example does not use the verbose mode of testcs. From the output, you can conclude that

- All the kernel components are running well (manager, database server and data collector).

- The control system is distributed on 4 hosts and uses 34 device servers.
- The deice server gpib/dummy is not running
- The device servers wxbpm/mcd and ud\_daemon/ud\_atte have not been started.





## Chapter 17

# Adding Private Commands, Errors and XDR Data Types

*by J.Meyer and A.Götz*

### 17.1 Introduction

For more flexible and memory saving architecture, commands, errors and XDR data types are treated as follows :

1. error strings can be generated dynamically by the server and returned to the client as part of the `dev_putget()` call.
2. in addition strings can be stored as resources in the resource database.
3. a split up of the command and error numbers into several fields allows private specifications for a device server.
4. there is a small kernel of general XDR data types which has to be linked to every device server or client. All other data types are declared private and must be explicitly loaded in a server or client process.

### 17.2 Dynamic Errors

TACO V8.18 supports dynamic error strings. This means error strings can be generated dynamically by the server and returned to the client using the `dev_error_push()` call. This allows for much more flexible error treatment e.g. errors can be generated in situ with very clear dynamically generated text indicating the exact error. Error messages can be stacked on the server side to indicate the device or class where the error was first detected. NOTE: when using dynamic error strings the error code is ignored when retrieving the error string (obviously) but the client can still use it to detect the type of error. For more details see the DSAPI section of this manual. Example of using `dev_error_push()` :

```
long MyClass::my_cmd(MyClass my_device, void *vargin, void *vargout, long *error);
{
static char error_str[256];
long argin;

argin = *(long*)vargin;
```

```

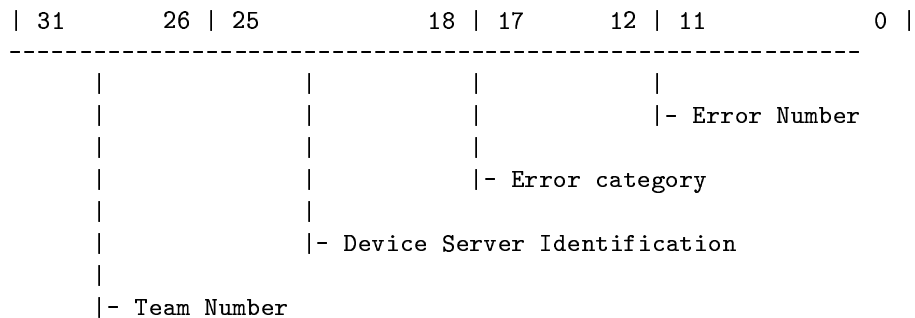
if (argin > my_device.maximum)
{
    sprintf("MyClass::my_cmd(): argin = %d exceeds maximum value allowed (max=%d)\n",
    argin, my_device.maximum);
    dev_error_push(error_str);
    *error = DevErr_CommandFailed;
    return(DS_NOTOK);
}

.
.
.
}

```

### 17.3 Error Numbers

The error number, defined as a long word, is split into four different fields:



- Team Number:

A unique number which is assigned to each developer team in the include file **DserverTeams.h**. These numbers are managed by the machine control team.

```

#ifndef _DserverTeams_h
#define _DserverTeams_h

/*
 * Definitions to code and decode the error and command numbers.
 */

#define DS_TEAM_SHIFT      26
#define DS_IDENT_SHIFT    18

#define DS_TEAM_MASK      0x3f
#define DS_IDENT_MASK     0xff

/***** Device server development Teams definitions *****/

#define CntrlTeamNumber    (1 << DS_TEAM_SHIFT) /* CS - Machine Control */
#define DasTeamNumber      (2 << DS_TEAM_SHIFT) /* CS - Data Acquisition */
#define ProgTeamNumber     (3 << DS_TEAM_SHIFT) /* Experiments -Programming */
#define CrgTeamNumber      (4 << DS_TEAM_SHIFT) /* External - CRG */
#define BlcTeamNumber      (5 << DS_TEAM_SHIFT) /* CS - Beam Line Control */

```

```
#endif /* _DserverTeams_h */
```

- Device Server Identification:

