TACO

DSAPI

Version 6

A.Götz and J.Meyer

13 November, 1997

DSN/101 Revision: 2.1

Contents

1 Introduction **5**

2 What is DSAPI ? **5**

3 Getting Started **6**

3.1 "*Hello World*" (synchronous) example . . . 6

3.1.1 *Step 1* . . . 6

3.1.2 *Step 2* . . . 6

3.1.3 *Step 3* . . . 7

3.1.4 *Step 4* . . . 8

3.1.5 Example code - `helloworld.c` . . . 8

3.2 "*Hello World*" (asynchronous) example . . . 9

3.2.1 *Step 1* . . . 10

3.2.2 *Step 2* . . . 10

3.2.3 *Step 3* . . . 10

3.2.4 *Step 4* . . . 11

3.2.5 Example code - `helloworld_asyn.c` . . . 11

3.3 Common Pitfalls . . . 13

3.3.1 Nethost . . . 13

3.3.2 Shared Libraries . . . 14

3.3.3 Makefiles . . . 15

3.3.4 Memory Allocation . . . 17

3.4 Advanced Features . . . 19

3.4.1 Timeouts . . . 19

3.4.2 Protocol . . . 19

4 C Library **21**

4.1 Synchronous Client API . . . 21

4.1.1 dev_cmd_query() . . . 21

4.1.2 dev_free() . . . 22

4.1.3 dev_import() 22
4.1.4 dev_inform() 22
4.1.5 dev_put() 23
4.1.6 dev_put_asyn() 23
4.1.7 dev_putget() 24
4.1.8 dev_putget_raw() 24
4.1.9 dev_rpc_protocol() 25
4.1.10 dev_rpc_timeout() 25
4.1.11 dev_xdrfree() 25
4.2 ASynchronous Client API 26
4.2.1 dev_asynch_timeout 26
4.2.2 dev_pending 26
4.2.3 dev_putget_asyn() 26
4.2.4 dev_synch() 27
4.3 Server 27
4.3.1 dev_cmd() 27
4.3.2 ds__create() 28
4.3.3 ds__destroy() 28
4.3.4 dev_export() 28
4.3.5 ds__method_finder() 29
4.3.6 ds__method_search() 29
4.3.7 ds__svcrun() 29
4.4 General Purpose Functions 29
4.4.1 dev_printerror_no() 29
4.4.2 dev_error_str() 30
4.4.3 dev_printdebug() 30

5 XDR types **32**
5.1 Kernel Types 32
5.2 Simple C Types 32

5.3 Combinations of Simple Types 33
5.4 Variable Length Arrays 33
5.5 Exotic Types 35

6 Changes **36**

6.1 Version 6.0 36
6.2 Version 5.1 36
6.3 Version 4.1 36
6.4 Version 3.37 36
6.4.1 An Asynchronous dev_put() 36
6.4.2 Destroying Objects 36
6.4.3 Accessing Process Internal Devices 37
6.4.4 Dynamic Memory Allocation 37
6.4.5 The Device ID 38

7 Platforms supported **39**

8 Interfaces to other Languages **40**

1 Introduction

The DSAPI is the TACO[1] Device Server Application Programmer's Interface for C and C++ programs. It is used by clients and servers to import, execute commands on, free and explore TACO devices. It uses the ONC-RPC (SUN remote procedure call) as underlying communication protocol. This document describes the latest version of the DSAPI V6.0.

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

2 What is DSAPI ?

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

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

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

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

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

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

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

[1] for more information about TACO see http://www.esrf.fr/computing/cs/taco/taco.html

3 Getting Started

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

3.1 "*Hello World*" (synchronous) example

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

3.1.1 *Step 1*

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

The command we will use in this example is DevHello.

3.1.2 *Step 2*

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

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

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

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

- `#include <API.h>` - include file required by all clients (and servers). Necessary to prototype all DSAPI functions, and to define symbols and types. API.h will include other include as necessary.
- `devserver hs` - variable which will contain the device handle. Every device has to have a device handle. It is passed as input parameter to every DSAPI call. It contains all information necessary to communicate with the device on the network (network address, protocol, security etc.) as well as various bookkeeping information (device name). The device handle is initialised on the first successful call to `dev_import()` (cf. below).

- `dev_import()` - initialise the device handle. This call takes as input the device name and permission level requested. It checks the database to see if the device is defined and if so it asks the database for the device's network address. Then it tries to contact the device server. All this information is stored in the device handle and returned to the user. If the device is not defined in the database or the user does not have the necessary permission to use the device `dev_import()` will return an error and the device handle will be NULL. The import is stateless this means the routine will not fail if the device server is not running. The 2nd parameter is used for security (this is discussed in the C library reference).
- `dev_putget()` - execute a command on the device. This call is the workhorse of DSAPI. It is used to execute a command on a device synchronously i.e. the client sends her request to the device and then waits for the command to be executed and for the answer to be returned before continuing. For the asynchronous version see below. The client has to specify the input and output arguments and their types. This information is normally obtained from the DSUG but can be constructed dynamically (using `dev_cmd_query()`. All parameters are passed as pointers. If the output arguments contain any pointers in them the client can choose to allocate space for the result himself or let DSAPI allocate space. (by setting the pointer to NULL) In the latter case it is up to the client to free the space allocated by DSAPI. The question of when to allocate and when to free is a tricky one and is treated in more detail in the section *"Common Pitfalls"*.
- `dev_free()` - free the device handle. This call will try to inform device server that the client is not connected to this device anymore. If this is the client's last network connection to the device server it will free the socket connection to the device server. Finally it will free the device handle structure allocated by `dev_import()`.

3.1.3 *Step 3*

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

Unix and OS9

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

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

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

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

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

to be filled in ...

3.1.4 *Step 4*

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

3.1.5 Example code - `helloworld.c`

```
static char RcsId[] = "@(#)$Header: /segfs/dserver/doc/notes/DSN101/RCS/DSN101.tex,v 2.1 1997/11/13 14:16:40 goetz Exp $";
/*********************************************************************

 File       :   helloworld.c

 Project    :   Device Server

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

 Author(s)  :   Andy Goetz

 Original   :   November 1997

 $Revision: 2.1 $
 $Date: 1997/11/13 14:16:40 $

 $Author: goetz $

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

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


 *-*******************************************************************/

#include <Admin.h>
#include <API.h>

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

```
        devserver hw;
        long access = WRITE_ACCESS, error, status;
        char *ch_ptr,helloworld[256], dev_name[256];

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

        status = dev_import(dev_name,access,&hw,&error);
        printf("dev_import(%s) returned %d\n",dev_name,status);

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

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

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

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

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

3.2 "*Hello World*" (asynchronous) example

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

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

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

3.2.1 *Step 1*

Understanding the device - same as Step 1 above.

3.2.2 *Step 2*

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

The new calls are :

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

3.2.3 *Step 3*

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

Not support under Windows-NT (yet).

3.2.4 *Step 4*

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

Not supported under Windows-NT (yet).

3.2.5 **Example code -** `helloworld_asyn.c`

```
static char RcsId[] = "@(#)$Header: /segfs/dserver/doc/notes/DSN101/RCS/DSN101.tex,v 2.1 1997/11/13 14:16:40 goetz Exp $";
/*+*********************************************************************

 File       :   helloworld_asyn.c

 Project    :   Asynchronous Device Server's

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

 Author(s)  :   Andy Goetz

 Original   :   January 1997

 $Revision: 2.1 $
 $Date: 1997/11/13 14:16:40 $

 $Author: goetz $

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

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


 *-*********************************************************************/

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

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

 Description:   callback function to be called asynchronously after executing
```

```
                the DevHello commands

*********************************************************************-*/

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

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

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

        return;
}

/*+********************************************************************
 Function   :   main()

 Description:   main function to test asynchronous DSAPI.

*********************************************************************-*/


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

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


        switch (argc)        {
                case 1:
                        printf("enter device name [\"exp/hello/world\"]? ");
```

```
                        if(NULL==gets(dev_name) || '\0'==dev_name[0])
                                strcpy(dev_name,"exp/hello/world");
                        break;
                case 2:
                        strcpy(dev_name,argv[1]);
                        break;

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

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

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

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

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

        status = dev_putget_asyn(hw,DevHello,
                                 &helloworld,D_STRING_TYPE,
                                 &ch_ptr,D_STRING_TYPE,
                                 (DevCallbackFunction*)void_callback,
                                 (void*)user_data, &asynch_id,
                                 &error);
        printf("\nDevHello dev_putget_asynch(%d) returned %d\n",asynch_id, status);
        if (status < 0) dev_printerror_no(SEND,NULL,error);

/*
 * wait for answer from client (waits for a max of 25 s)
 */
        status = dev_synch(&timeout_25s, &error);


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

}
```

3.3 Common Pitfalls

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

3.3.1 Nethost

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

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

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

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

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

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

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

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

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

3.3.2 Shared Libraries

Another common error is not finding the DSAPI shared libraries.

If your application dies with the following message :

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

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

For Solaris and Linux use :

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

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

For HP-UX use :

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

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

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

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

3.3.3 Makefiles

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

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

- `__hp9000s700` - for HPPA 1.0 systems running HP-UX 9.x
- `__hpux10` - for HPPA 1.0 systems running HP-UX 10.2
- `_solaris` - for Solaris
- `linux` - for Linux
- `vw68k` - for Motorola 68k systems running VxWorks
- `vwx86` - for Intel x86 systems running VxWorks
- `_UCC` - for OS9 systems using the Ultra C and C++ compiler
- `unix` - for HP-UX, Solaris, Linux and VxWorks platforms

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

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

```
ifdef _UCC
LIBHOME = $(DSHOME)/lib/os9
endif # _UCC
ifdef vw68k
LIBHOME = $(DSHOME)/lib/vw68k
endif # vw68k
ifdef vwx86
LIBHOME = $(DSHOME)/lib/vwx86
endif # vwx86
#
# include files home directory
#
INCLDIRS = -I$(DSHOME)/include \
           -I$(DSHOME)/include/private
#
# compiler flags - platform dependant
#
ifdef __hpux10
CC = /bin/cc
CFLAGS  = -Aa -g -DEBUG -Dunix -D_HPUX_SOURCE -D__hpux10 -DBSD=199704 \
          -c $(INCLDIRS)
endif # __hpux10
ifdef _solaris
CC = /opt/SUNWspro/SC4.0/bin/cc
CFLAGS  = -Xa -g -Dsolaris -DEBUG -c $(INCLDIRS)
endif # _solaris
ifdef linux
CC = gcc
CFLAGS  = $(INCLDIRS) -Dlinux -Dunix -ansi -DEBUG -g -c
endif # linux
ifdef _UCC
CC = xcc
CFLAGS =  -mode=c89 -g -D EBUG -to osk -tp 020 -x il -e as=. $(INCLDIRS)
endif # _UCC
ifdef vw68k
CC = cc68k
CFLAGS =       -Dvxworks -Dunix -DCPU=MC68020 -ansi -m68030 \
               -msoft-float -DEBUG -e $(INCLDIRS) -g
endif # vw68k
ifdef vwx86
CC = cc386
CFLAGS  =       -v -c -Dvxworks -Dunix -DCPU=I80386 -ansi \
                -DEBUG $(INCLDIRS) -g
endif # vwx86
#
# library flags
#
ifdef __hpux10
LFLAGS = -L$(LIBHOME) -ldsapi -ldsxdr -ldbapi -lm
endif # __hpux10
ifdef _solaris
LFLAGS = -L$(LIBHOME) -ldsapi -ldsxdr -ldbapi -lnsl -lsocket -lm
endif # _solaris
ifdef linux
```

```
LFLAGS = -L$(LIBHOME) -ldsapi -ldsxdr -ldbapi -lm
endif # linux
ifdef _UCC
LFLAGS = -L$(LIBHOME) -l dsapi -l dsxdr -l dbapi -l rpclib -l netdb_small \
         -l socklib.l -l sys_clib.l -l unix.l
endif # _UCC
#
#------------------------main-target-to-make------------------------------
#
all : helloworld

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

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

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

3.3.4 Memory Allocation

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

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

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

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

To illustrate the above rules here are some examples :

- **simple C types**

```
devserver ps;
long status, error;
float readvalue;
.
.
.
status = dev_putget(ps, DevReadCurrent, NULL, D_VOID_TYPE,
                    &readvalue, D_FLOAT_TYPE, &error);
printf("current %6.3f\n",readvalue);
.
.
.
```

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

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

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

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

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

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

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

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

3.4 Advanced Features

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

3.4.1 Timeouts

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

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

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

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

3.4.2 Protocol

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

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

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

4 C Library

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

4.1 Synchronous Client API

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

4.1.1 dev_cmd_query()

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

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


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

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

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

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

4.1.2 dev_free()

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

Dev_free() closes the connection to a device associated with the passed client handle.

4.1.3 dev_import()

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

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

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

1. `READ_ACCESS` for read-only access
2. `WRITE_ACCESS` for read and write access (**default**)
3. `SI_WRITE_ACESS` for single user write access
4. `SU_ACCESS` for super-user access
5. `SI_SU_ACCESS` for single user super-user access
6. `ADMIN_ACCESS` for administrator access

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

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

4.1.4 dev_inform()

```
typedef struct {
                char device_name[80];
                char device_class[32];
                char device_type[32];
                char server_name[80];
```

```
                char server_host[32];
               } DevInfo;

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

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

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

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

4.1.5 dev_put()

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

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

4.1.6 dev_put_asyn()

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

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

4.1.7 dev_putget()

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

Dev_putget() executes a command synchronously on the device associated with the passed client handle. The device might be remote or local. Input and output data types must correspond to the types specified for this command in the device server's command list. Otherwise an error code will be returned. All arguments have to be passed as pointers.
Memory for outgoing arguments will be automatically allocated by XDR, if pointers are initialised to **NULL**. To free the memory allocated by XDR afterwards, the function **dev_xdrfree()** must be used.

4.1.8 dev_putget_raw()

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

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

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

4.1.9 dev_rpc_protocol()

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

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

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

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

4.1.10 dev_rpc_timeout()

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

Sets or reads the timeout for a RPC connection with UDP protocol. A request to set the timeout has to be asked with CLSET_TIMEOUT as request parameter and the timeout specified by the timeval structure dev_timeout. The timeout will be set without any retry. A request to read the timeout has to be asked with CLGET_TIMEOUT, and the current timeout will be returned in dev_timeout.

All devices implemented in the same server and imported by the client use the same RPC connection. Changing the timeout of a RPC connection with dev_rpc_timeout means changing the timeout value for all devices of the same server.

4.1.11 dev_xdrfree()

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

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

4.2 ASynchronous Client API

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

4.2.1 dev_asynch_timeout

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

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

4.2.2 dev_pending

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

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

4.2.3 dev_putget_asyn()

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

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

long dev_putget_asyn (ds,cmd,argin,argin_type,argout,argout_type,
                      callback, user_data, asynch_id, error)
        devserver       ds;            /* client handle */
        long            cmd;           /* command */
        DevArgument     argin;         /* pointer to input arguments */
```

```
        DevType         argin_type;     /* type of input arguments */
        DevArgument     argout;         /* pointer to output arguments */
        DevType         argout_type;    /* type of output arguments */
        DevCallbackFunction *callback;  /* pointer to callback function */
        void            *user_data;     /* pointer to user data to pass to callback */
        long            *asynch_id;     /* asynchronous id returned by call */
        long            *error;         /* error */
```

Dev_putget_asyn() executes a command asynchronously on the device associated with the passed client handle. The device must be remote and compiled with V6. Input and output data types must correspond to the types specified for this command in the device server's command list. Otherwise an error code will be returned. All arguments have to be passed as pointers.
Memory for outgoing arguments will be automatically allocated by XDR, if pointers are initialised to **NULL**. To free the memory allocated by XDR afterwards, the function **dev_xdrfree()** must be used.

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

4.2.4 dev_synch()

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

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

4.3 Server

4.3.1 dev_cmd()

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

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

With the extended functionality of **dev_putget** and **dev_put** the function should be used only to access objects which are not exported.
To access internal exported devices the unified interface must be used, to avoid access and security problems in the coming releases.

4.3.2 ds__create()

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

Ds__create() creates a new device server object of the class ds_class and will return a pointer on the object. Before creating the object (DevMethodCreate : obj_create(3x)) the class and all its superclasses are checked to see if they have been initialised. If not, then the DevMethodClassInitialise (class_init(3x)) is called for each uninitialised class.

4.3.3 ds__destroy()

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

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

The general destroy method will free the object correctly only, if no memory allocation was done for object fields outside the **DevServerPart structure** of the object. The device name, as a field of DevServerPart will be freed correctly bye the general device server class destroy method.
Also exported objects can be destroyed. They will be deleted from the list of exported devices and all client accesses will be stopped.

4.3.4 dev_export()

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

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

4.3.5 ds__method_finder()

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

Ds__method_finder() searches for a method in the class hierarchy of the object **ds** and returns a pointer to the method function. If the method was not found in the object's class, the search continues in all its superclasses up to the general device server class.

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

4.3.6 ds__method_search()

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

Ds__method_search() searches for a method in the class specified. It returns the pointer to the method function if the requested method was found in the class. If no such method was specified the status DS_NOTOK is returned.

4.3.7 ds__svcrun()

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

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

4.4 General Purpose Functions

4.4.1 dev_printerror_no()

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

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

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

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

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

- WRITE: Writes error message to buffer.
- SEND: Adds the last error message to the buffer, sends the buffer contents to an output device and clears the buffer.
- CLEAR: Clears the message buffer from all stored messages.

4.4.2 dev_error_str()

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

Dev_error_str() returns the error string for a given error number. If an appropriate error string can not be found in the data base, dev_error_str() returns a string, indicating the failure. The memory for the returned error strings is allocated by malloc() and can be freed by free().

4.4.3 dev_printdebug()

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

Dev_printdebug sends the debug information if the specified **debug_bits** are set. Possible debug_bits (debug flags) are:

```
        #define DBG_TRACE               0x1
        #define DBG_ERROR               0x2
        #define DBG_INTERRUPT           0x4
        #define DBG_TIME                0x8
        #define DBG_WAIT                0x10
        #define DBG_EXCEPT              0x20
```

```
#define DBG_SYNC               0x40
#define DBG_HARDWARE           0x80

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

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

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

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

5 XDR types

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

5.1 Kernel Types

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

5.2 Simple C Types

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

1. `D_VOID_TYPE`

   `typedef void DevVoid`
2. `D_CHAR_TYPE`

   `typedef char DevChar`
3. `D_BOOLEAN_TYPE`

   `typedef char DevBoolean`
4. `D_USHORT_TYPE`

   `typedef u_short DevUShort`
5. `D_SHORT_TYPE`

   `typedef short DevShort`
6. `D_USLONG_TYPE`

   `typedef u_long DevULong`
7. `D_LONG_TYPE`

   `typedef long DevLong`
8. `D_FLOAT_TYPE`

   `typedef float DevFloat`
9. `D_DOUBLE_TYPE`

   `typedef double DevDouble`
10. `D_STRING_TYPE`

    `typedef char* DevString`

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

5.3 Combinations of Simple Types

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

1. D_INT_FLOAT_TYPE

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

2. D_FLOAT_READPOINT

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

3. D_STATE_FLOAT_READPOINT

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

4. D_LONG_READPOINT

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

5. D_DOUBLE_READPOINT

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

5.4 Variable Length Arrays

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

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

where <Type> is the required type.

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

1. D_VAR_CHARARR

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

2. D_VAR_STRINGARR

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

3. D_VAR_USHORTARR

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

4. D_VAR_SHORTARR

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

5. D_VAR_ULONGARR

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

6. D_VAR_LONGARR

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

7. D_VAR_FLOATARR

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

8. D_VAR_DOUBLEARR

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

9. D_VAR_FRPARR

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

10. D_VAR_SFRPARR

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

11. D_VAR_LRPARR

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

5.5 Exotic Types

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

6 Changes

6.1 Version 6.0

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

6.2 Version 5.1

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

6.3 Version 4.1

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

6.4 Version 3.37

6.4.1 An Asynchronous dev_put()

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

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

6.4.2 Destroying Objects

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

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

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

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

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

6.4.3 Accessing Process Internal Devices

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

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

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

Attention:
This unified interface for device access works on all **exported** devices. Objects which are not exported, can be accessed only be dev_cmd().
To access process internal devices the unified interface must be used to avoid access and security problems in the coming releases.

6.4.4 Dynamic Memory Allocation

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

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

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

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

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

6.4.5 The Device ID

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

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

The **export counter** field becomes interesting only if you destroy an exported object and reexport another or the same object again. In the case of a destroyed object, the export counter is increased and all client connections on the old value are no longer valid. A newly exported device might take the place in the list of exported devices afterwards.
The **Local access flag** is set if the dev_import() detects a local device.
The split up of the device ID limits a device server to the following values:

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

7 Platforms supported

DSAPI is supported on the following platforms and compilers :

1. **HP-UX 9.x** and **HP-UX 10.2** with the HP native cc compiler for C and GNU g++ for C++.
2. **Solaris** with the SUN native cc compiler for C and GNU g++ for C++.
3. **Linux 2.x.y** on Intel platforms with the GNU gcc compiler for C and GNU g++ for C++.
4. **OS9** on Motorola platforms with the Ultra C compiler for C and the Ultra C++ compiler for C++.
5. **Windows-NT 4.0** on Intel platforms with the Visual C++ compiler for C.
6. **VxWorks 5.2** on Motorola and Intel platforms using the GNU gcc compiler for C.

8 Interfaces to other Languages

DSAPI has been interfaced to a number of other languages. This section lists what languages are supported, for further information refer to the respective manuals.

- **Java** - a Java interface has been written by W-D.Klotz. It uses the binary interface to C to call DSAPI. It can be used to write programs in Java (not applets).
- **Tcl/Tk** - a Tcl interface for DSAPI has been implemented by G.Pepellin. It can be used to write scripts and graphical user interfaces using the powerful Tk extension to Tcl.
- **ASCII** - an ascii version of the DSAPI has been implemented by F.Poncet. It is available as a C library to make calls to DSAPI in ascii.
- **VeeTest** - a binary interface to VeeTest (HP's LabView) has been implemented by F.Poncet, it uses the ascii interface to the DSAPI.