Device Server Programmer's Manual, Version 2.0

A. Götz

April 6, 1993

Abstract

Device servers are server processes which provide network access to devices. A device can be a logical or physical entity. Network access is provided by a set of remote procedure calls which implement a standard protocol. Clients access devices using a standardised application programmers interface. Device servers have to follow a certain style. Entry points are fixed so that the same application programmers interface can be used for all kinds of device. Each device is treated as an object belonging to a certain class. The classes are implemented in C and support object oriented programming techniques. This programmer's manual explains how to write a device server. It explains how to write a Device Server from scratch and what should go into a device server. It gives hints on how to abstract a device into a class, how to handle device startup, what commands to define and how to pass parameters. It also contains an elementary discussion on how to treat errors. The manual is aimed at the beginner and advanced device server programmer.

Contents

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.

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.

3 The Device Server Model

This section will present the device server model (see figure 1), 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).

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.

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.

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).

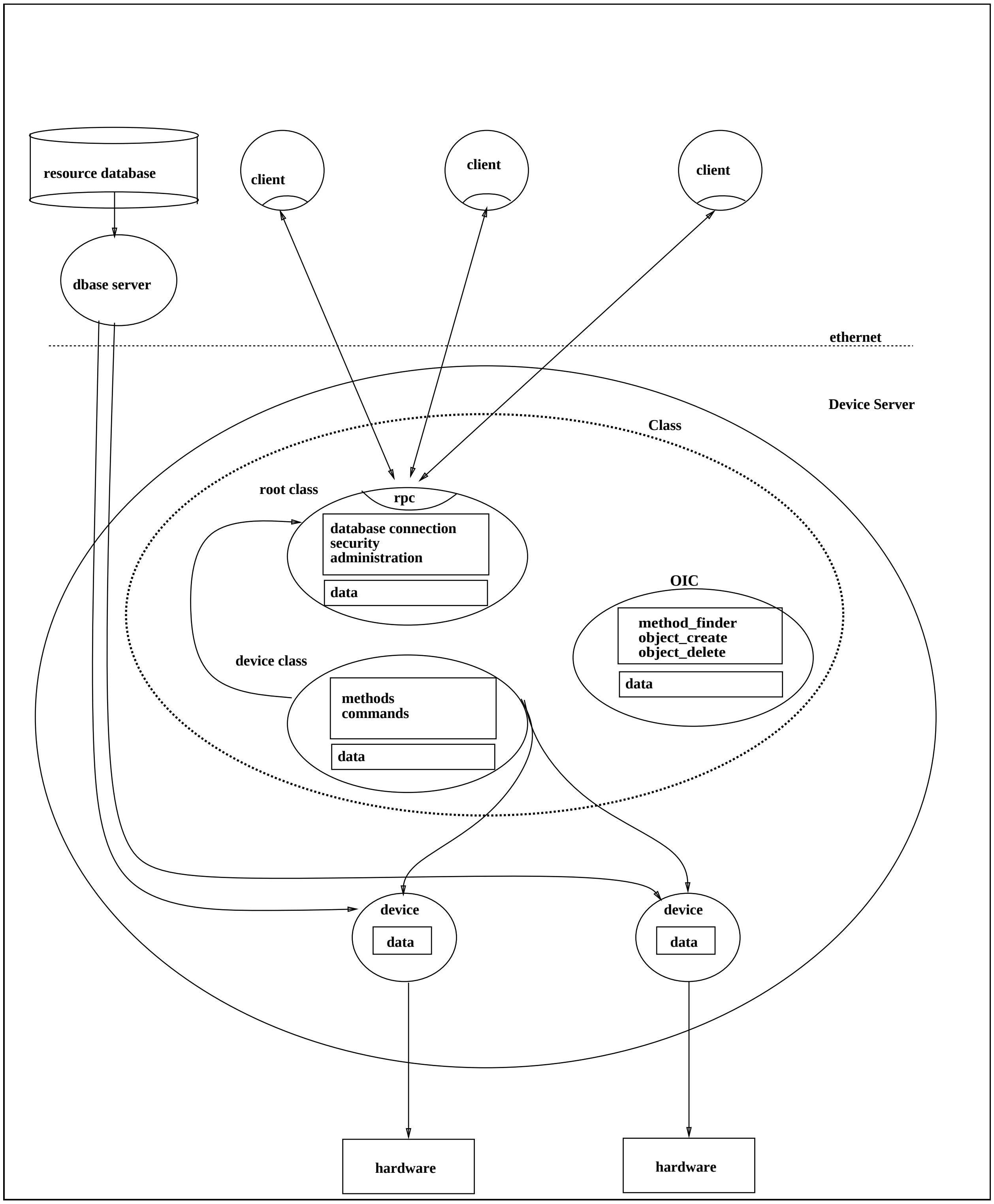

Figure 1: The Device Server Model

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.

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.

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.

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 2*.

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.

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.

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 :

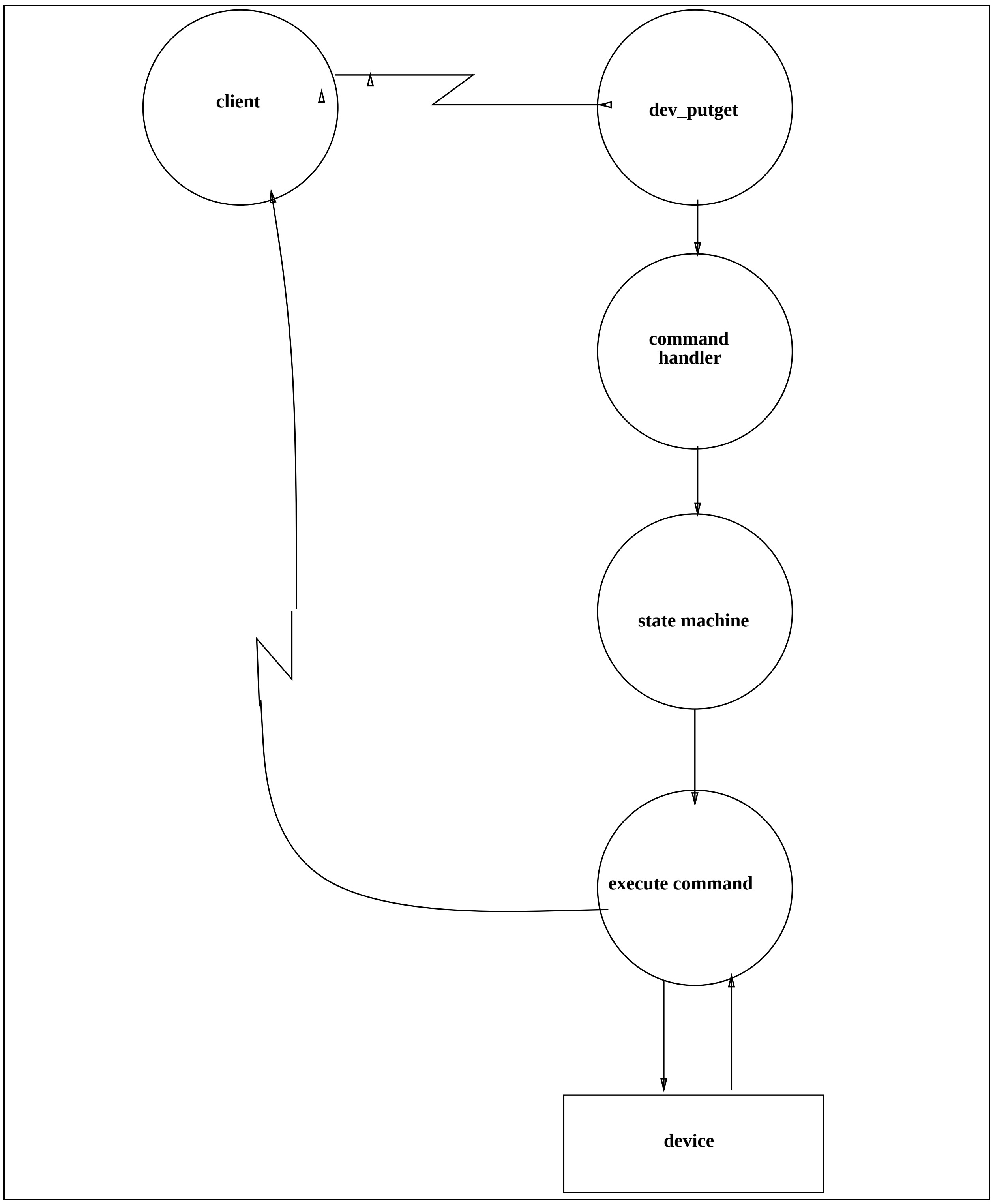

Figure 2: Flow of command execution

```
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.

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.

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.

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).

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*

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
```

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. *aGPowerSup-*

plyClass).

- 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
```

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);
```

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 3). 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

[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.

(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*.

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.

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

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

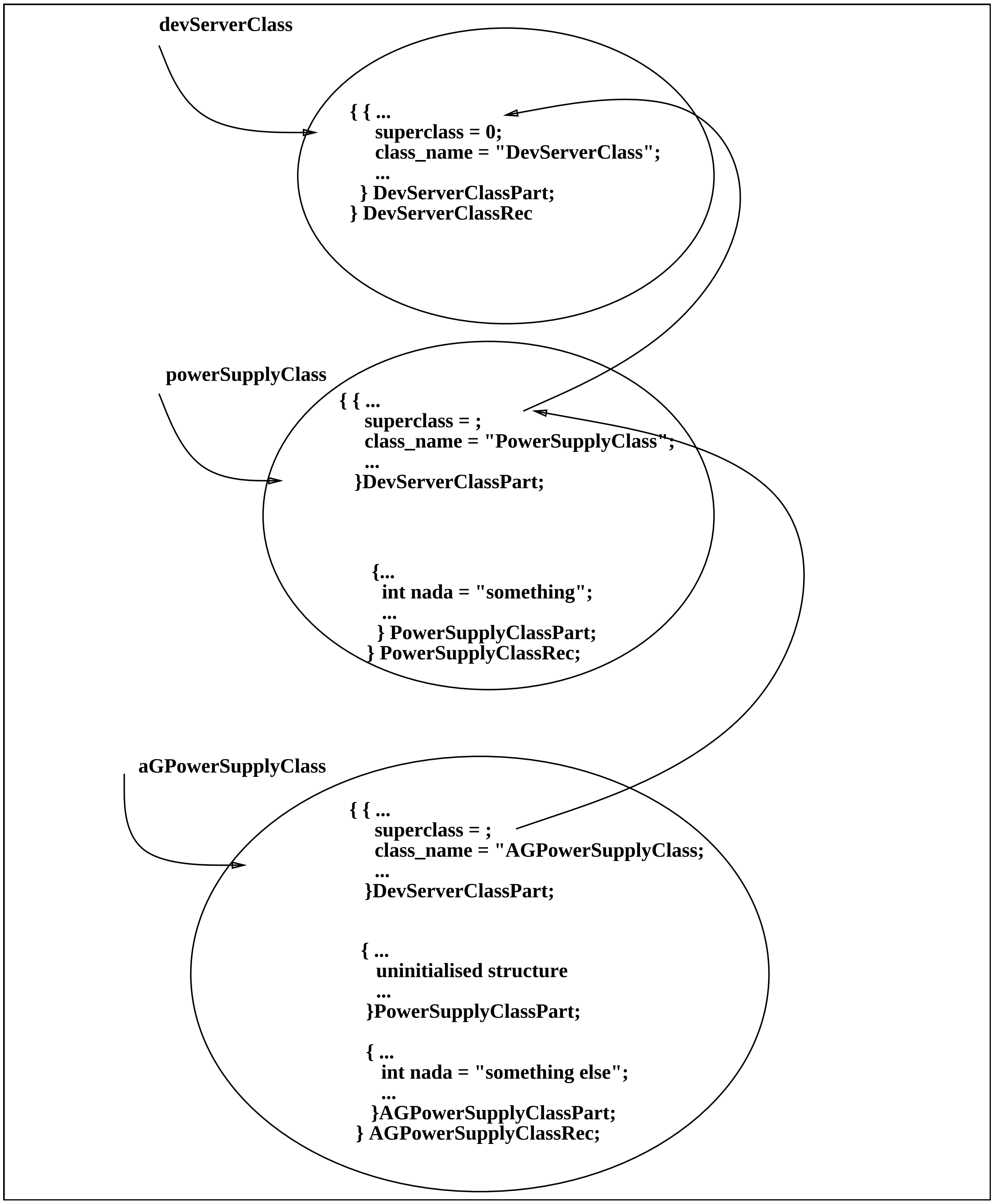

Figure 3: The device class structure for the AGPowerSupplyClass, a class with three levels of hierarchy. This diagram demonstrates the organisation of the partial structures of each class. It should be noted that each class structure has its own (initialised) copy of the DevServerPart. Note also that aGPowerSupplyClass has an (uninitialised) copy of PowerSupplyClassPart.

methods, its own superclass, 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.

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).

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.

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.

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.

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.

4.12.1 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)
```

4.12.2 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)
```

4.12.3 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)
```

4.12.4 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("AGPowerSupplyClass"));
   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);
}
```

4.12.5 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);
}
```

4.12.6 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);
}
```

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.

5.1 Synopsis

The process of writing a device server is an iterative one (see *figure 4*). 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.

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.

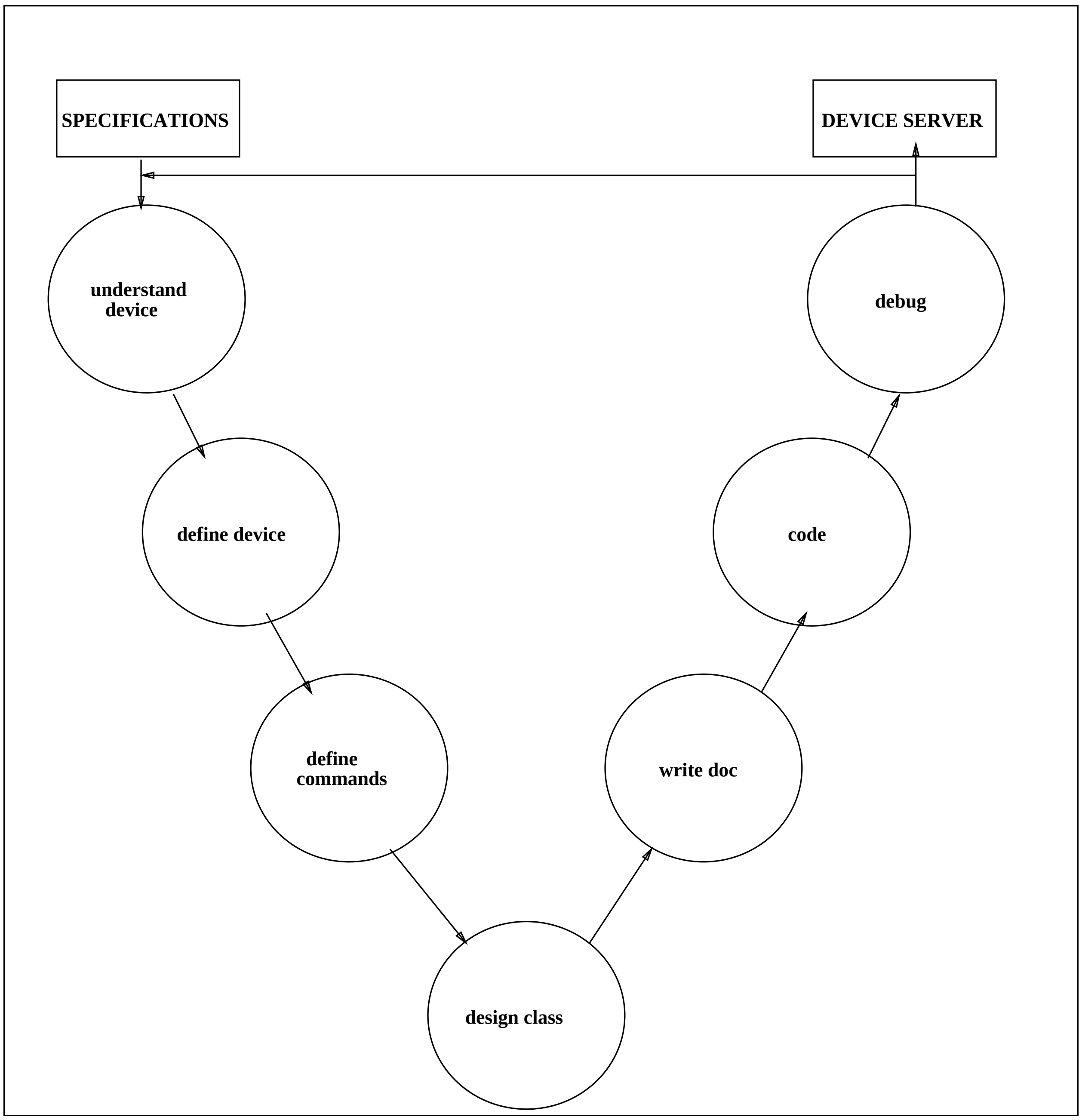

Figure 4: The iterative procedure for writing a device server

- **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).

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 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.

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.