A uniq number to identify a device server class and its private definitions. These numbers will be managed and assigned inside the programming teams. Example (DasDsNumbers.h):

```
#ifndef _DasDsNumbers_h
#define _DasDsNumbers_h

#include <DserverTeams.h>

/* ESRF-VDL */
#define DevVdlBase DasTeamNumber + (1 << DS_IDENT_SHIFT)
/* ELTEC-IC40 */
#define DevIpcBase DasTeamNumber + (2 << DS_IDENT_SHIFT)
/* NOVELEC-MCCE */
#define DevMcceBase DasTeamNumber + (3 << DS_IDENT_SHIFT)
/* ESRF - SKELETON */
#define DevSkelBase DasTeamNumber + (4 << DS_IDENT_SHIFT)
/* LECROY 1151 - COUNTER*/
#define DevCntBase DasTeamNumber + (5 << DS_IDENT_SHIFT)
/* ESRF - TDC CI022 */
#define DevTdcBase DasTeamNumber + (6 << DS_IDENT_SHIFT)
/* CAEN V462 - GATEGEN */
#define DevGategenBase DasTeamNumber + (7 << DS_IDENT_SHIFT)
/* ADAS ICV101 - ADC */
#define DevAdcicv101Base DasTeamNumber + (8 << DS_IDENT_SHIFT)
/* EC740 TFG */
#define DevTfgBase DasTeamNumber + (9 << DS_IDENT_SHIFT)
/* EC738 MCS */
#define DevMcsBase DasTeamNumber + (10 << DS_IDENT_SHIFT)
/* VVHIST */
#define DevHcBase DasTeamNumber + (11 << DS_IDENT_SHIFT)
/* HM - MM6326 */
#define DevHmBase DasTeamNumber + (12 << DS_IDENT_SHIFT)
/* Current Transformer */
#define DevCtBase DasTeamNumber + (13 << DS_IDENT_SHIFT)

#endif /* _DasDsNumbers_h */
```

- Error Category:

Not yet used.

Reserved for a future classification of error messages.

- Error Number:

The original error number to identify the error.

## 17.4 Command Numbers

The command number, defined as a long word, is split into three different fields:

31	26   25	18   17	0
-----			



## 17.7 The restructured XDR concept

In the last version all available XDR data types were known to servers and clients. This growing list was abandoned in the new release (version 3.30). It is replaced by a small kernel of general purpose data types and a dynamic list, which can hold private XDR data types used by servers or clients.

The set of data types in the kernel is always available and automatically loaded. All other XDR data types that should be used, must be explicitly loaded at startup time of a server or client.

The implemented general purpose data types are:

1. D\_VOID\_TYPE
2. D\_BOOLEAN\_TYPE
3. D\_SHORT\_TYPE
4. D\_LONG\_TYPE
5. D\_FLOAT\_TYPE
6. D\_DOUBLE\_TYPE
7. D\_STRING\_TYPE
8. D\_INT\_FLOAT\_TYPE
9. D\_FLOAT\_READPOINT
10. D\_STATE\_FLOAT\_READPOINT
11. D\_LONG\_READPOINT
12. D\_DOUBLE\_READPOINT
13. D\_VAR\_CHARARR
14. D\_VAR\_STRINGARR
15. D\_VAR\_SHORTARR
16. D\_VAR\_LONGARR
17. D\_VAR\_ULONGARR
18. D\_VAR\_FLOATARR
19. D\_VAR\_DOUBLEARR
20. D\_VAR\_FRPARR - Float Readpoint Array
21. D\_VAR\_LRPARR - Long Readpoint Array
22. D\_OPAQUE\_TYPE - Block of Bytes

To recompile your old software, which might use other XDR data types as the ones mentioned in the above list, you have two possibilities.

1. To **change the code** and load all necessary XDR descriptions as described in the next section. Like this you will link only with the XDR functions you really need. The size of the executable will reduce.

