TACO Manual

version 2.1

April 2002

Abstract

TACO is a toolkit for implementing distributed object oriented control systems originally developed at the European Synchrotron Radiation Facility (ESRF)[1] 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 accesses 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 persistant information and keeping track of where devices are running and an archiving database (based on Oracle). It is also possible to run TACO without the database (useful for embedded devices and very simple control systems). 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.

[1] http://www.esrf.fr

Contents

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[1] in Munich (Germany) to control the beamlines and at the Hartebeesthoek Radio Astronomy Observatory (HartRAO)[2] 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 accesses 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 persistant 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 uptodate on a regular basis and should be useful to new and experienced users of TACO.
TACO can be downloaded from the TACO website[3] 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 :

[1] http://www.frmii.de
[2] http://www.hartrao.ac.za
[3] 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 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.

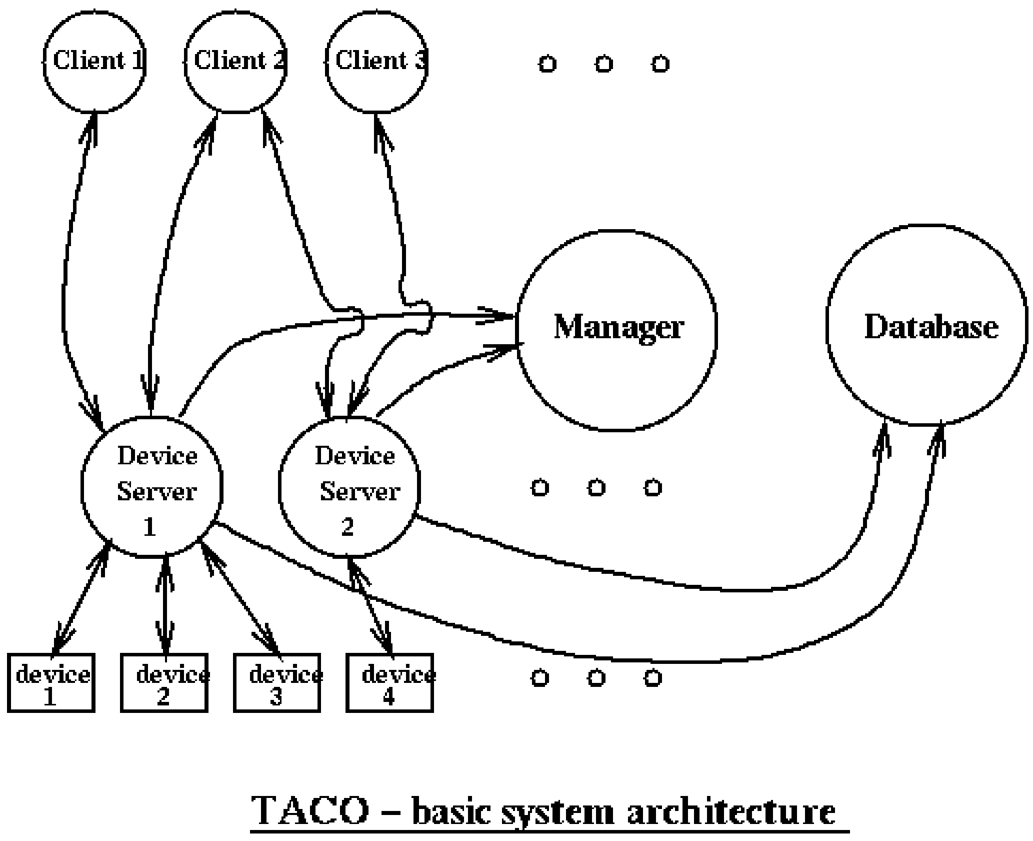


Figure 2.1: TACO system architecture

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 diiferent 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 a 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.

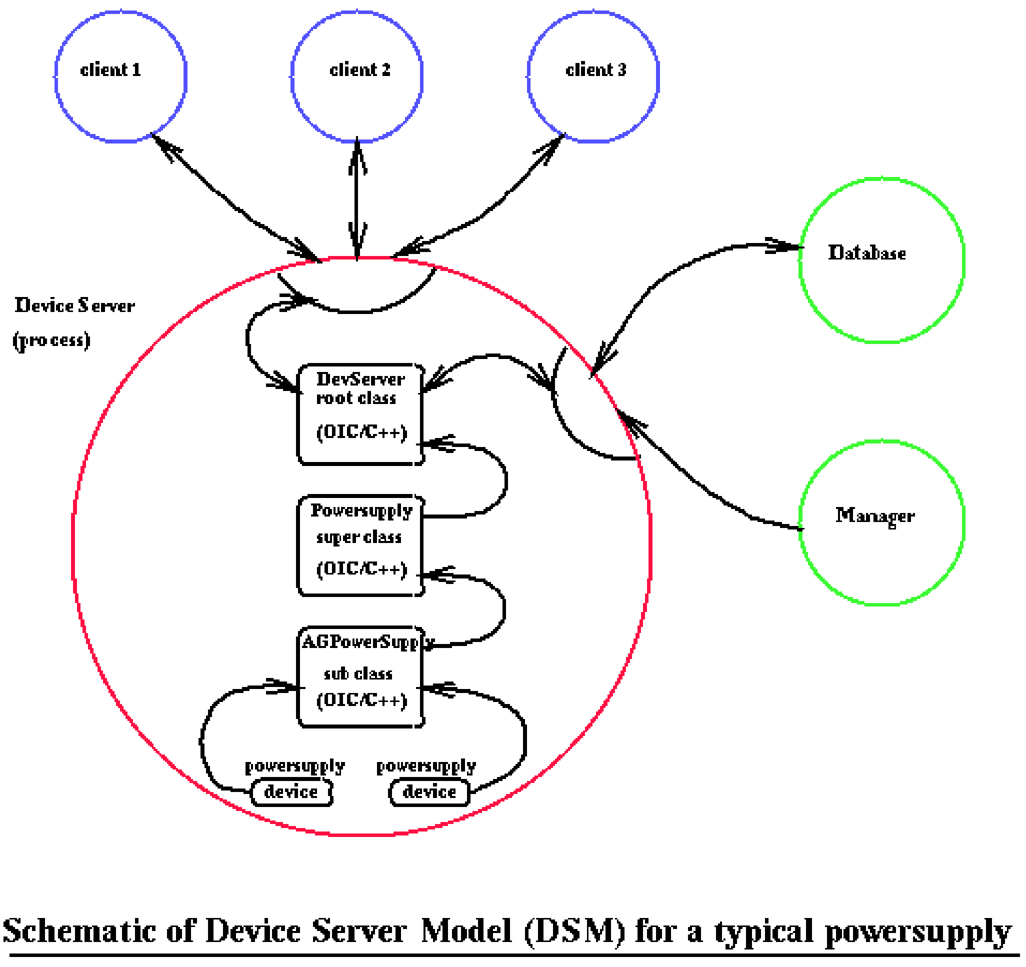


Figure 2.2: TACO Device Server Model for a typical PowerSupply

Chapter 3

Changes

Here is a list of changes in the TACO manual :

- **V2.1**
  - added chapter on "TACO Lite"
- **V2.0**
  - added chapter on "Device Servers in C"
  - added chapter on "Labview and TACO"
  - added chapter on "Python and TACO"
  - added appendix of "Device Server Catalog"
- **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
- Markhu 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) - rtdb, 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 -xzvf 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‘
   myds 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 povided 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 and 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 incldue 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 fundemental 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

TACO for Windows *by W-D.Klotz and A.Götz*

7.1 Introduction

TACO for Windows exists since November 1997. It is being used more and more to interface detector systems, OPC based SCADA system and other software running under Windows. The original Windows port was done using the version V5.15 of the DSAPI of TACO. It has been recently updated with the latest versions of all the libraries - DSAPI=V8.29, DBAPI=V6.12, and DSXDR=V5.20. The main difference is that TACO on Windows now supports asynchronism and events. It is therefore fully compatible with the most recent versions of TACO for Windows. A new port of TACO C++ device server library to Windows is on the way and will hopefully be finished before the end of November 2001. This chapter describes the V8.29 port of the C version DSAPI and associated TACO libraries to Windows. It supports writing TACO device servers and clients under Windows 95/98/NT and 2000. A port of the ndbm version of the TACO database server and manager exists but is not available as package. It can be downloaded on request (send an email to `taco@esrf.fr`). The following text describes how to download and install the Windows port of TACO and an example device server.
This document describes how to install and use the device server libraries (libdsapi.lib, libdbapi.lib, libdsxdr.lib) and the ONC RPC (oncrpc.lib) on Windows NT and Windows 95/98. Once installed on your target computer, you can develop TACO device servers and TACO clients written in ANSI C.
The libraries have been developed and tested on Windows 95/98 and Windows NT4.0. With the exception of the ONC RPC library, libraries are created as static libraries. All libraries have been compiled with MS Visual C++ Version 6.0, and are delivered as Release versions.
The libraries are distributed in two packages:

1. Binary, comprising the libraries, header files and a sample client/server application with sources, project- and makefiles.

2. Source, comprising in addition to the binary distribution all source code, that allows you to rebuild the libraries yourself on your target system.

All executables and libraries in this distribution have been compiled as Win32-Release versions. For details on compiler flags etc. you have to look into the corresponding makefiles, i.e. files with the extension .mak.
The current libraries are based on DSAPI revision 8.29

The zip file for the TACO Windows release can be found on our ftp server[1]. Additional archives mentioned below are also available on our ftp server[2]. Additional products that you need The Taco device server libraries are based on SUN's ONC RPC. This RPC has been ported to 32 bit Windows starting from the original source code, which is freely available for UNIX from SUN. The library is named oncrpc.lib and linked into a dynamic link library oncrpc.dll. It is based on the Windows Socket definition from MicroSoft. You can get the tested ONC RPC library from our ftp server. The C/C++ compiler used to build this release was Microsoft Visual C++ version 6.0. For the Text-to-Speech sample server you need Microsoft's Speech SDK and an additional DDE server (TTSApp.exe). Both are available from our ftp server.

7.2 Installation

All files are bundled in a single zip archive (16 MB). You have to extract files from the archive. During extraction files and directories will be created relative to the directories /taco/dbase and /taco/dserver. If you do not extract to the root directory of your current disk, you have to change path-specifications in the corresponding makefiles or project files. To avoid this work, it is strongly recommended to install everything in the root directory! Doing so you will get the following directory tree structure:

```
C:\TACO\DBASE
+---res
    +---clnt
    +---svc
    +   +---rtdb
    +---win32
    +   +---Debug
    +   +---Release
    +---include

C:\TACO\DSERVER
+---dev
+   +---classes
+   +   +---main
+   +   +   +---src
+   +   +   +---include
+   +---system
+   +   +---api
+   +   +   +---admin
+   +   +   +   +---include
+   +   +   +---apilib
+   +   +   +   +---include
+   +   +   +   +---src
+   +   +   +   +---win32
+   +   +   +   +   +---Debug
+   +   +   +   +   +---Release
+   +   +   +---cmds_err
+   +   +       +---include
+   +   +       +---res
```

[1]ftp://ftp.esrf.fr/pub/cs/taco/taco_win32_v8_29.zip
[2]ftp://ftp.esrf.fr/pub/cs/taco/dsapiNT/

```
+   +   +       +---src
+   +   +---xdr
+   +   +   +---include
+   +   +   +---src
+   +   +   +---win32
+   +   +       +---Debug
+   +   +       +---Release
+   +   +---dc
+   +   +   +---include
+---include
+---classes
+   +---powersupply
+   +   +---ag
+   +   +   +---include
+   +   +   +---src
+   +   +   +---win32
+   +   +       +---ps_menu
+   +   +       +   +---Release
+   +   +       +---Release
+   +   +---src
+   +   +---include
+   +---TextTalker
+       +---Release
+       +---src
+       +---include
+       +---TextTalker_menu
+           +---Release
+---lib
+   +---win32
+       +---Debug
+       +---Release
```

7.3 Binary distribution

If you plan to develop new device servers and client applications only, you should use the binary distribution. To get the binary distribution, you have to extract the following directories with all their subdirectories:

- taco/dserver/classes
- taco/dserver/include
- taco/dserver/lib

  In the directory /taco/dserver/lib/win32/Release you will find the files:
- DSMain.res a resource file that has(!) to be linked with every device server (do not modify it!);
- libdbapi.lib a library to access the static data base;
- libdsapi.lib the main DSAPI library;
- libdsxdr.lib a library with XDR filter routines.
- libtts.lib a library used by the TextTalker sample application.

- oncrpc.lib the ONC RPC import library.
- oncrpc.dll the ONC RPC dll.

In the directory /taco/dserver/lib/win32/Debug you find the same files as WIN32 Debug versions. The directory /taco/dserver/include comprises all .h header files of this release. Building the AGPowersupply sample client/server pair. In the directory dserver/classes/powersupply/ag/win32 you will find

- Agpsds.mak the common makefile , and
- Agpsds.dsw the common project file
- Agpsds.hpj the help project file to genereate Agpsds.hlp.

for the ag-powersupply device server and the ag-powersupply menu client.
Look into the Win32 Release configuration of the makefile if you want to understand how to set compiler flags when you build a server or a client. When you compile the source files on NT or Windows 95/98 you have to define the preprocessor macros _NT, WIN32,_WINDOWS.
In the directories

- taco/dserver/classes/powersupply/ag/win32/Release
- taco/dserver/classes/powersupply/ag/win32/ps_menu/Release

you will find a ready to run server - AGpsds.exe, and a client that knows this server's commands - ps_menu.exe.
Hint: To satisfy the precompiler on my machine, I had to set the /I compiler directive as follows: /I "../include" /I "../../include" /I "/taco/dserver/include" /I/ "/taco/oncrpc/win32/include"
Hint: To satisfy the linker, we had to link with the following libraries:

```
libdsapi.lib libdsxdr.lib libdbapi.lib oncrpc.lib version.lib
wsock32.lib kernel32.lib user32.lib gdi32.lib winspool.lib
comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib
uuid.lib odbc32.lib odbccp32.lib comctl32.lib
```

and set the library path to:

```
/libpath:"/taco/dserver/lib/win32/Release"
```

Hint: Use the find facility to search for the correct location of header files and libraries if you don't succeed to compile and link the samples. And study the makefiles!

7.3.1 Building the Text-to-Speech example

In the directory taco/dserver/classes/TextTalker you will find

- TextTalker.mak the common makefile for server and client
- TextTalker.dsw the common project file for server and client
- TextTalker.hpj the help project file to generate TextTalker.hlp.

Look into the Win32 Release configuration of the makefile if you want to understand how to set compiler flags when you build a server or a client. When you compile the source files on NT or Windows 95/98 you have to define the preprocessor macros _NT, WIN32,_WINDOWS.
To link the server we had to set the following linker options:

```
libdsapi.lib libdbapi.lib libdsxdr.lib libTTS.lib oncrpc.lib
version.lib wsock32.lib kernel32.lib user32.lib gdi32.lib
winspool.libcomdlg32.lib advapi32.lib shell32.lib ole32.lib
oleaut32.lib uuid.libodbc32.lib odbccp32.lib comctl32.lib
/subsystem:windows /incremental:no/ /pdb:"$(OUTDIR)/TextTalkerds.pdb"
/debug /machine:I386/ /out:"$(OUTDIR)/TextTalkerds.exe"
/libpath:"/taco/dserver/lib/win32/Release"
```

The server (TextTalkerds.exe) needs a slave server (TTSApp.exe) to run correctly. TTSApp.exe is a DDE server, that receives requests either interactively from it's GUI or through DDE messages. TextTalkerds.exe uses these DDE messages as interface to Microsoft's text to Speech engine. That means that you have to install Microsoft's Speech SDK and the TTSApp.exe server before you can use TextTalkerds.
You find Microsoft's Speech SDK as a self-extracting archive on our ftp server called sdk30s.exe (11,799KB) and the TTSApp project called ttsapp.zip. (166KB)
To start TextTalkerds.exe, you first start manually TTSApp.exe, and then TextTalkerds.exe, or you copy TTSApp.exe into your system PATH. TextTalkerds will launch TTSApp automatically if it is in the system PATH.

7.4 Source distribution

If you want to rebuild the libraries, you have to use the source distribution. When you extract the directories

- taco/dbase
- taco/dserver/dev

with all their subdirectories from the archive, you get the source distribution.
Look into the files

- taco/dbase/res/win32/dbapilib.mak
- taco/dserver/dev/system/api/apilib/win32/libdsapi.mak
- taco/dserver/dev/system/xdr/win32/xdrlib.mak

for the makefiles of these libraries.
Look into the files

- taco/dbase/res/win32/dbapilib.dsw
- taco/dserver/dev/system/api/apilib/win32/libdsapi.dsw
- taco/dserver/dev/system/xdr/win32/xdrlib.dsw

for the project files for Microsoft Visual C++ 6.0.
Compiling the libraries is harder. You should first have successfully compiled the sample applications in the binary distribution, before attempting that.

7.5 The ONC RPC library

The ONC RPC library is packaged in the same zip file. When you unzip this archive in the root directory of your hard disk you get the following directory structure:

```
C:\TACO\ONCRPC
+---win32
    +---drivers
    +   +---etc
    +---librpc
    +   +---lib
    +       +---Release
    +       +---Debug
    +---rpcgen
    +---rpcinfo
    +---service
    +---test
    +---wintest
    +   +---vers1
    +       +---Release
    +---bin
    +---include
        +---rpc
```

In the project file for Microsoft Visual C++ 6.0[3] oncrpc.lib and oncrpc.dll are found in the Release and Debug directories, respectively. All .h header files are placed in: /oncrpc/win32/include.

You have to copy oncrpc.lib to /taco/dserver/lib/win32/Release to build the Release versions of the DSAPI libraries or sample applications. You also have to copy ./Release/oncrpc.dll to the Windows system directory c:/winnt/system32, before running the Release versions. The same holds for the corresponding files in ./Debug if you want to build and run the Debug versions.

If you want to build the oncrpc library yourself, you have to define the preprocessor macro _X86_ on Intel platforms.

7.5.1 Portmapper.exe

Portmapper has to run on your computer before you start any device servers. The makefile and sources for portmapper are in /taco/oncrpc/win32/service. The makefile or project file creates two applications portmap.exe and inst_pm.exe. For W/NT portmapper has to be started as a system service. To register portmapper as system service you use the helper inst_pm. With the ControlPanel/Services utility you can define the statup mode of portmapper. On Windows 95/98 portmap.exe is a different executable!! You have to start portmap.exe on reboot by an entry in 'autoexec.bat' When you rebuild portmap.exe you have to modify the makefile according to your system. Read the first lines of the makefile! You have to set OS either to _NT or _W95.

7.5.2 Rpcinfo.exe

rpcinfo.exe is a utility known to UNIX users. It allows you to interrogate portmapper's port tables on you local or any remote host that runs portmapper. In the makefile you have to set OS either to _NT or _W95 if you want to build the executable from scratch.

[3] /taco/oncrpc/win32/librpc/lib/oncrpc.dsw

7.5.3 Rpcgen.exe

rpcgen.exe is the RPC IDL compiler. It generates C-stub source code and xdr-filter source code according to your protocol definition in your IDL-file. In the makefile you have to set OS either to _NT or _W95 if you want to build the executable from scratch. Don't mind the many warning messages during compilation, rpcgen.exe works nevertheless!

7.5.4 RPC sample programs

cou_svc.exe and do_cou.exe are a server/client pair to test the ONC RPC library. They are both simple console applications. cou_svc.exe in the directory wintest/vers1 is the same RPC server as a Windows application.

7.6 Tips and Tricks for developers

7.6.1 Printing debug messages

The standard text output I/O library functions printf, fprintf, ? etc do not work on Windows. TextOut(0 is Windows way to display a string in a window's client area at specified coordinates. Apart from that Windows provides only minimal support for text output to the client area of a window. To simplify this problem, the DSAPI library provides a set of functions similar to printf.
The library provides a global integer that can take values between 0 - 4 to describe different debug levels:

```
0     no debug output, like standard printf
1     level adds error messages,
2     level adds trace messages,
3     level adds more details on trace and errors,
4     level adds dumps of data.
```

```
extern int giDebugLevel;     // 0 is default
```

The library provides two functions to manipulate this global:

```
extern void SetDebugLevel(int i);
extern int GetDebugLevel();
```

The library provides a replacement function to printf that accepts as the first argument a format string compatible with the formats printf uses, followed by a variable list of arguments:

```
extern void cdecl DbgOut(LPSTR lpFormat, ...);
```

Instead of calling DbgOut directly, you should use one of the following macros in your code for the corresponding debug level as stored in giDebugLevel.

```
#define dprintf                             DbgOut
#define dprintf1 if (giDebugLevel >= 1) DbgOut
#define dprintf2 if (giDebugLevel >= 2) DbgOut
#define dprintf3 if (giDebugLevel >= 3) DbgOut
#define dprintf4 if (giDebugLevel >= 4) DbgOut
```

There is another helpful macro defined in the header file macros.h:

```
#ifdef _NT
:
#ifdef WIN32
#define PRINTF(a)      MessageBox(NULL,a,NULL,MB_OK|MB_ICONASTERISK);
#endif

#else   /* not _NT */
#define PRINTF(a)      printf(a)
:
#endif  /* _NT */
```

To make printf(char *format,?) compatible with Windows, you should use sprintf(buff, char* format,?) first and PRINTF(buff) afterwards instead. This provides the standard printf functionality on UNIX, but pops up a MessageBox on Windows instead.

7.6.2 The startup.c file

The developer can assign two function pointers in the server's startup routine. One to perform delayed actions during startup and the other for clean server shut down. Since the server's main calls the startup routine to initialize TACO's RPC services before initializing Windows and creating window handles, you have the possibility to continue the startup after the creation of window classes and main window handles. The function pointer (*DelayedStartup)() will be invoked by the DSAPI library after Windows has finished it's initialization.
If the application uses other Windows services like OLE, DDE or whatever, it has to shut them down in a clean manner. For that purpose the developer can assign the function pointer (*OnShutDown)(), which will be invoked when the main window receives a WM_CLOSE windows message.
Here the definitions in DevServer.h:

```
/* Function called from 'libdsapi' for delayed startup. Useful for
 * Windows applications to perform startup operations when Window's
 * GUI has been initialized. If function pointer is NULL, no delayed
 * startup will take place.
 */
extern long (*DelayedStartup)();
/*
 * Function called from 'libdsapi' for clean shutdown. Useful for
 * Windows applications to perform shutdown operations before the
Window's
 * process is shutdown. If function pointer is NULL, no delayed
 * startup will take place.
 */
extern void (*OnShutDown)();
```

There is the possibility to pass some lines of text to the application's startup. This text will be displayed in the main Windows's backdrop and can be used to inform the user of the server's identity and version.
Here the definition of the corresponding structure in DevServer.h:

```
/* an array of strings to be displayed on the main window backdrop */
typedef struct {
int lines;
char **text;
} MainWndTextDisplay;
extern MainWndTextDisplay gMWndTxtDisplay;
```

If you want text to appear in the main window you have to place something similar like that into the startup routine:

```
:
:
/*
 * Here is the place to define what to put into
 * the main window's backdrop.
 */
static char* info[]= {
{"TACO Server that speaks ASCII text"},
{"32 bit Version rev. 1.0 for Windows 95/98/NT, Oct 2001"},
{"ESRF, BP 220, 38043 Grenoble, France"}
};
:
:
/*
 * Here is the place to assign what to put into
 * the main window's backdrop.
 */
gMWndTxtDisplay.lines= 3;
gMWndTxtDisplay.text= info;
:
:
```

7.6.3 Important Window handles

If you want to extend the server's GUI, you need to know the following handles which are declared as globals in NT_debug.h:

```
extern HWND ghWndMain;           // the main window handle
extern char* gszAppName;         // the application's name
extern HINSTANCE ghAppInstance;  // the application's module handle
```

7.7 Limitations

The libraries do not provide asynchronous calls nor do they provide calls to the TACO Data Collector nor device servers in C++. The ONC RPC library has been tested at it's best, but one never knows.. If you encounter any bug, try to fix it, and please let me know it! The device server comprises two threads now. A main thread that handles GUI- and Window- events, and a worker thread, that runs the svc_run() function, i.e. dispatches all RPC requests. There is no thread synchronization for the time being. Therefore, if you call RPC-service routines from the main thread, for example as a result of an interactive user input via the GUI, you may run into troubles. Closing remarks In Windows jargon, the sample client application ps_menu is a so called console application. That means that the MFC Framework supplies its own WinMain function, upon which you have no influence what so ever. Apparently that does not conflict with the fact, that the DSAPI library contains also a WinMain entry point, i.e. the server's main. We hope (we haven't tested it yet) that this will stay like that, if you write a standard Windows client, i.e. when you provide your own WinMain for your client, or when you write a non console client with the MFC Framework.
The Device Server's WinMain function has been rewritten, and is much cleaner now. It takes note of small differences between W/NT and W/95. With the new

ONC RPC library, we have now better control on the interplay of Windows events and RPC requests. The device server handles all Window events in a main thread, that updates the GUI, whereas a second worker thread handles the svc_run() loop for RPC requests.
Both sample device servers have now their own help support.
The next step will be to support C++ device servers on Windows.
In case of problems or requests/proposals for modifications contact `klotz@esrf.fr` or `götz@esrf.fr`.

Chapter 8

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 9

TACO Lite

What is TACO lite ? TACO lite is the ability to run TACO servers and clients without the database. Original TACO needs a database (ndbm, Oracle or MySQL) for storing network address and permanent configuration information and settings. Sometimes however all you want is a single device server and client running and using the TACO protocol. For example in a small lab or for an embedded or portable device. In this case it can be a constraint to run the manager, database and message server as well.
Running a device server without the database means you have to fix the network address (host and program number) for the server and inform the client, somehow tell the server which devices it has to server and program the server not to use the resources in the database e.g. by providing sensible defaults in the program.
This is exactly what TACO lite does. TACO lite is supported in DSAPI V8.32 and later. To use TACO lite all you have to do is :

1. on the server side start the device server as follows

   ```
   MyServer name -nodb -pn 123456 -device mydevice1 mydevice2 ...
   ```

   where `name` is the personal name of the device server, `123456` is the program number the device server will use[1] and `mydevice1, mydevice2, ...` are the names of the device to be served (note they do not have to have to respect the domain/family/member nomenclature).

2. on the client side import the device(s) in the device server using the following syntax for the name

   ```
   //host/mydevice1?123456
   ```

   where `host` is the name of the host machine where the client is running, `mydevice1` is the name of the device given on the command line to the server, and `123456` is the program number specified on the command line of the server.

Clients can mix devices without database with devices with database in the same process. Using TACO lite it is now possible to distribute servers and clients as static binaries to run on green sites where TACO is not installed.
This is just the first version of running TACO servers and clients without a database. Comments are welcomed. Further evolutions could include providing support for a flat file for resources for example.

[1]make sure this is free beforehand by doing a `rpcinfo -p host` on the host where it will run

Chapter 10

Device Servers in `C++`
by A.Götz and E.Taurel

10.1 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.

10.2 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 invidual 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.

10.3 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.

10.3.1 `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_inited;

   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 */
```

10.3.2 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_inited != 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 subclasses. 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);
};
```

10.4 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.

10.4.1 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 ();

};
```

10.4.2 `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);
}
```

10.5 `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.

10.5.1 `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 ();

};
```

10.5.2 `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 !)[1]

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

   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);
```

[1]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);
}
```

10.6 `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 form 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);
}
```

10.7 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++.

10.7.1 `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 */
```

10.7.2 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);
}
```

10.8 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 :

```
#ifndef __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))
```

10.9 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 HPUX (i.e. the C++ libraries are compiled only with the g++ compiler[2]).

10.10 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 HPUX and Ultra C++

[2]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

10.11 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.

10.12 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**, DOOCS and **cdev**.

10.13 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).

10.14 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 (DOOCS or cdev?).

5. if **templates** and **exception handling** become standard on all compilers then investigate how they can be best used,

10.15 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.

10.16 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 11

Device Server in C *by A.Götz*

THIS CHAPTER IS THE ORIGINAL DEVICE SERVER MANUAL FOR WRITING DEVICE SERVERS IN C. IT FIRST APPEARED IN 1995 AND IS THEREFORE SOMEWHAT DATED. HOWEVER MOST OF IT IS STILL APPLICABLE FOR DEVICE SERVERS WRITING IN C USING THE OIC METHODOLOGY. SOME OF THE MATERIAL ABOUT ASYNCHRONISM IS OUT OF DATE NOW. IT IS INCLUDED FOR REFERENCE PURPOSES.

11.1 Introduction

Device servers were first developed at the European Synchrotron Radiation Facility (ESRF) for controlling the 6 GeV synchrotron radiation source. This document is a Programmer's Manual on how to write device servers. It will not go into the details of the ESRF, nor its Control System nor any of the specific device servers in the Control System. Various papers describe these topics already. Readers are referred to *Laclare (1983)* for a description of the ESRF, to *Götz et al (1991)* for the Control System and the *Device Server User Guides* (or DSUGs) for specific device servers. The role of this document is to help programmers faced with the task of writing device servers.
Device servers have been developed at the ESRF in order to solve the main task of the Control System viz. provide read and write access to all devices in a distributed system. The problem of distributed device access is only part of the problem however. The other part of the problem is providing a programming framework for a large number of devices programmed by a large number of programmers each having different levels of experience and style.
Device servers have been written at the ESRF for a large variety of different devices. Devices vary from serial line devices to devices interfaced by field-bus to memory mapped VME cards to entire VME/VXI data acquisition systems. The definition of a device depends very much on the user's requirements. In the simple case a device server can be used to hide the serial line protocol required to communicate with a device. For more complicated devices the device server can be used to hide the entire complexity of the device timing, configuration and acquisition cycle behind a set of high level commands.
A model (referred to as the Device Server Model or DSM) has been developed to satisfy the main two requirements. In order to do this the DSM has a number of parts to it. It defines the concept of a generic device which is created and managed in a server – a *device server*. The device is accessed by an *application*

programmers interface (api) which is network transparent. Device specific details get treated in the device servers thereby freeing applications to do application-oriented work. Multiple access is implemented by queuing requests – the queuing is handled automatically by the network software.
In this manual the process of how to write device servers will be treated. The manual has been organised as follows – chapter 2 presents an historical account of device servers. The device server model (DSM) is treated in chapter 3. This is followed by a chapter on Objects in C (the Object Oriented Programming methodology used to implement the device servers). Chapter 5 describes how to write a device server. Chapter 6 is devoted to techniques in using classes. Chapter 7 is reserved for Frequently Asked Questions. Finally there is a discussion of limitations in the present device server model and what improvements are planned.
Throughout this manual examples of source code will be given in order to illustrate what is meant. The examples have been taken from the AGPowerSupplyClass - a simulation of a powersupply which illustrates how a typical device server for a powersupply at the ESRF functions. The simulation runs under OS9 and Unix operating systems and requires no hardware in order to run.

11.2 History

The concept of using servers to access devices was first proposed by W.D.Klotz in 1989. To demonstrate this a simulation of a powersupply was implemented which ran as a server. Clients wanting to use the powersupply connected to the server which then forked a copy of the server for the client. Unique data was stored in shared memory. This first version was based on Berkeley sockets and suffered from the drawback that no machine independent data format was used and that for large numbers of clients the number of forked process soon became a limiting factor. The original version was released to application programmers and served a useful purpose.
A.Götz (the author) took over the original server in the late Spring of 1990. The first goals were to replace the Berkeley sockets with the CERN NC/RPC interface, to write servers for real devices, and to setup a team of programmers who would write the servers. This was just the time that X11 and MIT Widgets started appearing on commercial platforms. The Widget model (implemented by MIT's Intrinsics Toolkit) struck the author as being very appealing. It is easy to use, very powerful and manages to hide the complexity of the implementation from the user. It also demonstrated how Classes and Objects can be implemented in C.
Armed thus with the original powersupply api and the Widget model from MIT work begun (mid-1990) in earnest on the device server concept. Assistance was provided by R.Wilcke (who ported the CERN NC/RPC software to OS9) and H.Witsch (who acted as the first guinea-pig device server programmer). The first device server implemented the same functionality as the WDKPowerSupply. The server ran on OS9 and the client on HPUX.
Today (almost three years later) more than 500 device servers exist for the ESRF's Machine and Beamline Control Systems and for Data Acquisition Systems. They run on a range of Operating Systems i.e. OS9, HPUX and SunOS. There are approximately 16 programmers involved in writing device servers. The CERN NC/RPC has been replaced by the SUN NFS/RPC thanks to J.Meyer. A resource database has been added which is accessible via a standard set of rpc calls developed by E.Taurel. Device servers are implemented using classes and clients access devices via a standardised api. If the powersupply server process is considered as the first prototype and the NC/RPC based device servers as the first generation, then it would be true to say that device servers are now well into their second generation.

The term "*device server*" first appeared in an internal ESRF document by W.D.Klotz and S.M.Keogh in June 1989. It reappeared in a paper written for a GULAP (Group for Upper Level Applications Programming) Meeting in January 1990 and has been a common word in the ESRF daily vocabulary ever since.

11.3 The Device Server Model

This section will present the device server model (see figure **??**), hereafter referred to as DSM. It will describe each of the basic features of the DSM and their function. The DSM can be divided into the following basic elements – the *device*, the *server*, *Objects In C*, the *root class*, the *resource database*, the *commands*, *local access*, *network access*, and the *application programmers interface*. This chapter will treat each of the above elements (except for OIC which is treated in the next chapter) separately. More details on the DSM can be found in *Götz et. al.* (from which most of the information for this chapter have been taken).

11.3.1 The model

The basic idea of the DSM is to treat each device as an **object** which is created and stored in a process called a server. Each device is a separate entity which has its own data and behaviour. Each device has a unique name which identifies it in network name space. Devices are configured via resources which are stored in a database. Devices are organised according to **classes**, each device belonging to a class. Classes are implemented in C using a technique called **Objects In C**. All classes are derived from one root class. The class contains a generic description of the device i.e. what actions can be performed on the device and how to implement them. The actions are available via commands. Commands can be executed locally i.e. in the same process, or remotely i.e. across the network. Network access is implemented using a remote procedure call which is accessed via an application programmers interface.

11.3.2 The device

The device is at the heart of the DSM. A device is an abstract concept defined by the DSM. In reality it can be a piece of hardware (e.g. an interlock bit) a collection of hardware (e.g. a screen attached to a stepper motor) a logical device (e.g. a taper) or a combination of all these (e.g. an accelerator). Each device has a unique name. At the ESRF a three field name space has been adopted consisting of **DOMAIN/FAMILY/MEMBER**. A document (see *Taurel (1993)*) exists which describes device names for the ESRF's Machine Control System. A similar document exists (see *Pepellin (1993)*) for the Beam Line Control Systems.

11.3.3 The server

Another integral part of the DSM is the server concept. The server is a process whose main task is to offer one or more services to one or more clients. To do this the server has to spend most of its time in a *wait* loop waiting for clients to connect to it. This division of labour is known as the **client-server** concept. It is used extensively in many systems today (see *Mullender* (1990) for a good overview of the state of client-server technology today).

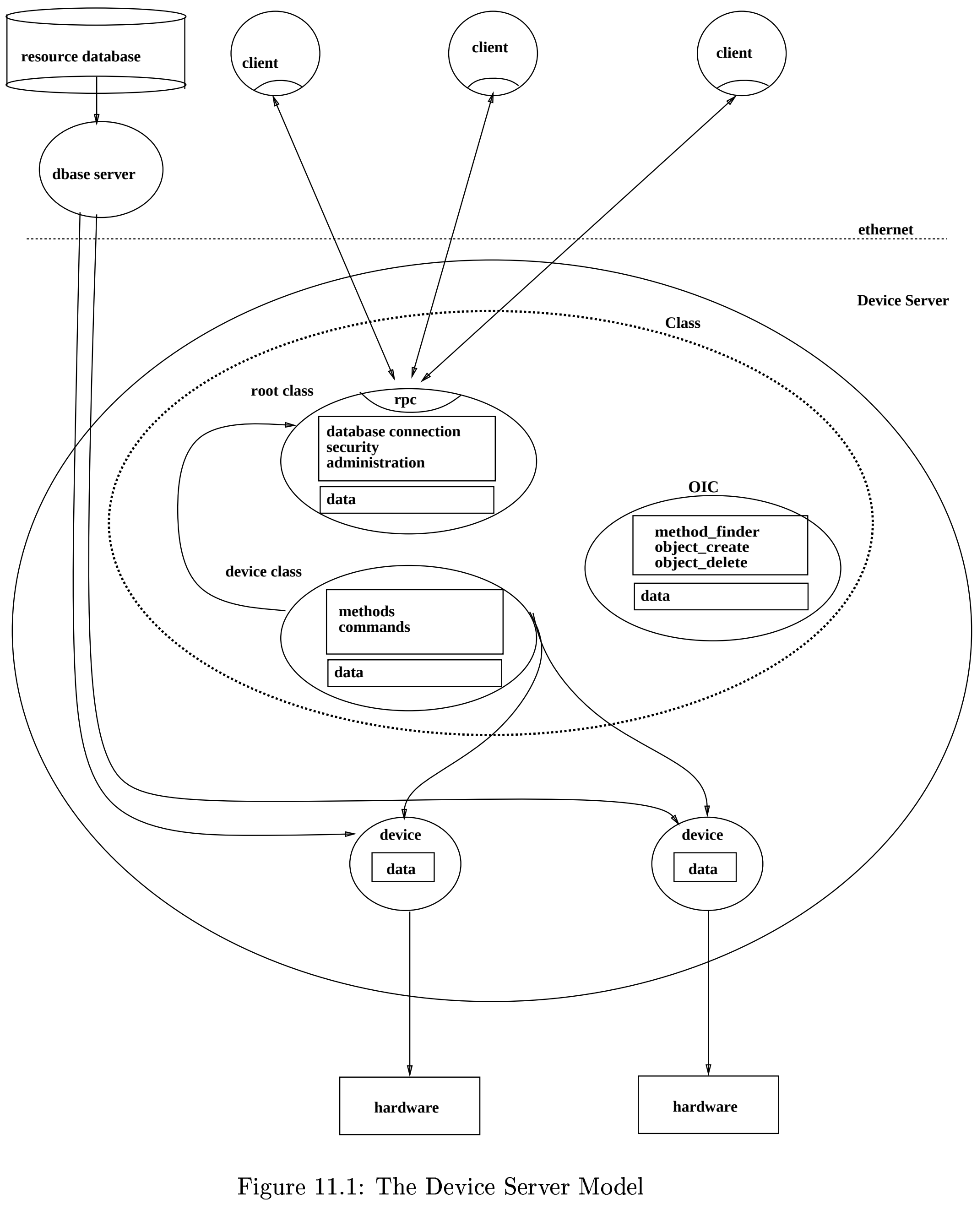


Figure 11.1: The Device Server Model

11.3.4 The root class

All device classes are derived from the same class, the **root class** called the **DevServerClass**. The DevServerClass contains all common device server code. This includes all code related to the applications programmer interface, the database connection, security, administration and so on. Because all device classes are derived from this class they automatically inherit all this code. This makes maintenance and improvements to the DSM easy to carry out.

11.3.5 The device class

Devices are organised into classes in order to generalise on common features between devices while at the same time hiding device dependent details. The device class contains a complete description and implementation of the behaviour of all members of that class. New device classes can be constructed out of existing device classes. This way a new hierarchy of classes can be built up in a short time. Device classes can use existing devices as **sub-classes** or as **sub-objects**. The practice of reusing existing classes is classical for OOP and is one of its main advantages. It encourages code to be written only once and maintained only once. Implementing device access in device classes forces the programmer to implement a generic solution.

11.3.6 The resource database

To achieve complete device independence it is necessary however to supplement device classes with a possibility for configuring device dependencies at runtime. The utility which does this in the DSM is the **resource database**. Resources are identified by an ASCII string and the device name. The link between resource and the device is done using the device name. Each device class should support a certain number of device resources. A well written device class will implement all device dependencies as resources. At device initialisation time the device class interrogates the resource database for all resources associated with each device being created.

11.3.7 The commands

Each device class implements a list of **commands**. Commands are very important because they are the client's **dials** and **knobs** for controlling a device. Commands are like special methods. They difference being they cannot be inherited by subclasses and they have a fixed calling syntax - consisting of one input argument and on output argument. Arguments can be any C type varying from simple types to complicated structures. Commands can execute any sequence of actions. However because all commands are executed synchronously commands timing can become critical.

Commands are executed across the network using the application programmers interface function `dev_putget()`. `dev_putget()` calls a special method implemented in the root class - the *command_handler* method. The *command_handler* calls the *state_handler* method implemented in the device class before calling the command itself. The *state_handler* implements the **state machine** for all devices belonging to that device class. The state machine checks to see wether the command to be executed is compatible with the present state. The command function is only executed if the state handler returns **DS_OK**. The control flow at command execution is represented in *figure* **??**.

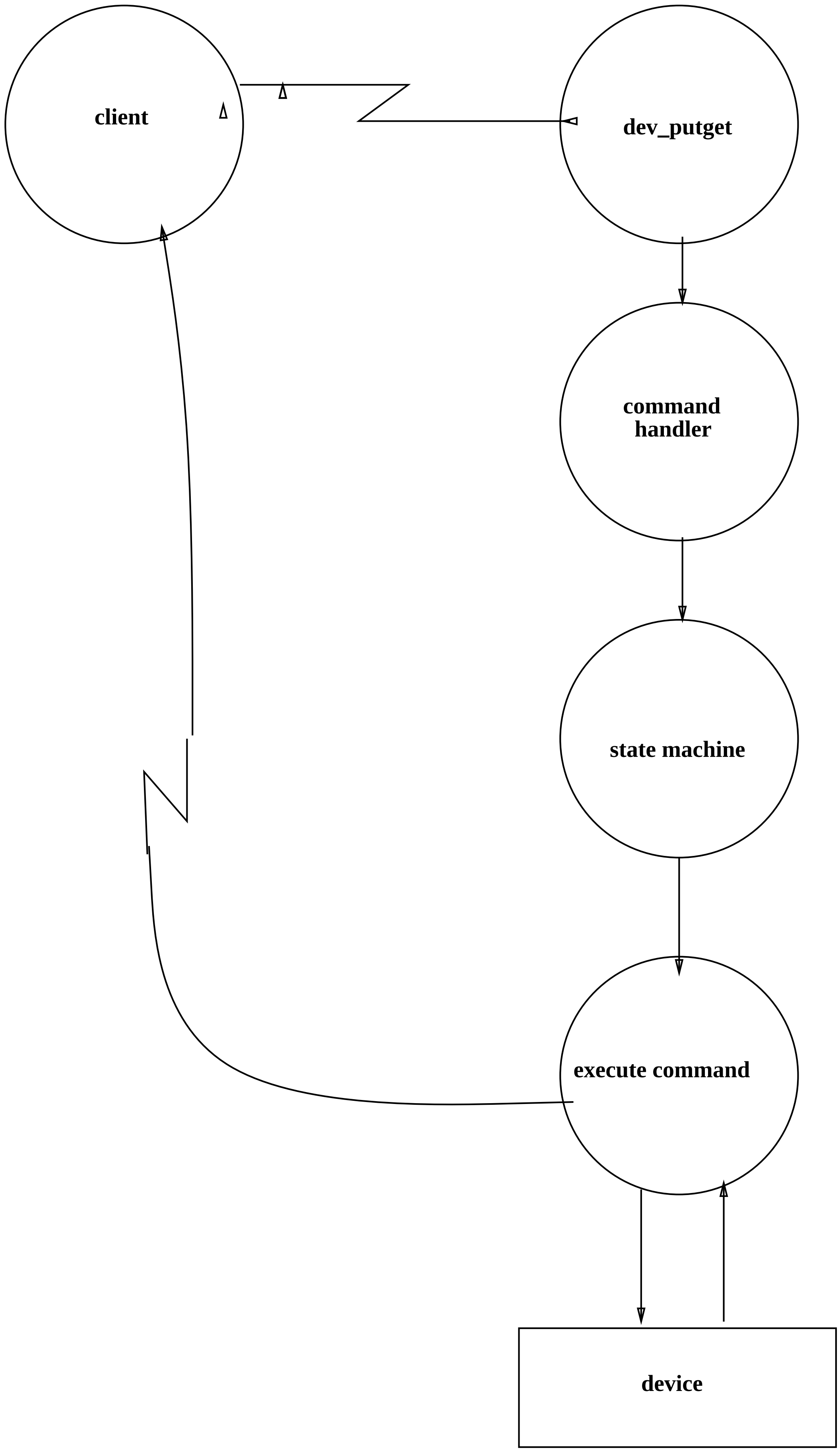


Figure 11.2: Flow of command execution

11.3.8 Executing commands locally

If a device is created in a class or in a program it is possible to execute the device's commands locally i.e. in the same process. The convenience function for executing commands locally is `dev_cmd()`. Commands executed locally do not have any overhead and are consequently much quicker than the same commands executed over the network. This allows programs or other device classes which have to run close to the hardware because of performance or hardware constraints to use existing device classes locally. Devices which are created as members of a class are referred to as **sub-objects**. For more information on this use of devices refer to the discussion in Chapter 6.

11.3.9 Executing commands over the network

Network access is implemented in the DSM in the root class. This is achieved with a **remote procedure call** (rpc). The DSM is presently use the rpc from **SUN** – the Network File System rpc or **NFS/RPC**. Data is transported in network format using the eXternal Data Representation (**XDR** format). The **XDR** routines are part of the NFS/RPC software. A library of routines is maintained for all basic C data types supported. This way not all device server programmers have to learn how to use the XDR routines. It is possible for the device server programmer to add new (exotic) types to this list.

11.3.10 The application programmers interface

Device server clients access devices using the application programmer's interface (API). For performance reasons the device server API is based on the file paradigm. The file paradigm is the open-read-write-close paradigm. The device server API paradigm uses the import-put-get-free paradigm.
The three fundamental API calls are –

1. A call to import a device :

```
dev_import (name,ds_handle,access,error)
char       *name;
devserver  *ds_handle;
long       access;
long       *error;
```

2. A call to execute a command on a device :

```
dev_putget (ds_handle,cmd,argin_ptr,in_type,argout_ptr,out_type,error)
devserver   ds_handle;
short       cmd;
DevArgument *argin_ptr;
DevType     in_type;
DevArgument *argout_ptr;
DevType     out_type;
long        *error;
```

3. A call to free a device :

```
dev_free (ds_handle,error)
devserver ds_handle;
long      *error;
```

Using these three calls clients can execute all commands implemented in the device class on an imported device. All calls are **synchronous** calls. This means that the clients waits for the call to complete before continuing. If the device server does not respond or the remote machine is down a timeout will occur. If the client continues to try executing `dev_putget()` calls and in the meantime the device server is running again the device will be automatically reimported. This last feature assumes that the device has been imported correctly before the connection was lost.

11.4 Objects In C

Very early during the design phases of the Device Server Model (DSM) it was recognised that the problem of device access is well-suited to Object Oriented Programming (OOP). The definition of a generic *device* which unifies all devices can be implemented with a *root device class* from which all other new classes can be derived. The root device class implements the basic functionality of the DSM while the device classes implement the device specific functionality. This means that for each new device class the device server programmer implements, only the new device related code has to be developed - the basic DSM functionalities (like network access, a command handler and so on) are automatically inherited by deriving the new class from the root class.

Although the advantages of OOP are obvious the choice of an OOP language is not always so obvious. Any choice made had to be compatible with the operating systems SunOS, HP-UX and OS9 (the operating systems being used presently at the ESRF). The lowest common denominator in this list is OS9. Operating system compatibility means compatibility with the OS9 **C** language compiler from Microware (the authors of OS9). This reduces the choice to a C-like OOP language (e.g. C++ or Objective C) or developing an OOP programming technique in C. Seeing as at the time the choice for an OOP language was made (1990) , none of the C-like OOP languages available on OS9 were compatible with the Microware C compiler the only solution left was to use OOP programming technique in C.

OOP programming techniques are numerous. This is partly due to the fact that C lends itself to OOP by its ability to support new types via the *typedef* construct. OOP techniques in C are 90% discipline and 10% implementation. The technique which has been developed for the DSM is called **Objects in C** or **OIC**. OIC is based on the MIT Widget programming model. This chapter will describe OIC and how to program in it. No prior knowledge is assumed about Widget programming. The reader is assumed to be conversant in C however.

11.4.1 MIT widgets

The MIT Widget model served as a starting point for the DSM. The MIT Widgets are a spinoff of the Athena Project and the HP Xray toolkit. For an in depth description of the Widget model readers are referred to *Asente and Swick* (1990).

The principal idea behind MIT's Widgets is to treat graphical interaction objects (e.g. a push button or a scrollbar) as separate objects. Each object is represented as a variable of a certain type. The code and data necessary to implement each graphical object are hidden from the user. The user has a set of functions for interacting with the object i.e. reading or setting any of its resources. Widgets are objects which can be created and destroyed. Every Widget belongs to a class. All Widgets are derived from the same root class - the CoreClass.

The advantage of this method is that all the common code (and data) which every Widget has to have (e.g. creating an X11 window and storing it's id) are provided

in the root class. For every new Widget written only the code which is new to this widget has to be written (and maintained).
In order for this to work it is necessary to be able to pass code automatically from one class (e.g. the root class) to other classes, this process is called **inheritance**. An elegant and natural way of doing this with classes is to implement sub-classes. By declaring a new class to be a sub-class of another class, code and data can be automatically inherited. Widgets implement classes in C using structures.

11.4.2 ESRF devices

Although the MIT Widget Model has some very attractive features it is not completely suited to providing network transparent device access. The Widget Model was invented mainly to provide a toolkit for high level X11 programming. Widgets are created and destroyed locally in a program. They don't belong to more than one process at a time. Their main purpose is to hide the complexity of X11 programming behind a simple to manage and understand interface. The device access problem in a distributed control system is a more critical global problem. It has to provide network access to a wide variety of different devices. There is only one copy of each device but there may be many clients at any one time. Errors from devices have to be correctly treated and recovered from - it doesn't help to simply kill the program and restart it. On the other hand there are features of the Widget Model which are compatible with the DSM however.
Instead of reinventing the wheel therefore it was decided to use the MIT Widget model as much as possible and only write/modify those parts which either did not exist or were not suited to the device server model.
Amongst those items which were adopted are (1) the Widget naming convention, (2) the organisation of the private include files, public include files and source code files and (3) the implementation of classes by structures. Amongst those things which were added are (1) a method finder which supports inheritance of methods by subclasses from superclasses, (2) a network manager and (3) a database accessible over the network. The remote database replaces the X11 resource database which is implemented in the X Server. The database is accessible over the network via a database server. The Widget root class (*CoreClass*) has been replaced by a new root class (the *DevServerClass*). DevServerClass has been designed to deal with the network and its resources instead of graphics. It implements (a) the remote procedure calls for the network access, (b) creates a connection to the static database (so the resources can be accessed), (c) keeps a list of exported devices (so that network clients can import devices). It also implements a number of standard methods required for the DSM (e.g. DevMethodCommandHandler, DevMethodExport, DevMethodDestroy).

11.4.3 Naming convention

Every software project needs a naming convention. The naming convention adopted for the DSM follows the X Toolkit Intrinsics naming convention for Widgets. The main reason for adopting the Xt naming convention is to be able to use classes in C as they are used in Widgets.
The following guidelines should be followed when writing device servers :

- Type and procedure names start with uppercase and use capitalization for compound words.
- Local procedures (i.e. static in C) are in lowercase and use underscores for compound words.

- Variable names are in lower case and can use underscores for compound words, but don't have to.
- Structure component names are all in lowercase and use underscores for compound words.
- Predefined symbols and constants are in upper case.
- New device classes start with a capital letter and use uppercase for compound words.
- Each device class has a number of C structures associated with it. Given a new class name *AGPowerSupply* the following structures and pointers to structures must be defined:
  - Partial device instance structure *AGPowerSupplyPart*
  - Complete device instance structure names *AGPowerSupplyRec* and *_AGPowerSupplyRec*
  - Device instance pointer type name *AGPowerSupply*
  - Partial Class structure name *AGPowerSupplyClassPart*
  - Complete Class structure names *AGPowerSupplyClassRec* and *_AGPowerSupplyClassRec*
  - Class structure variable *aGPowerSupplyClassRec*
  - Class pointer variable *aGPowerSupplyClass*

11.4.4 Private (P.h) include files

The private .h file for a device class is included by all device classes that are subclasses of it. It should contain :

- A reference to the public .h file for the class (e.g. *AGPowerSupply.h*).
- A reference to the private .h file for the superclass (e.g. *DevServerP.h*).
- The new fields that the device instance adds to the superclass's device structure.
- The complete device instance structure for this device (e.g. *AGPowerSupplyRec*).
- The new fields that this device class adds to the superclass's device class structure.
- The complete device class structure for this device (e.g. *AGPowerSupplyClassRec*).
- Symbol and constant definitions which are private to this class, i.e. which should be hidden from users of this class but which are nonetheless required by the class and its sub-classes.

Here is an example private include file `AGPowerSupplyP.h` -

```
/*static char RcsId[] = " $Header: AGPowerSupplyP.h.tex,v 1.1 93/04/05 18:16:00 goetz Exp

/*********************************************************************

 File:         AGPowerSupplyP.c

 Project:      Device Servers
 Description:  private include file for the class
               of AG simulated powersupplies.

 Author(s);    Andy Goetz

 Original:     March 1991

 $Log: AGPowerSupplyP.h.tex,v $
Revision 1.1  93/04/05  18:16:00  18:16:00  goetz (Andy Goetz)
Initial revision


 Copyright (c) 1991 by European Synchrotron Radiation Facility,
                      Grenoble, France


 ********************************************************************/

#ifndef _AGPOWERSUPPLYP_h
#define _AGPOWERSUPPLYP_h

/*
 * as subclass of the powerSupplyClass include PowerSupplyClass private
 * definitions
 */

#include <PowerSupplyP.h>

typedef struct _AGPowerSupplyClassPart {
                                         int nada;
                                        }
                AGPowerSupplyClassPart;

typedef struct _AGPowerSupplyPart {
                                    int nada;
                                   }
                AGPowerSupplyPart;

typedef struct _AGPowerSupplyClassRec {
                                         DevServerClassPart devserver_class;
                                         PowerSupplyClassPart powersupply_class;
                                         AGPowerSupplyClassPart agpowersupply_class;
                                       }
                AGPowerSupplyClassRec;

extern AGPowerSupplyClassRec aGPowerSupplyClassRec;

typedef struct _AGPowerSupplyRec {
                                    DevServerPart devserver;
                                    PowerSupplyPart powersupply;
                                    AGPowerSupplyPart agpowersupply;
                                  }
                AGPowerSupplyRec;
```

```
/*
 * private constants to be used in the AGPowerSupplyClass
 */

#define AG_MAX_CUR       100.0
#define AG_MIN_CUR       0.0
#define AG_PER_ERROR     0.001

/* fault values */

#define AG_OVERTEMP      0x01
#define AG_NO_WATER      0x02
#define AG_CROWBAR       0x04
#define AG_RIPPLE        0x08
#define AG_MAINS         0x10
#define AG_LOAD          0x20
#define AG_TRANSFORMER   0x40
#define AG_THYRISTOR     0x80

#endif _AGPOWERSUPPLYP_h
```

11.4.5 Public (.h) include files

The public .h file for a device class is included by other device classes or programs which create devices belonging to this class. It contains :

- A reference to the public .h files for the device class' superclass (e.g. *DevServer.h*).
- The class structure pointer that is used to create devices of this class (e.g. *aGPowerSupplyClass*).
- The instance structure pointer of the template device that is used to initialise new devices of this class(e.g. *aGPowerSupply*).
- The C type that is used to declare device instances of this class (e.g. *AGPowerSupply*)
- Symbols and constants which are related to this class and are of interest to the device classes and/or programs which create local copies of this device.

Here is an example public include file `AGPowerSupply.h` -

```
/*static char RcsId[] = " $Header: AGPowerSupply.h.tex,v 1.1 93/04/05 18:15:57 goetz Exp $

/*********************************************************************

 File:         AGPowerSupply.h

 Project:      Device Servers

 Description:  public include file for implementing the class
               of AG simulated powersupplies.
 Author(s);    Andy Goetz

 Original:     March 1991

 $Log: AGPowerSupply.h.tex,v $
Revision 1.1  93/04/05  18:15:57  18:15:57  goetz (Andy Goetz)
Initial revision


 Copyright (c) 1991 by European Synchrotron Radiation Facility,
                     Grenoble, France

 ********************************************************************/

#ifndef _AGPowerSupply_h
#define _AGPowerSupply_h

typedef struct _AGPowerSupplyClassRec *AGPowerSupplyClass;
typedef struct _AGPowerSupplyRec *AGPowerSupply;

extern AGPowerSupplyClass aGPowerSupplyClass;
extern AGPowerSupply aGPowerSupply;

/*
 * public symbols
 */


#endif
```

11.4.6 Source (.c) code files

The source code can be divided into two parts (a) the code to implement the device class (e.g. AGPowerSupply.c), and (b) the code to implement the startup procedure. The startup is only required when a server process is being customised. This will be treated in chapter 5.
The source file implementing the device class normally contains the entire code for implementing the device class. The class implementation is private and is meant to be accessed only via the class structure i.e. via its methods. For this reason all functions appearing in this file, especially the class methods and all device server commands are declared as `static` in C. The source file initialises the class structure defined in the Private include file. The *method_list* and *n_methods* variables are initialised by static assignments before load time. All other initialisation is done at runtime - this is more flexible and makes the code upwards compatible.
Here is an example of the header and all related declarations for the AGPowerSupplyClass .c file -

```
static char RcsId[] = "@(#) $Header: class_header.c.tex,v 1.1 93/04/05 18:16:11 goetz Exp $ "

/*********************************************************************

 File:         AGPowerSupply.c

 Project:      Device Servers
 Description:  Code for implementing the AG Power Supply class
               The AG Power Supply is a simulation of a typical
               power supply at the ESRF. This means it has two
               main state DEVON and DEVOFF, DEVSTANDBY is unknown.
               All the common power supply commands are implemented.
               The simulation runs under OS9 and Unix. It has been
               developed for application program developers who want to
               test their applications without accessing real devices

 Author(s);    A. Goetz

 Original:     March 1991

 $Log: class_header.c.tex,v $
Revision 1.1  93/04/05  18:16:11  18:16:11  goetz (Andy Goetz)
Initial revision

 * Revision 1.1  91/05/02  08:25:31  08:25:31  goetz (Andy Goetz)
 * Initial revision
 *

 Copyright (c) 1991 by European Synchrotron Radiation Facility,
                       Grenoble, France



 *********************************************************************/

#include <API.h>
#include <DevServer.h>
#include <DevErrors.h>
#include <DevServerP.h>
#include <PowerSupply.h>
#include <AGPowerSupplyP.h>
#include <AGPowerSupply.h>

/*
 * public methods
 */

static long class_initialise();
static long object_create();
static long object_initialise();
static long state_handler();

static DevMethodListEntry methods_list[] = {
 {DevMethodClassInitialise, class_initialise},
 {DevMethodCreate, object_create},
 {DevMethodInitialise, object_initialise},
 {DevMethodStateHandler, state_handler},
};

AGPowerSupplyClassRec aGPowerSupplyClassRec = {
   /* n_methods */     sizeof(methods_list)/sizeof(DevMethodListEntry),
   /* methods_list */  methods_list,
};

AGPowerSupplyClass aGPowerSupplyClass =
                   (AGPowerSupplyClass)&aGPowerSupplyClassRec;

/*
 * public commands
 */

static long dev_off();
static long dev_on();
static long dev_state();
static long dev_setvalue();
static long dev_readvalue();
static long dev_reset();
static long dev_error();
static long dev_local();
static long dev_remote();
static long dev_status();
static long dev_update();

static DevCommandListEntry commands_list[] = {
 {DevOff, dev_off, D_VOID_TYPE, D_VOID_TYPE},
 {DevOn, dev_on, D_VOID_TYPE, D_VOID_TYPE},
 {DevState, dev_state, D_VOID_TYPE, D_SHORT_TYPE},
 {DevSetValue, dev_setvalue, D_FLOAT_TYPE, D_VOID_TYPE},
 {DevReadValue, dev_readvalue, D_VOID_TYPE, D_FLOAT_READPOINT},
 {DevReset, dev_reset, D_VOID_TYPE, D_VOID_TYPE},
 {DevStatus, dev_status, D_VOID_TYPE, D_STRING_TYPE},
 {DevError, dev_error, D_VOID_TYPE, D_VOID_TYPE},
 {DevLocal, dev_local, D_VOID_TYPE, D_VOID_TYPE},
 {DevRemote, dev_remote, D_VOID_TYPE, D_VOID_TYPE},
 {DevUpdate, dev_update, D_VOID_TYPE, D_STATE_FLOAT_READPOINT},
};

static long n_commands = sizeof(commands_list)/sizeof(DevCommandListEntry);

/*
 * a template copy of the default powersupply that normally gets created
 * by the DevMethodCreate. it is initialised in DevMethodCLassInitialise
 * to default values. these defaults can also be specified in the resource
 * file or via an admin command.
 */

static AGPowerSupplyRec aGPowerSupplyRec;
static AGPowerSupply aGPowerSupply =
                     (AGPowerSupply)&aGPowerSupplyRec;

/*
 * template resource table used to access the static database
 */
```

```
db_resource res_table[] = {
            {"state",D_LONG_TYPE},
            {"set_val",D_FLOAT_TYPE},
            {"channel",D_SHORT_TYPE},
            {"n_ave",D_SHORT_TYPE},
            {"conv_unit",D_STRING_TYPE},
            {"set_offset",D_FLOAT_TYPE},
            {"read_offset",D_FLOAT_TYPE},
            {"set_u_limit",D_FLOAT_TYPE},
            {"set_l_limit",D_FLOAT_TYPE},
            {"polarity",D_SHORT_TYPE},
                          };
int res_tab_size = sizeof(res_table)/sizeof(db_resource);
```

11.4.7 The device class C structure

In OIC each device class is represented by a C structure. Understanding this structure is vital to understanding OIC. This section will describe the various components of the class structure. The next section will describe how they should be initialised. Each class structure is made up of a number of fields (cf. figure **??**). Each of these fields is in itself a structure, called a partial structure. Each device class defines (in the private include file) its own partial structure. The partial structure contains all data which are common to all members of that class.
A class hierarchy is defined by the hierarchy of partial structures. For example if a class Z contains the partial structures X, Y and Z (in that order) then one knows that it belongs to the root class X, is a member of the subclass Y and is itself the class Z. Because all device classes are members of the root class DevServerClass, the first partial structure of any device class must be the DevServerClass partial structure, DevServerClassPart.
The DevServerClassPart plays a very special role in the implementation of OIC. It defines the fields necessary for implementing and inheriting methods. This is a fundamental part of OIC because **the Objects In C method finder depends completely on the first partial structure of every class structure being of type DevServerClassPart**.
The fact that the DevServerClass has a dual purpose i.e. implementing *methods* in OIC and *device access* can be confusing. The reasons for this are (as usual) historical. These two functions could have been implemented separately[1]. In OIC this has not been done and device server programmers have to be aware of this. The implications of this are that today only one root class exists - the DevServerClass, and that OIC is used only to implement device servers.
In the same way that the first partial structure of any device class has to be the root class (DevServerClass) so the device classes own partial structure should be the last partial structure. All partial structures in between should be in hierarchical order of the superclasses of the class.
A copy of each device classes C structure is created (space is reserved and it is initialised) once in every program where the device class is used. The structure has the same name as the device classes structure type except that the first character is a small letter. For example the class *AGPowerSupplyClass* has the device class structure type *AGPowerSupplyClassRec* whereas the copy of the device class structure is called *aGPowerSupplyClassRec.*

[1]One obvious way of doing this in OIC would have been to define two partial structures – one called ClassPart which contains the fields required by the method_finder and one called DevServerClassPart which contains only the fields necessary for the DevServerClass.

For each device class structure there is a corresponding device class. The device class is a pointer to the copy of the class structure[2]. The same convention is followed for the device class as for device class structure. For example for the device class type *AGPowerSupplyClass* the actual device class (which must begin with a small letter) is *aGPowerSupplyClass.* A program which wants to instantiate a device of a certain class or wants to use a device class as one of its superclasses uses the pointer to the copy (the one which starts with a small letter) of the device class (*aGPowerSupplyClass* in this example). The device class pointer is defined as external in the public include file and defined and initialised in the .c source code file which implements the device class. Referring to the class pointer in a program forces the loader to link the object code for the class being referred to with the program. This simple but efficient mechanism allows classes to be linked with a program without referring to any of the class source code.

11.4.8 Initialising the device class structure

Each device class is a subclass of DevServerClass (the root class). This means that the first structure within a device class structure is the partial part of the DevServerClass i.e. DevServerClassPart. DevServerClassPart structure contains :

```
typedef struct _DevServerClassPart {
   int                  n_methods;       /*number of methods*/
   DevMethodList        methods_list;    /*pointer to list of methods*/
   DevServerClass       superclass;      /*pointer to superclass*/
   DevString            class_name;      /*name of class*/
   DevBoolean           class_inited;    /*flag indicating if class initialised*/
   int                  n_commands;      /*number of commands*/
   DevCommandList       commands_list;   /*pointer to list of commands*/
   DevString            server_name;     /*server name*/
   DevString            host_name;       /*host name*/
   long                 prog_number;     /*NFS/RPC program number of server*/
   long                 vers_number;     /*NFS/RPC version number of server*/
                                  }
               DevServerClassPart;
```

All device classes have their own copy of this structure pointed to by the class pointer e.g. aGPowerSupplyClass. This is necessary so that each class can have its own list of implemented methods, its own superclass, its own class name, its own class_inited flag and its own commands list. The server name, host name, program number and version number are stored only once - in the DevServerClassPart of the DevServer class.
The *n_methods* and *methods_list* are crucial for the implementing of classes. The *method_finder* (cf. below) uses these two fields to locate the method which will be executed. In order not to be tied down by the definition of the DevServerClassPart structure it was decided very early on in the development of the device servers that these two fields will be the only ones which are initialised at compile time i.e. in static data area. The other fields will be initialised in the *class_initialise* method by assignment statements. This makes existing code upwards compatible even if the DevServerClassPart structure is reorganised or other fields added to it in the future. The fields *n_methods* and *methods_list* have to be initialised with the number of methods and the list of methods in the .c file before any code is executed i.e. at compile and load time.

[2]O woe betide the device server programmer who has not understood pointers and structures in C !

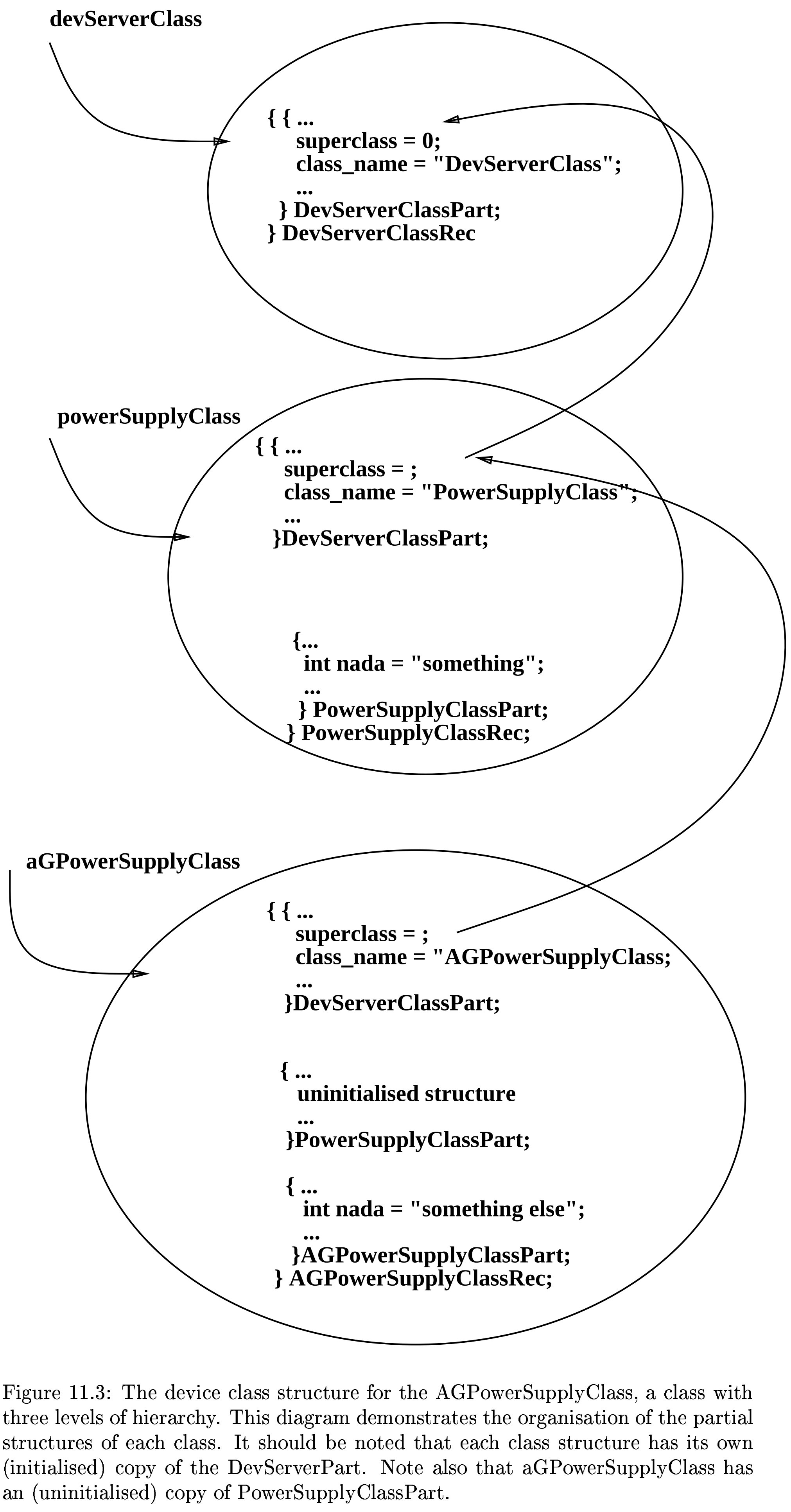


Figure 11.3: The device class structure for the AGPowerSupplyClass, a class with three levels of hierarchy. This diagram demonstrates the organisation of the partial structures of each class. It should be noted that each class structure has its own (initialised) copy of the DevServerPart. Note also that aGPowerSupplyClass has an (uninitialised) copy of PowerSupplyClassPart.

The following fields of DevServerClassPart are initialised in the *class_initialise* method -

- The *superclass* field should contain a pointer to the class structure of the device classes superclass. This pointer **has** to be initialised correctly, otherwise the *method_finder* will not be able to follow the class tree and will fail the first time the application tries to execute a method. A null pointer implies this is a root class. At present only the DevServerClass (the only root class) has a superclass pointer set to NULL.
- The *class_name* is an ascii string containing the name of the class. This should be equivalenced to a symbol defined in the Private include file.
- The *class_inited* field is a flag set to 1 by *class_initialise*. It is used by the *method_finder* to determine whether the *class_initialise* method has been called or not.
- The *n_commands* field should be initialised to the number of commands implemented for this class.
- The *commands_list* should be initialised to point to the list of commands.

After initialising the DevServerClassPart the class should initialise its own partial part. Taking the same example as used above - this means initialising AGPowerSupplyClassPart (of the structure pointed to by aGPowerSupplyClass).

11.4.9 The device C structure

Just like each for each device class there is a C structure in OIC, so for each device created in a program using OIC there is a device structure. The device structure is created (in the *object_create* method implemented in the device class) by allocating memory for it and initialising it. Once created the device is referred to by its pointer. Each device belongs to a class. Each device contains a pointer to its class. Any methods or commands implemented in the device's class can be executed by calling the *method_finder* and specifying the device.
Each device has its own copy of the device structure. This means each device has its own copy of the all data stored in the device structure. Programmers should be aware of this when defining the device structure. Any data which is common to all devices should be stored in the device's class structure. Devices only contain data not code. All code implemented in a class is common to all devices.

11.4.10 Initialising the device structure

The first part of each device structure is the DevServerPart structure. DevServerPart contains the following fields :

```
typedef struct _DevServerPart {
   char           *name;          /*name of device*/
   char           dev_type[24];   /*pointer to string containing device type*/
   DevServerClass class_pointer;  /*pointer to class type*/
   long           state;          /*device state*/
   long           n_state;        /*next device state*/
       }
                DevServerPart;
```

All these values (except the *n_state* variable) have to be initialised at device creation time. The most important is the *class_pointer* variable which is used by the method finder to locate the device class structure.

11.4.11 The template device

Each device is initialised at creation time with default values. These default values can be defined (in order of precedence) either -

1. in the class source code (the socalled code defaults), or

2. in the resource database as class default values (the socalled class defaults), or

3. in the resource database as device default values (the socalled device defaults).

Each class has a template copy of a device which is initialised at runtime by the *class_initialise* method. It is used for initialising devices of this class. Analogous to the class pointer, aGPowerSupplyClass, the template device is called aGPowerSupply. It is defined and should only be accessible from the classes source code. It should be initialised in the `class_initialise()` to the predefined code defaults which can be overridden by the class defaults stored in the static database. This means the `class_initialise()` should access the database after it has initialised the default object. This object will be used to initialise all newly created objects of that class. In C this is achieved by a single structure assignment statement. The defaults in the template object can be overridden by the devices defaults stored in the database.

11.4.12 Methods

Methods are C functions but with a difference. The main difference is that they are accessed via a *method_finder* which uses run-time binding and not by being called directly like with traditional functions which implies compile-time binding. Methods are identified by a symbol representing an integer. All symbols start with the preffix **DevMethod**. The naming convention is to distinguish compound words by capital letters. The symbols for methods are stored in the DevServer.h file - new methods can be added here or in the public include file.
Methods, like traditional C functions, do not have a fixed argument syntax. A function implementing a method may use any argument syntax. The application executing the method should know what arguments are required, their type and in what order to pass them.
All methods should return an integer value which reflects the execution status of the method. For the return value the following convention has been adopted - *DS_NOTOK* is returned if the method fails to execute correctly, and *DS_OK* if it succeeds. These symbols are defined in the `DevServer.h` file.
All methods should be defined as being static in C i.e. only directly accessible from within the classes source code file. Doing this forces applications and subclasses to use the *method_finder* to execute a class's methods. This is what is commonly termed code-hiding and is one of the advantages of Object Oriented Programming. Another advantage of the *method_finder* is that it supports code inheritance. A subclass can inherit code from its superclass(es). This is the case for all subclasses of the DevServerClass for example - they inherit the network interface code which is implemented in DevServerClass.

The method finder function

Methods are accessed using the *method_finder*. The *method_finder* is implemented in the `ds__method -_finder()` function. The *method_finder* lies at the heart of the OIC methodology. It searches the objects class structure for a specifed method. If it doesn't find the method in the objects class it searches the *methods_list* of the

object's superclass. If it can't find the method there it searches the superclass of the superclass and so on until it reaches the root class (**devServerClass**). The function pointer for the first method found which matches the desired method is returned and can be executed. It is up to the calling routine to know and specify the correct syntax. The present implementation of the `ds__method_finder()` does an `exit()` if no matching method is found.
The method finder has following calling syntax -

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

`DevMethodFunction` is defined as a pointer to function i.e.

```
typedef long int (*DevMethodFunction)();
```

The returned function pointer points to the function implementing the desired method which was found by the *method_finder*. It is then necessary to call the function. An example of using the method finder to search for and execute the *object_initialise* method is -

```
ds__method_finder(ds_list[i],DevMethodInitialise)(ds_list[i],error)
```

The device create function

The convenience function `ds__create()` is used to create an device of a given device class. The create function serves a special function. A device class is accessed via its devices and not directly. Each device therefore contains a pointer to its class structure. In order for this to be true each device needs to be created and initialised first. A dedicated method exists in each class for doing this (DevMethodCreate). The `ds__create()` function finds and execute this method for a given class. `ds__create()` is a special version of the `ds__method_finder()` which finds and executes the DevMethodCreate for a class. It takes a device class pointer (e.g. aGPowerSupplyClass) and the device's name as parameters and returns a pointer to the created device.
The device create function has following calling syntax -

```
long ds__create (char *name, void *ptr_ds_class, void *ptr_ds_ptr, long *error)
```

The device destroy function

The convenience function `ds__destroy()` is used for destroying objects. Devices are rarely destroyed in a device server. This is mainly because the device server's main task is to serve all devices as long as it exists. In the event that a device has to be destroyed however the DevMethodDestroy is called. `ds__destroy()` uses the `ds__method_finder()` for finding and executing DevMethodDestroy. The DevServerClass has DevMethodDestroy implemented - its main role is to deallocate the space occupied by the device structure.
The device destroy function has following syntax -

```
long ds__destroy (void *ptr_ds_class, void *ptr_ds, long *error)
```

The class initialise method

All device classes must have at least the *class_initialise* method. This method is called once by the *method_finder* the first time a device of this class (or a device belonging to a subclass of this class) is created. This is taken care of by the method_finder. It is used to initialise class specific data which are required by that class. Amongst other things it should initialise the class structure and the default

object (cf. above). It can also be used to do things required by the class like forking other processes. The *class_initialise* method of a class should not rely on the *class_initialise* method of any of its superclasses. *Class_initialise* methods are called from bottom to top i.e. *class_initialise* of the DevServerClass is called last.
The class initialise method must have following calling syntax -

```
static long class_initialise(long *error)
```

Here is an example class initialise method (for the AGPowerSupplyClass) -

```
/*======================================================================
 Function:      static long class_initialise()

 Description:   Initialise the AGPowerSupplyClass, is called once for
                this class per process. class_initialise() will initialise
                the class structure (aGPowerSupplyClass) and the default
                powersupply device (aGPowerSupply).

 Arg(s) In:     none

 Arg(s) Out:    long *error - pointer to error code if routine fails.
 =======================================================================*/

static long class_initialise(error)
long *error;
{
   AGPowerSupply ps;
   int state;

/*
 * AGPowerSupplyClass is a subclass of PowerSupplyClass
 */

   aGPowerSupplyClass->devserver_class.superclass = (DevServerClass)powerSupplyClass;
   aGPowerSupplyClass->devserver_class.class_name = (char*)malloc(sizeof("AGPowerSupplyCla
   sprintf(aGPowerSupplyClass->devserver_class.class_name,"AGPowerSupplyClass");

/*
 * commands implemented for the AG PowerSUpply class
 */

   aGPowerSupplyClass->devserver_class.n_commands = n_commands;
   aGPowerSupplyClass->devserver_class.commands_list = commands_list;

   aGPowerSupplyClass->devserver_class.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).
 */

   aGPowerSupply->devserver.class_pointer = (DevServerClass)aGPowerSupplyClass;
/*
 * 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.
 */
   aGPowerSupply->devserver.state = 0;
   aGPowerSupply->devserver.n_state = aGPowerSupply->devserver.state;
   aGPowerSupply->powersupply.set_val = 0.0;
   aGPowerSupply->powersupply.read_val = 0.0;
   aGPowerSupply->powersupply.channel = 1;
   aGPowerSupply->powersupply.n_ave = 1;
   aGPowerSupply->powersupply.conv_unit = (char*)malloc(sizeof("AMP"));
   sprintf(aGPowerSupply->powersupply.conv_unit,"AMP");
   aGPowerSupply->powersupply.set_offset = 0.0,
   aGPowerSupply->powersupply.read_offset = 0.0;
   aGPowerSupply->powersupply.set_u_limit = AG_MAX_CUR;
   aGPowerSupply->powersupply.set_l_limit = AG_MIN_CUR;
   aGPowerSupply->powersupply.polarity = 1.0;

/*
 * interrogate the static database for default values
 */

   ps = aGPowerSupply;
   res_table[0].resource_adr = &(ps->devserver.state);
   res_table[1].resource_adr = &(ps->powersupply.set_val);
   res_table[2].resource_adr = &(ps->powersupply.channel);
   res_table[3].resource_adr = &(ps->powersupply.n_ave);
   res_table[4].resource_adr = &(ps->powersupply.conv_unit);
   res_table[5].resource_adr = &(ps->powersupply.set_offset);
   res_table[6].resource_adr = &(ps->powersupply.read_offset);
   res_table[7].resource_adr = &(ps->powersupply.set_u_limit);
   res_table[8].resource_adr = &(ps->powersupply.set_l_limit);
   res_table[9].resource_adr = &(ps->powersupply.polarity);

   if(db_getresource("CLASS/AGPS/DEFAULT",res_table,res_tab_size,error))
   {
      printf("class_initialise(): db_getresource() failed, error %d\n",error);
      return(DS_NOTOK);
   }
   else
   {
      printf("default values after searching the static database\n\n");
      printf("CLASS/AGPS/DEFAULT/state         D_LONG_TYPE    %6d\n",
              ps->devserver.state);
      printf("CLASS/AGPS/DEFAULT/set_val       D_FLOAT_TYPE   %6.0f\n",
              ps->powersupply.set_val);
      printf("CLASS/AGPS/DEFAULT/channel       D_SHORT_TYPE   %6d\n",
              ps->powersupply.channel);
      printf("CLASS/AGPS/DEFAULT/n_ave         D_SHORT_TYPE   %6d\n",
              ps->powersupply.n_ave);
      printf("CLASS/AGPS/DEFAULT/conv_unit     D_STRING_TYPE  %6s\n",
              ps->powersupply.conv_unit);
      printf("CLASS/AGPS/DEFAULT/set_offset    D_FLOAT_TYPE   %6.0f\n",
              ps->powersupply.set_offset);
      printf("CLASS/AGPS/DEFAULT/read_offset   D_FLOAT_TYPE   %6.0f\n",
              ps->powersupply.read_offset);
      printf("CLASS/AGPS/DEFAULT/set_u_limit   D_FLOAT_TYPE   %6.0f\n",
              ps->powersupply.set_u_limit);
      printf("CLASS/AGPS/DEFAULT/set_l_limit   D_FLOAT_TYPE   %6.0f\n",
              ps->powersupply.set_l_limit);
      printf("CLASS/AGPS/DEFAULT/polarity      D_SHORT_TYPE   %6d\n",
              ps->powersupply.polarity);
   }

   printf("returning from class_initialise()\n");
   return(DS_OK);
}
```

The device create method

Classes that can be instantiated require a **DevMethodCreate** method for creating devices of a device class. A convenience function, `ds__create()`, exists for calling this method. The DevMethodCreate method has the job of allocating the space for the new device and initialising it with its name and the contents of the template device. The method can also be used to do any other static housekeeping which might be required by the newly created device.
The device create method must have following calling syntax -

```
static long object_create(char *name, DevServer *ds_ptr, long *error)
```

Here is an example object create method (for the AGPowerSupplyClass) -

```
/*======================================================================
 Function:      static long object_create()

 Description:   create a AGPowerSupply object. This involves allocating
                memory for this object and initialising its name.

 Arg(s) In:     char *name - name of object.

 Arg(s) Out:    DevServer *ds_ptr - pointer to object created.
                long *error - pointer to error code (in case of failure)
 =======================================================================*/

static long object_create(name, ds_ptr, error)
char *name;
DevServer *ds_ptr;
long *error;
{
   AGPowerSupply ps;

   printf("arrived in object_create(), name %s\n",name);

   ps = (AGPowerSupply)malloc(sizeof(AGPowerSupplyRec));

/*
 * initialise server with template
 */
```

```
   *(AGPowerSupplyRec*)ps = *(AGPowerSupplyRec*)aGPowerSupply;

/*
 * finally initialise the non-default values
 */

   ps->devserver.name = (char*)malloc(strlen(name));
   sprintf(ps->devserver.name,"%s",name);

   *ds_ptr = (DevServer)ps;

   printf("leaving object_create() and all OK\n");

   return(DS_OK);
}
```

The device initialise method

DevMethodInitialise is called by the application after the device has been created. It is used to retrieve the device related parameters from the database and to do the active (i.e. physical device related) initialisation of the device.
The device initialise method should have following calling syntax -

```
static long object_initialise(DevServer ds, long *error)
```

Here is an example device initialise method (for the AGPowerSupplyClass) -

```
/*======================================================================
 Function:      static long object_initialise()

 Description:   initialise a AGPowerSupply object. This involves
                retrieving all resources for this device from the
                resource database.

 Arg(s) In:     AGPowerSupply *name - name of object.

 Arg(s) Out:    long *error - pointer to error code (in case of failure)
 =======================================================================*/

static long object_initialise(ps,error)
AGPowerSupply ps;
long *error;
{
   printf("arrived in object_initialise()\n");
/*
 * initialise powersupply with values defined in database
 */

   res_table[0].resource_adr = &(ps->devserver.state);
   res_table[1].resource_adr = &(ps->powersupply.set_val);
   res_table[2].resource_adr = &(ps->powersupply.channel);
   res_table[3].resource_adr = &(ps->powersupply.n_ave);
   res_table[4].resource_adr = &(ps->powersupply.conv_unit);
   res_table[5].resource_adr = &(ps->powersupply.set_offset);
   res_table[6].resource_adr = &(ps->powersupply.read_offset);
   res_table[7].resource_adr = &(ps->powersupply.set_u_limit);
   res_table[8].resource_adr = &(ps->powersupply.set_l_limit);
   res_table[9].resource_adr = &(ps->powersupply.polarity);

   if(db_getresource(ps->devserver.name,res_table,res_tab_size,error))
   {
      printf("class_initialise(): db_getresource() failed, error %d\n",error);
      return(DS_NOTOK);
   }
   else
   {
      printf("initial values after searching the static database for %s\n\n",
     ps->devserver.name);
      printf("state         D_LONG_TYPE    %6d\n",ps->devserver.state);
      printf("set_val       D_FLOAT_TYPE   %6.0f\n",ps->powersupply.set_val);
      printf("channel       D_SHORT_TYPE   %6d\n",ps->powersupply.channel);
      printf("n_ave         D_SHORT_TYPE   %6d\n",ps->powersupply.n_ave);
      printf("conv_unit     D_STRING_TYPE  %6s\n",ps->powersupply.conv_unit);
      printf("set_offset    D_FLOAT_TYPE   %6.0f\n",ps->powersupply.set_offset);
      printf("read_offset   D_FLOAT_TYPE   %6.0f\n",ps->powersupply.read_offset);
      printf("set_u_limit   D_FLOAT_TYPE   %6.0f\n",ps->powersupply.set_u_limit);
      printf("set_l_limit   D_FLOAT_TYPE   %6.0f\n",ps->powersupply.set_l_limit);
      printf("polarity      D_SHORT_TYPE   %6d\n",ps->powersupply.polarity);

/*
 * interpret the initial state of the powersupply
 */
      if (ps->devserver.state == 1)
      {
         printf("switching ON\n");
         dev_on(ps,NULL,NULL,error);
/*
 * if switched ON then set the current too
 */
         dev_setvalue(ps,&(ps->powersupply.set_val),NULL,error);
      }
      else
      {
         printf("switching OFF\n");
/*
 * default is to assume the powersupply is OFF
 */
         dev_off(ps,NULL,NULL,error);
      }
   }

   return(DS_OK);
}
```

11.5 How to write a Device Server

Writing a device server can be made easier by adopting the correct approach. This section will describe how to write a device server. It is divided into the following parts - the team, understanding the device, abstracting the device, defining device commands, designing, coding, debugging, and a general section on standard functions.

11.5.1 Synopsis

The process of writing a device server is an iterative one (see *figure* ??). It starts off with the device documentation and a set of specifications which describe what is required. Often these specifications will not contain anything about logical devices and their commands nor about the class hierarchies. This is natural because of the specifications do not come from a device server programmer but from a user. It is the job of the device server programmer then to understand what the device represents and how it should be defined logically. At this stage the programmer should start thinking about class hierarchy and structure. Maximum use should be made of existing device classes, either as superclasses or as sub-objects (refer to the list of existing DSUGs for more information about device classes). Once the device has been well understood and defined thought should go into the commands to be implemented. At both stages the programmer should confer with the user to see whether the proposed implementation corresponds with the user's requirements and eventual evolutions of the software. The next stage is designing the software and writing the documentation. Only once these two tasks have been finished should coding start. The final phase is testing the code - **never** deliver **untested** software to the client.
Hopefully after going through the seven stages the programmer has a device server ready to be used. However the world is not a perfect place - the device might not correspond to what the user had in mind or the user might change his mind. Consequently even if the device server is ready and tested it might be necessary to modify it, add new functionalities to it or even rewrite it. Do not be afraid to go back to stage 1 and start again. The important thing is to deliver useful software which does the job well. The process should be repeated as often as necessary.

11.5.2 The team

When writing a device server it is useful to consider the people who will be involved and/or concerned -

- **Equipment Responsible** is the person who is responsible for the hardware, he is the lower level client of the Device Server Programmer.
- **Device Server Programmer** is the person who will write the device class and encapsulate it in a server.
- **Applications Programmer** is the person who will use the device server to talk to the hardware, she is the upper level client of the Device Server Programmer.
- **Device Server Maintenance Person** is responsible for maintaining the device server, is not necessarily the Device Server Programmer (but is often heard cursing the Device Server Programmer).

11.5.3 Understanding the device

The first step before writing a device server is to develop an understanding of the hardware to be programmed. The Equipment Responsible should have description of the hardware and its operating modes (manuals, spec sheets etc.). The Equipment Responsible must also provide **written specifications** of what the device

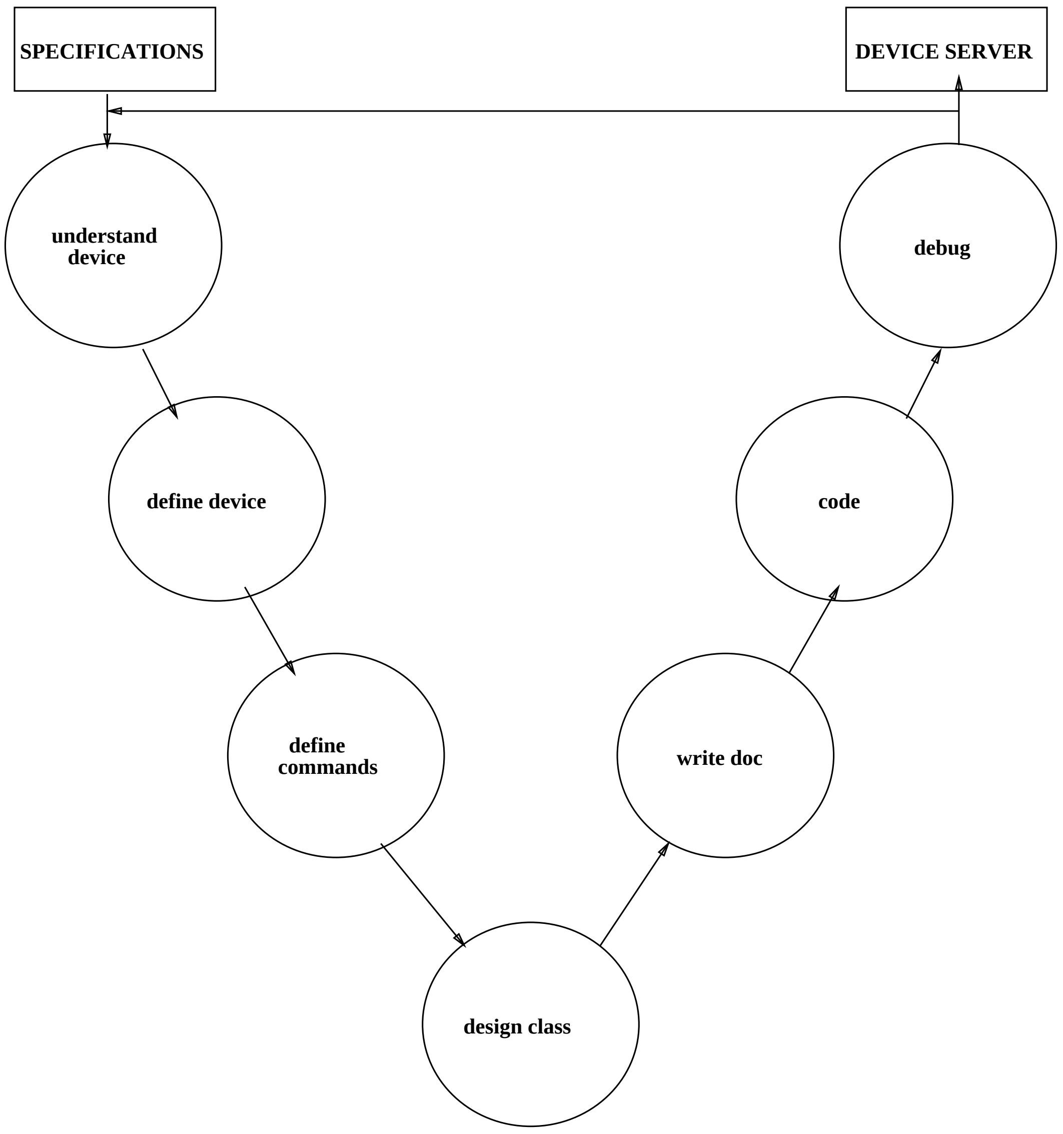


Figure 11.4: The iterative procedure for writing a device server

server should do. The Device Server Programmer should demand an exact description of the registers, alarms, interlocks and any timing constraints which have to be kept.
It is often hard to get hold of this essential information. But the Device Server Programmer should not give up. In very difficult cases it might be necessary to apply pressure on the Equipment Responsible (by explaining the situation to his superior for example) to produce the relevant information (i.e. device description and specifications).
It is very important to have a good understanding of the device interfacing before starting designing a new class. Some of the classic interfaces encountered while writing device servers are -

- *serial line*
- *field bus*
- *Ethernet*
- *IEEE 488*
- *memory mapped*

The most efficient interface being the memory mapped ones of course. Serial lines and IEEE interfaces although very common are notorious for being laborious to program and inefficient in their use. Many devices do not require speed and one often finds serial line and IEEE interfaces for devices. Where possible however I advise programmers to use memory mapped interfaced devices instead.

11.5.4 Abstracting the device

Once the Device Server Programmer has understood the hardware the next important step is to define what is a logical device i.e. what part of the hardware will be abstracted out and treated as a logical device. In doing so the following points of the DSM should be kept in mind -

- Each device is known and accessed by its ascii name.
- The device is exported onto the network to be imported by applications.
- Each device belongs to a class.
- A list of commands exists per device.
- Applications use the device server api to execute commands on a device.

The above points have to be taken into account when designing the level of device abstraction. The definition of what is a device for a certain hardware is primarily the job of the Device Server Programmer and the Applications Programmer but can also involve the Equipment Responsible. The Device Server Programmer should make sure that the Applications Programmer agrees with her definition of what is a device.
Here are some guidelines to follow while defining the level of device abstraction -

- **efficiency**, make sure that not too fine a level of device abstraction has been chosen. If possible group as many signals together to form a device. Discuss this with the Applications Programmer to find out what is efficient for her application.

- **hardware independency**, one of the main reasons for writing device servers is to provide the Applications Programmer with a *software* interface as opposed to a *hardware* interface. Hide the hardware structure of the device. For example if the user is only interested in a single channel of a multichannel device then define each channel to be a logical device. The user should not be aware of hardware addresses or cabling details. The user is very often a scientist who has a physics-oriented worldview and not a hardware-oriented worldview. Hardware independency also has the advantage that applications are immune to hardware changes to the device

- **object oriented worldview**, another *raison d'etre* behind the device server model is to build up an object oriented view of the world. The device should resemble the user's view of the object as closely as possible. In the case of the ESRF's Machine Control System for example the devices should resemble an operator's view of the machine.

- **atomism**, each device can be considered like an atom - is a independent object. It should appear independent to the client even if behind the scenes it shares some hardware or software with other objects. This is often the case with multichannel devices where the user would like to see each channel as a device but it is obvious that the channels cannot be programmed completely independently. The logical device is there to hide or make transparent this fact. If it is impossible to send commands to one device without modifying another device then a single device should be made out the two devices.

- **tailored *vs* general**, one of the philosophies of the DSM is to provide tailored solutions. For example instead of writing one *serial line* class which treats the general case of a serial line device and leaving the device protocol to be implemented in the client the DSM advocates implementing a device class which handles the protocol of the device. This way the client only has to know the commands of the class and not the details of the protocol. Nothing prevents the device class from using a general purpose serial line class if it exists of course.

11.5.5 Defining device commands

Each device has a list of commands which can be executed by the Application across the network or locally. These commands are the Application Programmer's network knobs and dials for interacting with the device.
The list of commands to be implemented depends on the capabilities of the hardware, the list of sensible functions which can be executed at a distance and of course the functionality required by the application. This implies a close collaboration between the Equipment Responsible, Device Server Programmer and the Application Programmer.
When drawing up the list of commands particular attention should be paid to the following points -

- **performance**, no single command should monopolise the device server for a long time (a nominal value for long is one second). Commands should be implemented in such a way that it executes immediately returning with a response. At best try to keep command execution time down to less than the typical overhead of an rpc call i.e. 20 milliseconds. This of course is not always possible e.g. a serial line device could require 100 milliseconds of protocol exchange. The Device Server Programmer should find the best trade-off between the users requirements and the devices capabilities. If a command implies a sequence of events which could last for a long time then implement the sequence of events in another process - don't block the device server.

- **robustness**, should be provided which allow the client to recover from error conditions and or do a warm startup.

The command names should be drawn up by the Device Server Programmer and the Device Server Source Custodian. All commands are presently stored in a single header file (**DevCmds.h**) and therefore the names have to be chosen so that there is no name clash and that they fit in with the naming convention used. It is often possible to reuse existing commands for new devices. The argument types which need to be passed for the commands should also be discussed with the Device Server Source Custodian. He can give advice where necessary on what types can be used how.
The list of device commands should be written and discussed with the Equipment Responsible and the Applications Programmer **before** any coding is started. The commands list should be used as the basis for the Device Servers User Guide, a document which has to exist for each device server. Don't forget documentation is part of the design and as such should be finished before the program.
The commands and the types they use are defined in the header of the class .c file. Here is an example of the commands defined for the AGPowerSupplyClass -

```
static DevCommandListEntry commands_list[] = {
 {DevOff, dev_off, D_VOID_TYPE, D_VOID_TYPE},
 {DevOn, dev_on, D_VOID_TYPE, D_VOID_TYPE},
 {DevState, dev_state, D_VOID_TYPE, D_SHORT_TYPE},
 {DevSetValue, dev_setvalue, D_FLOAT_TYPE, D_VOID_TYPE},
 {DevReadValue, dev_readvalue, D_VOID_TYPE, D_FLOAT_READPOINT},
 {DevReset, dev_reset, D_VOID_TYPE, D_VOID_TYPE},
 {DevStatus, dev_status, D_VOID_TYPE, D_STRING_TYPE},
 {DevError, dev_error, D_VOID_TYPE, D_VOID_TYPE},
 {DevLocal, dev_local, D_VOID_TYPE, D_VOID_TYPE},
 {DevRemote, dev_remote, D_VOID_TYPE, D_VOID_TYPE},
 {DevUpdate, dev_update, D_VOID_TYPE, D_STATE_FLOAT_READPOINT},
};

static long n_commands = sizeof(commands_list)/sizeof(DevCommandListEntry);
```

Standard commands

A minimum set of standard commands should exist for all devices. These are the commands -

- **DevState** - returns the state of the device as a long integer.
- **DevStatus** - the state and any additional useful information of the device as a formatted ascii string (in English).

Examples of these commands for the AGPowerSupplyClass are -

```
/*======================================================================
 Function:      static long dev_state()

 Description:   return state of simulated power supply.

 Arg(s) In:     AGPowerSupply ps - object on which to execute command.
```

```
                DevVoid *argin - void.

 Arg(s) Out:    DevShort *argout - state returned as short integer
                long *error - pointer to error code (in case routine fails)
 =======================================================================*/

static long dev_state (ps, argin, argout, error)
AGPowerSupply ps;
DevVoid *argin;
DevShort *argout;
long *error;
{

/* this command can be always executed independent of the state  */

      *argout = ps->devserver.state;

      return (DS_OK);
}

/*======================================================================
 Function:      static long dev_status()

 Description:   Return the state as an ASCII string. Interprets the error
                flag as well if the status is FAULT.

 Arg(s) In:     AGPowerSupply ps - object on which to execute command.
                DevVoid *argin - void.

 Arg(s) Out:    DevString *argout - status is returned as a string.
                long *error - pointer to error code (in the case of failure)
 =======================================================================*/

static long dev_status (ps, argin, argout, error)
AGPowerSupply ps;
DevVoid *argin;
DevString *argout;
long *error;
{
      static char mess[1024];
      int fault = ps->powersupply.fault_val;
      long p_state;


      p_state = ps->devserver.state;

      switch (p_state) {

      case (DEVOFF) : sprintf(mess,"%s","Off");
                      break;

      case (DEVON) : sprintf(mess,"%s","On");
                     break;
      case (DEVLOCAL) : sprintf(mess,"%s","Local");
                        break;

      case (DEVFAULT) : sprintf(mess,"%s","Fault\n");
                        break;

      default : sprintf(mess,"%s","Unknown");
                break;
      }

/* translate fault into a string */

      if ((fault != 0) && (p_state == DEVFAULT))
      {
      if ((fault & AG_OVERTEMP) != 0)
      {
        sprintf(mess+strlen(mess)," %s","Overtemp");
      }
      if ((fault & AG_NO_WATER) != 0)
      {
         sprintf(mess+strlen(mess)," %s","No Cooling");
      }
      if ((fault & AG_CROWBAR) != 0)
      {
         sprintf(mess+strlen(mess)," %s","Crowbar");
      }
      if ((fault & AG_RIPPLE) != 0)
      {
         sprintf(mess+strlen(mess)," %s","Ripple");
      }
      if ((fault & AG_MAINS) != 0)
      {
         sprintf(mess+strlen(mess)," %s","Mains");
      }
      if ((fault & AG_LOAD) != 0)
      {
         sprintf(mess+strlen(mess)," %s","Load");
      }
      if ((fault & AG_TRANSFORMER) != 0)
      {
         sprintf(mess+strlen(mess)," %s","Transformer");
      }
      if ((fault & AG_THYRISTOR) != 0)
      {
         sprintf(mess+strlen(mess)," %s","Thyristor");
      }
      }

      *argout = mess;

      return(DS_OK);
}
```

Standard commands ensure uniform behaviour of all devices and allow standard utilities to be used for interrogating and displaying device status to be developed. Subsets of standard commands exist for devices belonging to the same superclass. For example all powersupplies should implement the same minimum set of commands. The reader is referred to the Device Server Notes for a description of the major superclasses.

New commands

Where a device requires new commands to be defined because they don't exist in the list of standard commands they should be defined in the public .h file. A scheme has been proposed on how these commands should be defined (cf. F.Epaud DSN/???). As soon as it is adopted it will be included here. Basically it consists of dividing the 32 bit long word for commands into sub-fields and then reserving certain sub-fields for certain classes/groups.

11.5.6 Command data types

All commands have one input parameter and one output parameter. In theory parameter types can be any C type i.e. simple types or composite types represented by a structure. All parameters are passed by pointer. This is to ensure efficiency and homogeneity. Parameters have a sense of direction. For input parameters the memory required is allocated by the calling function (the rpc routine in the case of a remote client). For output parameters the memory occupied by the parameter has to be allocated by the command itself. This should be done in `static` storage so that the space is not deallocated on exiting the command. See the example for the `dev_status()` command for the AGPowerSupplyClass above.
All data types supported by the device servers require conversion routines for **serial**ising and **deserial**ising data from local format to network format (**XDR** format). For this reason in practice only a subset of data types are supported. The list of supported types can be found in `xdr_typelist.h` and in the related `_xdr.h` files.
All basic C types and also variable arrays thereof exist. Programmers should try as much as possible to restrict themselves to only these types. This reduces the number of data types which have to be supported and makes it easier to interface device servers to other software packages. These basic types are -

- typedef void DevVoid; **D_VOID_TYPE**,
- typedef char DevChar; **D_CHAR_TYPE**,
- typedef boolean DevBoolean; **D_BOOLEAN_TYPE**,
- typedef short DevShort; **D_SHORT_TYPE**,
- typedef long DevLong; **D_LONG_TYPE**,
- typedef float DevFloat; **D_FLOAT_TYPE**,
- typedef double DevDouble; **D_DOUBLE_TYPE**,
- typedef char *DevString; **D_STRING_TYPE**,
- typedef struct {short length; char *sequence} DevVarCharArray; **D_VAR_CHARARR**,
- typedef struct {short length; short *sequence} DevVarShortArray; **D_VAR_SHORTARR**,
- typedef struct {short length; long *sequence} DevVarLongArray; **D_VAR_LONGARR**,

- typedef struct {short length; float *sequence} DevVarFloatArray; **D_VAR_FLOATARR**

Normally it is possible to format all command parameters into one of the above types. In the special cases where a new type is required the device server programmer should develop the XDR conversion routine and add it to the library of XDR routines.
When using variable length data types don't forget that network transfers are restricted to 8 kbytes for UDP/IP protocol exchanges. If it is necessary to transfer more data then use TCP/IP. The switching between the two protocols occurs on the client side.
Presently all known data type conversion routines are linked with every device server. This is not at all efficient and wastes quite a lot of memory. In the future (summer 1993) a scheme will be introduced where only the basic types will be linked with each server and/or client and any additional types will require including an include file which contains the type definitions.

11.5.7 Designing

Device servers (like all software) needs to be designed. Some device servers are very simple and do not need an elaborate design. Others device servers are more complicated e.g. multiple processes which communicate with each other, and therefore need a more detailed design. Whatever the case is every device server needs a design.
The design can be in terms of a simple description (if the device server is simple) or it can consist of **data flow diagrams** and **algorithms**. The design should be documented in a computer readable form and this documentation stored with the device server source.
When designing a device server account should be taken of the DSM. The device server is primarily there to accept and execute commands from the network. It spends most of its time waiting for commands or clients to connect on the network and then to serve these requests. Because only one process exists per device server if the device server spends a lot of time doing something else all connections to it (and thereby all devices served by it) are blocked. In severe cases this can cause clients to timeout. Consequently the device server should not spend a lot of time executing any one command. All commands should be executed immediately so that the device server can go back to servicing the same or other clients.
Where the device server is required to treat other events which might be time consuming or require their own polling it is best to consider using a multi-process solution. Time consuming commands should be relegated to independent processes which do not block the device server. The capabilities of the operating system should be used to communicate between the device server and its coprocesses. Most operating systems offer an adequate range of possibilities for synchronising and communicating between process (for example shared memory, events, signals fifos etc.). Although the DSM constrains the programmer to a single event loop within the device server it does not prevent the device server from using the operating system to its fullest. Refer to the section on *Advanced programming techniques* (later on) and to the **Device Server Notes** for solutions already in use by existing device servers.

11.5.8 Documentation

Every device server has to be documented. The documentation should be viewed as part of the design and should therefore be written before the code.
The following documents should exist for each device server

- A **Device Server Users Guide** (DSUG) for the Applications Programmer. This guide is the Application Programmers interface and "how-to" manual for the device server. It should contain amongst other things a description of the device, the commands implemented for the device and how to implement complete sequences in an application. Refer to **DSN/067** for the format and a fuller description of a standard DSUG. A template exists for a standard DSUG.

- A **Class Manual Page** describing the device's class. This is intended for other Device Server Programmers who want to use the class locally in their programs. Refer to **DSN/059** for what should go into a typical class manual page. A template exists of a standard manual page.

- A **Design Document** which describes the design used for the device server. In simple cases plain text (English please) will suffice. For more complicated (especially multiprocess) designs it is recommended to include diagrams and algorithms explaining the design.

11.5.9 Coding

Coding should only be started once the above tasks have been completed or a first version thereof at least. Coding should be done, where possible, in **ANSI C**. All functionalities of the ANSI C compiler which improve the reliability of programs should be used, for example function prototyping.
Coding is best done using the **automatic class generator** written by *Laurent Claustre* (1992). Two versions of this exist (1) an ascii version (**classgen**) which requires file input, and (2) a Motif/X11 based version (**xclassgen**) which uses a graphic interface. Consult the user's manual for the class generator for details on how to use it.
It is also possible to take an existing class and use it as the starting point for a new device class. A global edit can very quickly turn an existing class into the beginnings of a new class.

11.5.10 Debugging

Debugging should be configurable i.e. turned ON and OFF. Use the C precompiler conditional statements, printf's and the debugging functions implemented in the standard device server api library. One precompiler option which should always be used is **DS_DEBUG**. This can be used to print general information about the device and/or class. New classes which require special debugging options should add them as they need e.g. **SL_DEBUG** for serial line debugging.
Symbolic debuggers exist on all platforms and can be used to assist the debugging process. Debugging options should be described in the Design Documentation.
It is useful to always have a debugging version of each class always ready so that in the case of doubt or problems this version can be loaded and used to identify the problem(s).

11.5.11 Testing

Each device server has to be tested. Therefore a test program has to exist for each device server. The standard test programs for device servers are client **menu programs**. The menu programs should allow the user to execute any of the commands implemented in the device class.
Here is an example of menu program for the AGPowerSupplyClass -

```
/*********************************************************************

 File:          ps_menu.c

 Project:       Device Servers

 Description:   Code for a menu driven test program for AGPowerSupplies.
                Allows each command to be executed on a given device.
                Device name is specified on the command line.

 Author(s);     A. Goetz

 Original:      March 1991

 $Log: ps_menu.c.tex,v $
Revision 1.1  93/04/05  18:16:41  18:16:41  goetz (Andy Goetz)
Initial revision

 * Revision 1.1  91/05/02  08:25:31  08:25:31  goetz (Andy Goetz)
 * Initial revision
 *

 Copyright (c) 1991 by European Synchrotron Radiation Facility,
                       Grenoble, France


 *********************************************************************/

#include <API.h>
#include <DevServer.h>

/*
 * include AGPowerSupply public file to get DevRemote command definition
 */
#include <AGPowerSupply.h>


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

      devserver ps;
      DevArg arg;
      long readwrite = 0, error;
      int cmd, status, nave, chan;
      float setcurrent, setvoltage;
      DevFloatReadPoint readcurrent, readvoltage;
      DevStateFloatReadPoint statereadpoint;
      short devstatus;
      char *ch_ptr,cmd_string[256];


      if (argc < 2)
      {
         printf("usage: %s device-name\n",argv[0]);
         exit(1);
      }

      status = dev_import(argv[1],readwrite,&ps,&error);
      printf("dev_import(%s) returned %d\n",argv[1],status);

      if (status != 0) exit(1);

      while (1)
      {
         printf("Select one of the following commands : \n\n");
         printf("0. Quit\n\n");
         printf("1. On          2. Off          3. State\n");
         printf("4. Status      5. Set          6. Read\n");
         printf("7. Update      8. Local        9. Remote\n");
         printf("10.Error       11.Reset\n\n");

         printf("cmd ? ");
/*
 * to get around the strange effects of scanf() wait for something read
 */
         for( ; gets(cmd_string) == (char *)0 ; );
         status = sscanf(cmd_string,"%d",&cmd);

         switch (cmd) {

         case (1) : status = dev_putget(ps,DevOn,NULL,D_VOID_TYPE,NULL,
                                        D_VOID_TYPE,&error);
                    printf("\nDevOn dev_put() returned %d\n",status);
                    if (status < 0) dev_perror(NULL);
                    break;


         case (2) : status = dev_putget(ps,DevOff,NULL,D_VOID_TYPE,NULL,
                                        D_VOID_TYPE,&error);
                    printf("\nDevOff dev_put() returned %d\n",status);
                    if (status < 0) dev_perror(NULL);
                    break;

         case (3) : status = dev_putget(ps,DevState,NULL,D_VOID_TYPE,
                                        &devstatus,D_SHORT_TYPE,&error);
                    printf("\nDevState dev_putget() returned %d\n ",status);
                    if (status == 0)
                    {
                       printf("status read %d , %s \n",devstatus,DEVSTATES[devstatus]);
                    }
                    break;

         case (4) : status = dev_putget(ps,DevStatus,NULL,D_VOID_TYPE,
                                        &ch_ptr,D_STRING_TYPE,&error);
                    printf("\nDevStatus dev_putget() returned %d\n ",status);
                    if (status == 0)
```

```
                {
                   printf(" %s \n ",ch_ptr);
                }
                break;

      case (9) : status = dev_put(ps,DevRemote,NULL,D_VOID_TYPE,&error);
                 printf("\nDevRemote dev_put() returned %d\n",status);
                 if (status < 0) dev_perror(NULL);
                 break;


      case (5) : printf("set current to ? ");
                 for( ; gets(cmd_string) == (char *)0 ; );
                 sscanf(cmd_string,"%f,",&setcurrent);
                 status = dev_putget(ps,DevSetValue,&setcurrent,D_FLOAT_TYPE,NULL,NULL,&er
                 printf("\nDevSetValue dev_putget() returned %d, ",status);
                 printf("current should be set to %6.2f amps\n",setcurrent);
                 if (status < 0) dev_perror(NULL);
                 break;

      case (6) : status = dev_putget(ps,DevReadValue,NULL,D_VOID_TYPE,
                                      &readcurrent,D_FLOAT_READPOINT,&error);
                 printf("\nDevReadValue dev_putget() returned %d, ",status);
                 printf("current set to %6.3f read %6.3f\n",readcurrent.set,
                           readcurrent.read);
                 if (status < 0) dev_perror(NULL);
                 break;

      case (11) : status = dev_put(ps,DevReset,NULL,D_VOID_TYPE,&error);
                  printf("\nDevReset dev_put() returned %d\n",status);
                  if (status < 0) dev_perror(NULL);
                  break;

      case (10) : status = dev_put(ps,DevError,NULL,D_VOID_TYPE,&error);
                  printf("\nDevError dev_put() returned %d\n",status);
                  if (status < 0) dev_perror(NULL);
                 break;

      case (8) : status = dev_put(ps,DevLocal,NULL,D_VOID_TYPE,&error);
                 printf("\nDevLocal dev_put() returned %d\n",status);
                 if (status < 0) dev_perror(NULL);
                 break;

      case (7) : status = dev_putget(ps,DevUpdate,NULL,D_VOID_TYPE,
                           &statereadpoint,D_STATE_FLOAT_READPOINT,&error);
                  printf("\nDevUpdate devputget() returned %d (error %d)\n",status,error)
                  if (status >= 0)
                  {
                    printf("status read %d , %s \n",statereadpoint.state,DEVSTATES[statere
                    printf("current set to %6.3f read %6.3f\n",statereadpoint.set,
                           statereadpoint.read);
                  }
                  break;
```

```
         case (12) : dev_free(ps,&error);
                     exit(0);

         default : break;
         }
      }
}
```

Which provides the user with the following menu -

```
$ ps_menu tl1/ps-d/d
dev_import() returned 0
Select one of the following commands :

0. Quit

1. On          2. Off          3. State
4. Status      5. Set          6. Read
7. Update      8. Local        9. Remote
10.Error       11.Reset

cmd ?
```

11.5.12 Database support

One of the requirements of the device servers is that they be database driven. No constants which could change during the lifetime of the device server should be hardwired into the program. All weak constants plus any parameters and variables should be configurable from a socalled **resource file**. The resource file is a standard feature of the DSM. It is an ascii file with a flat structure where resources (basic C types) can be stored according to class names or device names. A Database api exists for retrieving resources from the database, refer to **DSN/046**.
An example of using the resource database can be found above under the **object initialise** section.

11.5.13 State machine

Each device class must have a state machine implemented in the *state_handler* method. It is called by the *command_handler* to determine whether the requested command can be executed or not. If the *state_handler* returns a non-zero status then the command is not executed.
For very simple devices the *state_handler* has very little to do - any command can be executed at anytime. For other more complicated devices the *state_handler* is used to reflect the internal state of the device.
All devices must however use the *state_handler* to control access to the device. It should be used to reflect the availability of the device. Any device (even the simplest) should support the following two states

- **DEVON** or an equivalent state to indicate that the device is ready to receive commands.

- **DEVFAULT** to indicate that a fault has been detected and the device needs attention, if the fault has been solved a DevReset command should return the device back to a non fault, standard configuration. A fault can also arise when the device hardware has been disconnected.

An example state machine for the AGPowerSupplyClass is given below, it implements the state diagram depicted in figure **??** -

```
/*=======================================================================
 Function:      static long state_handler()

 Description:   Check if the command to be executed does not violate
                the present state of the device.

 Arg(s) In:     AGPowerSupply ps - device to execute command to.
                DevCommand cmd - command to be executed.

 Arg(s) Out:    long *error - pointer to error code (in case of failure).
 =======================================================================*/

static long state_handler( ps, cmd, error)
AGPowerSupply ps;
DevCommand cmd;
long *error;
{
   long iret = DS_OK;
   long int p_state, n_state;

   p_state = ps->devserver.state;

/*
 * before checking out the state machine assume that the state
 * doesn't change i.e. new state == old state
 */
   n_state = p_state;

   switch (p_state) {

   case (DEVOFF) :
   {
      switch (cmd) {

      case (DevOn) : n_state = DEVON;
                     break;

      case (DevError) : n_state = DEVFAULT;
                        break;

      case (DevLocal) : n_state = DEVLOCAL;
                        break;

/* following commands are ignored in this state */

      case (DevSetValue) :
      case (DevReadValue) : iret = DS_NOTOK;
                            *error = DevErr_CommandIgnored;
                            break;

/* default is to allow commands */
```

```
      default : break;
      }

      break;
   }

   case (DEVON) :
   {
      switch (cmd) {

      case (DevOff) : n_state = DEVOFF;
                      break;
      case (DevError) : n_state = DEVFAULT;
                        break;
      case (DevLocal) : n_state = DEVLOCAL;
                        break;

/* following commands violate the state machine */

      case (DevRemote) :
      case (DevReset) : iret = DS_NOTOK;
                        (*error) = DevErr_AttemptToViolateStateMachine;
                        break;

/* default is to allow commands */

      default : break;
      }

      break;
   }
   case (DEVLOCAL) :
   {
      switch (cmd) {

      case (DevRemote) : n_state = DEVOFF;
                         break;

/* the following commands violate the state machine */

      case (DevOn) :
      case (DevOff) :
      case (DevRun) :
      case (DevReset) :
      case (DevStandby) :
      case (DevError) : iret = DS_NOTOK;
                        (*error) = DevErr_AttemptToViolateStateMachine;
                        break;

/* following commands are ignored */
      case (DevSetValue) : iret = DS_NOTOK;
                           *error = DevErr_CommandIgnored;
                           break;
```

```
/* default is to allow commands */

      default : break;

      }

      break;
   }
   case (DEVFAULT) :
   {
      switch (cmd) {

      case (DevReset) : n_state = DEVOFF;
                        break;

/* the following commands violate the state machine */

      case (DevOff) :
      case (DevRemote) :
      case (DevOn) :
      case (DevLocal) : iret = DS_NOTOK;
                        (*error) = DevErr_AttemptToViolateStateMachine;
                        break;

/* following commands are ignored */

      case (DevSetValue) :
      case (DevReadValue) : iret = DS_NOTOK;
                            *error = DevErr_CommandIgnored;
                            break;

/* default is to allow commands */

      default : break;

      }
      break;
   }

   default : break;
   }

/*
 * update powersupply's private variable n_state so that other methods
 * can use it too.
 */

   ps->devserver.n_state = n_state;

#ifdef DS_DEBUG
  printf("state_handler(): p_state %2d n_state %2d, iret %2d\n",
          p_state,n_state, iret);
#endif
```

```
   return(iret);
}
```

Sub-classes of the same super-class usually represent similar devices and should therefore have the same or similar state machine. A diagram (like in fig. ??) representing the state machine of each new class should be included as part of the standard device server documentation.

11.5.14 Errors, Faults and Alarms

Errors, faults and alarms all represent problems of some kind. In the DSM an attempt is made to distinguish between the three classes of problems. This section will describe the difference between error, faults and alarms and explain how to treat them.

- **Errors** indicate that a command has not been able to be executed to completion correctly. This can be due to a partial hardware failure, an incorrect parameter or a bad configuration. Errors are the least serious of the three problem types which have been defined. They should be signalled to the client and can be treated or corrected by the client. They do not require any privileged intervention from above i.e. operator or device server administrator.

  Errors in command execution should be indicated by the status **DS_NOTOK** being returned.

- **Faults** indicate a serious failure of the hardware which needs to be signalled to the operator. The device should change its state to **DEVFAULT** and should not permit further commands until the cause of the fault (bad or missing hardware for example) has been remedied. The fault mode requires execution of a special command (normally DevReset) to put the device back in an operational state.

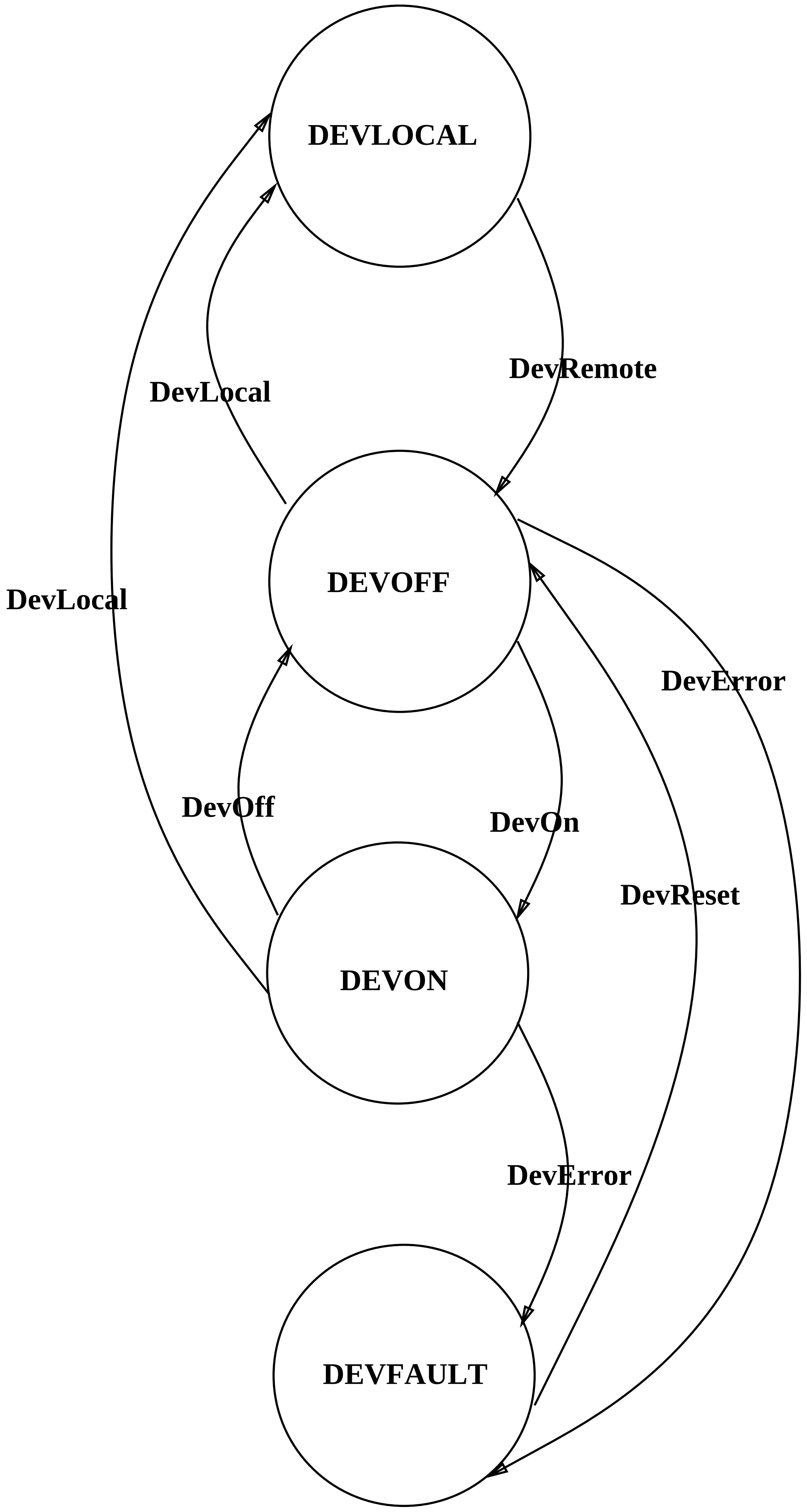
  Faults in the device which prevent a command from being executed should be signalled by the status **DS_NOTOK** being returned.

- **Alarms** indicate faults which can endanger personal or equipment safety. They are context dependent and should therefore be treated by a dedicated application. Alarms should not be treated inside the device servers because (1) the device servers are not aware of the context in which they are used, and (2) device servers are "*dumb*" and not supposed to be polling devices eternally nor waiting for interrupts.

Commands should be provided for recovering from any of the above conditions. It should be possible to bring the device back into a usable state (as far as the hardware permits of course).

11.5.15 Device server startup

In C all programs have a `main()` function. It is the first function called at runtime. This is no different for device servers. However because device servers spend most of their lives sleeping waiting for clients to access them (in an NFS/RPC routine called `rwait()`) the main has to be implemented in a special manner. Rather than providing the user with the source code for the device server main the programmer is given an entry point, the `startup()` routine, which will be called by the common main.


DEVLOCAL
DevRemote
DevLocal
DEVOFF
DevLocal
DevError
DevOff
DevOn
DevReset
DEVON
DevError
DEVFAULT

The startup() has the job of creating all devices of a given class and exporting them onto the network. It can also be used to do global initialisation or other non-standard actions like exporting sub-objects. The startup should return a long status which indicates whether the startup has worked or not. A non-zero status will be interpreted as a failure and the main will do an exit.
The startup function is called by main() with the following syntax -

```
long startup(char *svr_name, long *error)
```

Where svr_name is the personal name referred to below.
An example is the startup for the AGPowerSupplyClass -

```
/*********************************************************************

 File:          startup.c

 Project:       Device Servers

 Description:   Startup procedure for AGPowerSupplyClass. The
                startup procedure is the first procedure called
                from main() when the device server starts up.
                All toplevel devices to be created for the device
                server should be done in startup(). The startup
                should make use of the database to determine which
                devices it should create. Initialisation of devices
                is normally done from startup().

 Author(s);     A. Goetz

 Original:      March 1991

 $Log: startup.c.tex,v $
Revision 1.2  93/04/05  18:16:44  18:16:44  goetz (Andy Goetz)
*** empty log message ***


 Copyright (c) 1990 by European Synchrotron Radiation Facility,
                       Grenoble, France



 *********************************************************************/

#include <Admin.h>
#include <API.h>
#include <DevServer.h>
#include <DevErrors.h>
#include <DevServerP.h>
#include <AGPowerSupplyP.h>
#include <AGPowerSupply.h>

/***************************/
/* AG PowerSupply startup  */
/***************************/
```

```
long startup(svr_name, error)
char *svr_name;
long *error;
{
   AGPowerSupply ps_list[MAX_NO_OF_DEVICES];
   int i,status;
/*
 * pointer to list of devices returned by database.
 */
   char **dev_list;
   int dev_no;

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

/*
 * create, initialise and export all devices served by this server
 */

   for (i=0; i < dev_no; i++)
   {
      if (ds__create(dev_list[i], aGPowerSupplyClass, &(ps_list[i]),error) != 0)
      {
         break;
      }

/*
 * initialise the newly created powersupply
 */

      if (ds__method_finder(ps_list[i],DevMethodInitialise)(ps_list[i],error) != 0)
      {
         break;
      }

/*
 * now export it to the outside world
 */

      printf("created %s, going to export it\n",dev_list[i]);
      if (dev_export(dev_list[i],ps_list[i],error) != 0)
      {
         break;
      }
      printf("export worked !\n");

   }

   printf("left startup and all's OK\n");
   return(DS_OK);
}
```

Retrieving a list of device names

At startup time the device server needs to know which devices it should create and export onto the network. This is done with the use of the **static database** and the device server's **personal name**. Each device server is started with at least one parameter - the personal name. The personal name is an ascii string which is used to identify device server in the database. The list of devices which should be created by a device server are stored in the static database as a resource with a special resource name **device** and attached to the device server executable name and its personal name. An example from the ESRF Transfer Line one demonstrates the principle -

```
AGPSds/TL1/device: TL1/PS-D/D
```

The above resource attaches the device `TL1/PS-D/D` to the device server `AGPSds` which is started with the personal name **TL1**.
It is the job to the device server to retrieve the list of device names from the resource database using the database function `db_getdevlist()`. The syntax for db_getdevlist is -

```
db_getdevlist(char *svr_name,char ***dev_list,long *n_devices,long *error)
```

This call returns a list of device names which the startup can then create and initialise.

Exporting devices on the network

Once a list of devices have been created and initialised they can be exported onto the network to be exported by clients who want to execute commands on them. Not all created devices are necessarily exported onto the network. Sometimes devices are created in the startup for internal use and never exported onto the network.
To export devices onto the network their are two possibilities - (1) calling the DevMethodDevExport directly with the method finder, or (2) using the convenience function `dev_export()` which calls the method finder. It is possible to export a device onto the network with a different name to its device name. This option is reserved for perverse device server programmers. For a device to be exportable it has to appear somewhere in a list of devices in the resource database (cf. above example).

11.5.16 Advanced programming techniques

The basic techniques for writing device classes and device servers are required by each device server programmer. In certain situations it is however necessary to do things out of the ordinary. For example simple device servers spend all their time in a wait loop waiting for commands across the network. A device server might need to do other things in addition to waiting for commands from clients. This section will look into programming techniques which permit the device server serve more than simply the network.

Receiving signals

It is possible for device servers to receive signals from drivers or other processes even while in the main wait loop. The DSM supports signals via a single unified call `ds__signal()`. This call has the same syntax on all operating systems (even OS9) and has been modelled on the Posix call `signal()`.
ds__signal has the following syntax -

```
long ds__signal (int signo, void (*action)(), long *error);
```

This call is used to register a function **action** for the signal **signo**. As soon as the device server receives a signal, it checks to see whether an action has been registered under this signal number and then calls it. Only one action can be registered per signal.
Signals allow the device server to set up asynchronous actions (e.g. timers) during execution of a command and return control to the client. On receipt of the signal (at a later time) the device server can then take appropriate action.
ds__signal() is the only to register actions with signals for device servers. This is because the device server has to exit gracefully and is always programmed for the signal **SIGTERM**. On receipt of the signal SIGTERM the device server will first check to see whether the class has registered its interest for this signal. If so it will call the corresponding function. After that it will exit gracefully by unregistering the device server from the static database.
For more information see the manual page **ds__signal**.

Time sharing

It is sometimes necessary for the device server to only serve the network i.e. commands coming from clients, only a part of its time. Instead of spending all its time in a wait loop waiting for commands it is possible for the device server to poll the network at regular intervals to see if there are any commands to be executed. The call which allows this is `ds__svcrun()`.
The calling syntax is -

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

ds__svcrun will check all open sockets to see if there are any commands waiting to be executed and will then execute the next command. If there are no commands waiting the function will timeout after 10 ms (1 sec for OS9 !).

Multi-processing

Sometimes the fact that device servers are "*dumb*" beasts which sleep most of the time is too limiting for the performance requirements or other requirements demanded of the device server. It is also sometimes not possible to satisfy the requirements of the device server by using a classic design (mono-process) and the two calls described above. In cases like this it is no problem to implement the device server using multiple processes. One process, the device server proper, will be dedicated to dealing with client requests while the other process(es) can be dedicated to other tasks e.g. data taking, monitoring etc. To communicate between the processes shared memory is normally used. A library of basic calls exist for creating and synchronising processes via shared memory called the **dataport** (see *D.Carron*, 1993). Two process device servers are supported in a standard way by the class generator.

11.6 Using Classes

The **Objects in C** model used in device servers supports and even forces Device Server Programmers to write classes. Programming with classes is becoming more common nowadays and many articles can be found on Object Oriented Programming in the literature. Refer to the bibliography listed at the end of this manual for further reading. Although a general philosophy of Object Oriented Programming exists (cf. Yourdon, 1991) the exact technique varies with the implementation.
This section will describe some basic philosophy and techniques for implementing classes, subclasses and superclasses in **OIC**.

11.6.1 classes

The technique of implementing a class has already been treated extensively in this paper eg. contents of include files, source files etc. Nonetheless there are some techniques which are useful when implementing classes which belong to the "*art of good (device server) class programming*" which have not been touched upon yet. The topic of this subsection is to treat these more esoteric techniques.
Classes programming represents a new approach to programming. Until recently the approach to programming was to use traditional languages (e.g. FORTRAN, PASCAL or C) to break down the problem into smaller problems. These smaller problems were then solved and coded up to produce libraries of subroutines, blocks or functions depending on the language used. Programs based on these functions consisted of a series of calls to the function (to use the C paradigm) implemented in the library(ies).
Classes represent a new approach to programming. A class can be best viewed as a *generic description* and *solution* for a particular problem. The art of good class programming is to find the description which best describes the problem. Instead of breaking down the problem into subproblems the problem is broken down into subclasses i.e. shorter descriptions, until eventually one arrives at an ensemble of generic descriptions which by taking specific instances of these descriptions will behave in such a way that they solve the particular problem.
Identifying which classes need to be implemented is not only an attempt at providing a generic description of a solution to a problem or task but also a hierarchical description of the solution. Programming classes are very closely modelled on biological classes. In this respect a class can be a member of other classes. However it is rarely an equal member. A designer of classes tries to organise her classes in order of rank. Some classes are more general than other classes. At the top of the hierarchy one finds a root class. This root classes contains a description of the characteristics and behaviour which are common to all members or sub-members of that class. The set of classes which constitute the solution cannot be defined by a single class. There will always be characteristics and behaviour patterns which are specific to only certain members. Therefore instead of having a top-heavy solution new classes are defined which inherit all or part of what is defined in the root class and then add what is new i.e. specific to them. These new classes are called **subclasses** because they inherit characteristics and behaviour from other (more generic) classes and because they appear lower within the hierarchical structure. A class which has subclasses is known as a **superclass**. An instance of a class which is used in another class is known as a **subobject**.

11.6.2 subclasses

In **OIC** a class can have as many subclasses as it wants. Each class defines a hierarchy. A hierarchy is composed of a root class and all subsequent classes up to

the final class. Because in **OIC** each class can only belong to one superclass, class hierarchies in OIC are one dimensional tree structures (other languages like C++ for example, support multiple inheritance and are therefore two dimensional tree structures). Figure 3 represents a typical OIC tree structure.
The first step in implementing a subclass is to specify its requirements. Subclasses are implemented by modifying the definition of the superclass (defined in the superclasses *private include* file). A definition is required for the partial object and class record structures. The partial object record structure contains those variables and constants which each member of the new class requires a personal copy thereof. The partial class structure contains those variables and constants which are required for the implementation of the class and which can be shared by all members of the class. The classes object and class record structures contain the full description of the class hierarchy. They are formed by adding the partial object and class record structures to the object and class record structures respectively of the superclass.
Once the class and object structures of the subclass are defined then the class behaviour can be implemented in the source file. This means implementing the minimum methods required by each class (e.g. class_initialise)plus the new methods which the new subclass requires. If the new subclass will be instantiated then it will also implement a list of device server commands.
It is necessary that each subclass initialises the root class (DevServerClass) class structure with at least the superclass pointer (*superclass_pointer*) in order for **OIC** to work. This is done in the *DevMethodClassInitialise* method implementation. The classes partial class structure is also initialised in the *DevMethodClassInitialise* method.
One very important implementation detail of **OIC** is that because it is only a programming technique and not a compiler a certain amount of redundancy exists which could confuse the beginner device server programmer. The class record definition includes the partial class record structures for the root class, all superclasses and the class itself. Because the superclasses do not know about the subclasses it is impossible for them to initialise 'their' partial class structures of each of their subclasses. Consequently the partial classes structures of the superclasses of a subclasses remain *uninitialised.* This (possibly confusing for beginner programmers) aspect has been retained in the **OIC** model for two reasons :

1. to provide better readability of the class record structure definition in the private include file, and

2. to maintain symmetry with the object record structure definition.

If the class needs to access data in one of its superclasses it should do so by following the superclasses *class_pointer* in the root class partial structure and thereby access the initialised copy of the superclasses partial structure.
The same doesn't apply to the object record structure however. Each object has its own private copy of the object record. A subclass can access the data defined in the superclasses object partial structure directly. All object data is accessible this way. This is because **OIC** does not distinguish between private and public data.

11.6.3 superclasses

As mentioned above classes are hierarchically organised generic descriptions. If a class has subclasses then it is automatically a *superclass.* Because all device server classes belong to the root class *DevServerClass* they are also automatically *subclasses* and their implementation is as for any other subclasses. It would be wrong to treat all subclasses of the root class serve in the same way. Not all classes are supposed to be instantiated, some classes exist only as superclasses

for other subclasses. The best example of this is the root class *DevServerClass*. There are other examples (e.g. the *PowerSupplyClass*). The philosophy behind these **superclasses** is different from classes which occur only as subclasses of other classes and never as superclasses.
The idea behind a superclass is to abstract out what is common to a number of subclasses and implement this in a single class. This has the advantage of having only a single source to maintain. It also enforces reusability of code. Superclasses can be thought of as abstract classes which serve as place holders for data and a single common source for code. They are essential for implementing classes i.e. hierarchically organised generic descriptions.
Experience with class programming has shown that it is not a good idea to have too many levels of hierarchy. Nesting classes too deeply (i.e. more than five superclasses) is difficult to follow and dissuades programmers from reusing existing superclasses. The ideal level of nesting is **three** or in rare cases **four** levels of class hierarchy. Keep class hierarchies simple. It is more efficient to opt for a flat class structure with many toplevel classes than to go for heavily nested classes. Reusing existing classes implies reusing them as objects rather than as superclasses i.e. as subobjects.

11.6.4 subobjects

The most common way of reusing existing classes in new classes is to instantiate members of the existing classes in the new class. The instantiated member is referred to as a **subobject**. The object has to be created in the *object_create* method of the new class. The object exists locally as a subobject of the new object. This means commands and methods can be executed on the subobject. Thereby allowing the new class to profit from the existing classes implementation without knowing any of the details of implementation.
To execute commands locally use the convenience function `dev_cmd()`. Syntax for dev_cmd is -

```
long dev_cmd (short cmd, DevArgument *argin_ptr, DevType in_type,
              DevArgument *argout_ptr, DevType out_type, long *error);
```

It is also possible to use remote devices as subobjects in a class by importing them (as opposed to creating them locally). This has the advantage that a new class can use existing classes across the network i.e. it is not obliged to be on the same physical machine as the imported device. It has the disadvantage that executing commands on the remote device takes longer because of the network overhead. Another disadvantage of this method is that methods cannot be executed remotely. Nonetheless it can be very useful sometimes to import devices in classes and it is done quite often.

11.7 Discussion

Up to now this manual has presented the device server model and how to use the **Objects In C** programming method to write new device servers. There remain a number of topics which have not been treated however. In particular the questions of device server programmers when first confronted with the task of writing device classes.
This section will treat some of the most *Frequently Asked Questions* which device server programmers pose. It will also include a discussion on the limitations of the present device server programming model and improvements which need to be made to the present method of writing device server classes.

11.7.1 Frequently Asked Questions

1. *What is the difference between a device server and a device class ?*

   A device server is a single process which instantiates and exports object(s) of one or more classes. Once the object(s) have been exported the process waits for requests on the network to execute commands.

   A device class is a software class which implements the generic behaviour and characteristics of a logical device. It is implemented in C using a method called **Objects In C**.

2. *Can device servers still be used to solve problems where fast timing is a critical issue ?*

   This question demonstrates a misunderstanding of the work of a Device Server Programmer. Device Server Programmers are writing software classes which describe and implement device access. These classes can be used by other classes or in conventional procedure base software. A device server (i.e. the process which serves a or many devices) is simply a way of packaging these classes into a process which provides a procedural interface on the network. If network access is requested and if timing is a problem subclasses can be combined in superclasses in such a way that all critical timing takes place within a single class (i.e. locally in one process), thereby removing the network access part from the critical path. Alternatively the critical code can be implemented in an independent processes and device server can be used to provide network access.

   The DSM in no way prevents the programmer from using the operating system to its fullest - in theory it is possible to achieve the same response with a device server as with any other local process running under OS9 or Unix.

3. *What is the difference between a method and a command ?*

   A method is a special function implemented in a class in the **OIC** programming methodology which can be inherited by subclasses of that class.

   A command is a special function in the DSM which can be executed across the network using the device server api call **dev_putget()**. All commands have a fixed calling syntax. Commands as opposed to methods cannot be inherited by subclasses. The only way to inherit a command to a subclass is to implement it as a method.

4. *Are device servers complicated to implement ?*

   A device server is only as complex as the device it has to implement and serve. The advantage of the DSM is that all common functions related to network access are standardised. What might appear complicated to beginners is the object oriented aspect of class programming. The advantages of class programming (e.g. hierarchical structuring, generic solutions, re-using code) are sufficient however that it is worth investing the time in learning how to write classes.

   Device servers should *not* be used as an excuse not to write complex but maintainable software.

5. *Do device servers replace replace device drivers ?*

   The answer is NO. In an ideal world both should exist. A device driver should be written to access the physical hardware by exploiting up to a maximum the I/O channels of the operating system. For example fast queued access with arbitration is offered by OS9 for drivers. A device server takes over from where the device driver leaves off. It offers higher level functions and network access. Although sometimes compared to a networked version of a device driver it is at a much higher level in terms of the way it presents information and the commands it offers.

11.7.2 Limitations

One of the limitations to the DSM is the **OIC** methodology. The adoption of the **OIC** programming method was decided on because of the desire to have a programming environment which supports object oriented programming under an operating system (*OS9*) for which no commercially available and viable Object Oriented Language could be found. It is a limitation because it is home-brewed (at the ESRF) and is unknown. It is not a language and therefore consists of 10 percent code and 90 percent discipline. Paradoxically OIC is also one of the strengths of the DSM. It is a strength because it is portable and Operating System independent. It is implemented in C and therefore completely compatible with the existing Unix-like programming environment. It is easy for programmer's proficient in C to use OIC. This is not necessarily the case for C++ for example which requires programmers to be proficient in C and C++.

Another limitation of the DSM is the lack of **multiple inheritance**. Multiple inheritance is the ability of a class to be derived from multiple superclasses at the same level. This limitation is due to the use of OIC. It can be partially overcome by using multiple superclasses arranged hierarchically but will only be completely overcome by either adding multiple inheritance to OIC or by implementing the DSM in an OOP language which supports multiple inheritance.

Timing is another area in the DSM which is treated in a limited way. A device server spends most of its time waiting for client requests. When a request is received it is executed completely i.e. synchronously, before the server goes back to waiting for client requests. The server by definition has only one thread of execution. If a server wants to communicate with other processes it has to use the mechanisms offered by the operating system or some of the advanced calls developed as part of the device server library (see *Advanced programming techniques* above). Timing has to be taking into account when designing device classes.

11.7.3 Improvements

One of the major advances in the DSM would be to implement it in a 'real' Object Oriented Language, for example in *C++*. This way the compiler implements the Object Oriented-ness and the programmer can concentrate on the class implementation. Implementing device servers in *C++* would have the advantage of adopting a *de facto* standard as compared to the **OIC** programming method (which even if derived from the MIT Widgets model exists only at the ESRF and is poorly documented compared to C++).

Other improvements which are planned are in the device server api. An **asynchronous** dev_putget will be added to complement the existing synchronous call. The asynchronous call be compatible with main event loop in X11/Motif applications. A second improvement to the api is the addition of a reliable protocol based on UDP/IP. To date only UDP/IP and TCP/IP are supported. The former is connectionless but not reliable while the latter is connection oriented and reliable. The aim is to add a third protocol which is connectionless but reliable i.e. based on UDP/IP. These improvements are planned for the summer of 1993.

11.8 Conclusion

This manual describes how to write a device server. Device servers vary enormously in their complexity and it is difficult without writing a thesis on the topic to treat all the possibilities of device servers in a single document. It is hoped however that the manual describes sufficiently the process of writing device server that beginners can start being effective quite soon after reading this manual. The best way to start is to actually write a device class and then encapsulate it in a server. To do this all that is necessary is a device, this manual and the class generator. Once the basics of device server writing have been grasped the programmer will see how simple the entire process is actually.
Very few manuals are perfect and this is surely not one of them. The author will gladly accept any useful or constructive criticism on how to improve it.

Bibliography

[1] P.J.Asente and R.R.Swick, *X Window System Toolkit*, Digital Press, 1990.

[2] A.D.Birell and B.J.Nelson, "Implementing Remote Procedure Calls" in *ACM Transactions on Computer Systems* 2(1), February 1987.

[3] D.Carron, "Using Dataports as an Interprocess Communication Mechanism", *ESRF internal document*, 1993.

[4] L.Claustre, "The Automatic Class Generator", *ESRF internal document*, 1993.

[5] P.Coad and E.Yourdon, *Object-Oriented Analysis*, Prentice-Hall, 1991.

[6] A.Götz, J.Meyer and W.D.Klotz, "Object Oriented Programming Techniques Applied to Device Access and Control", *Proceedings of the International Conference on Accelerator and Large Experimental Physics Control Systems*, Tsukuba (Japan), November 1991.

[7] A.Götz, W.D.Klotz, J.Meyer, E.Taurel, M.Schofield, P.Mäkijärvi andM.Karhu, "A Distributed Control System based on the Client-Server Model" in *Proceedings of The International Conference cum Workshop on Control and Data Acquisition*, Calcutta (India), November 1991.

[8] K.E.Gorlen, S.M.Orolow and P.S.Plexico, *Data Abstraction and Object Oriented Programming in C++*, John Wiley & Sons, 1990.

[9] A.I.Holub, *C+C++ Programming with Objects in C and C++*

[10] F.L.Laclare, "Overview of the European Synchrotron Light Source" in *IEEE Particle Accelerator Conference*, Washington D.C. (USA), March 1987, pp. 417-421.

[11] S.Mullender, *Distributed Systems*, ACM Press, 1990.

[12] S.Oualline, *Practical C Programming*, O'Reilly & Associates, Inc., 1991.

[13] G.Pepellin, "ESRF Beamline Device Names Compendium", *ESRF internal document*, 1993.

[14] T.Plum, *C Programming Guidelines*, Plum Hall Inc., 1989.

[15] E.Taurel, "ESRF Machine Device Names Compendium", *ESRF internal document*, 1993.

Chapter 12

DSAPI
by J.Meyer and A.Götz

12.1 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.

12.2 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.

12.3 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).

12.3.1 "*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 2.1 199
/*+*******************************************************************

 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(%s) 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);
}
```

12.3.2 "*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 callbacks 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_printerror_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);

}
```

12.3.3 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.

12.3.4 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`

12.3.5 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 and 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).

12.3.6 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
- `vwx86` - 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.l -l sys_clib.l -l unix.l
```

```
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 dependancies in one file can prove to be a time saver when developing on multiple platforms.

12.3.7 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 argments 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.

12.3.8 Advanced Features

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

12.3.9 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_rpctimeout()` call (cf. the C library reference). This can be necessary if the request is know 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 fixed by simply changing from UDP to TCP protocol (see next section).

12.3.10 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).

12.4 C Library

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

12.4.1 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_ACESS` 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 correpsonds 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_asyn()

```
long dev_put_asyn (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_asyn() is similar to 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.

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.

12.4.2 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 appropiate 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 returing immediately.

12.4.3 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 bye 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.

12.4.4 General Purpose Functions

dev_printerror_no()

```
void dev_printerror_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()**[1].

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
```

[1]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_ASYNCH            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.

12.5 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.[2] Servers should use ONLY these types as input and output arguments.

12.5.1 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.

12.5.2 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`

[2]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

12.5.3 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;
   ```

12.5.4 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 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;
```

12.5.5 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).

12.6 Changes

12.6.1 Version 8.0

Version 8 introduces support for TANGO. TANGO[3] 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).

12.6.2 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.

12.6.3 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).

12.6.4 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.

12.6.5 Version 4.1

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

12.6.6 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 */
```

[3]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 bye 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 be 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.

```
    | 31 | 30             20 | 19        12 | 11          0 |
    -------------------------------------------------------------
      |            |                |          |
      |            |                |          |- Position in the
      |            |                |             list of exported
      |            |                |             devices.
      |            |                |
      |            |                |- Position in the
      |            |                   list of client
      |            |                   connections to the
      |            |                   device.
      |            |
      |            |- Export counter
      |
      |- Local access flag
```

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 13

Database guide - ndbm
by E.Taurel

13.1 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.

13.2 Device and resource definition

13.2.1 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.

13.2.2 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.

13.2.3 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. Nevertheles, 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.

13.3 Greta

Greta (**G**raphical **R**esource **E**ditor for **TA**co) 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

13.3.1 The device window

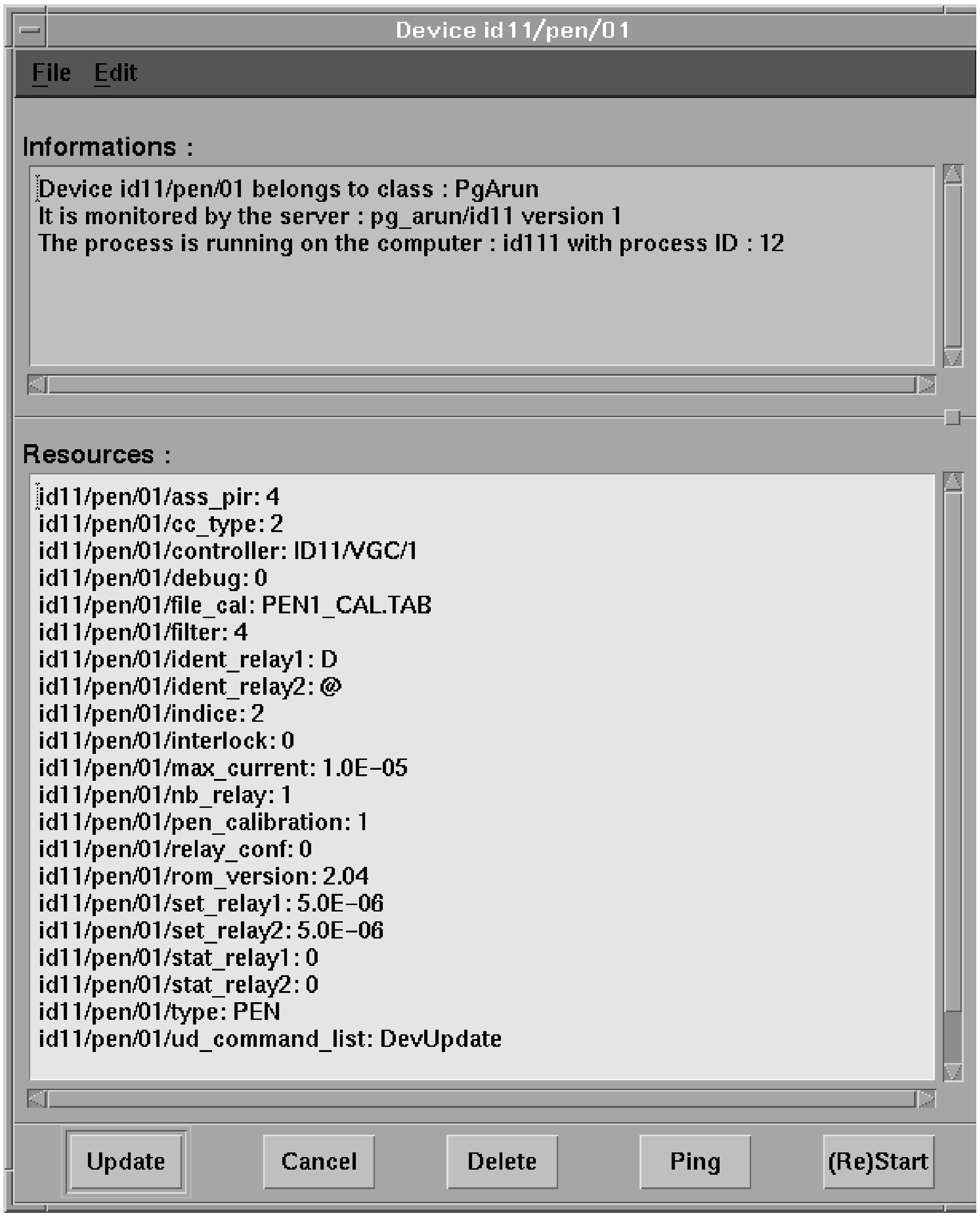
To open a device window, click on File–Open device. A database device browsing window is poped-up. Once a device is selected (by double click on the field name or by pressing the filter button), pressing the open button or a double click on the Member field will poped-up a device window.
The Informations 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 poped-up
- **Cancel** to close the window without any database change
- **Delete** to delete the device from database. A window is poped 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 poped_up. In all cases, a confirmation window is poped up.

Under the window File button, it is possible to :

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

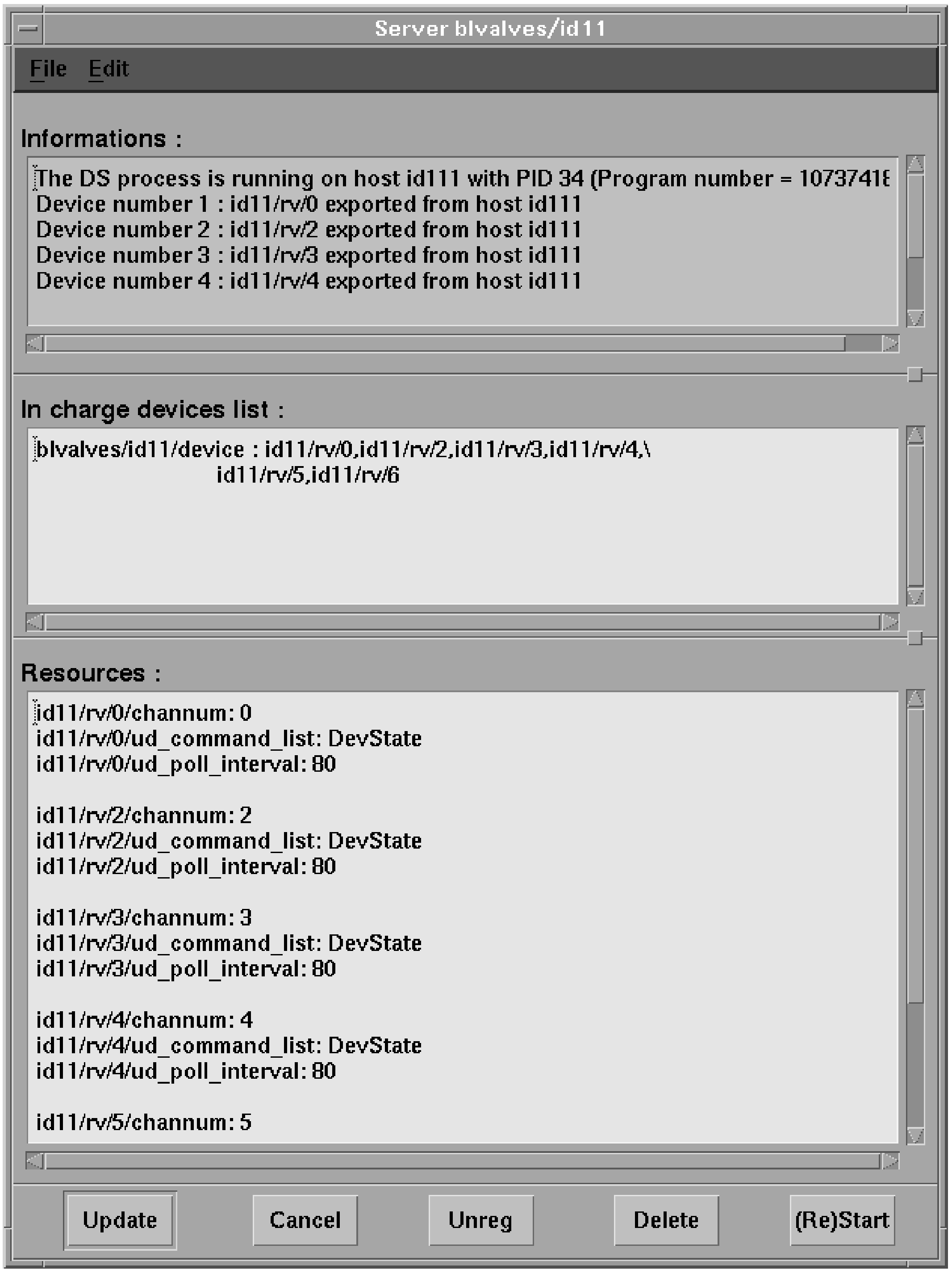
Under the Edit button, the user will find the classical edit features plus the "insert device resource" button. If some device resources are device name, by selecting this device name and clicking in Edit–insert device resource, all the resources belonging to the newly selected device will be added at 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.



```
Device id11/pen/01 belongs to class : PgArun
It is monitored by the server : pg_arun/id11 version 1
The process is running on the computer : id111 with process ID : 12
```

```
id11/pen/01/ass_pir: 4
id11/pen/01/cc_type: 2
id11/pen/01/controller: ID11/VGC/1
id11/pen/01/debug: 0
id11/pen/01/file_cal: PEN1_CAL.TAB
id11/pen/01/filter: 4
id11/pen/01/ident_relay1: D
id11/pen/01/ident_relay2: @
id11/pen/01/indice: 2
id11/pen/01/interlock: 0
id11/pen/01/max_current: 1.0E-05
id11/pen/01/nb_relay: 1
id11/pen/01/pen_calibration: 1
id11/pen/01/relay_conf: 0
id11/pen/01/rom_version: 2.04
id11/pen/01/set_relay1: 5.0E-06
id11/pen/01/set_relay2: 5.0E-06
id11/pen/01/stat_relay1: 0
id11/pen/01/stat_relay2: 0
id11/pen/01/type: PEN
id11/pen/01/ud_command_list: DevUpdate
```

Figure 13.1: Greta device window



```
The DS process is running on host id111 with PID 34 (Program number = 1073741
Device number 1 : id11/rv/0 exported from host id111
Device number 2 : id11/rv/2 exported from host id111
Device number 3 : id11/rv/3 exported from host id111
Device number 4 : id11/rv/4 exported from host id111
```

```
blvalves/id11/device : id11/rv/0,id11/rv/2,id11/rv/3,id11/rv/4,\
                       id11/rv/5,id11/rv/6
```

```
id11/rv/0/channum: 0
id11/rv/0/ud_command_list: DevState
id11/rv/0/ud_poll_interval: 80

id11/rv/2/channum: 2
id11/rv/2/ud_command_list: DevState
id11/rv/2/ud_poll_interval: 80

id11/rv/3/channum: 3
id11/rv/3/ud_command_list: DevState
id11/rv/3/ud_poll_interval: 80

id11/rv/4/channum: 4
id11/rv/4/ud_command_list: DevState
id11/rv/4/ud_poll_interval: 80

id11/rv/5/channum: 5
```

Figure 13.2: Greta server window

13.3.2 The server window

To open a server window, click on File–Open server. A database server browsing window is poped-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 poped-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 poped-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 poped-up.
- **Delete** to delete the server from database. A window is poped 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 poped_up. In all cases, a confirmation window is poped up.

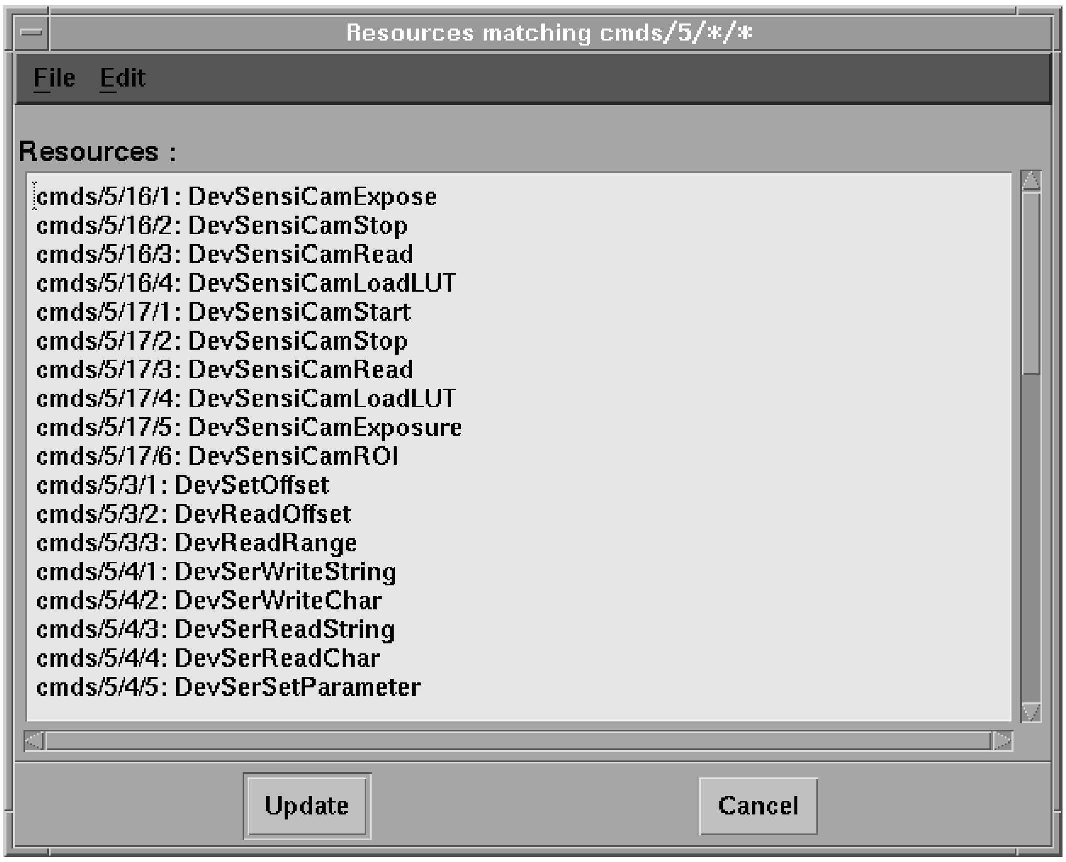
Under the window File button, it is possible to :

- 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.

13.3.3 The resource window

To open a resource window, click on File–Open resources. A database resource browsing window is poped-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 poped-up a server window. It is always proposed by greta to use the wildcard * as Member and/or Name field.



```
cmds/5/16/1: DevSensiCamExpose
cmds/5/16/2: DevSensiCamStop
cmds/5/16/3: DevSensiCamRead
cmds/5/16/4: DevSensiCamLoadLUT
cmds/5/17/1: DevSensiCamStart
cmds/5/17/2: DevSensiCamStop
cmds/5/17/3: DevSensiCamRead
cmds/5/17/4: DevSensiCamLoadLUT
cmds/5/17/5: DevSensiCamExposure
cmds/5/17/6: DevSensiCamROI
cmds/5/3/1: DevSetOffset
cmds/5/3/2: DevReadOffset
cmds/5/3/3: DevReadRange
cmds/5/4/1: DevSerWriteString
cmds/5/4/2: DevSerWriteChar
cmds/5/4/3: DevSerReadString
cmds/5/4/4: DevSerReadChar
cmds/5/4/5: DevSerSetParameter
```

Figure 13.3: Greta resource window

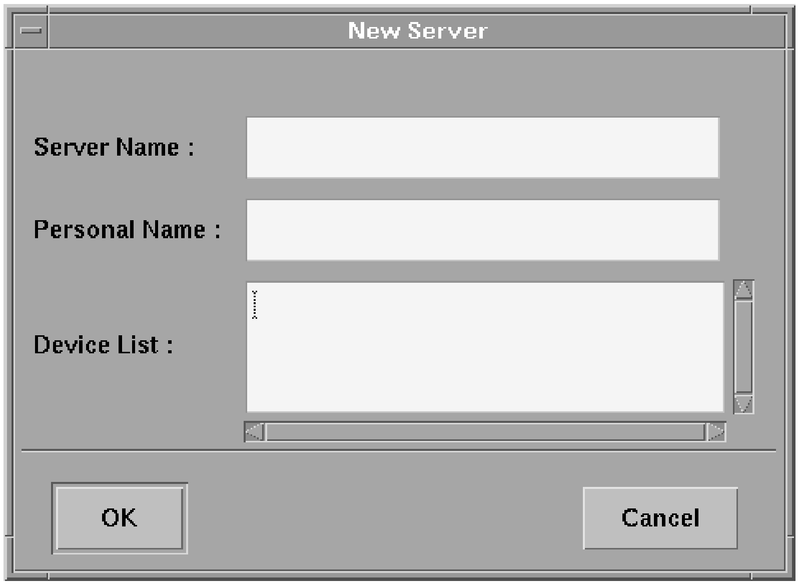


Figure 13.4: Greta new server window

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 poped-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

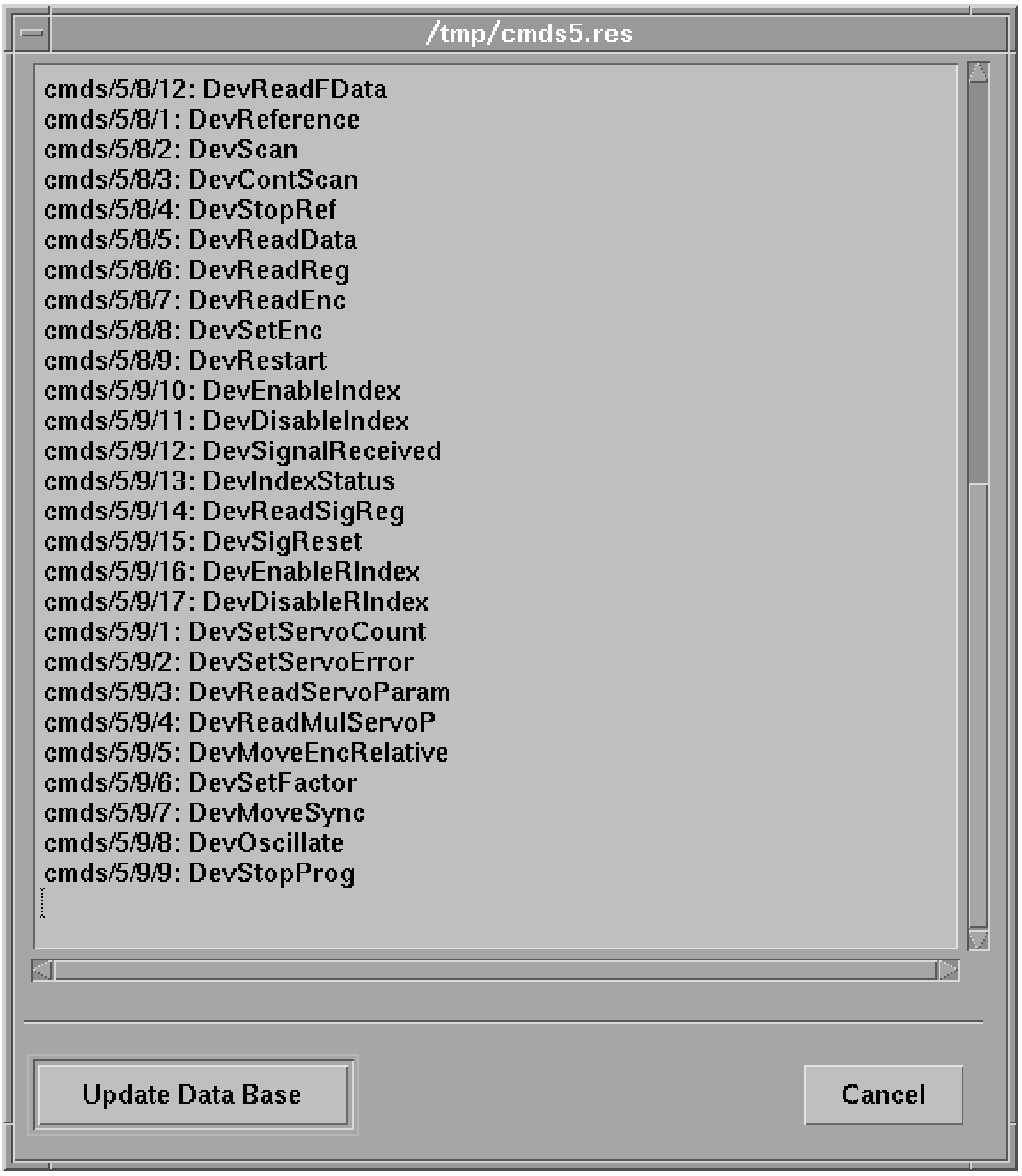
Under the Edit button, the user will find the classical edit features.
It is possible to open up to 10 different resources windows.

13.3.4 The new server window

The new server window allows a user to create new device server within the database. This window is poped-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.

13.3.5 The load file window

Once a file as 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 :



```
cmds/5/8/12: DevReadFData
cmds/5/8/1: DevReference
cmds/5/8/2: DevScan
cmds/5/8/3: DevContScan
cmds/5/8/4: DevStopRef
cmds/5/8/5: DevReadData
cmds/5/8/6: DevReadReg
cmds/5/8/7: DevReadEnc
cmds/5/8/8: DevSetEnc
cmds/5/8/9: DevRestart
cmds/5/9/10: DevEnableIndex
cmds/5/9/11: DevDisableIndex
cmds/5/9/12: DevSignalReceived
cmds/5/9/13: DevIndexStatus
cmds/5/9/14: DevReadSigReg
cmds/5/9/15: DevSigReset
cmds/5/9/16: DevEnableRIndex
cmds/5/9/17: DevDisableRIndex
cmds/5/9/1: DevSetServoCount
cmds/5/9/2: DevSetServoError
cmds/5/9/3: DevReadServoParam
cmds/5/9/4: DevReadMulServoP
cmds/5/9/5: DevMoveEncRelative
cmds/5/9/6: DevSetFactor
cmds/5/9/7: DevMoveSync
cmds/5/9/8: DevOscillate
cmds/5/9/9: DevStopProg
```

Figure 13.5: 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

13.3.6 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.

13.3.7 Other features

Some miscellaneous features are also incorporated into greta.
Global–Informations : 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

13.4 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.

13.5 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.

13.6 Database administration commands

13.6.1 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 alittle 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
```

13.6.2 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
```

13.6.3 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 alittle 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|unittype|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
```

13.7 Database user commands

13.7.1 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
```

13.7.2 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
```

13.7.3 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
```

13.7.4 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 wil 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
```

13.7.5 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
```

13.7.6 db_resdel

```
db_resdel <device name/resource name>
```

This command deletes a resource from the database. Example :

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

13.7.7 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
```

13.7.8 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
```

13.8 Security commands

13.8.1 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.

13.8.2 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.

13.8.3 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.

13.9 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.

13.10 Resource oriented calls

All the following calls are linked to resources

13.10.1 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.

13.10.2 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.

13.10.3 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.

13.11 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.

13.11.1 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.

13.11.2 db_freedevexp()

```
int db_freedevexp (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.

13.12 Device oriented calls

The following functions are device oriented.

13.12.1 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.

13.12.2 db_dev_import()

```
int db_dev_import (name, tab, dev_num, error)
    char           **name;         /* Device(s) name to be imported */
    Db_devinf_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.

13.12.3 db_dev_export()

```
int db_dev_export (devexp, dev_num, error)
    Db_devinf       *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.

13.12.4 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".

13.12.5 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.

13.12.6 db_devicedeleteres()

```
long db_devicedeleteres (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.

13.12.7 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,....

13.13 Server oriented calls

The following functions deals with device server.

13.13.1 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.

13.13.2 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.

13.13.3 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....

13.13.4 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.

13.13.5 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.

13.14 Database browsing oriented calls

All the following 11 calls allows database browsing

13.14.1 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.

13.14.2 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.

13.14.3 db_getdevmemberlist()

```
long db_getdevmemberlist(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 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.

13.14.4 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.

13.14.5 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.

13.14.6 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.

13.14.7 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.

13.14.8 db_getresresoval()

```
long db_getresresoval(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 (*).

13.14.9 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.

13.14.10 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.

13.14.11 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.

13.15 Pseudo device oriented calls

13.15.1 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.

13.15.2 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.

13.16 Database update calls

13.16.1 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.

13.16.2 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.

13.16.3 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.

13.17 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.

13.18 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 acess 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 14

Events
by A.Götz

14.1 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.

14.2 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`

14.3 API

The event API consists of three additional calls which are distributed as part of the DSAPI. The API consists of a client part and a server part. The client part allows a client to register its interest in events , to receive events and to unregister once it is finished. The server part allows servers to dispatch events to clients. The server has to program how to trigger events.

14.3.1 Client side

- dev_event_listen() - register a callback for an event type

  ```
  long dev_event_listen (devserver ds,  long event_type,
                         DevArgument argout, DevType argout_type
                         DevCallbackFunction *callback,
                         void *user_data, long *event_id_ptr,
                         long *error)

  devserver ds - device from which client wants to receive events
  long event_type - type of event to receive
  DevArgument argout - pointer to argout data (if any) which will be sent with event
  DevType argout_type - argout type
  DevCallbackFunction *callback - pointer to callback function
  void *user_data - pointer to user data to pass to callback function
  long *event_id_ptr - pointer to event id (returned by dev_event_listen())
  long *error - pointer to error code (if any)
  ```

- dev_event_unlisten() - unregister a callback for an event type

  ```
  long dev_event_unlisten (devserver ds, long event_type,
                           long event_id, long *error)

  devserver ds - device from which to unregister client's interest in event
  long event_type - event type to unregister
  long event_id - event id (returned by dev_event_listen())
  long *error - pointer to error code (if any)long dev_event_fire
  ```

- dev_synch() - poll network to check if any events have arrived and trigger callback

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

  struct timeval *timeout - pointer to maximum time to wait while polling
  long *error - pointer to error code (if any)
  ```

14.3.2 Server side

- dev_event_fire() - a server call to diispatch a user event to all clients which have registered their interest in that event with this server

  - C using Objects in C :

    ```
    long dev_event_fire (DevServer ds, 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)
```

– 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)
```

14.4 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.

14.5 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_asynch_timeout()) it wll 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 wil silently remove it from the list of registered clients.

14.6 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;

#if defined(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 appropiate point in the device server e.g. during class_initialise() or object_create().

14.7 Performance

The performance of events depends naturally on what type of system the device server is rnning on. Tests have been caried 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 tothe 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 millseconds wih 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
```

14.8 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 15

The Signal Interface
by J.Meyer and J-L.Pons

15.1 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.

15.2 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 sup- plies) 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.

15.3 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 ...).

15.4 The Server Side

15.4.1 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
```

15.4.2 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 DevMethodCheck-Alarms and the method used in the DevStatus command is DevMethodReadAlarms. See the Multi Signal Class Users Guide for more information.

15.5 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 fsigmon, 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:

15.5.1 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.
```

15.5.2 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.
```

15.5.3 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.
```

15.5.4 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.
```

15.5.5 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.
```

15.6 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 properies 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;
```

15.7 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);
}
```

15.8 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 16

LabView for TACO
by A.Götz

16.1 Introduction

This document describes the LabView interface to TACO control systems. It describes the DSAPI client interface and how to write TACO device servers in the LabView graphical programming language G.

16.2 Getting started

In order to get started you need access to the following tools :

1. LabView on Unix (Linux, HP-UX, Solaris or Windows)
2. for Unix : the LabView-TACO shared libraries lv_dsapi.so and lv_dsclass.so (only needed if you plan to write a device server in TACO) in your shared library path ($LD_LIBRARY_PATH for Linux/Solaris and$SHLIB_PATH for HP-UX)
3. for Windows : the LabView-Taco shared library lv_dsapi.dll and oncrpc.dll in your $PATH e.g. in c:
   Windows (you can download these two libraries as a zip file[1])
4. start LabView (normally by typing labview) and open one of the example VI's or wire your own following the instructions in this document.

16.3 LabView-TACO interface

16.3.1 Clients

The following shared library calls have been implemented to interface TACO clients to LabView :

1. `lv_dev_putget()` - will execute a command on a device, one input and output argument is l passed, an error code is returned and a status. The call has the following calling syntax :

[1]ftp://ftp.esrf.fr/pub/cs/taco/lv_taco.zip

- lv_dev_putget (char *name, char *cmd, void *argin, void *argout, long *error)
- name : device name e.g. "id11/oregon/1" (passed as C string type)
- cmd : command to execute e.g. "DevMoveRelative" (passed as C string type)
- argin : input argument e.g. array of floats (passed as "Adapt to Type" (specify "Handles by Value" on Labview 6i) type, this means the G program has to wire the correct input type expected by the command to the input argument e.g. if a double array is expected the LabView program has to provide a double array as input, failure to do so can result in a core dump of the LabView program ! NOTE: the input has to be wired even it is not used to a dummy type of the correct type; this is because LabView does not differentiate between not used input types when calling library functions and will not allow the program to run if an input is not wired - the RUN arrow is broken)
- argout: output argument e.g. array of floats (passed as "Adapt to Type" (specify "Handles by Value" on Labview 6i), see text above for advice on how to use this type. NOTE: the input has to be wired even it is an output to a dummy type of the correct type; this is because LabView does not differentiate between output only types when calling library functions and will not allow the program to run if an input is not wired - the RUN arrow is broken)
- error : output error if any (passed as pointer to 32 bit integer. See NOTE above for wiring the output error)
- returns : a 32 bit integer indicating the status of the command (0 = OK, -1 = NOT OK)
- NOTE : refer to section on types to know which are supported for argin and argout

2. `lv_dc_devget()` - will retrieve the result of a command on a device from the data collector (the TACO data cache), one output argument is passed, an error code is returned and a status. The call has the following calling syntax :

   - lv_dc_devget (char *name, char *cmd, void *argout, long *error)
   - name : device name e.g. "sr/d-ct/1" (passed as C string type)
   - cmd : command to execute e.g. "DevReadSigValues" (passed as C string type)
   - argout: output argument e.g. array of floats (passed as "Adapt to Type" (specify "Handles by Value" in Labview 6i), see text above for advice on how to use this type. NOTE: the input has to be wired even it is an output to a dummy type of the correct type; this is because LabView does not differentiate between output only types when calling library functions and will not allow the program to run if an input is not wired - the RUN arrow is broken)
   - error : output error if any (passed as pointer to 32 bit integer. See NOTE above for wiring the output error)
   - returns : a 32 bit integer indicating the status of the command (0 = OK, -1 = NOT OK)
   - NOTE : refer to section on types to know which are supported for argin and argout

3. `lv_dev_protocol()` - change the RPC protocol on a device to TCP or UDP. NOTE: the default is UDP. Changing to TCP will make client connections more reliable and allow correct error detection (e.g. server down error). The call has the following syntax :
   - lv_dev_protocol(char *name, char*protocol, long *error);
   - name : name of device e.g. "id11/oregon/1" (passed as C string type)
   - protocol : RPC protocol e.g. "tcp" or "udp" (passed as C string type)
   - error : error if any (passed a pointer to 32 bit integer)
   -
4. `lv_dev_timeout()` - change RPC timeout on a device. The call has following syntax :
   - lv_dev_timeout(char *name, long timeout, long *error);
   - name : name of device e.g. "id11/oregon/1" (passed as C string type)
   - timeout : new timeout in milliseconds (pass as 32 bit integer)
   - error : error code if any (passed as pointer to 32 bit integer)
5. `lv_dev_free()` - close down all connections to a device. This call is useful for releasing access to devices not used anymore. It closes all open network connections to the device thereby also saving open file descriptors. The first call to the device will reimport the device. The call has following syntax :
   - lv_dev_free(char *name, long *error);
   - name : name of device e.g. "id11/oregon/1" (passed as C string type)
   - error : error code if any (passed as pointer to 32 bit integer)
6. `lv_dev_error_str()` - return TACO error string corresponding to error_no. The call has following syntax :
   - char *lv_dev_error_str(long error_no);
   - error_no : error number
7. `lv_dev_cmd_query()` - return the list of commands supported by a device and their types. This call is mainly useful as info for the LabView programmer to know what commands and what data types are supported for a device . The call has following syntax :
   - long lv_dev_cmd_query(char *device, void* cmd_list, long *error);
   - device : the name of the device
   - cmd_list : list of commands and their input/output types returned as an array of strings (passed as "Adapt to Type", specify "Handles by Value" in Labview 6i)
   - error : pointer to error code in case routine fails
8. `lv_dc_cmd_query()` - return the list of commands polled by the data collector for a device and their types. This call is mainly useful as info for the LabView programmer to know what commands and what data types are supported for a device in the data collector. The call has following syntax :
   - long lv_dc_cmd_query(char *device, void* cmd_list, long *error);
   - device : the name of the device

- cmd_list : list of commands and their input/output types returned as an array of strings (passed as "Adapt to Type", specify "Handles by Value" in Labview 6i)
- error : pointer to error code in case routine fails

9. `lv_db_getdevexp()` - return the list of exported devices whose names satisfy the filter. The call has following syntax :
   - long lv_db_getdevexp(char *filter, void* device_list, long *error);
   - device : name filter of format D/M/F where either of D, F or M can be the wildcard *
   - device_list : list of device names returned as an array of strings (passed as "Adapt to Type", specify "Handles by Value" in Labview 6i)
   - error : pointer to error code in case routine fails

16.3.2 Servers

It is possible to write TACO device servers in LabView (described below). HOWEVER the preferred method to make Labview callable from TACO clients is to generate a shared library (DLL in Windows parlance) of your Labview program using the Application Builder and to call the shared library from a TACO device server. If you cannot generate a shared library from Labview then use the technique below.
TACO device servers in Labview work by implementing a device server loop in a part of the G program which polls the network periodically (using the lv_ds_cmd_get() call) to see if there are any client requests. Once a request is detected the LabView program has to execute it and then return the answer to the client (using the lv_ds_cmd_put()) call. Clients and servers communicate using the TACO RPC protocol on the network. The TACO devices have to be defined beforehand in the TACO database. The following shared library calls have been implemented to write TACO device servers in G :

1. `lv_ds_init()` - create and initialise a LabView TACO device server, to be called once in a LabView device server before calling lv_ds_cmd_get(). The call has the following calling syntax :
   - lv_ds_init(char *server, char*name);
   - server : device server executable name e.g. "StressRigds" (passed as C string type)
   - name : device server instance/personal name e.g. "id11" (passed as C string type)
2. `lv_ds_cmd_get()` - poll network to see if a client request has arrived, if yes command returned is non-zero. Can only be called after lv_ds_init() has been called. The call has the following syntax :
   - lv_ds_cmd_get(long *command, void *argin);
   - command : command received, can be one of DevLVIOStringDevState (1) takes array of strings as input and returns array of strings as output, DevLVIODouble (2) takes array of doubles as input and returns array of doubles as output, DevReadValue (3) returns an array of doubles as output, DevSetValue (4) takes array of doubles as input, DevState (5) and DevStatus (6) or 0 if no client request. DevState and DevStatus are handled automatically by the LabView device server

   - argin : passed as "Adapt to Type" (specify "Handles by Value" on Labview 6i).

3. `lv_ds_cmd_put()` - return result to client after completing executing of command. Must only be called after a lv_ds_cmd_get() has returned a non-zero command value. The call has the following calling syntax :
   - lv_ds_cmd_put(long command, void *argout);
   - command : command returned by lv_ds_cmd_get() (passed as 32 bit integer)
   - argout : double array to be passed back to client if command was DevIOLVString, DevIOLVDouble or DevReadValue (passed as "Adapt to Type" (specify "Handles by Value" on Labview 6i))

16.3.3 Debugging

An additional command exists to set the debugging flag in the library in order to display debugging information in a graphical window. The command is :

1. `lv_ds_debug(long debug)` : sets the debug flag. Setting the debug flag to a non-zero value switches on the debugging, zero switches it off.

16.4 Types

All TACO kernel types and motor types are supported. The following types are supported as input and output :

1. D_VOID_TYPE
2. D_SHORT_TYPE
3. D_LONG_TYPE
4. D_FLOAT_TYPE
5. D_DOUBLE_TYPE
6. D_STRING_TYPE
7. D_VAR_STRINGARR
8. D_VAR_SHORTARR
9. D_VAR_LONGARR
10. D_VAR_ULONGARR
11. D_VAR_FLOATARR
12. D_VAR_DOUBLEARR

The following types are supported as input only :

1. D_MOTOR_FLOAT
2. D_MULMOVE_TYPE

The following arguments are supported in output only :

1. D_LONG_READPOINT
2. D_FLOAT_READPOINT
3. D_DOUBLE_READPOINT
4. D_STATE_FLOAT_READPOINT
5. D_VAR_LRPARR
6. D_VAR_FRPARR
7. D_VAR_SFRPARR
8. D_OPAQUE_TYPE

4 Examples
There are examples of calling all the functions as well as examples of a TACO LabView device server (device_server.vi) and client (device_client.vi) are available. Study them to find out how to write your own clients and servers in LabView.

16.5 Known Problems

Some of the known problems with the TACO LabView interface are :

1. out of memory - sometimes LabView gives this error message when calling the lv_dev_putget() function. I don't know what this is due to. It seems to be occur when a string is passed as output. The only solution I have found is to rewire the corresponding (dummy) input for the output to another string type or to recreate it as a constant. This problem has been mostly occurring on HP-UX. If anyone finds a better explaination/solution for this problem let me know.
2. limited number of devices - the present interface is limited to 1000 TACO devices in a LabView session
3. not unregistered LabView device servers - there is no routine yet for unregistering LabView device servers. This is not a major problem. It simply means the number of reserved program numbers will go up every time a LabView device server is started.
4. blocked lv_dev_putget() call - this can happen on HPUX when trying to access a device running on a host which is not reachable from the host on which LabView is running. The present implementation will block. This problem does not occur on Linux.
5. client cannot run in same Labview session as server - the present version does not support running clients in the same Labview session as the device server.
6. too many files open - this problem can be encountered in Labview applications which have too many devices open. The latest (V1.5) version of the TACO interface sets the limit of the number of open files to the maximum operating system allowed value at the first call. If you still have problems then your application is too big ! Previous versions (¡V1.5) are limited by the default limit for no. of open files e.g. 60 on HP-UX.
7. Labview hangs - this can happen when accessing a device with UDP protocol and the server/host are down. The solution is to switch to TCP protocol.

8. Windows port is based on DSAPI V5.15. Data collector interface and lv_dsclass have not been ported (yet). Could be if the need arises.

9. the resizing of variable length string arrays as output does not work properly. It will downsize the input array but not upsize it. To get around this problem pass a string array initialised with more elements than you know will be returned. The array will then come back with the correct size. Apparently there is a leak with string arrays. If you see this please let me know so I can reproduce it and try to fix it.

16.6 Future developments

This is the sixth release of the LabView TACO interface. A number of improvements been made e.g. adding support for the data base, dynamic device management, increasing limit on open files. In the future we plan to offer a VI library for TACO which will reduce the programming effort on clients even more. If any readers have ideas for other improvements they should send their comments to the author (goetz@esrf.fr) or even better add them to the source code themselves and then send the code to the author !

Chapter 17

Python and TACO
by M.C.Dominguez and J.Meyer

17.1 Introduction

Python is fast becoming a very popular language for doing almost anything including writing TACO clients and servers. This chapter documents (briefly) the TACO Python interface for clients (written by M-C.Dominguez) and servers (written by Jens Meyer).

17.2 Client interface

The TACO client interface in Python is based on an object model. The commands to a device are dynamically added to the list of methods for the device. Here is an example of using the TACO client interface in Python :

```
file TacoDevice.py

x=Device("MCD/maxe032_1/1")
print x
x.CommandList()
x.tcp()
x.udp()
print x.timeout()
x.timeout(2)
a = x.DevReadEncPos(2)
aa=array([0,1,2,3,4,5,6,7],Float32)
x.DevReadMulVel(0,1,2,3,out=aa)
bb=DevReadMulVel(0,1,2,3,outtype='numeric')

dev_putresource("MCD/maxe032_1/1","toto","5")
dev_getresource("MCD/maxe032_1_1/1","axe_ident0")
dev_delresource("MCD/maxe032_1/1","toto")
```

In addition to the dev.command() interface the following calls are defined in the interface :

1. `dev_debug(flag)` - sets python functions debug flag
   - *input* : debug flag (0: no trace, else: trace)

- *returns* : - 0: error - 1: OK

2. `dev_init(mdevname)` - inits the class instance used by x=Device("MCD/maxe032_1/1")
   - *input* : device name
   - *returns* : list ([] if error, or [devname,cpt] devname: mdevname in lowercase cpt: index in C device table)
3. `dev_unref(mdevname)` - decrement reference to that object in Tab_dev table If reference becomes 0, calls the C dev_free routines:
   - *input* : device name
   - *returns* : - 0 : error, - 1 : OK
4. `dev_query(mdevname)` - asks for database device command list
   - *input* : device name in lower case
   - *returns* : dictionnary (if error else cmd_name:[cmd,in_type,out_type], ... where cmd_name: command string, cmd: command numeric value, in_type: input type, and out_type: output type)
5. `dev_tcpudp(mdevname,mode)` - Sets mode tcp/udp for the device
   - *input* : device name in lower case, mode= "tcp" or "udp"
   - *returns* : - 0 if error, - 1 if OK
6. `dev_timeout(mdevname,*mtime)` - Asks for or set the device timeout
   - *input* : device name in lower case, - mtime: optional argument: - if not existing: read timeout required - if exists: time in second for setting timeout
   - *returns* : - 0 if error, - time in sec (read or set) if OK
7. `dev_getresource(mdevname,resname)` - Gets a device resource
   - *input* : mdevname= device name, resname= resource name
   - *returns* : value packed as a string (resource value if OK, else None if error)
8. `dev_putresource(mdevname,resname,value)` - Sets a device resource
   - *input* : mdevname=device name, resname=resource name, value=the resource value packed as a string
   - *returns* : value (1 if OK, 0 if error)
9. `dev_delresource(mdevname,resname)` - removes a device resource
   - *input* : mdevname=device name, resname=resource name
   - *returns* : value (1 if OK, 0 if error)
10. `dev_io(mdevname,mdevcommand,*parin,**kw)` - sends a command to a device
    - *input* : mdevname=device name in lower case, parin=list of optional INPUT parameters, kw=dictionnary of optional OUTPUT parameters
    - *returns* : value (1 if no device ARGOUT or OUTPUT param provided, device ARGOUT if device ARGOUT and no OUTPUT param)

List of taco types handled by the C interface :

```
D_VOID_TYPE
D_BOOLEAN_TYPE
D_USHORT_TYPE
D_SHORT_TYPE
D_ULONG_TYPE
D_LONG_TYPE
D_FLOAT_TYPE
D_DOUBLE_TYPE
D_STRING_TYPE

D_INT_FLOAT_TYPE
D_FLOAT_READPOINT
D_LONG_READPOINT
D_DOUBLE_READPOINT
D_MOTOR_LONG
D_MOTOR_FLOAT
D_STATE_FLOAT_READPOINT
D_MULMOVE_TYPE

D_VAR_CHARARR
D_VAR_STRINGARR
D_VAR_USHORTARR
D_VAR_SHORTARR
D_VAR_ULONGARR
D_VAR_LONGARR
D_VAR_FLOATARR
D_VAR_DOUBLEARR
D_VAR_FRPARR
D_VAR_SFRPARR
D_VAR_LRPARR
D_OPAQUE_TYPE
```

For numeric types, the correspondance C to Python numeric is :

```
D_VAR_CHARARR         Int8
D_VAR_USHORTARR         Int16
D_VAR_SHORTARR         Int16
D_VAR_ULONGARR         Int32
D_VAR_LONGARR         Int32
D_VAR_FLOATARR         Float32
D_VAR_DOUBLEARR         Float64
```

17.3 Server interface

Thanks to the work by Jens Meyer it is possible to write TACO device servers in Python. The documentation is in the form of examples (yes it is so easy).

17.3.1 Example 1

The first example is :

```
TacoServer  import *
```

```
class MyServer (TacoServer):
    "This is a test class"

#    Common variables for a class

    my_cmd_list = { DevState :    [D_VOID_TYPE, D_SHORT_TYPE , 'state'],
             DevStatus:       [D_VOID_TYPE, D_STRING_TYPE, 'status'],
             DevOn:       [D_VOID_TYPE, D_VOID_TYPE ,  'on'],
             DevOff:       [D_VOID_TYPE, D_VOID_TYPE ,  'off'],
             DevSetValue:    [D_FLOAT_TYPE, D_VOID_TYPE , 'set'],
             DevSetParam:    [D_VAR_FLOATARR, D_VOID_TYPE , 'set_array'],
             DevReadValue:    [D_VOID_TYPE, D_FLOAT_TYPE,  'read'],
             DevReadSigValues:    [D_VOID_TYPE, D_VAR_FLOATARR,'read_signals'],
             DevGetDevs:    [D_VOID_TYPE, D_VAR_STRINGARR,'read_names'],
             DevSetDevs:    [D_VAR_STRINGARR, D_VOID_TYPE,'set_names'] }

    class_name = "TestClass"

    value      = 123.4
    names      = ('no', 'input')
    array       = (1,2,3)

#    rem_device = Dev('id/python/test4')

    def __init__ (self, name):
        TacoServer.__init__ (self, name, command_list=self.my_cmd_list)

        res = dev_getresource (self.dev_name, "value")
        if res != None:
            self.value = float(res);
        return

    def state (self):
#        print 'remote device status:'
#        print self.rem_device.DevStatus()
        return self.dev_state

    def status (self):
        if self.dev_state == DEVUNKNOWN:
            self.dev_status = "The device is in an unknown state"
        elif self.dev_state == DEVON:
            self.dev_status = "The device is switched ON"
        elif self.dev_state == DEVOFF:
            self.dev_status = "The device is switched OFF"
        return self.dev_status

    def on (self):
        self.dev_state = DEVON

    def off (self):
        self.dev_state = DEVOFF

    def read (self):
        return self.value

    def set (self, x):
        print x

        if x > 100:
            Server.error.taco_error = DevErr_ValueOutOfBounds
            raise Server.error

        elif x < 0:
            x / 0

        self.value = x
        return

    def read_signals (self):
        return self.array

    def read_names (self):
        return self.names

    def set_names (self, in_names):
            #
        # A copy to a new tuple is needed here!!!
        # self.names = in_names
        # will result in a memory fault when
        # executing read_names!
        #
        self.names = ()
        for i in in_names:
            self.names = self.names + (i, )
        print self.names
        return

    def set_array (self, x):
            #
        # A copy to a new tuple is needed here!!!
        # self.array = x
        # will result in a memory fault when
        # executing read_signals!
        #
        self.array = ()
        for i in x:
            self.array = self.array + (i, )
        print self.array
        return
```

And a script to create the server and start it :

```
#
# Example script how to start a
# Taco device server written in Python
#

import TacoServer
import MyServer

def start():
    #
    # Create two device objects
    #
    x=MyServer.MyServer ('id/python/test1')
    y=MyServer.MyServer ('id/python/test2')

    #
    # Put the two objects to be exported
    # on the network in a tuple
    #
    dev=(x,y)

    #
    # Export to the network and start the
    # device server thread
    #
    # With a device server definition in the resource
    # database as:
    # Python/test/device:    id/python/test1 \
    #             id/python/test2
    #
    #
    TacoServer.server_startup (dev, process_name='Python', server_name='test')
```

17.3.2 Example 2

Here is a second example on how to write a device server in Python :

```
from TacoServer  import *


class YourServer (TacoServer):
    "This is another test class"

#    Common variables for a class

    cmd_list = { DevState :        [D_VOID_TYPE, D_SHORT_TYPE , 'state'],
             DevStatus:        [D_VOID_TYPE, D_STRING_TYPE, 'status'],
             DevOpen:        [D_VOID_TYPE, D_VOID_TYPE ,  'open'],
             DevClose:        [D_VOID_TYPE, D_VOID_TYPE ,  'close'],
             DevReadSigValues:    [D_VOID_TYPE, D_VAR_FLOATARR,'read_signals']}

    class_name = "YourTestClass"
    value      = 123.4

    def __init__ (self, name):
        TacoServer.__init__ (self, name)
        return

    def state (self):
        return self.dev_state
    def status (self):
        if self.dev_state == DEVUNKNOWN:
            self.dev_status = "The device is in an unknown state"
        elif self.dev_state == DEVOPEN:
            self.dev_status = "The device is Open"
        elif self.dev_state == DEVCLOSE:
            self.dev_status = "The device is Closed"
        return self.dev_status

    def open (self):
        self.dev_state = DEVOPEN

    def close (self):
        self.dev_state = DEVCLOSE

    def read_signals (self):
        signals = (self.dev_state, self.value)
        return signals
```

For more information please contact the authors directly - `domingue@esrf.fr` and `meyer@esrf.fr`.

Chapter 18

Access Control and Security *by J.Meyer*

18.1 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].

18.2 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.

18.3 The Model

18.3.1 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 18.1: The control system access table

18.3.2 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.
- SI_SU_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.

18.3.3 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/Member	Name	Acess
user	SR/V-RV/C1-3	meyer	SU_ACCESS
user	SR/V-RV	meyer	SI_WRITE_ACCESS
		taurel	WRITE_ACCESS
user	SR	operator	WRITE_ACCESS
		meyer	WRITE_ACCESS
group	SR/V-RV/C1-3	dserver	ADMIN_ACCESS
group	SR/V-RV	vacuum	SI_SU_ACCESS
group	SR	dserver	WRITE_ACCESS
		operator	WRITE_ACCESS
default			READ_ACCESS

Figure 18.2: 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.

18.3.4 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 off a device. By verifying this key all parameters for the open client connection to a device can guaranteed unchanged. Nothing can be modified on the connection. Parameters necessary to check the device and command access are send 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.

18.4 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.

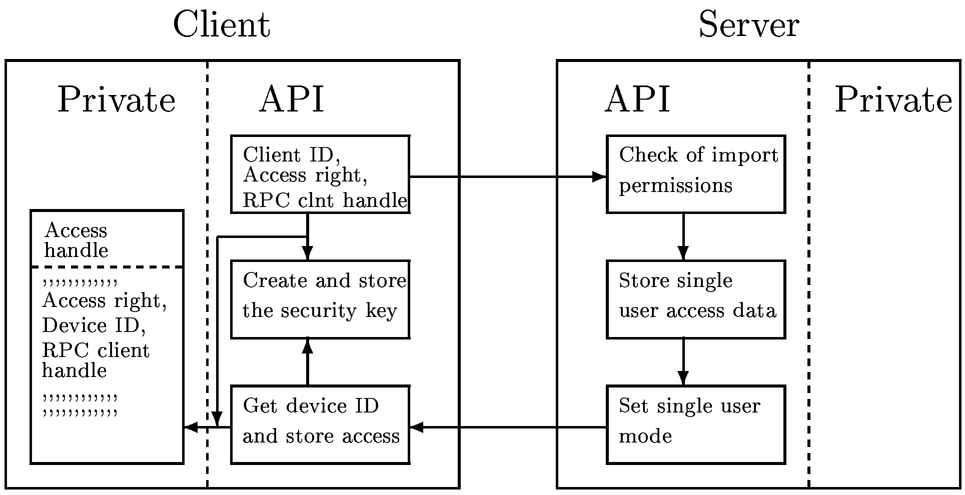

Client
Server
Private
API
API
Private
Client ID, Access right, RPC clnt handle
Check of import permissions
Access handle
,,,,,,,,,,,,
Access right, Device ID, RPC client handle
,,,,,,,,,,,,
,,,,,,,,,,,,
Create and store the security key
Store single user access data
Get device ID and store access
Set single user mode

Figure 18.3: The security key creation

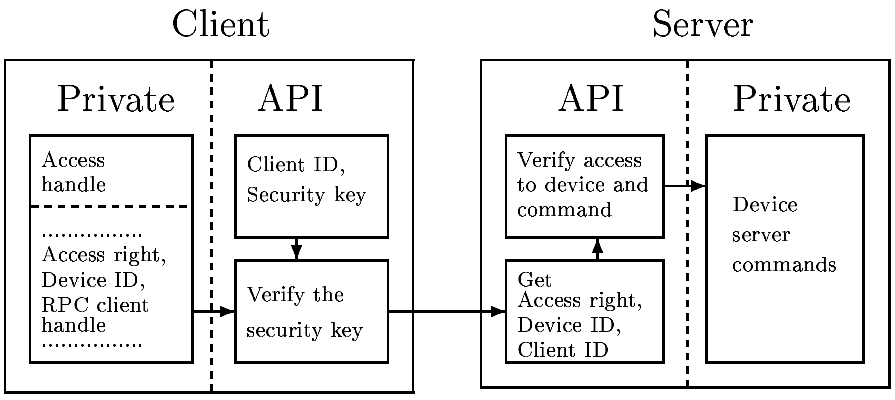


Figure 18.4: Access control with the Security Key

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.

18.5 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

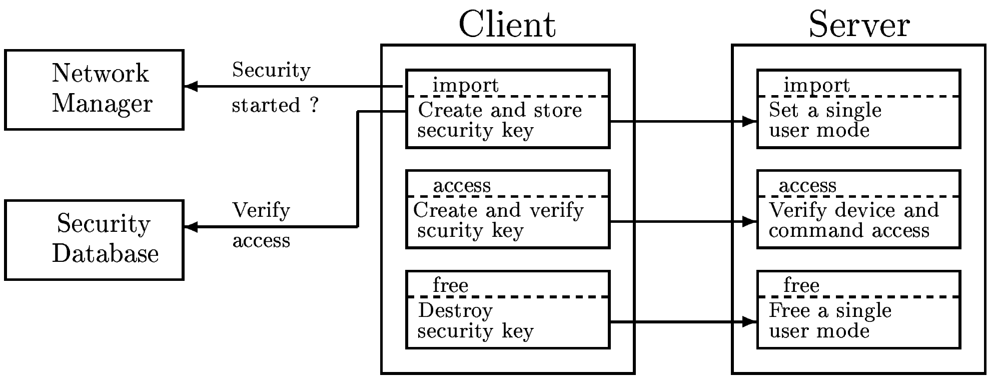


Figure 18.5: The security system integration to the API

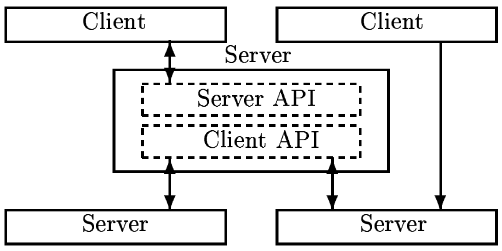


Figure 18.6: Server - server connections

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.

18.6 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

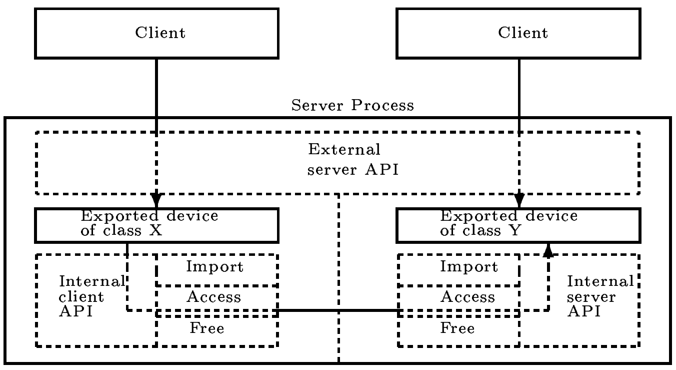


Figure 18.7: Internal and external API

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.

18.6.1 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!

18.6.2 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 predefined 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 specifyed 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 SI_WRITE_ACCESS and SI_SU_ACCESS will set the device into single user mode.
- Trying to import a device with SI_WRITE_ACCESS or SI_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))
```

18.6.3 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 19

Standard Makefiles using GNU make (gmake) *by A.Götz*

19.1 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.

19.2 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 philisophy 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 dependancies at make time. The latter approach is the one adopted for TACO and described in this chapter.

19.3 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 substition 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.

19.4 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 Motorala 68040
10. `_UCC` - (new) Ultra C/C++ compiler for OS9
11. `sun` - SunOS running on SPARC
12. `OSK` - (old) Unibridge compiler for OS9

19.5 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")

19.6 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 simple define the appropriate variables for that platform in the Makefile and then call gmake without any arguments.

19.7 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
OBJS_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.l \
                -l netdb_small.l \
                -l socklib.l \
                -l sys_clib.l \
                -l unix.l

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 -O 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$(VERSION)
#               RCS check out options
RCSCO   =       co -r$(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
ifdef unix
                ar rv $(LIB_HOME)/$(CLASS_LIB) $(CLASS_OBJS)
endif


#
#                 install executables
#
ifdef _UCC
install:        $(SERVER) $(CLIENT) $(CLASS_LIB)
                cp $(SERVER) $(INSTALL_HOME)/$(SVC_INST)
                cp $(CLIENT) $(INSTALL_HOME)/$(CLN_INST)
endif
ifdef 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:
                $(RCSCO) $(SRC)
                cd ../include; $(RCSCO) $(INCL); cd ../src

ci:
                $(RCSCI) $(SRC)
                cd ../include; $(RCSCI) $(INCL); cd ../src
```

19.8 Further Reading

1. *GNU Make* by Richard M. Stallman and Roland McGrath

Chapter 20

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.[1] 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".

[1] 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 21

A tool to test a TACO control system
by E.Taurel

21.1 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 netwok 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.

21.2 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.

21.3 Usage example

21.4 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
$
```

21.5 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
$
```

21.6 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).

21.7 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 exmaple does not use the verbose mode of testcs. From the output, you can conclude that

- All the kernel conponents 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 22

Adding Private Commands, Errors and XDR Data Types

by J.Meyer and A.Götz

22.1 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.

22.2 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);
}

.
.
.
}
```

22.3 Error Numbers

The error number, defined as a long word, is split into four different fields:

```
| 31       26 | 25             18 | 17      12 | 11            0 |
-------------------------------------------------------------
      |              |            |            |
      |              |            |            |- Error Number
      |              |            |
      |              |            |- Error category
      |              |
      |              |- Device Server Identification
      |
      |- Team Number
```

- Team Number:
  A uniq 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.

22.4 Command Numbers

The command number, defined as a long word, is split into three different fields:

```
| 31        26 | 25           18 | 17                0 |
---------------------------------------------------
```

```
|                |              |
|                |              |- Command Number
|                |
|                |
|                |
|                |- Device Server Identification
|
|- Team Number
```

The distribution of **Team Number** and **Device Server Identification** is the same as described in the last section.

22.5 Database Support

To avoid linking with long lists of error messages or command name strings, all this text information is now stored as resources, in two new tables, of the static database. The new tables are **ERROR** for error messages and **CMDS** for command name strings. To specify a private error and command in a device server class, the following defines and resources are necessary.
Define the error code:

#define DevErr_MyError   DevMyBase + Error_Number

Specify the error string as a resource in the database. Use the Team Number as defined in DserverTeams.h and the Device Server Identification as defined, for your class, in your programming team's identification file, in the resource path :

ERROR/Team_Number/DS_Identification/Error_Number: "Error Message"

Example:

```
#define DevErr_SetHighLimit      DevMcceBase + 15
ERROR/2/3/15:                    "Unable to set polarization high limit"

#define DevSetHighLimit          DevMcceBase + 15
CMDS/2/3/15:                     "DevSetHighLimit"
```

All general errors and commands as they are defined in the include files **DevErrors.h** and **DevCmds.h** are loaded in the database as resources with the Team_Number = 0 and the DS_Identification = 0. Only the error messages for API and database errors are kept in a global error list.
In all versions of the API-library, starting with version **3.20**, the functions **dev_printerror_no()**, **dev_error_str()** and **dev_cmd_query()** use error and command resource definitions. To relink older software should not cause problems, as long as these functions are used and the global lists are not directly accessed.

22.6 Time Stamp for Error Messages

All error strings created by the API-library functions **dev_printerror_no** and **dev_error_str()** include a time stamp before the error message. The returned error strings are in the format:

"Sun Sep 16 01:03:52 1993   This is my error message."

A description of the two error functions can be found in the man page **dev_error.3x**.

22.7 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.

22.8 Private XDR Data Types

22.8.1 Data Type Numbers

The data type number, defined as a long word, is split into three different fields:

```
| 31       26 | 25           18 | 17               0 |
---------------------------------------------------
      |               |              |
      |               |              |- Data Type Number
      |               |
      |               |
      |               |
      |               |- Device Server Identification
      |
      |- Team Number
```

The distribution of **Team Number** and **Device Server Identification** is the same as described in section 2.

22.8.2 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);
}
```

22.8.3 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.

22.8.4 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_AXEREAD, D_VAR_PARREAD, D_VAR_ERRREAD
21. grp_xdr.h : D_GRPFP_TYPE
22. pin_xdr.h : D_PINSTATE_TYPE

22.9 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

22.9.1 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.

22.10 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_Identifaction 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 23

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`) or Marie-Christine Dominguez (`domingue@esrf.fr`)
- **Tcl** - contact Gilbert Pepellin (`pepellin@esrf.fr`)
- **MathLab** - contact Laurent Farvacque (`laurent@esrf.fr`) or Francis Epaud (`epaud@esrf.fr`)
- **LabView** - contact Andy Götz (`goetz@esrf.fr`)
- **SPEC** - contact Gerry Swislow (`info@certif.com`)

For more information refer to the website or contact the person involved directly.

Appendix A

Device Server Catalogue

A.1 Introduction

Looking for a device server ? Thinking of writing a device server but you don't know if it is written ? Maybe this catalogue can help you. Literally hundreds of TACO device servers exist (over two hundred alone at the ESRF). This list presents device servers which could be of general interest to other users (most of them support commercial hardware) written at the ESRF and other sites using TACO (FRMII, HartRAO, Lure, etc.). Device servers for site specific hardware are not listed here, refer to the each site's documentation for these. The present list is far from complete so if you know of a device server which is not listed on this page but you think could be of interest to other users please send an email to *goetz@esrf.fr*.

A.2 Motors

A.2.1 Oregon (VME/PC-104)

- **description** : a device server for the multiple axes motor controller Oregon VME58 (VME format) and PC68 (PC-104 format) cards from OMS (http://www.omsmotion.com). The device server supports the Maxe device server interface. The same device server supports both the VME and PC-104 card.
- **author(s)** : Andy Götz (goetz@esrf.fr)
- **documentation** : none
- **hardware** : VME, VME58 (VME motor control from OMS), CC133 (VME relative encoder) [optional], or PC-104 + PC68 (PC-104 motor controller from OMS)
- **platforms** : Linux/68k (VME) and Linux/x86 (PC-104)
- **language** : written in C++
- **note** : maximum steprate is 1 MHz

A.2.2 Galil (VME)

- **description** : a device server for the DMC 1300 DC motor controller VME card from Galil. Device server supports an external gate synchronised to the motor position. Microprograms can be downloaded and executed.

- **author(s)** : M.Perez (perez@esrf.fr)
- **documentation** : DSUG174
- **hardware** : VME + DMC1300 motor card.
- **platforms** : OS9
- **language** : written in C
- **note** :

A.2.3 Flexmotion (cPCI)

- **description** : a device server for the multiple axes motor controller FlexMotion from National Instruments for compact PCI. The device server supports the Maxe device server interface. Presently no support for encoders or micro-programming.
- **author(s)** :Andy Götz (goetz@esrf.fr)
- **documentation** : none
- **hardware** : cPCI, FlexMotion (cPCI card)
- **platforms** : Linux/x86
- **language** : written in C++
- **note** : only used in the lab

A.2.4 Huber (GPIB)

- **description** : a device for controlling Huber motors via GPIB.
- **author(s)** : V.Rey (rey@esrf.fr)
- **documentation** : DSUG106
- **hardware** : Huber motors + IOTech SCSI-GPIB controller
- **platforms** : Solaris
- **language** : written in C
- **note** :

A.2.5 Berger (serial line)

- **description** : device server for Berger motor controller via serial line.
- **author(s)** : C.Penel (penel@esrf.fr)
- **documentation** : DSUG048
- **hardware** : VME + serial line + Berger motor controller
- **platforms** : OS9
- **language** : written in C
- **note** :

A.2.6 VPAP (VME)

- **description** : device server for the ESRF developed 8-axes motor controller for VME
- **author(s)** : M.C.Dominguez (domingue@esrf.fr) (originally T.Mettälä)
- **documentation** : DSUG093
- **hardware** : VME + VPAP
- **platforms** : OS9 and Linux
- **language** : C (OS9) and C++ (Linux)
- **note** :

A.3 CCD Cameras

A.3.1 Sensicam (PC/Windows)

- **description** : device server for the fast readout 12 bit Sensicam CCD cameras from Optimas.
- **author(s)** : Vicente Rey (rey@esrf.fr), Andy Götz (original version)
- **documentation** : none
- **hardware** : Windows PC + Sensicam interface card + CCD camera
- **platforms** : Windows
- **language** : written in C
- **note** :

A.3.2 Matrox (PC/Windows)

- **description** : device server for Matrox family of video grabbers.
- **author(s)** : Jens Meyer (meyer@esrf.fr) + Holger Witsch (witsch@esrf.fr)
- **documentation** : none
- **hardware** : Windows PC + Matrox frame grabber + CCD camera
- **platforms** : Windows
- **language** : written in C
- **note** : supports provided for doing image analysis on acquired image

A.3.3 Medoptics (PC/Windows)

- **description** : device server for 16 bit CCD camera from Medoptics.
- **author(s)** : Andy Götz (goetz@esrf.fr)
- **documentation** : none
- **hardware** : Windows PC + Medoptics interface card + CCD camera
- **platforms** : Windows
- **language** : written in C
- **note** :

A.3.4 Imagepro (PC/Windows)

- **description** : device server for image acquisition and analysis software ImagePro Plus from Media Cybernetics. Supports acquiring images and calling macros.
- **author(s)** : Andy Götz (goetz@esrf.fr)
- **documentation** : DSUG205
- **hardware** : Windows PC + ImagePro + CCD camera
- **platforms** : Windows
- **language** : written in C
- **note** : ImagePro supports a large number of interface cards and cameras e.g. Matrox, Sensicam, Photonics Science, TWAIN, ..., and has been extended at the ESRF to support the Frelon and Medoptics.

A.3.5 Mar (PC/Linux)

- **description** : device server for the Mar CCD camera. Device server is usually forked by the Mar GUI application. Communication is via pipes.
- **author(s)** : A. Götz
- **documentation** : DSUG211
- **hardware** : Linux PC + Mar CCD
- **platforms** : Linux/x86
- **language** : written in C++
- **note** :

A.3.6 Frelon (PCI/Linux/Solaris)

- **description** : device server for the ESRF developed Frelon CCD camera. This is a true 14 bit camera with tens of millisecond readout times. 1kx1k or 2kx2k (Frelon 2000).
- **author(s)** : D.Fernandez (dfernandez@esrf.fr)
- **documentation** : ?
- **hardware** : Linux PC or Solaris workstation + Frelon CCD
- **platforms** : Linux/x86 or Solaris
- **language** : written in C
- **note** :

A.4 Data Analysis

A.4.1 Matlab (Unix/Windows)

- **description** : a device server which starts a Matlab engine and allows clients to send and get arrays and/or strings and evaluate Matlab commands. This allows remote clients to collect data, send it to Matlab and analyse it in an automatic fashion.
- **author(s)** : A.Götz (goetz@esrf.fr)
- **documentation** : DSUG212
- **hardware** : Matlab licence
- **platforms** : Windows and Linux
- **language** : two version, one written in C and one in C++
- **note** : C version for Windows, C++ version for Linux

A.5 Sample Environment

A.5.1 Linkam Thermal Stage (serial line)

- **description** : a sample temperature environment controller from Linkam for controlling temperature.
- **author(s)** : Andy Götz (goetz@esrf.fr)
- **documentation** : DSUG213
- **hardware** : Linkam device + serial line
- **platforms** : Linux/x86
- **language** : C++
- **note** : `http://www.linkam.co.uk`

A.5.2 Impac Pyrometer (serial line)

- **description** : a pyrometer for reading temperature at a distance from Impac. Models exist for low (300 to 1200) and high (300 to 3000) temperatures.
- **author(s)** : Andy Götz (goetz@esrf.fr)
- **documentation** : ?
- **hardware** : Impac pyrometer + serial line
- **platforms** : Linux/x86
- **language** : C++
- **note** : `http://www.ir-impac.com`

A.6 Input/Output

A.6.1 Wago (serial line / ethernet)

- **description** : a device server for the Wago modbus 750 series of input/output modules
- **author(s)** : A.Götz (goetz@esrf.fr)
- **documentation** : Wago.pdf
- **hardware** : serial line or ethernet
- **platforms** : Linux/x86
- **language** : written in C++
- **note** : `http://www.wago.com`

A.6.2 Redlion Thermocouple (serial line)

- **description** : a device server for the Redlion thermocouple controller.
- **author(s)** : A.Götz (goetz@esrf.fr)
- **documentation** : Redlion.pdf
- **hardware** : serial line
- **platforms** : Linux/x86
- **language** : written in C++
- **note** : `http://www.redlion-controls.com`

A.6.3 ICV150 (VME)

- **description** : a device server for the multichannel VME ADC card ICV150 from ADAS.
- **author(s)** : A.Beteva (beteva@esrf.fr)
- **documentation** : DSUG177
- **hardware** : VME + ICV150
- **platforms** : OS9
- **language** : written in C
- **note** :

A.6.4 ICV712/ICV716 (VME)

- **description** : a device server for the 8/16 channel DAC VME card ICV712/ICV716 from ADAS
- **author(s)** : A.Beteva (beteva@esrf.fr)
- **documentation** : DSUG046
- **hardware** : VME + ICV712/ICV716
- **platforms** : OS9
- **language** : written in C
- **note** :

A.6.5 ICV101 (VME)

- **description** : a device server for the fast (MHz) analog to digital VME card ICV101 with on board memory from ADAS.
- **author(s)** : F.Epaud (epaud@esrf.fr), T.Mettälä (original version)
- **documentation** : DSUG109
- **hardware** : VME + ICV101
- **platforms** : OS9
- **language** : written in C
- **note** :

A.6.6 DM5210 (PC-104)

- **description** : a device server for the PC-104 analog and digital input/ output card DM5210.
- **author(s)** : A.Götz (goetz@esrf.fr)
- **documentation** : none
- **hardware** : PC-104 + DM5210
- **platforms** : Linux/x86
- **language** : written in C++
- **note** :

A.7 Counters/Timers

A.7.1 Lecroy 1151 (VME)

- **description** : device server for the VME 1151 counter from Lecroy.
- **author(s)** : F.Epaud (epaud@esrf.fr)
- **documentation** : DSUG087
- **hardware** : VME + Lecroy 1151
- **platforms** : OS9
- **language** : C
- **note** :

A.7.2 CAEN V462 (VME)

- **description** : device server for the VME V462 gate generator from CAEN.
- **author(s)** : F.Epaud (epaud@esrf.fr)
- **documentation** : DSUG088
- **hardware** : VME + V462
- **platforms** : OS9
- **language** : C
- **note** :

A.8 Multichannel Analysers

A.8.1 Canberra AIM (PC/Windows or Unix)

- **description** : device server for the AIM MCA from Canberra. The AIM is interfaced via Ethernet. The device server can run on Windows using the Canberra libraries or on Unix (HP-UX and Solaris) using libraries written by the ESRF.
- **author(s)** : A.Beteva (beteva@esrf.fr)
- **documentation** : DSUG146
- **hardware** : AIM module + PC/Workstation
- **platforms** : Windows + HP-UX + Solaris
- **language** : C
- **note** :

A.9 Image Plates

A.9.1 MAR345 (PC/Linux)

- **description** : a device server for the MAR345 image plate scanner from Mar.
- **author(s)** : L.Claustre (claustre@esrf.fr)
- **documentation** : DSUG207
- **hardware** : MAR345 scanner
- **platforms** : Linux
- **language** : C++
- **note** :

Appendix B

Licence