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);
```

5.5.1 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.

5.5.2 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.

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.

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.

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.

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.

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).

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 program**s. 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,&error);
           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[statereadpoint.state]);
              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 ?
```

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.

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 5 -

```
/*======================================================================
 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. 5) representing the state machine of each new class should be included as part of the standard device server documentation.

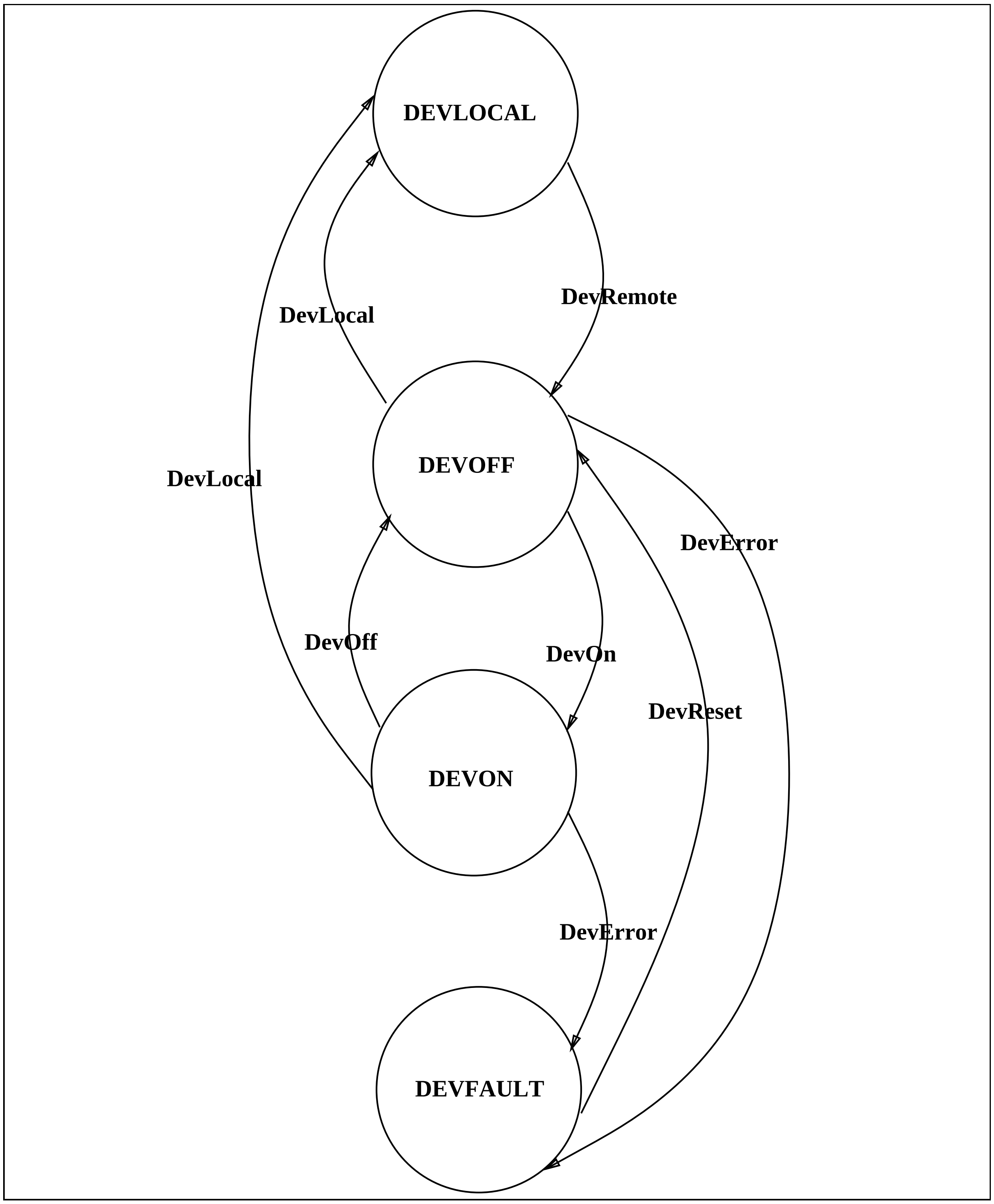

Figure 5: The State Diagram for the AGPowerSupplyClass

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).

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.

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);
}
```

5.15.1 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.

5.15.2 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).

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.

5.16.1 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**.

5.16.2 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 !).

5.16.3 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.

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**.

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**.

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.

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.

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.

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.

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.

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.

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.

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.

References

[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.