2. To **change the include files** (see section 7.4) and the Makefile to link with the library

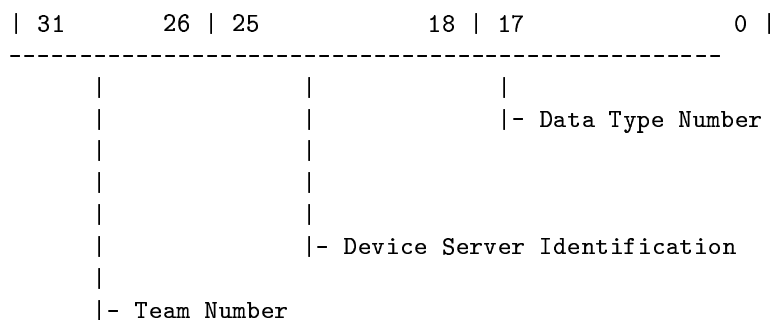
**libdsxdr\_all.a or os9-dsxdr\_alllib.l**

which will load all XDR data types known in the last versions up to 3.29.

## 17.8 Private XDR Data Types

### 17.8.1 Data Type Numbers

The data type number, defined as a long word, is split into three different fields:



The distribution of **Team Number** and **Device Server Identification** is the same as described in section 2.

### 17.8.2 What is a Complete XDR Data Type Definition?

A XDR data type definition consists of a **.h** and a **.c** file. In the include file are the C type definition, the declaration of the XDR functions, the declaration of the XDR length calculation functions (for the data collector), the definition for the data type number and the definition of the load macro.

Example (ct\_xdr.h):

```
#include <DasDsNumbers.h>

/*
 * definitions for current transformer data type
 */

struct DevCtIntLifeTime {
    float DeltaIntensity; /* delta-intensity for this measure */
    float LifeTime;      /* value of the life-time */
    long  DateTicks;      /* date in ticks since midnight */
    long  DeltaTused;     /* delta-T used for calculations */
};

typedef struct DevCtIntLifeTime DevCtIntLifeTime;
/* The declaration for the xdr function */
bool_t xdr_DevCtIntLifeTime ();
/* The declaration for the xdr length calculation function */
long xdr_length_DevCtIntLifeTime ();
```

```

struct DevVarCtIntLifeTimeArray {
    u_int          length;
    DevCtIntLifeTime *sequence;
};
typedef struct DevVarCtIntLifeTimeArray DevVarCtIntLifeTimeArray;
/* The declaration for the xdr function */
bool_t          xdr_DevVarCtIntLifeTimeArray ();

/* The declaration for the xdr length calculation function */
long            xdr_length_DevVarCtIntLifeTimeArray ();

/* The definition of the data type number */
#define D_CT_LIFETIME          DevCtBase + 1

/* The definition of the load macro */
#define LOAD_CT_LIFETIME(A)      xdr_load_type ( D_CT_LIFETIME, \
                                                xdr_DevVarCtIntLifeTimeArray, \
                                                sizeof(DevVarCtIntLifeTimeArray), \
                                                xdr_length_DevVarCtIntLifeTimeArray, \
                                                A )

```

The `.c` file contains the XDR functions and the XDR length calculation functions for the data type.

More information on how to write a XDR function can be found in the HP, SUN or OS9 documentation of **NFS/RPC**. In addition to the standard XDR functions, all translation functions of the defined general purpose data types can be reused. The XDR length calculation functions are structured in the same way as the XDR functions. The length of each structure field has to be summed up to find the length of the structure in XDR format. Reusable XDR length calculation functions are available for all defined general purpose data types.

Example (ct\_xdr.c):

```

#include <dev_xdr.h>
#include <ct_xdr.h>

bool_t
xdr_DevCtIntLifeTime (xdrs, objp)
    XDR *xdrs;
    DevCtIntLifeTime *objp;
{
    if (!xdr_float(xdrs, &objp->DeltaIntensity)) {
        return (FALSE);
    }
    if (!xdr_float(xdrs, &objp->LifeTime)) {
        return (FALSE);
    }
    if (!xdr_long(xdrs, &objp->DateTicks)) {
        return (FALSE);
    }
    if (!xdr_long(xdrs, &objp->DeltaTused)) {
        return (FALSE);
    }
    return (TRUE);
}

```



```

}

long
xdr_length_DevCtIntLifeTime(objp)
    DevCtIntLifeTime *objp;
{
    long    length = 0;

    length = length + xdr_length_DevFloat (&objp->DeltaIntensity);
    length = length + xdr_length_DevFloat (&objp->LifeTime);
    length = length + xdr_length_DevLong  (&objp->DateTicks);
    length = length + xdr_length_DevLong  (&objp->DeltaTused);

    return (length);
}

bool_t
xdr_DevVarCtIntLifeTimeArray(xdrs, objp)
    XDR *xdrs;
    DevVarCtIntLifeTimeArray *objp;
{
    if (!xdr_array(xdrs, (char **)&objp->sequence,
        (u_int *)&objp->length, ~0, sizeof(DevCtIntLifeTime),
        xdr_DevCtIntLifeTime)) {
        return (FALSE);
    }
    return (TRUE);
}

long
xdr_length_DevVarCtIntLifeTimeArray (objp)
    DevVarCtIntLifeTimeArray *objp;
{
    long    length = 0;

    /*
     * four bytes for the number of array elements
     */

    length = length + xdr_length_DevLong (&objp->length);

    /*
     * now calculate the length of the array
     */

    length = length + (objp->length *
        xdr_length_DevCtIntLifeTime(&objp->sequence[0]) );

    return (length);
}

```

### 17.8.3 How to Integrate a New Data Type?

The integration of a new, private XDR data type must be done in two steps. First, the load macro of the data type must be called once at startup time of a server or a client. The best place in a device server is the method **DevMethodClassInitialise** to execute all necessary load macros. In a client the same macros have to be executed before the data types are used.

Example:

```
long *error;

.....

if ( LOAD_CT_LIFETIME(error) == DS_NOTOK )
{
    return (DS_NOTOK);
}

.....
```

Second, the XDR functions of the data type must be linked to server and client. This should be done locally first to test the data transfer. Afterwards the new XDR data type can be used completely local for server and client, or can be integrated to the XDR library. To make the data type visible to other clients who want to use the service.

### 17.8.4 Available Data Types

The XDR library contains the data type for the kernel as described in section 6 and a number of hardware specific data types. Here is a list of all data types not referenced in the kernel and their include files with the type definitions.

1. bpm\_xdr.h : D\_BPM\_POSFIELD, D\_BPM\_ELECFIELD
2. ws\_xdr.h : D\_WS\_BEAMFITPARM
3. vgc\_xdr.h : D\_VGC\_STATUS, D\_VGC\_GAUGE, D\_VGC\_CONTROLLER
4. ram\_xdr.h : D\_NEG\_STATUS, D\_RAD\_DOSE\_VALUE
5. thc\_xdr.h : D\_VAR\_THARR, D\_LIEN\_STATE
6. haz\_xdr.h : D\_HAZ\_STATUS
7. vrif\_xdr.h : D\_VRIF\_WDOG, D\_VRIF\_STATUS, D\_VRIF\_POWERSTATUS
8. gpib\_xdr.h : D\_GPIB\_WRITE, D\_GPIB\_MUL\_WRITE, D\_GPIB\_RES, D\_GPIB\_LOC
9. bpss\_xdr.h : D\_BPSS\_STATE, D\_BPSS\_READPOINT, D\_BPSS\_LINE, D\_STATE\_INDIC
10. pss\_xdr.h : D\_PSS\_STATUS
11. rf\_xdr.h : D\_RF\_SIGCONFIG
12. ct\_xdr.h : D\_CT\_LIFETIME
13. daemon\_xdr.h : D\_DAEMON\_STATUS, D\_DAEMON\_DATA
14. seism\_xdr.h : D\_SEISM\_EVENT, D\_SEISM\_STAT

15. `slit_xdr.h` : `D_BLADE_STATE`, `D_PSLIT_STATE`
16. `atte_xdr.h` : `D_ATTE_TYPE`
17. `maxe_xdr.h` : `D_MULMOVE_TYPE`, `D_MOTOR_LONG`, `D_MOTOR_FLOAT`
18. `icv101_xdr.h` : `D_VAR_PATTERNARR`, `D_ICV_MODE`
19. `mstat_xdr.h` : `D_VAR_MSTATARR`
20. `m4_xdr.h` : `D_VAR_LONGFLOATSET`, `D_VAR_LONGREAD`, `D_VAR_POSREAD`,  
`D_VAR_AXERREAD`, `D_VAR_PARREAD`, `D_VAR_ERRREAD`
21. `grp_xdr.h` : `D_GRPFP_TYPE`
22. `pin_xdr.h` : `D_PINSTATE_TYPE`

## 17.9 Numbering Limits

Due to the length of the bit fields in an error or command number the numbering limits are:

Bit Field	Bits	Possible Numbers
Team Number	6	0 - 63
DS Identification	8	0 - 255
Error Category	6	0 - 63
Error Number	12	0 - 4095
Command Number	18	0 - 262143
XDR Data Type Number	18	0 - 262143

### 17.9.1 Master Copies

The master copy of all sources can be found under the path

`DSHOME=libra:/users/d/dserver`

Important files and pathes are:

- `$DSHOME/include/DserverTeams.h`  
Containing all predefined programming team numbers.
- `$DSHOME/include/CntrlDsNumbers.h`  
Containing the machine control groups's device server identifications.
- `$DSHOME/include/DasDsNumbers.h`  
Containing the data acquisition groups's device server identifications.
- `$DSHOME/include/BlcDsNumbers.h`  
Containing the beam line control groups's device server identifications.
- `$DSHOME/system/api/cmds_err/res/dev_errors.res`  
Containing all error default error strings, which have to be loaded into the resource database. The database table `ERROR` must be defined!
- `$DSHOME/system/api/cmds_err/res/dev_cmds.res`  
Containing all default command name strings, which have to be loaded into the resource database. The database table `CMDS` must be defined!

- \$DSHOME/dev/system/xdr  
The subdirectories include and src contain all **.h** and **.c** files for the XDR data types which are available in the XDR library **libdsxdr.a**.
- libdsxdr\_all.a or os9-dsxdr\_alllib.l  
The version of the API-library which loads automatically all XDR data types which were available up to version 3.29.

## 17.10 Conclusion

The new versions of the API-and XDR-library, give the possibility to define private commands, errors and XDR data types. The only condition is to respect the correct Team\_Number and DS\_Identification for definitions and the resource pathes.

**Attention:**

If the numbering scheme is not respected resources of other classes or general resource definitions will be deleted. The ERROR and CMD tables in the resource database are not yet protected.

Despite private definitions, the wheel should not be reinvented. Errors and commands should be reused as long as an appropriate definition can be found in the general files DevErrors.h and DevCmds.h.

Also, first try to reuse already existing XDR data types before creating new ones. In 80% of all cases the general purpose data types are sufficient.



## Chapter 18

# Interfaces

TACO has been interfaced to a number of other languages and programs. The main interfaces are C and C++ and are described in a separate chapter. In addition to these two languages the following languages/programs have been interfaced to TACO :

- **Python** - contact Jens Meyer ([meyer@esrf.fr](mailto:meyer@esrf.fr)) or Marie-Christine Dominguez ([domingue@esrf.fr](mailto:domingue@esrf.fr))
- **Tcl** - contact Gilbert Pepellin ([pepellin@esrf.fr](mailto:pepellin@esrf.fr))
- **MathLab** - contact Laurent Farvacque ([laurent@esrf.fr](mailto:laurent@esrf.fr)) or Francis Epaud ([epaud@esrf.fr](mailto:epaud@esrf.fr))
- **LabView** - contact Andy Götz ([goetz@esrf.fr](mailto:goetz@esrf.fr))
- **SPEC** - contact Gerry Swislow ([info@certif.com](mailto:info@certif.com))

For more information refer to the website or contact the person involved directly.



# Appendix A

## Licence

TACO is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

### GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc.  
59 Temple Place - Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

#### Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute



copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

#### TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you

may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- \* a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- \* b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- \* c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- \* a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- \* b) Accompany it with a written offer, valid for at least three years,

to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

- \* c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not

excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

## NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS