The UBC OSI Distributed Application Programming Environment [1]

User Manual

by TR 90-37

Gerald W. Neufeld
Murray W. Goldberg
Barry J. Brachman

Technical Report 90-37
January, 1991

The OSI Laboratory
Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1W5

Email: goldberg@cs.ubc.ca

[1]This work was made possible by grants from the Canadian National Science and Engineering Council, the UBC Centre for Integrated Systems Research and UBC Research Services and Industry Liaison.

Contents

Acknowledgement

The authors would like to acknowledge the work of Barry J. Brachman and Yeuli Yang for their contribution to the design and development of portions of the programming environment. We would also like to recognize the constructive suggestions and comments of EAN Group member Eric Lau, and OSIWARE employee Duncan Stickings.

Introduction

This manual describes DAPE (the Distributed Application Programming Environment) and gives instructions for its use. DAPE is a software package which provides the remote operations service defined by the CCITT standard X.219 (ISO standard 9072-1) using the protocol defined by the CCITT standard X.229 (ISO standard 9072-2). It also supports communication through use of a Presentation layer interface. These protocol layers, as well as the OSI transport, session and association control layers are also provided. DAPE provides access to CASN1, an ASN.1 compiler. This compiler enables DAPE applications to encode application level data into an external representation suitable for transmission over a heterogenous computer network. All DAPE applications execute in the Threads sub-kernel environment. Threads provides a convenient coding environment for a group of cooperating processes. Finally, DAPE provides an object storage facility for persistent storage of variable size C data objects.

This manual is divided into parts. The first part describes the Threads environment. The features of Threads are outlined and the Threads application interface is detailed. The second part discusses the communication facilities of DAPE. The main sections of this part discuss application association establishment, release and information transfer using remote operations or presentation data transfer primitives. This part also describes the functionality and use of CASN1, the ASN.1 compiler. The final part of this manual presents the persistent object storage facility. Examples are given throughout the manual.

Part I

Operating System Support

Chapter 1

The Threads Sub-Kernel

1.1 Introduction

Threads is a sub-kernel running inside a UNIX process. The purpose behind writing Threads was to provide a pleasant and convenient coding environment for communication protocols. The result though is such an environment for any set of cooperative communicating processes.

The threads environment has the following properties. First, light-weight processes may be created and destroyed at will. All threads processes run in a shared memory space. Simple interprocess communication primitives have been supplied. A sleep facility has been provided. Memory management routines have been provided to allow quick memory allocation. A threads process has access to all existing libraries and UNIX system calls, though the more popular blocking primitives have been replaced so only the calling threads process is blocked, not the entire UNIX process.

A program written to run in the threads environment has few differences from one written to run directly under UNIX. The main difference, of course, is that the capabilities mentioned above are available for use. All such programs must include the header file "os.h". Also, instead of a program having to have a "main()" routine, a program designed to run under threads must instead have a "mainp()" routine. This gives access to argv and argc just as "main()" does. It should be noted that very little other than process creation

should be done in the "mainp()" routine. "mainp()" is not actually a threads process, and therefore the proper task for it is the creation of other threads processes. Once the "mainp()" routine returns, the created routines will be allowed to run, and "mainp()" will never be heard from again.

The following is a simple example of code written to use threads:

```
#include "standards.h"
#include "os.h"

/*************************************************************************/
/* this process awaits a message from other processes, prints the      */
/* message to stdio, and replies to the sender. This process will      */
/* terminate when the received message begins with  the character "!". */

PROCESS writer()
{
    PID  who;
    int  msglen;
    char *buf;
    char printbuf[100];
    char firstchar;

    do
        {
        /* await a message from some other process                     */
        buf = Receive( &who, &msglen );

        /* remember the first character so we can test it later        */
        firstchar = buf[0];

        /* null terminate the message                                  */
        buf[msglen - 1] = (char)0;

        /* send message to stdio indicating message arrival            */
        sprintf(printbuf, "Received message %s, length %d\n", buf, msglen);
        WriteN( 1, printbuf, strlen(printbuf) );
```

```
        /* return a reply to unblock sender                              */
        strcpy( buf, "thank you" );
        if( ( Reply(who, buf) ) < 0)
            lilerr("writer on rpl");
        }
    while(firstchar != '!');
}


/*************************************************************************/
/* this is a termination subroutine called when the reader terminates  */
printThis( string )
char *string;
{
    WriteN( 1, string, strlen( string ) );
}

/*************************************************************************/
/* this process reads some text from stdin and sends the text to the   */
/* writer process for printing to stdio. This process will terminate   */
/* when the first character of the read text is a "!". Note that this  */
/* process is passed a single argument on creation.                    */

PROCESS reader( arg )
char *arg;
{
    PID  writer;
    char buf[100];
    char *rplbuf;
    int  numread;
    char printbuf[100];
    char firstch;

    /* indicate that printThis is to be called when this proc finishes */
    OnTermination( printThis, "Goodbye Cruel World" );

    /* print out the create argument, just for fun                      */
    WriteN( 1, arg, strlen(arg) );
```

```
    /* ask Threads for the PID of the "writer" process                */
    writer = NameToPid("writer");

    do
        {
        /* read text from stdin and record the first character        */
        numread = Read(0, buf, 100);
        firstch = buf[0];

        /* send the text to the writer for writing to stdio           */
        if ( (rplbuf = Send(writer, buf, numread)) < 0)
            lilerr("reader on send");

        /* write (to stdio) the reply received from the writer        */
        sprintf( printbuf, "got reply %s\n", rplbuf );
        WriteN( 1, printbuf, strlen(printbuf) );
        }
    while(firstch != '!')
}

mainp(argc, argv)
int argc;
char **argv;
{
    /* create two processes, each with a 3k stack size, at normal priority */
    Create(reader, 3000, "reader", "readers arg", NORM);
    Create(writer, 3000, "writer", 0, NORM);
}
```

1.2 Process Management, Scheduling and Context Switching

Threads process scheduling is performed on a round-robin, multi-priority basis. All ready processes of a common priority level are scheduled on a round-robin basis. A process will

only be allowed to run if there are no higher priority ready processes.

Threads scheduling is neither time-slicing nor preemptive. Context switches are performed only on the basis of threads system calls made by a threads process. The system calls which potentially cause a context switch are the following: Pexit(), Send(), Receive(), Reply(), Read(), Write(), ReadN(), Recvfrom(), WriteN(), Accept(), Connect() and Sleep(). Also, a process may explicitly release control of the processor by calling Sched(). The Sched() threads call requires no parameters. This property of knowing when a context switch is likely to occur is actually very useful when programming cooperating processes. A critical section can be written with little worry about other process interference. If some operation on shared memory is performed, the writer only needs to be cautious about which threads system calls are made during the critical section.

Lack of time slicing is only a problem in the case of non-cooperating processes (for example - a multi-user system). In this case a threads process may easily neglect or refuse to relinquish control of the cpu for as long as it wishes. This would be a serious problem for the other users (processes) in the system.

Context switches are implemented through the switching of process stacks. Each process has its own stack. When a process is about to give up control of the system, two things happen. First, most processor registers are saved onto the process' stack. Next, the actual stack pointer is saved into a variable in the process' process control block. To load the next ready process, the processor registers are loaded from the ready process' process control block, the ready process' saved stack pointer is decremented to reflect the popping of the registers, and the decremented saved stack pointer is loaded into the hardware stack pointer. Many of these steps require the use of assembly language. The effect of this is that when the Threads context switching subroutine returns, the return will occur to a location taken from the newly installed stack, and therefore a context switch occurs. This necessitates the creation of a "fake" stack at process creation time. This fake stack will cause a completed process (running off the end or returning) to return to a system subroutine which does all necessary clean-up.

Threads makes the assumption that there is always at least one runnable process in the system. This assumption is satisfied by the doIO process which runs at the lowest priority and is guaranteed never to block.

The headers for the process management subroutines are as follows:

```
PID MyPid()

PID NameToPid(name)
char *name;

int PExists(pid)
PID pid;

PID Kill(pid)
PID pid;

Pexit()

PID Create(addr, stksize, name, arg, prio)
int (*addr)();
int stksize;
char *name;
int arg;
PPRIO prio;

OnTermination( subr, arg )
int (*subr)();
int arg;
```

MyPid() requires no parameters and returns the process identifier of the calling process.

NameToPid() requires a single parameter which is a pointer to a null terminated character string. This string represents a process name (see Create()). This routine searches the existing processes in the system for one with name *name*. The process identifier (PID) of the first process found with such a name is returned.

PExists() requires the single argument *pid*. This routine checks for a process with a process identifier of *pid*. If such a process exists in the system, the integer 1 (one) is returned. Otherwise the integer 0 (zero) is returned.

Kill() requires the single argument *pid*. Kill() searches for the process identified by *pid*, and if found, removes all traces of this process from the system returning the PID of the killed process. Otherwise, a zero is returned.

Pexit() causes the calling process to exit from the system. This routine is always successful and when called is the last instruction executed by the calling process. Note that it is not necessary for a process to make a call to Pexit(). A process may also terminate its own existence by simply returning from, or falling off the end of its main subroutine.

Create() causes the creation of a new Threads process. Create requires five parameters. The first parameter, *addr*, is a pointer to the routine which acts as the main subroutine for the newly created process. In the C language this is accomplished by simply using the name of the desired subroutine (without parens) for this parameter. The *stksize* parameter is the size, in bytes, of the new process' stack. The stack size requirements vary with the depth of subroutine calls made by the new process. They also vary according to the number of local variables and parameters of routines called by the process. A minimum requirement is generally about 2K bytes, though some processes require as much as 10K bytes or more. The third parameter, *name*, points to a text string which acts as a user supplied identifier for this process. This string must be null terminated and currently has a length restriction of 17 bytes including the null-terminator. Any number of processes may be identified by the same name. The name is copied by the Create() routine and therefore the memory containing this routine may be released by the caller on return from Create(). The fourth parameter, *arg*, acts as an argument (parameter) to the newly created process. This argument is passed transparently to the process and may be of any (4 byte or smaller) type, although it should be cast to an integer on call. The main routine for the new process receives this argument as a parameter, or may instead not declare any parameters if no creation-time arguments are required. The final argument to Create()

is *prio*. This arguments indicates the priority of the newly created process. The possible process priorities are HIGH, NORM and LOW. Except under unusual circumstances, user processes should be created at normal (NORM) priority.

OnTermination() requires two arguments, *subr* and *arg*. This routine registers a subroutine to be executed for the calling process when that process terminates (returns, exits or is killed). The *subr* parameter identifies the entry point of the subroutine to be called on process termination. As in the case of Create(), the entry point is identified by using the subroutine name as the argument. The *arg* parameter is a single argument to be provided to the called subroutine. The registration of the subroutine may be cancelled by calling OnTermination() again with a NULL for the *subr* parameter. It should be noted that this subroutine will be called by the process which is terminating only in the cases that this process falls off the end or calls PExit(). If the process is terminated using the Kill() routine, then the termination subroutine is performed by the process making the call to Kill().

1.3 Memory Management

Threads memory management provides fast memory allocation and deallocation. Threads provides two forms of memory management: a general memory allocation scheme, and a per-process allocation scheme.

The general scheme works as follows. A request for memory allocation is rounded up to the next size allowed by Threads. These sizes, and the number of such sizes are configurable within Threads. Threads keeps a list of available memory blocks of each size. Initially, each list of memory blocks is empty. When a process requests memory, Threads checks the list to see if there are any appropriately sized blocks on the list. If there are, the first block is dequeued and returned. Otherwise, a request for memory is made to UNIX. When memory is returned from a Threads process, it is not returned to UNIX, but is instead queued onto the appropriate list. This way, once a sufficient free pool of memory has been

established, subsequent allocations and deallocations are very fast. If a request is made for a block of memory larger than the largest configured size, the request is passed directly to UNIX. When this block is freed, the free request is also passed directly to UNIX. If more memory is required from UNIX and UNIX cannot satisfy the request, all free memory blocks are returned to UNIX, and the process of building the free block pool begins again. This can help recover from serious memory fragmentation. Memory allocated in this way is not associated with any Thread process. If the process that allocated the memory dies, exits or gets killed, the memory still exists.

The calls to allocate and release this type of memory are as follows:

```
BYTE *Malloc( size )
int size;

Free( mem )
BYTE *mem;
```

The parameter to Malloc(), *size*, indicates the number of bytes required. Malloc() will return the address of the allocated memory. The parameter to Free() is the address of the memory to be freed. This address must have been previously returned by Malloc().

Threads also has a per-process memory facility. Each Threads process has associated with it a stack of memory frames. If a process exits or is killed, all of the memory allocated from its stack frames is returned to the system. Per-process memory may be allocated only from a process' top memory frame. There are also operations to create, destroy and swap frames. Frames may also be passed from one process to another. An example of one way that this is useful follows. Say process A wishes to create a linked list of structures and pass it to another process - process B. Process A can create a new memory frame, and allocate all of the linked list nodes from this top frame. It then can pass this frame (and all the nodes contained within) to process B. Process B consumes the list and can free all of the storage associated with the list using a single call which frees the memory frame. This feature is especially useful for complex structures which would be time consuming to

free. Its implementation is very efficient in that freeing a memory frame (no matter how many blocks have been allocated on it) requires little more time than freeing one memory block.

The headers of the subroutines which operate on per-process memory are as follows:

```
NewFrame()

FreeFrame()

SwapFrame()

void *PopFrame()

PushFrame( frame )
void *frame;

int TransferTempMem( topid )
PID topid;

BYTE *TempMalloc( size )
int size;

FreeTempMem()
```

NewFrame() creates a new memory frame and pushes it onto the calling thread's memory frame stack.

FreeFrame() frees all memory associated with the calling process' top memory frame, pops the frame and discards it.

SwapFrame() swaps the top two memory frames of the calling process. If zero or one frames exist for this process then SwapFrame() has no effect.

PopFrame() pops and returns a pointer to the top memory frame of the calling process. This routine should be used with caution, generally in conjunction with PushFrame(). The

reason for caution is that a memory frame which is not currently on any process' memory stack is essentially an orphan. This memory will not be returned to the system should its creator or owner exit.

PushFrame() takes a pointer to a frame and pushes it on the calling thread's memory stack. The PopFrame() / PushFrame() pair can be used to transfer per-process memory from one process to another, or to perform stack rearrangement functions. Note that a stack frame which does not currently reside on any process' stack is in danger of becomming uncollectable garbage. Normally, when a memory frame exists on some process' memory stack, the memory is returned to the system when the process exits or dies. Note also that a memory frame cannot exist on more than one memory stack at a time. If an application wishes to move the top memory frame from one process to another, this may be done using PopFrame(), PushFrame() and inter-process communication, but it is preferable to use TransferTempMem() instead.

TransferTempMem() takes as an argument the PID *topid.* This operation transfers the top memory frame of the calling process to the top of the memory stack of process *topid.* This routine avoids the time interval between a PopFrame() and a PushFrame() when a memory frame does not belong to any process.

TempMalloc() takes an integer parameter *size.* This routine allocates memory from the top memory frame of the calling process. This memory cannot be released using Free(). Per-process memory is instead returned to the system using FreeFrame() (discussed previously) or FreeTempMem(). An important feature of TempMalloc() is that if no memory frame currently exists on the calling process' memory stack, a new one is created and pushed. In this case, the allocated memory is taken from the new frame.

FreeTempMem() returns the calling process' per-process memory to the system. The memory from each of the calling process' memory frames (not just the top one as in the case of FreeFrame()) is returned. This routine also pops all memory frames from the calling process leaving it with none.

Finally, there is one routine which is common to both general memory allocation and per-process memory allocation. This is the Realloc() routine. The header for this routine is as follows:

```
BYTE *Realloc( oldptr, newlength )
BYTE *oldptr;
int  newlength;
```

Realloc requires two parameters. *Oldptr* is pointer to memory (previously allocated using Malloc() or TempMalloc()), and *newlength* is an integer. This routine allocates a new block of memory of length *newlength*, copies the contents of the original memory to the new memory (to the extent of the old or new memory sizes - whichever is smaller), and returns a pointer to the new memory. Realloc() also frees the original memory block. If the original block was per-process memory, the new block will be allocated from the same memory frame as the original.

1.4 I/O

Routines which support I/O in threads include NonBlkRead(), NonBlkWrite(), Read(), Write(), ReadN(), WriteN(), Accept(), Connect(), Recvfrom(), Open(), Close() and Socket(). Each of these routines is meant to replace corresponding UNIX routines (see individual routine descriptions), though some are provided for different reasons than others.

The routines Open(), Close() and Socket() are provided for the reason that threads must keep account of the number of open file descriptors or sockets. Each UNIX process is allowed to have open at any one time no more than *getdtablesize()* descriptors. This causes a problem for the UNIX *accept()* command which allocates a new descriptor. UNIX *accept()* cannot be called if its completion would require more than the available number of descriptors. In order to avoid this situation, each descriptor allocated from and returned

to UNIX must be counted, and a check of this number must be made before the threads Accept() makes its call to UNIX accept().

Threads Accept() has a more important job than simply checking the number of available descriptors and calling UNIX accept(). This routine, like the remainder of the above routines (NonBlkRead(), NonBlkWrite(), Read(), Write(), ReadN(), WriteN(), Recvfrom() and Connect()) are re-implemented in threads for a more important reason. Each of these routines has the trait that it has the potential to block the UNIX process once called. This could be a significant problem for the rest of the threads processes running in the system. It would not be appropriate for all Threads processes to have to wait for a single process' I/O.

To avoid this problem, threads replaces common blocking UNIX system calls with similar threads calls. The replacement I/O calls have the effect of blocking the calling threads process without blocking the other processes sharing the same UNIX process. This is accomplished via the UNIX *select()* call. All Read(), Receive(), Accept(), Write() and Connect() calls (with a small exception) place the calling process on the I/O blocked queue. There it stays until its I/O is satisfied. How does the I/O become satisfied? The way this is accomplished is by first marking all I/O descriptors as non-blocking using the *fcntl()* UNIX call. Then, a threads system process called "doIO" periodically checks the I/O blocked queue. If there is anything on this queue, doIO builds read and write masks for use with select(). If select() indicates that any of the descriptors are ready for reading or writing, then the appropriate operation is performed. If the performance of the operation completes the requested threads operation (eg. if the correct number of bytes have been read), then the process making the original call is taken off the I/O blocked queue and readied by doIO.

The interval at which doIO checks the I/O blocked queue depends on the priority at which the doIO process is created. At present, doIO is created at low priority, and therefore I/O is only performed once all higher priority processes have blocked or completed. This seems to be a suitable arrangement as I/O is comparatively slow.

The headers for these I/O routines are as follows:

```
int Open( file, flags [,mode] )
char *file;
int  flags;
int  mode;

int Close( fd )
int  fd;

int Socket( domain, type, protocol )
int  domain, type, protocol;

int Connect (fd, name, namelen)
int  fd;
void *name;    /* some address type */
int  namelen;

int Accept (fd)
int  fd;

int Read (fd, buf, numbytes)
int  fd;
char *buf;
int  numbytes;

int Recvfrom (fd, buf, numbytes, flags, from, fromlen)
int  fd;
char *buf;
int  numbytes;
int  flags;
char *from;
int  *fromlen;

int ReadN (fd, buf, numbytes)
int  fd;
char *buf;
int  numbytes;
```

```
int NonBlkRead (fd, buf, numbytes)
int  fd;
char *buf;
int  numbytes;

int Write (fd, buf, numbytes)
int  fd;
char *buf;
int  numbytes;

int WriteN (fd, buf, numbytes)
int  fd;
char *buf;
int  numbytes;

int NonBlkWrite(fd, buf, numbytes)
int  fd;
char *buf;
int  numbytes;
```

The Open(), Close(), Socket(), Connect() and Accept() routines all provide the same interface and service as their corresponding UNIX routines. As indicated, Open(), Close() and Socket() are only provided to keep track of the number of file descriptors currently in use. The Connect() and Accept() routines are both provided so that a call to one of these blocks only the calling thread rather than the entire UNIX process.

The interface to Read(), ReadN() and NonBlkRead() are the same as for the UNIX read() routine. Read() waits until some data is available for reading, reads the available data, and returns the number of bytes read. Unless some error has occurred, the number of bytes read will be between one and the number requested. ReadN() is similar except that it waits until it reads exactly the number of bytes requested. NonBlkRead() will attempt a non-blocking read and return any data that is immediately available for reading. The number of bytes read will vary between zero and the number of bytes requested. These

routines are provided in order that calls made do not block the entire UNIX process.

The interface and service provided by Recvfrom() is the same as the corresponding UNIX routine. This routine is provided so that a call to it does not block the entire UNIX process.

The interface to Write(), WriteN() and NonBlkWrite() are the same as for the UNIX write() routine. Write() waits until it is possible to write some data, writes the data, and returns the number of bytes written. Unless some error has occurred, the number of bytes written will be between one and the number requested. WriteN() is similar except that it waits until it can write exactly the number of bytes requested. NonBlkWrite() will attempt a non-blocking write and return the number of bytes which could be written immediately. The number of bytes written will vary between zero and the number requested. These routines are provided in order that calls made do not block the entire UNIX process.

1.5 Sleep

Processes in the threads sub-kernel have the ability to put themselves to sleep with a timer resolution of CLKRES seconds. In the present implementation CLKRES is set to one tenth of a second.

When the threads Sleep() routine is called, a check is first made to be sure that the caller wants to sleep for more than 0 seconds. If this is not the case, Sleep() returns immediately. Otherwise, a calculation is made of the correct wakeup time by adding the desired sleep duration in seconds to the current time as supplied by the UNIX library function *time(0)*. This value is loaded into the process control block of the calling process, and that process is queued into a blocked queue.

The mechanism by which a process is taken out of the sleep queue and readied will depend on the number of active processes in the system, their cpu intensity, and their state. First of all, it should be mentioned that because threads is not a time-slicing system, there is no guarantee that a sleeping process will be waken up within a deterministic time

of the requested wake time. Obviously, some other cpu intensive process could decide not to relinquish control of the cpu for an extended period of time, and therefore the sleeping process would remain asleep until the running process gave up the cpu. This is not really a fault with the sleep logic, but more a by-product of the fact that there is no time-slicing. For the great majority of applications with cooperating processes the lack of time-slicing is not a weakness, but is often rather a benefit. If there are many ready processes in the system, then it is likely that the sleeping process will be waken up as a result of a context switch. Every time a context switch occurs, a check is made of the sleep queue, and if there is a process ready to be waken, it will be placed on the ready queue at that time. If there are few or no ready processes in the system, then the bulk of the time will be spent by threads in the doIO process making the UNIX *select* call. It has been arranged that *select* will time-out every CLKRES seconds, and at this time the sleep queue is checked and appropriate processes readied.

The process of checking the sleep queue for processes to awaken is done at every context switch and therefore must be very fast. This is accomplished by ordering the sleeping processes in order of wake time. When a check is made, first the head of the sleep queue is checked to see if there are any sleeping processes. If there are, it is only necessary to check the wake-time of the head process in the queue, and this is done by means of a simple comparison with the current time. If the process at the head of the queue is ready to be awaken, then the rest in line are checked and readied until one is found whose time has not yet come.

The header of this routine is as follows:

```
Sleep(secs)
long secs;
```

As indicated, this routine puts the calling thread to sleep for the number of seconds indicated in the argument *secs*.

1.6 Signal Handling

Threads provides a way for a threads process to block awaiting the arrival of a UNIX signal. The implementation of this is fairly simple. A call to the SigWait() routine records the signal to be waited for, informs UNIX of a routine to be called on the arrival of this signal (using the UNIX signal() routine), and then blocks the calling thread.

The arrival of this signal causes a global variable to be set. The doIO threads process will check this variable periodically for the arrival of some signal. If one has arrived then each process blocked awaiting that signal is readied.

This method of handling signals is crude in the sense that if signals arrive at very short intervals it is possible that one of them will be missed.

The header of the SigWait() routine is as follows:

```
SigWait( sig )
int sig;
```

As indicated, this call blocks the calling process until the arrival of the signal indicated by *sig*. The valid values for *sig* are those given in the include file <signal.h>.

1.7 Interprocess Communication

Threads processes communicate via Send() / Receive() / Reply() primitives. Because all threads processes share one memory space, no copying of data is done. Instead, a pointer and a length (or actually any two, four byte values) are sent in Send() and Reply().

Send works by first checking to see if the destination process is currently waiting for a message (via Receive()). If so, the pointer and length are transferred, and the receiver is returned to the appropriate ready queue. The sender is placed on the "waiting for reply" queue. If instead the destination process is not waiting for a message, then a check is made

to verify that the destination process exists. If it does, the sender is blocked (on the send blocked queue) pending a Receive() operation by the destination.

Receive is very similar to send. First, Threads checks to see if some process is blocked waiting to send to the receiving process. If so, the pointer and length are transferred, and the receiver is readied. The sender is placed on the "waiting for reply" queue. Otherwise, the receiving process is blocked pending some other process sending to it.

Reply is very simple. Unless a problem has occurred, Reply will find that its replied-to process is waiting on the "wait for reply" queue. A verification of this is performed, and the reply pointer is transferred. Also, at this point the original sender is returned to the appropriate ready queue. The Reply() operation may be used by any process. It does not have to be the one which originally received the message.

There is also a routine which allows a potential receiver of messages to test (in a non-blocking fashion) whether there are messages waiting for it. This is the MsgWaits() subroutine.

The subroutine headers to Send(), Receive(), Reply() and MsgWaits() are as follows:

```
char *Send ( to, msg, len )
PID  to;
char *msg;
int  len;

char *Receive( pid, len )
PID  *pid;
int  *len;

int Reply(pid, msg)
PID  pid;
char *msg;

int MsgWaits()
```

The Send() routine sends a message pointed to by the parameter *msg*, of length *len*, to process *to*. If successful, Send() returns a pointer to the message returned by the Reply() operation. If the proposed destination does not exist, Send() will return the value NOSUCHPROC.

Receive() blocks the calling process until some message is sent to it using Send(). The parameters *pid* and *len* should point to memory locations large enough to hold a PID and integer respectively. The *pid* parameter must point to a valid location. The *len* parameter may have the value NULL if the length of the received message is not required. On return, the *pid* location will contain the PID of the process which sent the message. The *len* location (if provided) will contain the length parameter provided with the sent message. Receive returns a pointer to the message received.

The Reply() operation returns a message to, and unblocks a process which previously performed the Send() operation. The parameter *pid* should contain the PID of the blocked sender, and the parameter *msg* is a pointer to the returned message (if any). Reply() returns 0 on success or NOSUCHPROC if the destination of the reply does not exist or is not blocked awaiting a Reply().

The MsgWaits() routine returns a 1 if there is at least one message waiting to be received by the calling process. Otherwise, a 0 is returned. This call does not block.

Part II

Communication Support

Chapter 2

Association Management

2.1 Introduction

This chapter describes the communication component (DASCOM) of DAS and gives instructions for its use.

DASCOM supports all association classes (1, 2 and 3), and supports operation classes 1 and 2. It also supports direct use of the presentation layer interface for data transfer. The association control service conforms to CCITT recommendation X.227. The presentation service provides the kernel functional unit of CCITT recommendation X.226. The session service provides the full-duplex basic combined subset of functional units described by CCITT recommendation X.225. Class 0 of the transport layer of CCITT recommendation X.224 is also provided. The transport layer can make use of several network services. Currently supported interfaces are those to sunlink X.25, UBC X.25, and TCP/IP.

DASCOM provides interface routines that enable clients and servers to communicate using the remote operations protocol or presentation data units directly. A server is an application process which blocks awaiting events such as connection requests or remote operations requests. A client is an application process which generates these events by requesting a connection or requesting that a server perform some operation. Access to the association control and data transfer services is provided through calls to DASCOM interface subroutines. DASCOM allows any reasonable number of application processes

(APs) to exist within one UNIX process. Each AP runs as an individual sub-process (thread) of the UNIX process. Each UNIX process may have a main thread which is the recipient of all connection requests directed at that UNIX process. This thread may handle the connection itself or may transfer the connection to any other thread within the process. In this way the main thread may field incoming connection requests by creating a worker thread and passing the connection to the newly created worker. It is possible to set up a server thread on one or both ends of a connection so that remote operation requests may flow in both directions. Each client and server may have 0 or more connections with APs in the same or different UNIX processes. It is also possible for multiple clients to share a connection directing multiple outstanding requests at the same server.

The remainder of this part discusses the use, behaviour, parameters and return values for the DASCOM interface routines.

2.2 Initialization Steps

Before the services of DASCOM may be used, two initialization steps are required. The first step is a call to the DASCOM initialization subroutine InitComm(). All users of DASCOM, whether they intend to be connection initiators, responders or both, must make a call to InitComm() before DASCOM may be used. The header for this subroutine is as follows:

```
int InitComm( paddr )
PresentationAddress *paddr;
```

The first parameter, *paddr* represents a presentation address. If the initializing AP wishes only to make connection requests of other AP's, and not receive any connection requests, then the value of this parameter should be NULL. Otherwise, this parameter reflects the address on which the initializing AP is willing to receive connection requests. The format of the presentation address is found in section 2.3.

The second initialization step need only be executed by an AP wishing to act as a responder (wishing to receive incoming connection requests). This is the RegisterServer() DASCOM interface subroutine. This subroutine requires no parameters. This routine is to be called by the thread wishing to act as the main thread for the UNIX process. It is this thread which is notified each time an incoming connection request is received. Aside from this unique property, the main thread is no different from any other thread. On receipt of a connection request, the main thread may either handle the connection itself, or pass the connection to some other thread.

Once InitComm() and (possibly) RegisterServer() have been called, the APs are free to make use of the DASCOM services.

2.3 Addressing

Presentation addresses are required for several purposes when using DASCOM. First, a presentation address is used when an AP initializes DASCOM to indicate to DASCOM which address it must accept incoming connection requests on. Also, presentation addresses are used when an AP is establishing an association with another AP in order to identify both the called and calling APs. The format of an DASCOM presentation address is as follows:

```
typedef struct PresentationAddress
{
        OCTS            *pSelector;
        OCTS            *sSelector;
        OCTS            *tSelector;
        LIST            *nAddress;   /* list of OCTS */
}               PresentationAddress;
```

Each of *pSelector*, *sSelector* and *tSelector* are pointers to ASN.1 octet string structures (see section 4.2.1 for the format of the OCTS structure). The *nAddress* field is a list of octet string structures. This list is constructed using the list routines described in section 4.2.1.

If the initialization is performed correctly, a 0 (zero) value is returned. Otherwise, a -1 is returned. For InitComm() to be successful, at least one network address on the list of network addresses (which forms part of the presentation address) must be in a format recognizable to DASCOM. DASCOM delivers connection requests received on any of the valid network addresses in the list. Currently, (subject to the communication services available on your machine) three network address formats are recognized by DASCOM, each of which is represented by a string.

Each octet string making up a valid network address is composed of two parts concatenated together: a network address type, and a network address value. The substrings which represent the valid address types are as follows:

- *Internet=*
- *ubcX25=*
- *sunX25=*

For an internet address, the address value is made up of three substrings. The first substring is either a machine name which is recognized locally, or an internet address in the *N. N. N. N* format where *N* is a 1 to 3 digit decimal number in the range 0 to 255. The second substring is the single character *+*. The third substring is a port number in decimal notation. Here are three examples of octet string values making up internet addresses:

- *Internet=koolaid.cs.ubc.ca+5101*
- *Internet=koolaid+55101*
- *Internet=128.189.97.62+555101*

For an UBC-X25 address, the address value is made up of several substrings all concatenated together. The first substring is optional and may be used to set the packet size to 128. It takes the following form: *p,* (the comma is part of the substring - i.e.

this is a two character substring). The second two-character substring is used to reverse the charges when making an outgoing call and takes the form *r,* (again, the comma forms part of the substring). This substring is also optional and only has meaning when constructing an address for an outgoing connection request (see the section on the Bind() DASCOM interface subroutine). The next substring is mandatory and represents the X25 address. This address is represented as a string of decimal digits. The following substring is mandatory and consists of the single character *+*. The final substring is also mandatory, and consists of an even number (between 2 and 32) of hexadecimal digits. Each pair of hexadecimal digits represents one byte of user data. Here are four examples of octet string values making up ubcX25 addresses:

- *ubcX25=p,8310141002+58353030*
- *ubcX25=p,r,123421920030045+58353030*
- *ubcX25=p,131106070013600+0301810023*
- *ubcX25=150527372000054+03018100*

The format for a Sunlink-X25 address is the same as for UBC-X25 addresses with the following exceptions. First, the substring indicating the network address type is, of course, *sunX25=* rather than *ubcX25=*. Secondly, the last two substrings (*+* and the user data) are both optional and if one is present, must both be present. Here are four examples of octet string values making up sunX25 addresses:

- *sunX25=p,8310141002+58353030*
- *sunX25=p,r,123421920030045*
- *sunX25=p,131106070013600*
- *sunX25=150527372000054+03018100*

2.4 The CONNINFO structure

All connection management operations require an extensive set of parameters and results passed between the user and the various communication layers of DASCOM. Therefore a

general purpose structure, called the *CONNINFO* structure, has been created as a placeholder for this information. This structure is used to pass parameters to and return results from connection management operations. Because it is used by several connection management routines, and because all routines do not require the same set of parameters, the fields of importance will vary according to which operation is being performed. The relevant fields are discussed in the sections describing the DASCOM interface subroutines. Here are the fields which are of interest to the user of DASCOM.

```
typedef struct          /* connection info passed between layers  */
    {
    /* the following for the session layer */
    BYTE *callingssur;  /* pointer to session calling user ref    */
    int  callingssurlen;
    BYTE *calledssur;   /* pointer to session called user ref     */
    int  calledssurlen;
    BYTE *commonssur;   /* pointer to session common user ref     */
    int  commonssurlen;
    BYTE *addtlref;     /* pointer to session additional reference*/
    int  addtlreflen;
    short ureq;         /* user requirements                      */
    BYTE reason;        /* disconnection/refusal reason           */

    /* the following for the presentation layer                   */
    int  pVersion;      /* presentation version                   */
    PresentationAddress *callingAddr;   /* addresses              */
    PresentationAddress *calledAddr;
    PresentationAddress *respondingAddr;
    LIST *pcdl;         /* pres context definition list           */
    LIST *pcil;         /* pres context identifier list           */
    Default_context_name *dcn; /* pres default context name       */
    int  *dcr;          /* default context result                 */
    BYTE presReqs;      /* presentation requirements              */
    int  *pReason;      /* abort reason, provider reason          */
    int  PDVchoice;     /* encoding choice for presentation data  */
    OID  *thisTSN;      /* transfer syntax name - only used in PCP*/
    int  thisPC;        /* presentation context for this data unit*/
    int  dataPC;        /* pres ctxt for data xfers, -1 for DCN   */
```

```
    int  useRose;        /* NOROSE or USEROSE  - use pdata or RO   */

    /* the following for ACSE */
    int  ACSEtype;       /* pdu type at acse level                 */
    int  ACSEversion;    /* ACSE version                           */
    OID  *acn;           /* application context name */
    UNIV *calledTitle;
    UNIV *calledQualifier;
    int  *calledAPInvId; /* application process Invocation id      */
    int  *calledAEInvId; /* application entity Invocation id       */
    UNIV *callingTitle;
    UNIV *callingQualifier;
    int  *callingAPInvId;
    int  *callingAEInvId;
    UNIV *respondingTitle;
    UNIV *respondingQualifier;
    int  *respondingAPInvId;
    int  *respondingAEInvId;
    UNIV ACSEuserInfo;
    int  *ACSEReason;    /* release request reason                 */
    int  ACSEResult;     /* associate response result              */
    int  REsource;       /* associate source diagnostic source     */
    int  REdiagnostic;   /* associate source diagnostic            */
    }    CONNINFO;
```

2.5 Waiting For An Event: The *Await()* call

An application process (AP) may receive incoming events by calling the *Await()* subroutine. The Await() subroutine call will block the calling AP until some event is received. This header for this subroutine is as follows:

```
PUBLIC OP Await( idp, invokeidp, argp )
int          *idp;        /* pointer to space for returned connection id */
InvokeIDType *invokeidp;  /* pointer to space for returned invokeid      */
void         **argp;      /* pointer to returned arguments               */
```

The following typedefs apply to this header:

```
typedef int     OP;             /* INTEGER */
typedef int     InvokeIDType;   /* INTEGER */
```

All three parameters are used to return results after the completion of the call. Prior to the subroutine call, the fields *idp* and *invokeidp* should each point to memory large enough to hold integers. The *argp* field should point to a memory location large enough to hold a single pointer value. The meaning of these results depends on the return value of Await(), though in general the space pointed to by *idp* will be filled with the association identifier (or connection id) of the association which generated the event, and the space pointed to by *invokeidp* will contain the invoke id (see X.219) of the associated event (if there is one). The *argp* pointer will be filled with a pointer to an argument received as part of the event. This could be a bind argument, invoke argument, etc. A side effect of this call is that on return, a new memory frame is pushed onto the AP's memory stack. All DASCOM-allocated results are allocated on this new memory frame. See the Threads documentation for a description of memory frames.

The return value of Await may be any one of the following:

- BINDOP,
- UNBINDOP,
- ROREJECTU,
- ABORTOP,
- PRESDATA,
- some other positive operation value.

A BINDOP return value indicates that some AP is attempting to establish an association. On return the *idp* location contains the identifier of the attempted association, the *invokeidp* location is meaningless, and the *argp* pointer location points to a CONNINFO

structure containing received bind arguments. The CONNINFO structure exists in memory allocated on the applications new top memory frame. The important fields of this structure are discussed in the section on connection establishment.

An UNBINDOP return value indicates that an associated AP is releasing the association. On return the *idp* location holds the identifier of the released association, *invokeidp* is meaningless, and the *argp* pointer location points to a CONNINFO structure (allocated from the new memory frame) containing received release arguments.

A ROREJECTU return value indicates that a problem with a previously sent remote operations Result() or Error() (discussed later) was detected by the associated AP. On return the *idp* location holds the identifier of the association over which the problem was detected, the *invokeidp* location contains the remote operation's invocation identifier (if one is received) of the Result() or Error() that caused the problem, and the *argp* location points to an OUTCOME structure (allocated from the new memory frame). The meaningful fields of this structure for a ROREJECTU are *problemType* and *problemVal*. *ProblemType* and *problemVal* can be one of the following pairs:

problemType GENPROB: *problemVal* can be any of: unrecognisedAPDU, mistypedAPDU or badlyStructuredAPDU.

problemType RRESPROB: *problemVal* can be any of: unrecognisedInvocation, resultResponseUnexpected or mistypedResult.

problemType RERRPROB: *problemVal* can be any of: unrecognisedInvocation, errorResponseUnexpected, unrecognisedError, unexpectedError or mistypedParameter.

The full OUTCOME structure is as follows: (though for a ROREJECTU return value, only fields *problemType* and *problemVal* are meaningful)

```
typedef struct
    {
    UNIV    arg;      /* result or error parameter */
    int     problemType;  /* for reject U or P    */
    int     problemVal;   /* for reject U or P    */
```

```
int     errorVal;     /* for error          */
} OUTCOME;
```

An ABORTOP return value indicates that an association has been abnormally terminated. The *idp* location holds the identifier of the aborted association, the *invokeidp* location is meaningless, and the *argp* pointer location either points to a CONNINFO structure (allocated from the new memory frame) or has the value NULL if no abort information is available.

A PRESDATA return value signals the arrival of presentation data (without the use of remote operations). The *idp* location holds the identifier of the connection over which the data arrived. The *invokeidp* location is meaningless, and the *argp* pointer location points to received presentation data.

These five possible return values are all negative integers. Any positive value returned from Await() constitutes an operation value and is the result of an associated AP invoking an operation on the awaiting entity. In this case the return value is the operation value of the requested operation. On return the *idp* location holds the association identifier of the association over which the operation request is received. The *invokeidp* location holds the invocation identifier of the requested operation, and the *argp* location holds the address of the received invoke arguments. The encoding and decoding of these arguments are explained below in the section concerning the use of the remote operations primitives.

2.6 Establishing An Association: The *Bind()*, *AcceptBind()* and *RefuseBind()* Calls, and the *CONNINFO* Structure.

There are two sides to any application association: the initiator and the responder. The initiator makes the connection attempt and the responder may either accept or refuse the requested connection. The normal sequence of events is as follows:

1. An initiating AP makes a connection request of some other AP by using the Bind() subroutine call. The initiator is blocked on the Bind() call. The desired responder must be waiting for events using the Await() subroutine call.

2. The Await() subroutine will return to the responder indicating that a connection is being requested.

3. The responder may accept the connection using the AcceptBind() subroutine call, or it may refuse the connection using the RefuseBind() subroutine call. Neither of these calls block the responder.

4. The initiator now returns from the call to Bind() and the return value indicates the status of the requested association.

2.6.1 The *Bind()* Subroutine

The first step in the process of establishing an association is for the initiator to issue an association request using the Bind() DASCOM interface subroutine. This routine sends an association request to the proposed responder. The Bind() subroutine header is as follows:

```
/* returns id, CONNREFUSED or ABORTOP */
PUBLIC int Bind( cinfo, result, server )
CONNINFO  *cinfo;
CONNINFO  **result;  /* result structure returned here */
PID       server;
```

On subroutine call, *cinfo* must point to a CONNINFO structure containing the Bind arguments. *result* must point to a memory location large enough to hold a pointer to a CONNINFO structure. It is in this returned CONNINFO structure that the received Bind results are returned (further information regarding the returned results of the Bind() interface subroutine are discussed below). A side effect of the call to Bind() is that the top memory frame of the calling AP is removed and passed to DASCOM (see section 1.3 for a description of memory frames). The *cinfo* structure should be allocated from memory on this top memory frame. The *server* parameter either contains a valid PID or a 0 ((PID) 0). If the initiator wishes an association of classes 2 or 3, this parameter contains the PID of the process which is to "Await()" operation requests over this requested association. If the requested association class is 1, then this parameter contains a 0.

The *cinfo* parameter points to a CONNINFO structure which contains the bind argument as well as all the parameters necessary for the communication layers to establish the connection. The fields of the CONNINFO (pointed to by *cinfo*) which must be provided by the initiating AP are as follows:

- For the session layer:
  - Session Connection Identifier
    - Calling SS-user reference and length
      - BYTE *callingssur; /* 64 bytes maximum*/
      - int callingssurlen; /* length in bytes */
    - Common Reference and length
      - BYTE *commonssur; /* 64 bytes maximum*/
      - int commonssurlen; /* length in bytes */
    - Additional Reference
      - BYTE *addtlref; /* 4 bytes maximum */
      - int addtlreflen; /* length in bytes */
  - Session User Requirements
    - short ureq; /* must currently be set to DUPLEX */
- For the Presentation Layer:
  - Calling Presentation Address
    - PresentationAddress *callingAddr;
  - Called Presentation Address
    - PresentationAddress *calledAddr;
  - Presentation Context Definition List
    - LIST *pcdl;
  - Default Context Name
    - Default_context_name *dcn;
  - Transfer Syntax Name for Presentation Connect User Data
    - OID *thisTSN;
  - Presentation Context id for Presentation Connect User Data
    - int thisPC; (-1 for simple encoding)

  - Presentation Context id for user-data PDUs
    - int dataPC; (-1 for simple encoding)

- For the Association Control Layer:
  - Application Context Name
    - OID *acn;
  - Calling AP Title
    - UNIV *callingTitle;
  - Calling AE Qualifier
    - UNIV *callingQualifier;
  - Calling AP Invocation-identifier
    - int *callingAPInvId;
  - Calling AE Invocation-identifier
    * int *callingAEInvId;
  - Called AP Title
    - UNIV *calledTitle;
  - Called AE Qualifier
    - UNIV *calledQualifier;
  - Called AP Invocation-identifier
    - int *calledAPInvId;
  - Called AE Invocation-identifier
    - int *calledAEInvId;
  - User Information
    - UNIV ACSEuserInfo; /* pointer to a "list of external" */

- For the Application Interface:
  - Application Context Name
    - int useRose;

Many of these fields require further explanation. The session calling, common and additional references may have any length up to the indicated maximum. Therefore, if these parameters are not to be included, the length of each should be set to zero, and

the pointers should be assigned NULL. The session user requirements must take the value DUPLEX. Any other value would be an error as this is the only session functional unit currently supported.

The calling and called presentation addresses are represented by the PresentationAddress structure. This structure has the same form, and is constructed in the same way as was outlined above in section 2.2.

At least one valid network address must exist on the network address list for the *calledAddr* field of the CONNINFO structure. All other fields (selectors) of the *calledAddr* structure, and all fields of the *callingAddr* structure (including the network address list field) are optional and may be assigned the value NULL (actually, the *callingAddr* field itself may be assigned the value NULL). DASCOM will scan the network addresses in the *calledAddr* list and attempt to connect to the first valid network address it encounters.

The *pcdl* field of the CONNINFO structure points to the presentation context definition list. This is a list of T02 structures. This list is constructed using the list routines described in section 4.2.1. The T02 structure is as follows:

```
typedef struct T02 {  /* element of presentation context definition list */
        Presentation_context_identifier r11;
        Abstract_syntax_name r12;
        LIST  *r13;     /* pointer to list of Transfer_syntax_name structures */
        } T02;
```

The related typedefs are as follows:

```
typedef int      Presentation_context_identifier;
typedef OID      *Abstract_syntax_name;
typedef OID      *Transfer_syntax_name;
```

For a description of the meaning of this parameter see X.216 - 10.2.1.4 and X.226 - 6.2.2.7 and 6.2.6.1. The list of transfer syntax names is constructed using the list routines described in section 4.2.1. This section also provides a description of the OID (object

identifier) structure. The presentation context definition list parameter relates to the *thisPC*, *thisTSN* and *dataPC* parameters (described later).

The *dcn* field of the CONNINFO structure points to a structure which represents the default context name proposed for this connection. This structure has the following form:

```
typedef struct Default_context_name {
        Abstract_syntax_name r11;
        Transfer_syntax_name r12;
        }  Default_context_name;
```

The related typedefs are as follows:

```
typedef OID     *Abstract_syntax_name;
typedef OID     *Transfer_syntax_name;
```

For a description of the meaning of this parameter see X.216 - 10.2.1.6 and X.226 - 6.2.2.8 and 6.2.6.2. See section refasn1oid for a description of the OID (object identifier) structure.

The *thisTSN* field is a pointer to an OID structure describing the transfer syntax name for the presentation connect data values. This field is optional (according to the rules specified in X.226 8.2.4.7) and the value should be set to NULL if the transfer syntax name is not required (i.e. if simple encoding is desired or if only one transfer syntax name was proposed for this presentation context). This parameter relates to *thisPC* described below and *pcdl* described above.

The *thisPC* field is an integer representing the presentation context of the connect presentation data values. If simple encoding is desired this field should be set to -1. See X.226 - 8.4.2.6 as well as the presentation context definition list parameter (*pcdl*) described previously.

The *dataPC* field is an integer representing the presentation context to be used for normal data transfer (after the connection is fully established). If simple encoding is

desired for presentation user data during the data transfer phase, then this parameter would be set to -1.

The *acn* field is a pointer to an object identifier structure representing the application context name. This is a mandatory field. Information on this parameter may be obtained from X.217 - 9.1.1.2 and X.227 - 7.1.4.2.

The fields *callingTitle*, *callingQualifier*, *calledTitle* and *calledQualifier* are all pointers to ASN.1 encoded data in the form of an IDX list. These parameters convey the calling and called AP titles and the calling and called AE qualifiers. The encoded ASN.1 data may be of any type. Information concerning these parameters may be found in X.217 9.1.1.3, 9.1.1.4, 9.1.1.7 and 9.1.1.8, and also in X.227 7.1.4.3, 7.1.4.4, 7.1.4.7 and 7.1.4.8. For information on the IDX structure see section 4.3. These are all optional fields and may therefore be set to NULL.

The fields *callingAPInvId*, *callingAEInvId*, *calledAPInvId* and *calledAEInvId* are all pointers to integers. These parameters convey the calling and called AP and AE invocation identifiers. Their meaning is described in X.217 9.1.1.5, 9.1.1.6, 9.1.1.9 and 9.1.1.10, and also in X.227. These are all optional fields and may therefore be set to NULL.

The *ACSEuserInfo* field is used to convey the user's bind arguments to the proposed responder. The form of this parameter is a pointer to a list of EXT (external) structures. This list is constructed using the list routines described in the ASN.1 documentation. The EXT structure is described in section 4.2.1 and contains the following fields:

```
typedef struct EXT {
        OID     *dref;   /* direct reference (optional)       */
        int     *iref;   /* indirect reference (optional)     */
        OCTS    *value;  /* data value descriptor (optional) */
        CODING  ed;      /* Encoding policy                   */
} EXT;
```

A description of the OID (object identifier) and OCTS structures are found in section 4.2.1. The meaning of the first three fields may be found in ISO 8824, 32.1 to 32.14.

The CODING structure has the following format:

```
typedef struct
    {
        int     choice;
        union
        {
                ANYtype single;         /* single ASN.1 type */
                OCTS    octet;          /* octet aligned     */
                BITS    arbitrary;      /* arbitrary         */
        }       data;
    } CODING;

#define single_asn1_tag   0xa0000000
#define octet_aligned_tag 0x80000001
#define arbitrary_tag     0x80000002
```

The three indicated defines are used by the *choice* field to indicate the type of encoding chosen (single, octet aligned or arbitrary). The *choice* field must take one of these values. The meaning of the three encoding choices is described ISO 8824, 32.1 to 32.14. If encoding of a single ASN.1 type is desired, then the single field should contain a pointer to an IDX list. If an octet-aligned or bit-aligned value is to be encoded, then the union is used as an OCTS or BITS structure respectively. The OCTS and BITS types are defined in section 4.2.1.

The final field is *useRose*. This field indicates to the application interface whether remote operations or presentation data are to be used for data transfer. The associated defines are as follows:

```
/* definitions for use or non use of ROSE */
#define USEROSE 0
#define NOROSE  1
```

If this field is set to USEROSE, then the remote operations primitives are available for use. The PData() data transfer primitive is not available. If, instead, useRose is set to

NOROSE, only the PData() data transfer primitive is available for information transfer.

Once the CONNINFO structure and other parameters are filled and Bind() is called, the caller is blocked pending the outcome of the connection (Bind()) request. The connection request and its associated parameters are transferred to the proposed responder.

2.6.2 Receiving a Connection Request

An AP wishing to receive requests for associations does so by calling the Await() subroutine. An incoming connection request will cause Await() to return the value BINDOP. The memory location provided for the *idp* parameter will now contain the identifier for this newly proposed connection and the pointer location provided for the *argp* parameter will contain a pointer to a CONNINFO structure (see section 2.5). The identifier returned must be retained as it is used in all subsequent references to this association.

The CONNINFO structure returned (actually, its pointer is returned) contains all the arguments as provided by the initiator of the association. The only exception to this is the *dataPC* filed which is provided to the initiating DASCOM for future data transfer and is not transferred on association creation. Optional fields which are not received as part of the bind request will contain a NULL pointer. Mandatory fields, and optional fields which are received in the bind request will follow the same format described above in the section on the Bind() interface subroutine. One effect of Await() returning is that a new memory frame (see section 1.3) is pushed onto the memory stack of the "Awaiting" AP. The CONNINFO structure described here is allocated from memory on this new memory frame.

2.6.3 Responding to a Connection Request

The AP on the receiving side of a BINDOP may either accept or refuse the requested connection. If accepted, the connection is open and may be used for information transfer (using remote operations primitives). If refused, no connection is established. The DASCOM interface subroutines for accepting and refusing a proposed association are AcceptBind() and

RefuseBind() respectively.

Accepting a Proposed Connection

The AcceptBind() DASCOM interface subroutine requires two parameters. The header for this routine is as follows:

```
int AcceptBind( cid, cinfo )  /* returns 0 or ABORTOP */
int      cid;
CONNINFO *cinfo;
```

AcceptBind() returns either 0 or ABORTOP. If the proposed connection is aborted between the time that the connection request is received and the time that it is accepted or refused, then ABORTOP is returned. Otherwise a 0 is returned. The parameter ***cid*** is an integer containing the connection identifier of the connection being accepted. ***cinfo*** is a pointer to connection response arguments. This structure is created by the accepting AP and should be contained in the topmost memory frame prior to the subroutine call. One effect of the call to AcceptBind() is that the topmost memory frame (containing the arguments) is popped from the calling AP's memory stack and passed to the DASCOM process. Therefore, nothing other than AcceptBind() arguments should be allocated from the top memory frame.

The fields of the CONNINFO structure which are provided by the responding AP are as follows:

- For the session layer:
  - session connection identifier
    - Called SS-user reference and length
      - BYTE *calledssur; /* 64 bytes maximum*/
      - int calledssurlen; /* length in bytes */
    - Common Reference and length
      - BYTE *commonssur; /* 64 bytes maximum*/

        - int commonssurlen; /* length in bytes */
      - Additional Reference and length
        - BYTE *addtlref; /* 4 bytes maximum */
        - int addtlreflen; /* length in bytes */
  - Session User Requirements
    - short ureq;

- For the Presentation Layer:
  - Responding Presentation Address
    - PresentationAddress *respondingAddr;
  - Presentation Context Definition Result List
    - LIST *pcdl;
  - Presentation Context id for Presentation Connect User Data
    - int thisPC; (-1 for simple encoding)
  - Presentation Context id for user-data PDUs
    - int dataPC; (-1 for simple encoding)

- For the Association Control Layer:
  - Application Context Name
    - OID *acn;
  - Responding AP Title
    - UNIV *respondingTitle;
  - Responding AE Qualifier
    - UNIV *respondingQualifier;
  - Responding AP Invocation-identifier
    - int *respondingAPInvId;
  - Responding AE Invocation-identifier
    - int *re dingAEInvId;
  - User Informa n
    - UNIV ACSEuserInfo;

- For the Application Interface:

- Application Context Name
  - int useRose;

Once again, these fields require further explanation. The called, common and additional references for the session layer follow the same format as the calling, common and additional references for the Bind() operation. The meaning of these fields may be found in X.215 - 12.1.2 and X.225 - 7.4.1, 8.3.4.3, 8.3.4.4 and 8.3.4.5. The session user requirements must take the value DUPLEX. Any other value would be an error as this is the only session functional unit currently supported (see X.225 8.3.4.13).

The responding presentation address follows the same format as the called and calling presentation addresses supplied to the Bind() interface routine. The meaning of this field is found in X.216 - 10.2.1.3 and X.226 - 6.2.3.3 and 6.2.3.4.

The *pcdl* field of the CONNINFO structure points to the presentation context definition result list. This is a list of T03 structures. This list is constructed using the list routines described in section 4.2.1. The T03 structure is as follows:

```
typedef struct T03 {
        Res             r11;                    /* result                */
        Transfer_syntax_name r12;               /* Transfer_syntax_name */
        int             *provider_reason;
}               T03;
```

The related typedefs are as follows:

```
typedef int     Res;
#define acceptance 0
#define user_rejection 1

typedef OID     *Transfer_syntax_name;
```

The *r11* field typedefs to an integer and may take the value *acceptance* or *user_rejection*. This field indicates the acceptance or rejection of the corresponding proposed presentation

context in the presentation context definition list received in the Bind() request. If the value is acceptance, then the *r12* field must point to a valid object identifier structure (described in section 4.2.1) containing one of the proposed transfer syntaxes for that presentation context. Otherwise, the *r12* pointer should take the value NULL. The *provider_reason* field is a pointer to an integer and must always take the value NULL. More information regarding the presentation context definition result list is found in X.216 - 10.2.1.5 and X.226 - 6.2.3.5.

The fields *thisPC* and *dataPC* take the same format and are used for the same purpose as described above for the Bind() interface routine. The only difference is that the values supplied are used for data transfer in the direction of responder to initiator.

The *acn* (application context name) field has the same format as that of the Bind() routine described above. More information regarding this parameter may be found in X.217 - 9.1.1.2 and X.227 - 7.1.5.2.

The *respondingTitle* and *respondingQualifier* fields have the same format as the *callingTitle*, *callingQualifier*, *calledTitle* and *calledQualifier* fields required for the Bind() routine. These fields convey the responding AP title and responding AE qualifier. More information regarding these parameters may be found in X.217 - 9.1.1.11 and 9.1.1.12 as well as in X.227 - 7.1.5.3 and 7.1.5.4.

The *respondingAPInvId* and *respondingAEInvId* fields have the same format as the *callingAPInvId*, *callingAEInvId*, *calledAPInvId* and *calledAEInvId* fields required for the Bind() routine. These fields convey the responding AP and AE invocation identifiers. More information regarding these parameters may be found in X.217 - 9.1.1.13 and 9.1.1.14 as well as in X.227 - 7.1.5.5 and 7.1.5.6.

The field *ACSEuserInfo* is used to convey the users bind results to the initiator. The form of this parameter is the same as that of the same field when used for the Bind() interface subroutine. Information regarding this parameter is found in X.217 - 9.1.1.15 and X.227 - 7.1.5.10.

The final field is *useRose*. This field indicates to the application interface whether remote operations or presentation data are to be used for data transfer. The associated defines are as follows:

```
/* definitions for use or non use of ROSE */
#define USEROSE 0
#define NOROSE  1
```

If this field is set to USEROSE, then the remote operations primitives are available for use. The PData() data transfer primitive is not available. If, instead, useRose is set to NOROSE, only the PData() data transfer primitive is available for information transfer.

Refusing a Proposed Connection

An AP wishing to refuse an incoming Bind() request does so using the RefuseBind() DASCOM interface routine. This routine requires two parameters. The header of this routine is as follows:

```
int RefuseBind( cid, cinfo )  /* returns 0 or ABORTOP */
int      cid;
CONNINFO *cinfo
```

RefuseBind() returns either 0 or ABORTOP. If the proposed connection is aborted between the time that the connection request is received and the time that it is refused, then ABORTOP is returned. Otherwise a 0 is returned. The parameter *cid* is an integer containing the connection identifier of the connection being refused. *cinfo* is a pointer to connection response arguments. This structure is created by the refusing AP and should be contained in the topmost memory frame prior to the subroutine call. One effect of the call to AcceptBind() is that the topmost memory frame (containing the arguments) is popped from the calling AP's memory stack and passed to the DASCOM process. Therefore, nothing other than RefuseBind() arguments should be allocated from the top memory frame.

The fields of the CONNINFO structure which must be provided by the refusing AP are as follows:

- From the session layer:
  - session connection identifier
    - Called SS-user reference
      - BYTE *calledssur; /* 64 bytes maximum */
      - int calledssurlen;/* length in bytes */
    - Common Reference
      - BYTE *commonssur; /* 64 bytes maximum */
      - int commonssurlen;/* length in bytes */
    - Additional Reference
      - BYTE *addtlref; /* 4 bytes maximum */
      - int addtlreflen;/* length in bytes */
  - Session User Requirements
    - short ureq;
- For the Presentation Layer:
  - Responding Presentation Address
    - PresentationAddress *respondingAddr;
  - Presentation Context Definition Result List
    - LIST *pcdl;
  - Default Context Result
    - int *dcr;
  - Presentation Context id for Presentation Connect User Data
    - int thisPC;
- For the Association Control Layer:
  - Application Context Name
    - OID *acn;
  - ASCE associate result
    - int ACSEResult;

- ACSE associate source diagnostic
  - int REdiagnostic;
- Responding AP Title
  - UNIV *respondingTitle;
- Responding AE Qualifier
  - UNIV *respondingQualifier;
- Responding AP Invocation-identifier
  - int *respondingAPInvId;
- Responding AE Invocation-identifier
  - int *respondingAEInvId;
- User Information
  - UNIV ACSEuserInfo;

All the session fields (*calledssur*, *calledssurlen*, *commonssur*, *commonssurlen*, *addtlref*, *addtlreflen* and *ureq*) follow the same format and have the same meaning as they do for the AcceptBind() interface routine. Information regarding these fields is found in X.215 - 12.1.2 and in X.225 - 8.3.5.3, 8.3.5.4, 8.3.5.5 and 8.3.5.7.

The presentation fields *respondingAddr*, *pcdl* and *thisPC* all follow the same format and have the same meaning as they do for the AcceptBind() interface routine. Information regarding the *respondingAddr* and *pcdl* parameters is found in X.216 - 10.2.1.3 and 10.2.1.5, as well as in X.226 - 6.2.4.2, 6.2.4.3 and 6.2.4.4. The *dcr* field is the default context result supplied in response to the default context name proposed in the Bind() request. Its form is a pointer to an integer. This is an optional field and if not desired must be set to NULL. If it is desired, *dcr* must consist of a pointer to an integer which contains either *acceptance* or *user_rejection*. More information regarding this parameter may be found in X.216 - 10.2.1.7 and X.226 - 6.2.4.5 and 6.2.6.2.

The ACSE fields *acn*, *respondingTitle*, *respondingQualifier*, *respondingQualifier*, *respondingAEInvId* and *ACSEuserInfo* all take the same format and carry the same meaning as their corresponding fields required for the AcceptBind() interface primitive. More information for these fields is found in the same references given for the AcceptBind() routine.

The ACSE fields *ACSEResult* and *REdiagnostic* indicate the type of, and reason for connection refusal. *ACSEResult* carries the ACSE associate result, while *REdiagnostic* holds the ACSE associate source diagnostic. *ACSEResult* is an integer, and may take the value *rejected_permanent* or *rejected_transient*. If some other value is supplied *ACSEResult* defaults to *rejected_permanent*. More information regarding *ACSEResult* is found in X.217 - 9.1.1.16 and X.227 - 7.1.5.7. *REdiagnostic* indicates the reason for connection refusal and is also an integer. It takes one of the following values:

- no_reason_given
- application_context_name_not_supported
- calling_AP_title_not_recognized
- calling_AP_invocation_identifier_not_recognized
- calling_AE_qualifier_not_recognized
- calling_AE_invocation_id_not_recognized
- called_AP_title_not_recognized
- called_AP_invocation_identifier_not_recognized
- called_AE_qualifier_not_recognized
- called_AE_invocation_id_not_recognized

If some other value is provided, *REdiagnostic* defaults to the value *null*. More information is found in X.217 - 9.1.1.18 and X.227 - 7.1.5.8.2 and 9.1.

2.6.4 Receiving Connection Request Results (or: Bind() Return Values)

Once the responder has either accepted or refused the Bind() request (or some problem has occurred), the initiator's Bind() subroutine call will return. One side effect of Bind() returning is that a new memory frame will be pushed onto the calling AP's memory stack. The return value from Bind() is one of the following:

- CONNREFUSED,
- ABORTOP or
- some positive value which is the connection identifier of a successfully established connection.

CONNREFUSED is returned if the destination AP will not accept the association, and ABORTOP is returned if the proposed destination could not be reached or if there is some problem at one of the protocol layers. These are both negative values. Any positive return value is the connection id of the successfully created connection. A return value of a positive connection identifier or CONNREFUSED is accompanied by a CONNINFO structure. The address of this structure is placed (by the Bind() subroutine) in the pointer location provided by the caller in the *result* parameter. The memory for this structure is taken from the new memory frame pushed by Bind() before it returns.

The contents of this returned CONNINFO structure will vary according to whether the connection was accepted or refused. If accepted, the fields will correspond in format and content to those described previously in the section on AcceptBind(). Likewise, if the connection is refused, the fields of the returned CONNINFO structure will correspond to those described in the section on RefuseBind(). There are, however, a couple exceptions.

The first exception is that the returned CONNINFO will not have a meaningful value in the *dataPC* field. The *dataPC* parameter is supplied for the AcceptBind() call as a local indication of which presentation context to use for future data transfers (of remote operations PDUs). It is not passed to the initiator in returned connection information.

The second exception is that there are a few fields returned to the initiator when Bind() returns which were not supplied by the rejecting responder. These fields are as follows:

```
BYTE reason;
int  *pReason;
int  REsource;
```

The first of these fields is *reason*. This field corresponds to the session layer reason code for session connection refusal and is only meaningful if the Bind() result is CONNREFUSED. More information regarding the meaning of this parameter is found in X.225 - 8.3.5.9.

The second of these fields is *pReason*. This field corresponds to the presentation layer provider refusal reason. It is an optional field and only points to a valid memory location if the connection was refused by the presentation protocol, otherwise it contains the value NULL. If the connection was refused by the presentation layer, this field gives the reason that the connection was refused and points to a memory location containing one of the following:

- reason_not_specified
- temporary_congestion
- local_limit_exceeded
- called_presentation_address_unknown
- protocol_version_not_supported
- default_context_not_supported
- user_data_not_readable
- no_PSAP_available

More information regarding the meaning of this parameter and its possible values is found in X.226 - 6.2.4.9.

The third field is *REsource*. This field corresponds to the association control layer's result source parameter. If the connection is refused by the ACSE protocol layer, this field takes the value *ServiceProvider*. If the connection is refused or accepted by the application, then the value is *ServiceUser*. More information regarding the meaning of this parameter is found in X.226 - 7.1.5.8.

2.7 Releasing An Association: The *Unbind()* and *AcceptUnbind()* Calls

An established association may only be released by its initiator. To do so, the initiator calls the DASCOM interface subroutine Unbind(). The Await() subroutine will deliver the unbind request to the responder by unblocking and returning the value UNBINDOP. The responder must then call the AcceptUnbind() interface subroutine to complete connection release. The Unbind() and AcceptUnbind() interface routine headers are as follows:

```
int Unbind( cid, cinfo, result )
int cid;
CONNINFO *cinfo;
CONNINFO **result;  /* result CONNINFO structure returned here   */

int AcceptUnbind( cid, cinfo )
int      cid;
CONNINFO *cinfo;
```

Both subroutines return 0 on success or ABORTOP if the association was abnormally released prior to the call.

For Unbind(), the *cid* parameter is the connection identifier of the connection to be released. *cinfo* is a pointer to association release parameters and information in the form of a CONNINFO structure. The *result* parameter must point to a memory location large enough to hold a pointer to a CONNINFO structure. On return this pointer location will contain the address of the returned CONNINFO structure. Two fields of the CONNINFO structure are of importance when releasing an association. These fields are as follows:

```
int  *ACSEReason;
UNIV ACSEuserInfo;
```

The *ACSEReason* field is optional, and reflects the ACSE connection release reason. If desired, this field points to a memory location containing one of the following integer values:

- normal
- urgent
- user_defined

If this field is not desired, *ACSEReason* takes the value NULL.

The *ACSEuserInfo* is also optional, and reflects user information to be conveyed with the release request. The format of this field is exactly the same as it is for the Bind() request. Further information regarding both the *ACSEReason* and *ACSEuserInfo* fields may be found in X.227 - 7.2.4.

This CONNINFO structure provided to the Unbind() subroutine must be on the top memory frame of the calling AP as this frame is popped and passed down to the communication process. On return, a new frame, containing the returned CONNINFO structure is pushed onto the calling AP's memory stack.

The Unbind() request will cause the connected responder to return from the Await() call with an UNBINDOP return value (see the section on the Await() subroutine call). The returned connection identifier contains the id of the connection being released. The returned CONNINFO structure contains the information provided by the releasing AP in the format described for Unbind(). The responder must now call the DASCOM interface subroutine AcceptUnbind() providing the identifier of the connection being released, as well as a CONNINFO structure. The important fields of the *cinfo* parameter (CONNINFO structure) are the same fields which are important (and are described above) for the Unbind() subroutine call. These fields are used to convey the AP's release response reason and user information. More information may be found regarding these fields for AcceptUnbind() in X.227 - 7.2.5.

The call to AcceptUnbind() unblocks the initiator's Unbind() call returning the CONNINFO structure provided by the responder. This structure is provided using the *result* parameter. The contents and format of the returned CONNINFO structure are the same as are provided to the AcceptUnbind() subroutine.

2.8 Owning and Transferring Associations

An incoming connection request is always directed at the main server thread of the UNIX process. If this connection is correctly established (the server accepts the association) then that server becomes the owner of the association. The owner of an association is the server thread which receives all incoming events related to that association. These events include incoming operation requests and unbind requests. It is possible for the owner of a connection to designate some other thread as the connection's new owner. This may be done before or after the connection has been completely established. Once the connection has been transferred to some other owner, that new owner receives all subsequent events associated with that connection. The only two ways that the main server could regain control of the association are if the worker explicitly transfers control back to the main server, or if the worker thread is killed while the association is still active. A typical scenario could be as follows:

- Await() returns to main server indicating new connection request.
- Main server creates a worker to deal with new association.
- Main server transfers the association to the worker. The association is now completely out of the main server's control.
- The first task of the worker is to accept the association by issuing an AcceptBind().
- The worker now performs a loop Await()-ing and performing requests.

A slight deviation from the above scenario would be for the main server to accept the connection before transferring it to a worker. In this way, the connection could be passed to an existing worker without the worker having to accept the connection before it can start Await()-ing events. This is useful if a worker is to handle more than one connection.

The DASCOM interface subroutine which transfers an association from one server to another is TransferAssociation(). The header for this routine is as follows:

```
/* returns 0 on success, -1 on failure */
```

```
int TransferAssociation( cid, pid )
int cid;          /* association id     */
PID pid;          /* pid to transfer to */
```

The first parameter is the identifier of the association (or potential association) being transferred. The second parameter is the identifier of the thread which is to be the new owner of the association. If the association is successfully transferred a 0 is returned, otherwise a -1 is returned. Association transfer may be done at any time during the life of an association (or by the responder even before the association has been accepted).

2.9 Examples

Please refer to section 3.3 for examples regarding association management.

Chapter 3

Data Transfer

Once the association has been established, the applications may transfer operation requests and responses using the remote operations protocol, or using the presentation layer directly. Remote operation association classes one, two and three are supported. Therefore, at the initiators option, operation requests may flow in either or both directions over the association.

3.1 Presentation Layer Interface

The presentation data transfer routine consists of the following:

```
PUBLIC int PData( cid, idx )
int cid;
IDX *idx;
```

This routine takes the encoded data (represented as an IDX list) and transfers it over the association identified by *cid*. A zero (0) is returned on success, or a -1 is returned if the association was aborted. One result of the call to PData() is that the top memory frame of the calling thread is popped and transferred to the communication server. Therefore it is wise to place the arguments to PData() in the top memory frame before the call. The

PData() interface may only be used over a connection which was appropriately configured at connection request and accept time. See sections 2.6.1 and 2.6.3 for details regarding the configuration of connections for the use of the PData() service.

3.2 Remote Operations Interface

The Remote Operations interface may only be used over a connection which was appropriately configured at connection request and accept time. See sections 2.6.1 and 2.6.3 for details regarding the configuration of connections for the use of the Remote Operations service.

The remote operation interface routines consist of the following:

- Invoke()
- Result()
- Error()
- RejectU()

The Invoke() interface subroutine is used to request an operation of an associated application. The association class of the connection determines whether the initiator or responder (or both) may issue an Invoke(). This class is determined by the Bind() parameters discussed in section 2.6.1.

Invoke() parameters consist of the connection identifier, the invocation identifier, the operation being requested and operation arguments. Information provided as the arguments to the operation request may be of any type but should be encoded in some external representation. The ASN.1 module provides the services necessary to perform this encoding. Once Invoke() is called, the client making the call is blocked pending the server's response. The Invoke() header is as follows:

```
int Invoke( cid, invokeid, operation, arg, outcomep )
int cid;                    /* association id          */
InvokeIDType invokeid;      /* invocation id           */
OP           operation;     /* operation requested     */
IDX          *arg;          /* must be an IDX list     */
OUTCOME      **outcomep;    /* ptr to OUTCOME structure */
```

The related typedefs are as follows:

```
typedef int     InvokeIDType;

typedef int     OP;

/* structure returned in outcome parameter of Invoke */
typedef struct
    {
    UNIV    arg;     /* result or error parameter */
    int     problemType;  /* for reject U or P    */
    int     problemVal;   /* for reject U or P    */
    int     errorVal;     /* for error            */
    } OUTCOME;
```

The *cid* parameter is an integer and represents the association over which the invoke is to be sent.

The *invokeid* parameter is an integer which is passed transparently to the server. If there is more than one Invoke() outstanding over this association (i.e. more than one client process is using this association) then it is important that the invoke ids are unique over that association. These identifiers are used by DASCOM when results are returned to find the client process which initiated the invoke. If two or more outstanding invokes over the same association have the same invoke id the results will be unpredictable. This also assumes that the server uses the correct invoke id in its responses.

The *operation* parameter is used to identify the operation to be performed. The value of this parameter is some positive integer recognizable to both client and server as representing some operation.

The *arg* parameter is a pointer to an IDX list. This parameter forms the arguments to the requested operation. For more information on the IDX list structure see section 4.3.

The *outcomep* parameter must point to a location large enough to hold a pointer to an OUTCOME structure. On return from Invoke(), this location points to an OUTCOME structure allocated from the top memory frame for this process. This structure has the fields listed above. The meaning of the fields depends on the return value of Invoke().

There are five possible return values from Invoke(). These are:

- RORESULT
- ROERROR
- REJECTU
- REJECTP
- ABORTOP

A RORESULT return value indicates the successful performance of the operation with the server responding to the operation request using the Result() interface routine. In this case, the only field of importance in the returned OUTCOME structure is *arg*. This field will point to a buffer (assuming one is returned) containing the result data as passed back by the server. If no result data is returned then this field will have the value NULL.

A ROERROR return value indicates the occurrence of some problem during the performance of the operation (for example, a divide by zero error). This is generated by the server responding using the Error() interface routine. In this case, the important fields of the OUTCOME structure are *arg* and *errorVal*. The *arg* field will point to data (if any) which accompanies the returned error. The *errorVal* field is an integer provided by the server to indicate the nature of the error.

A ROREJECTU return value indicates the occurrence of some problem with the request itself (rather than an error discovered in the performance of the request). This value is returned when the server responds using the RejectU() interface routine. ROREJECTP is returned if the protocol service detects a problem with the request. In either case (ROREJECTU or ROREJECTP) the fields of importance in the OUTCOME structure are *problemType* and *problemVal*. These are both integers. The possible values for these fields are as follows:

problemType GENPROB (ROREJECTP only): *problemVal* can be any of: unrecognisedAPDU, mistypedAPDU or badlyStructuredAPDU.

problemType INVPROB (ROREJECTU only): *problemVal* can be any of: duplicateInvocation, unrecognisedOperation, mistypedArgument, resourceLimitation, initiatorReleasing, unrecognisedLinkedID, linkedResponseUnexpected or unexpectedChildOperation.

More information regarding the meaning of these *problemType* and *problemVal* fields is found in X.229 - 7.4.4.2 and 7.5.4.2.

Finally, ABORTOP is returned if the association over which the Invoke() was to be sent is lost.

The Invoke() operation will cause the intended server to return from Await() with a return value equal to the operation being requested. The server is then free to respond to the operation (immediately or at some future time) using the Result(), Error() or RejectU() interface subroutines.

The Result() interface subroutine header is as follows:

```
int Result( cid, invokeid, operation, arg )  /* returns 0 or ABORTOP */
int          cid;
InvokeIDType invokeid;
OP           operation;
IDX          *arg;                    /* must be an encoded IDX list */
```

The *cid* field is used to indicate the association over which the result is to be sent. The *invokeid* field must match that of the incoming request to which this responds. The *operation* field indicates the operation performed for the client. Finally, the *arg* field points to an IDX list containing any result data to be passed back to the client. This field may contain the value NULL if no data is to be passed back with the result. The Result() interface subroutine returns 0 on success or ABORTOP if the association has been lost.

The Error() interface subroutine header is as follows:

```
int Error( cid, invokeid, error, arg ) /* returns 0 or ABORTOP */
int          cid;
InvokeIDType invokeid;
ERR          error;
IDX          *arg;              /* must be an encoded IDX list */
```

The related typedef follows:

```
typedef int     ERR;
```

The *cid* field is used to indicate the association over which the error indication is to be sent. The *invokeid* field must match that of the incoming request to which this responds. The *error* field is an integer which indicates to the client the reason for the error result. Finally, *arg* points to an IDX list containing any error data to be passed back to the client. This field may contain the value NULL if no data is to be passed back with the error indication. The Result() interface subroutine returns 0 on success or ABORTOP if the association has been lost.

The RejectU() interface subroutine header is as follows:

```
int RejectU( cid, invokeid, problemtype, problem ) /* returns 0 or ABORTOP */
int          cid;
InvokeIDType invokeid;
```

```
int          problemtype;
int          problem;
```

As in Result() and Error(), the *cid* and *invokeid* parameters reflect the connection identifier of the association on which to send the reject and the invocation identifier of the Invoke() being rejected. *Problemtype* and *problem* are both integers indicating the cause of the reject. RejectU() may be used to reject a received operation request (Invoke()) or to reject a received Result or Error response to a previously sent request. If the reject is used in response to an operation request, the permissible values for the *problemtype* and *problem* parameters are as follows:

problemtype INVPROB: *problem* can be any of: duplicateInvocation, unrecognisedOperation, mistypedArgument, resourceLimitation, initiatorReleasing, unrecognisedLinkedID, linkedResponseUnexpected or unexpectedChildOperation.

If RejectU() is used to reject a returned result or error, the permissible values for the *problemtype* and *problem* parameters are as follows:

problemtype RRESPROB: *problem* can be any of: unrecognisedInvocation, resultResponseUnexpected or mistypedResult.

problemtype RERRPROB: *problem* can be any of: unrecognisedInvocation, errorResponseUnexpected, unrecognisedError, unexpectedError or mistypedParameter.

Because each Invoke() blocks the calling thread, it may appear that asynchronous invokes are not possible. However, because multiple threads can concurrently issue invokes over the same association, class 2 operations are supported. Linked operations are handled in a similar way. The association initiator designates a thread to act as a server for the initiator side of the association. In this way, even though the initiator is blocked, the designated server is free to respond to a linked operation. The general design philosophy is that the application programmer does not have to deal directly with concurrency. Instead, concurrency is obtained by the creation of multiple threads resulting in a more simple interface.

3.3 Connection Management and Remote Operations Examples

Several examples are provided here to assist the application developer in the task of interfacing to the connection management and remote operations primitives provided by DASCOM. The convention used is that any text between pairs of three consecutive periods represents code which has been omitted for brevity.

This first example shows a client establishing a connection with a server, requesting one operation, then releasing the association. The client also creates a process to handle requests in the direction of the server to the client.

```
#define PSELECTOR   "p-selector goes here"
#define NADDRESS    "Internet=koolaid+556677"
#define ISO         1
#define STANDARD    0
#define BER         8825
#define ASN1        8824
#define CONNECTDATA "hey there - want to connect?"

/* this process designed to handle server requests destined for the client  */
PROCESS smallserver()
{
    int operation;
    int aid, invokeId, arg;

    do
        {
        /* wait for operation request from server */
        operation = Await( &aid, &invokeId, &arg );

        switch( operation )
            {
            case USEROP1:
                ... process the request and make the results ...

                Result( aid, invokeId, USEROP1, 0 );
```

```
                break;

            ...
            ...
            ...
            }
        }
    while( 1 );
}


/* this is the main client process which will bind to the main server        */
PROCESS client()
{
    CONNINFO            *cinfo, *rcinfo;
    PresentationAddress  destPaddr;
    OCTS                *octs, *dataocts;
    LIST                *list;
    T02                 t02;
    OID                 BERoid, ASN1oid;
    IDX                 *dataidx;
    PID                 sspid;
    int                 aid;
    OUTCOME             *outcomep;

    /* create new frame for bind arguments */
    NewFrame();

    cinfo = (CONNINFO *) TempMalloc( sizeof( CONNINFO ) );
    bzero( &cinfo, sizeof( CONNINFO ) );

    /*************************************************************************/
    /* first, set up the called and calling addresses                        */

    destPaddr.pSelector =(OCTS *)Malloc( sizeof(OCTS) + strlen(PSELECTOR) + 1);
    strcpy( destPaddr.pSelector->data, PSELECTOR );
    destPaddr.pSelector->len  = strlen( PSELECTOR );
    destPaddr.pSelector->next = NULL;
```

```
... do same for destPaddr.sSelector and destPaddr.tSelector ...

/* build the network address                                          */
octs = (OCTS *) Malloc( sizeof( OCTS ) + strlen( NADDRESS ) + 1 );
octs->next = NULL;
octs->len  = strlen(NADDRESS) + 1;
bcopy( NADDRESS, octs->data, strlen(NADDRESS) + 1);

/* construct the net. address list - this address being the only element */
list = NULL;
ListAdd( &list, octs );
destPaddr.nAddress = list;

cinfo->calledAddr   = &destPaddr;

... do similar setup for the calling address ...
cinfo->callingAddr  = &srcPaddr;


/*********************************************************************/
/* now make the presentation context definition list                 */

/* set up object identifiers for ASN.1 and BER */
ASN1oid.len = 3;
ASN1oid.oid[0] = ISO;
ASN1oid.oid[1] = STANDARD;
ASN1oid.oid[2] = ASN1;

... do same for BERoid ...

t02.r11 = 1;                    /* presentation context identifier   */
t02.r12 = &ASN1oid;             /* abstract syntax name              */

list = NULL;
ListAdd( &list, &BERoid );
t02.r13  = list;                /* list of transfer syntax names     */

list = NULL;
```

```
ListAdd( &list, &t02 );
cinfo->pcdl = list;              /* list of presentation contexts     */


/*************************************************************************/
/* now make some connect data                                            */
datalen  = strlen( CONNECTDATA );
dataocts = (OCTS *) Malloc( sizeof( OCTS ) + datalen );
dataocts->next = NULL;
dataocts->len  = datalen;
bcopy( DATAOCTS, dataocts->data, datalen );

/* encode this octet string */
dataidx = (IDX *) Encode_octs( 0, 0, 0, dataocts );

/* load bind data into CONNINFO structure (makeACSEuserInfo shown below) */
cinfo->ACSEuserInfo = (UNIV) makeACSEuserInfo( dataidx, &BERoid, NULL );


/*************************************************************************/
/* now make default context name                                         */
dcn.r11    = &ASN1oid;
dcn.r12    = &BERoid;
cinfo->dcn = &dcn;

/*************************************************************************/
/* indicate that we would like to use remote operations over this conn...*/
cinfo->useRose = USEROSE;

/*************************************************************************/
/* and fill in the rest - i.e.. application context name, session        */
/* requirements, called Title and Qualifier, this presentation context,  */
/* session references, etc ...                                           */
cinfo->calledTitle     = NULL;
cinfo->calledQualifier = NULL;
cinfo->calledAPInvId   = NULL;
cinfo->calledAEInvId   = NULL;
... and do the rest ...
```

```
/*************************************************************************/
/* now, create a process to handle requests directed at the initiator,   */
/* called "smallserver"                                                  */
sspid = Create( smallserver, 4000, "smallserver", 0, NORM );


/*************************************************************************/
/* go ahead and Bind(). The address of the returned CONNINFO is returned */
/* in rcinfo. The association id (or error code) is returned in aid.     */
/* Remember that Bind() will send out topmost memory frame to the comms  */
/* process.                                                              */
aid = Bind( cinfo, &rcinfo, sspid );

/* when Bind() returns, a new topmost memory frame will exist containing */
/* the Bind() results.                                                   */


/*************************************************************************/
/* now that the connection has been established (we are assuming that the*/
/* aid returned is a positive integer) we are free to request operations */
/* over this connection.                                                 */

/* first, build an octet string to send as an argument                   */
datalen  = strlen( OPERATIONARG );
dataocts = (OCTS *) Malloc( sizeof( OCTS ) + datalen );
dataocts->next = NULL;
dataocts->len  = datalen;
bcopy( OPERATIONARG, dataocts->data, datalen );

/* encode this octet string */
dataidx = (IDX *) Encode_octs( 0, 0, 0, dataocts );

/* now we have an IDX list to use as the operations arguments. For this  */
/* operation we will assume an operation value of USEROP1 and an         */
/* invocation identifier of 3. Remember that the Invoke() call will take */
/* the top memory frame and pass it to the {\em comms} process. It will, */
/* on return, also provide a new top memory frame containing the results.*/
/* therefore there is no net gain or loss of memory frames.              */
```

```
res = Invoke( aid, 3, USEROP1, dataidx, &outcomep );

/* when Invoke() returns {\em res} will hopefully equal RORESULT and the */
/* result arguments (if any) will be in outcomep->arg.                   */


/*************************************************************************/
/* if we are done with the connection, we can now use Unbind() to release*/
/* it.                                                                   */

/* first, make a CONNINFO structure to pass Unbind parameters and data   */
cinfo = (CONNINFO *) TempMalloc( sizeof( CONNINFO ) );

bzero( cinfo, sizeof( CONNINFO ) );

cinfo->ureq         = DUPLEX;
cinfo->ACSEuserInfo = NULL;

cinfo->thisPC = 1;
cinfo->thisTSN = NULL;
cinfo->ACSEReason   = NULL;

... etc. ...

/* now go ahead and call Unbind(). Unbind() will pass the top memory     */
/* to the {\em comms} process and supply a new one (with returned        */
/* values on return. So, as with Invoke() there is no net gain or loss   */
/* of memory frames for the calling process.                             */

Unbind( aid1, cinfo, &rcinfo );
```

The proposed responder (server) will be notified of the connection request if it is blocked on the Await() subroutine. A typical server would be as follows:

```
/* this is the server process which will respond to the clients requests     */
PROCESS server()
{
    int          aid;
    InvokeIDType invokeId;
    UNIV         arg;       /* universal or any other type */
    CONNINFO     *rcvdCinfo, *cinfo;
    PID          workerpid;


    RegisterServer();

    while(1)
        {
        operation = Await( &aid, &invokeId, &arg );
        /* remember that Await(), on return, will push a new memory frame */

        switch( operation )
            {
            case BINDOP:
                {
                /*************************************************************/
                /* for this example we will create a worker to handle the */
                /* new association. Remember that when Await() returned,  */
                /* there was a new memory frame pushed onto our memory    */
                /* stack containing (in this case) the Bind() arguments.  */
                /* TransferAssociation() will give our top memory frame to*/
                /* the association recipient.                             */

                rcvdCinfo = (CONNINFO *) arg;

                ... get whatever info is desired from rcvdCinfo ...

                NewFrame();    /* to pass to comms via AcceptBind()       */
                cinfo = (CONNINFO *) TempMalloc( sizeof( CONNINFO ) );

                ... fill cinfo with your Bind() result information in ...
                ... a way similar to that done for Bind() above. Be   ...
                ... sure to indicate the use of remote operations to  ...
```

```
            ... correspond with the initiator.                    ...

            /* now - create a worker to deal with the association     */
            workerpid = Create( worker, 8000, "worker", rcinfo, NORM );

            /* accept the connection                                  */
            AcceptBind( aid, &cinfo );

            /* and give it to the newly created worker                */
            if( TransferAssociation( aid, workerpid ) )
                printf("server: can't transfer association\n");
            }
            break;

        default:
            printf("got unknown operation = %d\n", operation);
            FreeTempMem();  /* to get rid of pushed memory frame      */
            break;
        }
    }
```

Note in the above example that this main server always creates workers to handle the new associations. Unless one of these workers is killed or dies with an active connection still in existence, the main server will never get an event other than a Bind indication. It will be the worker that will receive all subsequent events for the new association. Here is an example of what this worker might look like:

```
PROCESS worker( conninfo )
CONNINFO *conninfo;
{
    int           aid;
    OP            operation;
    InvokeIDType  invokeId;
    OUTCOME       *outcomep;
    UNIV          arg;
```

```
aid = conninfo->usercid;

... take whatever other information might be required ...
... from the received Bind() arguments                  ...

/* the transfer of the association gives this process one memory frame, */
/* and it will get another each time Await() returns.                    */
while( 1 )
    {
    operation = Await( &aid, &invokeId, &arg );

    switch( operation )
        {
        case UNBINDOP:
            {
            CONNINFO *cinfo;

            cinfo = (CONNINFO *) TempMalloc( sizeof( CONNINFO ) );

            ... fill cinfo with unbind results ...

            /* AcceptUnbind() passes top memory frame to comms process  */
            AcceptUnbind( aid, &cinfo );

            ... if this is our only connection we may want to Pexit() ...
            }
            break;

        case USEROP1:    /* some positive value */

            ... perform requested operation ...

            /* now send result which passes top memory frame to comms   */
            Result( aid, invokeId, USEROP1, 0 );
            break;

        case USEROP2:
            /* here, as part of service, do Invoke() back to client -   */
```

```
            /* in this example, samllserver will receive the Invoke()   */
            OUTCOME      *outcomep;

            ... do preliminary setup ...

            /* send a request back to the client                        */
            result = Invoke( aid, invokeId, USEROP1, 0, &outcomep );

            ... act on results received from the client ...

            ... send results to clients initial request using Result() ...

            break;

        ...
        ...
        ...

        case ABORTOP:
            Pexit();   /* if this is processes' only connection */
            break;

        default:
            FreeTempMem();
            break;

        }
    }
}
```

The following example subroutine makes a list of external for use in the *ACSEuserInfo* field of the CONNINFO structure:

```
/* this routine makes up the sequence of external required for ACSE userinfo */
/* It will only make a one element sequence, and encodes the user data as     */
/* an ANY (single-ASN1-type) rather than an OCTS (octet aligned) or BITS      */
```

```
/* (arbitrary ).                                                    */
LIST *makeACSEuserInfo( info, oid, pci )
IDX *info;     /* pointer to encoded user info                      */
OID *oid;      /* pointer to encoding rule OID or NULL (ISO 8824 - 32.6 )  */
int *pci;      /* presentation context identifier                   */
{
    LIST *list;    /* defined in the ASN.1 library */
    EXT  *node;    /* defines in the ASN.1 library */

    /* TempMalloc is a memory allocation routine - see Threads documentation */
    node = (EXT *) TempMalloc( sizeof( EXT ) );

    node->dref  = oid;
    node->iref  = pci;
    node->value = NULL;

    node->ed.choice      = single_asn1_tag;
    node->ed.data.single = (UNIV) info;

    list = NULL;

    /* see the ASN.1 documentation for the ListAdd subroutine */
    ListAdd( &list, node );

    return( list );
}
```

The resulting list may be assigned to the CONNINFO structure as follows:

```
cinfo.ACSEuserInfo = (UNIV) makeACSEuserInfo( dataidx, &BERoid, NULL );
```

The receiver of a CONNINFO structure containing *ACSEuserInfo* may decode this information as follows:

```
result = (void *)getACSEuserInfo(cinfo->ACSEuserInfo, &oid, &pci);
/* now decode result as required */
```

The routine getACSEuserInfo is as follows:

```
/* this routine will get the user information form the sequence of external  */
/* used for ACSEuserInfo, and return a pointer to a buffer. The routine will */
/* cope properly whether the single-ASN1-type or the octet-aligned type was  */
/* chosen for encoding - but will only deliver the data in the first element */
/* in the sequence regardless of how many exist.                             */
void *getACSEuserInfo( list, oid, pci )
LIST *list;
OID  **oid;  /* abstract syntax (direct reference) of received information   */
int  **pci;  /* presentation context identifier (indirect reference)         */
{
    EXT  *node;    /* external type - see ASN.1 documentation                 */
    void *retval;

    if( ! list )
        return( (void *) 0 );

    node = (EXT *) ListNext( list );

    if( oid )
        *oid = node->dref;

    if( pci )
        *pci = node->iref;

    switch( node->ed.choice )
        {
        case single_asn1_tag:
            retval = (void *) node->ed.data.single;
            break;

        case octet_aligned_tag:
            retval = (void *) node->ed.data.octet.data;
            break;

        default:
            sprintf(dbuf, "getACSEuserInfo: can't cope with encoding choice %d",
                    node->ed.choice );
            lilerr( dbuf );
            retval = (void *) -1;
```

```
            break;
        }

    return( retval );
}
```

Chapter 4

CASN1 - The ASN.1 Compiler

4.1 Introduction

CASN1 is a compiler for translating ASN.1 type specifications into C language definitions and for generating BER encoding/decoding routines for every defined type. The compiler takes as input an ASN.1 module specification. The compiler generates three files of importance. The first file contains C language type definitions which correspond to the ASN.1 types given in the input specification. The second file contains encode routines. These routines take as a parameter a C structure or variable which represents an instance of an ASN.1 type. From the values in this parameter the encode routines generate the corresponding BER (basic encoding rules) representation. The third file contains decode routines. These routines take, as input, a BER encoding. The BER encoding is decoded and a C structure or variable (of the same type that is generated by the compiler and used as input by the encode routines) is generated and filled with the decoded value(s).

4.2 The *defs.c* File

The most important part of using CASN1 is understanding the C language structures and type definitions generated by it. These defined structures and variables are used to pass values to the encode routines, and to return values from the decode routines. The

ASN.1 type language has some built-in types from which the ASN.1 programmer is able to compose new types. These new types may be created by renaming existing types or by using ASN.1 mechanisms for combining types together into structures such as sets or sequences. Regardless of how complex the user-defined types become, they may always be traced back to ASN.1 built-in types.

4.2.1 CASN1 Built-In Types

The C language structures and types generated by CASN1 follow the same type-composition idea as their ASN.1 counterparts. CASN1 defines a C language type for every built-in ASN.1 type. For example, a BITSTRING ASN.1 type is represented by CASN1 as a C structure containing fields for the bit string length, the values of the bits and a next pointer to form these BITS structures into a list. An INTEGER ASN.1 type is represented by the C type *int*. In order to understand the definitions generated by the CASN1 compiler, one must first be familiar with the CASN1 representations of the ASN.1 built-in types.

These types follow:

```
typedef unsigned char  bool;

typedef struct BITS
    {
    struct BITS *next;
    long        len;
    char        data[1];
    } BITS;

typedef struct OCTS
    {
    struct OCTS *next;    /* pointer to next OCTS structure       */
    long        len;      /* length of the octet string           */
    char        data[1];  /* pointer or value to the octet string */
    } OCTS;

typedef struct OID
```

```
    {
    int     len;     /* number of oid components < 20 */
    int     oid[20]; /* object identifier components  */
    } OID;

#define UTC_Z_TIME      0   /* UTC time with Z                    */
#define UTC_D_TIME      1   /* UTC time with differential         */
#define GNL_Z_TIME      2   /* Generalized time with Z            */
#define GNL_D_TIME      3   /* Generalized time with differential */
#define GNL_L_TIME      4   /* Generalized local time             */
typedef struct TIME
    {
    int     year;     /* year  :  19** or **, (0 .. ?)            */
    int     month;    /* month :  1 .. 12                         */
    int     day;      /* hour  :  1 .. 31                         */
    int     hour;     /* day   :  0 .. 23                         */
    int     minute;   /* minute:  0 .. 59                         */
    float   second;   /* second:  0 .. 59                         */
    float   diff;     /* difference between local time and UTCtime */
    int     zone;     /* flag for time type: (0, 1, 2, 3, 4)      */
    } TIME;


#define single_asn1_tag   0xa0000000
#define octet_aligned_tag 0x80000001
#define arbitrary_tag     0x80000002

typedef struct CODING
    {
    int     choice;
    union
        {
        ANYtype single;         /* single ASN.1 type */
        OCTS    octet;          /* octet aligned     */
        BITS    arbitrary;      /* arbitrary         */
        } data;
    }CODING;

typedef struct EXT
```

```
    {
    OID     *dref;   /* direct reference (optional)      */
    int     *iref;   /* indirect reference (optional)    */
    OCTS    *value;  /* data value descripter (optional) */
    CODING  ed;      /* Encoding policy choice            */
    } EXT;


#define Set LIST*
#define Seq LIST*
#define OF(type)
```

The BITS Structure

The BITS structure represents the ASN.1 BIT STRING type. This structure contains three fields. The first field, *next*, is a pointer to a BITS structure and allows the BITS structure to be formed into a list. The second field, *len*, indicates the length of the bitstring in units of bits. The final field, *data*, contains the actual bitstring. If a bit string equal to *110110001* is to be represented, the BITS structure could be created in the following manner:

```
BITS *bitsrep;

/* the data field will take two octets (9 bits) and therefore one extra */
/* octet must be allocated at the end of the BITS structure. Also, see  */
/* the Threads documentation for information regarding TempMalloc().    */

bitsrep = (BITS *) TempMalloc( sizeof(BITS) + 1 );

bitsrep->next    = NULL;
bitsrep->len     = 9;     /* 9 bits in this bitstring                */
bitsrep->data[0] = 0xd8;  /* hex value of first 8 bits in bitstring */
bitsrep->data[1] = 0x01;  /* hex value of last bit in bitstring     */
```

A number of operations on the BITS structure have been implemented for the convenience of the application programmer. The headers for these routines are as follows:

```
BITS* BITSInit(str, length)
char *str;
int length;

int BITSLen(bits, length, unb)
BITS *bits;
int *len;
int *unb;

char* BITSToChar(bits, b)
BITS *bits;
byte **b;

BITS *BITSDup(str)
BITS *str;

int BITSSet( str, whichBit )
BITS* str;
int whichBit;

bool BITSClr( str, whichBit )
BITS* str;
int whichBit;

bool BITSTest( str, whichBit )
BITS* str;
int whichBit;

int  BITSEqual( str1, str2 )
BITS* str1;
BITS* str2;
```

BITSInit() creates a BITS data structure given a character string (representing a sequence of bits) and a length specifying the number of bits in the string. This structure is allocated from the top memory frame of the calling process.

BITSLen() returns the number of bits in a BITS structure and the number of unused bits in the last octet. These values are returned by way of the *length* and *unb* parameters. On call, these parameters must point to sufficient memory to each hold an integer. The length is also returned as the return value of the subroutine.

BITSToChar() serializes a BITS structure into a character string.

BITSDup() allocates (from the top memory frame of the calling process) and returns a copy of the given BITS structure.

BITSSet() sets the specified bit in the BITS structure and returns a non-zero value on success.

BITSClr() clears the specified bit in the BITS structure and returns a non-zero value on success.

BITSTest() returns the value of the specified bit.

BITSEqual() compares two BITS structures and returns a non-zero value if equal.

The OCTS Structure

The OCTS structure represents the ASN.1 OCTET STRING type. This structure contains three fields. The first field, *next*, is a pointer to an OCTS structure and allows the OCTS structure to be formed into a list. The second field, *len*, indicates the length of the string in units of octets. The final field, *data*, contains the actual string.

A number of operations on this OCTS structure have been provided for convenience. The headers for these routines are as follows:

```
OCTS* OCTSInit(str, length)
char *str
```

```
int length

OCTS* OCTSBld(str)
char *str

int OCTSLen(octslist)
OCTS *octslist

char* OCTSToChar(str)
OCTS *str

OCTS* OCTSDup(str)
OCTS *str

OCTS* OCTSAppend(str1, str2)
OCTS *str1
OCTS *str2
```

OCTSInit() creates and returns an OCTS data structure. The data in the structure is that given by the argument *str* of length *length*. The memory for the new structure is allocated from the top memory frame of the calling process.

OCTSBld() performs the same function as OCTSInit() except that OCTSBld() takes as a parameter a null-terminated string to use as the OCTS data. Neither the length of the data nor the data itself in the OCTS structure include the null-terminator.

OCTSLen() counts and returns the total length of data in a list of OCTS structures.

OCTSToChar() returns the data contained in a list of OCTS structures. The data is returned in one contiguous buffer. A null-terminator is added to this buffer which is not part of the original OCTS data. The memory for the returned buffer is allocated from the top memory frame of the calling process.

OCTSDup() produces and returns a duplicate of the OCTS structure provided as the argument. If the original consists of a list of OCTS structures, the routine will return one OCTS structure containing the data from the list of structures concatenated together.

OCTSAppend() concatenates the OCTS list pointed to by *str2* onto the end of the list pointed to by *str1*. The new (combined) list is returned.

The OID Structure

The OID structure represents the ASN.1 OBJECT IDENTIFIER type. This structure contains two fields. The first field, *len*, contains the number of components of the object identifier being represented. The second field, *oid*, is an array of integers. This array contains the actual components of the identifier. Note that an object identifier represented using the OID structure may have a maximum of only 20 components.

The TIME Structure

The TIME structure is used to represent GeneralizedTime and UTCTTime values. This structure has eight fields. Field *year* is an integer representing the calendar year. The year may be given as 19xx or xx, where xx is in the range 0 to 99. *Month* is an integer field with possible values being 1 to 12. *Day*, *hour*, *minute* and *second* are all integer fields with possible values being 1 to 31, 0 to 23, 0 to 59 and 0 to 59 respectively. *Diff* is a float field representing the local time differential and must take a value in the range -59.99 to +59.99. Finally, *zone* is an integer variable representing the time type and must take one of the values *UTC_Z_TIME*, *UTC_D_TIME*, *GNL_Z_TIME*, *GNL_D_TIME* or *GNL_L_TIME*.

The EXT Structure

The ASN.1 EXTERNAL type is represented by the CASN1 EXT structure. This structure is made up of four fields. The *dref* field is a pointer to an object identifier (described above) representing the direct reference of the external type. The indirect reference is represented by the *iref* integer pointer field. The object descriptor portion of the ASN.1 external type is represented by the *value* field. This filed contains human-readable text and is a pointer to the OCTS type. All of these fields are optional and if not included are assigned the

value NULL. The fourth field, *ed*, contains the external data. This data is represented by the CODING CASN1 structure and may consist of any one of three types. The *choice* field of the CODING structure indicates the type of encoding for the external data. The choices are *single_asn1_tag*, *octet_aligned_tag* and *arbitrary_tag*. The choice field corresponds to the value of the *data* union in the CODING structure. If the external data is a single ASN.1 type then the *choice* field takes the value *single_asn1_tag* and the *single* field of the *data* union should point to an IDX list. If the external data is octet aligned then the *choice* field takes the value *octet_aligned_tag* and the *octet* field of the *data* union contains the data in the form of an OCTS structure. Finally, if the external data is neither an ASN.1 type nor octet aligned then the *choice* field takes the value *arbitrary_tag* and the *arbitrary* field of the *data* union contains the data in the form of a BITS structure.

The LIST Structure

There are four ways in the ASN.1 language to combine more than one type or instance into a new type. To create a new type which is an ordered collection of various other types, the SEQUENCE ASN.1 type is used. This is equivalent to a Pascal record or C structure. The ASN.1 SEQUENCE OF type generates a new type from an ordered collection of instances of one type. This is equivalent to a Pascal or C array. The ASN.1 SET type defines a new type from an unordered collection of various other types. Finally, the ASN.1 SET OF type creates a new type from an unordered collection of instances of one type.

CASN1 represents ASN.1 SEQUENCE and SET types using C structures. An ASN.1 SEQUENCE or SET which is composed of various other ASN.1 types will be translated into a C structure containing fields for each of the constituent types. For example, if one of the fields in a SEQUENCE is an OCTET STRING then there will be a field in the generated C structure of type OCTS or pointer to OCTS. In general if the OCTET STRING was optional the generated field will be a pointer to an OCTS structure (thus allowing its exclusion). Otherwise, the field will be of type OCTS.

CASN1 represents SET OF and SEQUENCE OF types using a list data structure.

The list nodes are defined by CASN1 (as type LIST) and various utility routines have been provided for the creation and manipulation of these lists. The definitions for the list structures follow:

```
typedef struct LIST_ITEM
    {
    UNIV             *item;
    struct LIST_ITEM *next;
    } LIST_ITEM;

typedef struct LIST
    {
    uint_32     count;
    LIST_ITEM   *top;
    LIST_ITEM   *next;
    } LIST;
```

Where the *count* field records the list length (number of list items), the *top* field points to the first node of the list and the *next* field points to the current node of the list. In structure LIST_ITEM, the *next* field is pointer to the next list item and the *item* field points to the item value.

The headers for the list utility routines are as follows:

```
LIST_ITEM *ListItemCreate()

LIST *ListCreate()

int  ListCount(list)
LIST *list;

UNIV *ListFirst(list)
LIST *list;

UNIV *ListNext(list)
```

```
LIST *list;

LIST *ListAdd(list, item)
LIST **list;
UNIV item;

LIST *ListAppend(list, item)
LIST **list;
UNIV item;

bool ListEOL(list)
LIST *list;

void ListRemove(list)
LIST *list;

bool ListEmpty(list)
LIST *list;

LIST *ListMerge(dst, list)
LIST *dst;
LIST *list;

LIST *ListCopy(list, copyItem)
LIST *list;
void *(*copyItem)();

void ListFor(list)
LIST *list;
```

ListItemCreate() allocates memory for a LIST_ITEM from the top memory frame of the calling process and initializes the *item* and *next* fields to NULL. A pointer to the created and initialized structure is returned.

ListCreate() allocates memory for a LIST structure from the top memory frame of the calling process and initializes the *top* and *next* fields to NULL. The *count* field is initialized to 0. A pointer to the created and initialized structure is returned.

ListCount() takes a pointer to a LIST as its argument. This routine counts the number of items in the list, sets the appropriate field in the LIST structure, and returns the number computed.

ListFirst() takes a pointer to a LIST as its argument. This routine returns the first item in the list (not the first LIST_ITEM structure, but the value in the *item* field of the first LIST_ITEM structure). Before returning, the *next* item in the list is set to be the second one.

ListNext() takes a pointer to a LIST as its argument. This routine returns the item in the LIST_ITEM currently pointed to by the LIST's *next* field. The *next* field is advanced one node.

ListAdd() requires two parameters, the address of a pointer to a list, and an item to be added to the list. If the list parameter points to a location containing the value NULL, a new LIST is created. In any case, a new LIST_ITEM is created and initialized so that the *item* field contains the value passed as the parameter item. This LIST_ITEM is then added at the front of the list. The *count* field of the list is incremented and the *next* field is set to point to the first LIST_ITEM.

ListAppend() performs the same function as ListAdd() except that instead of adding the new item to the front of the LIST, it is instead appended at the end of the LIST. The *next* field of the LIST is set to point to the item just appended.

ListEOL() takes a pointer to a LIST as its argument. This routine returns TRUE if the *next* field points to a LIST_ITEM (actually, if it has any value other than NULL) and FALSE otherwise.

ListRemove() takes a pointer to a LIST as its argument. This routine removes the LIST_ITEM from the LIST which is currently pointed to by the list's *next* field.

ListEmpty() takes a pointer to a LIST as its argument. This routine returns TRUE if the argument is NULL or if the *top* field of the supplied LIST is NULL. Otherwise, FALSE is returned.

ListMerge() takes two LIST pointers as its arguments. This routine appends the second LIST to the first LIST. The *count* field of the first list is updated to reflect the new elements in the list. While the second LIST is not altered, its elements are now part of both lists and changes made to one LIST may cause undesirable effects on the other.

ListCopy() requires two parameters. The first parameter is a pointer to a LIST, and the second parameter is a pointer to a subroutine. This routine creates a duplicate of the supplied LIST. The new LIST is allocated from the top memory frame of the calling process. The individual items are duplicated by this routine using the subroutine supplied as the second parameter. It is assumed by ListCopy() that the supplied subroutine takes an item as its single parameter and returns a duplicate item. The type of the parameter and result for the supplied subroutine is the same type as is held in the *item* field of LIST_ITEM nodes for the LIST supplied.

ListFor() takes a pointer to a LIST as its argument. This macro can be used in place of a for-loop construct to step through the items in the supplied LIST. The current item is referenced in the body of the for-loop by using the *next* field of the LIST structure.

This information should be helpful in understanding the C definitions generated by the CASN1 compiler. It is necessary that the CASN1 user understand these definitions as they are used to convey data to the encode routines, and to pass back results from the decode routines.

4.3 The *encode.c* File

The *.encode.c file generated by CASN1 contains subroutines to encode each of the ASN.1 type definitions included in the compiler specification. Each of these routines requires four parameters and returns a linked list of IDX nodes.

The first three parameters to the encode routines are the tag class, form and code for the type being encoded. These parameters are used by some encode routines, and ignored by others. The encode routines may be examined to determine whether these parameters

are used. The final parameter is an instance of a type defined in *.defs.c (generated by CASN1). This is the parameter which contains the data to be encoded. The type of this parameter corresponds to the name of the encode routine. For example, the routine Encode_TYPEX() requires as its third argument either a variable of type TYPEX or a pointer to a variable of type TYPEX. As indicated previously, this structure or simple variable must be filled with the values to be encoded. It is important that all fields and sub fields are correctly initialized. If a field is optional, its value should either be NULL or a valid pointer.

The returned value is a pointer to the first node in a linked list of IDX nodes. These nodes contain the encoding generated by the encoding routine. This list is used as a parameter to the remote operations data transfer routines.

If it is necessary to serialize the data contained within the IDX list into a single contiguous buffer, SerIDX() may be called. The header for this routine is as follows:

```
byte *serIDX(P, B, LEN)
IDX  *P;
byte **B;
int  *LEN;
```

This routine takes a pointer to an IDX list as the first parameter. Parameters *B* and *LEN* are pointers to memory large enough for a byte pointer and integer respectively. SerIDX() serializes the IDX list into a single contiguous buffer, returning the location of the buffer in the space pointed to by parameter *B*, and the length of the buffer in the location pointed to by parameter *LEN*. A pointer to the serialized buffer is also returned.

4.4 The *decode.c* File

The *.decode.c file generated by CASN1 contains subroutines to decode ASN.1 encodings (encoded using the basic encoding rules). Each of these routines requires five parameters

and returns a structure corresponding to the ASN.1 type being decoded.

The first three parameters to the decode routines are the class, form and code for the initial tag being decoded. These parameters are only used by some of the generated decode routines. The routines themselves may be examined to determine when these parameters are meaningful. The fourth parameter is always necessary. This parameter contains the address of a pointer variable. The pointer variable whose address is passed must contain the address of the first byte to be decoded by the routine. On return from the decode routine, the pointer variable points immediately past the decoded portion of the buffer. The final parameter is a pointer to a structure allocated to hold the decode results. This parameter is optional. If a NULL value is passed as this parameter the decode routine will allocate storage for the result structure from the top memory frame of the calling process. The type of this parameter is the C type generated by CASN1 which corresponds to the type of data being decoded. The name of the decode routine reflects the type of this parameter. For example, the routine Decode_CTYPE() will require either a variable of type CTYPE or a pointer to a variable of type CTYPE. This routine returns a pointer to a variable of type CTYPE which is filled with the decoded values.

4.5 Examples

The following is a simple example of the use of CASN1. The ASN.1 source for this example consists of the following listing:

```
Sample DEFINITIONS::=
BEGIN

-- a person may be a male or female

PERSON      ::= CHOICE {
                    female [1] FemaleInfo,
                    male   [2] MaleInfo
                    }
```

```
FemaleInfo ::= SEQUENCE
                    {
                    maidenName   NameType OPTIONAL,
                    husbandsName NameType OPTIONAL,
                    genericInfo  GenericInfoType
                    }

MaleInfo   ::= SEQUENCE
                    {
                    wifesName    NameType OPTIONAL,
                    genericInfo  GenericInfoType
                    }

GenericInfoType ::= SEQUENCE
                    {
                    address    AddressType,
                    age        INTEGER,
                    name       NameType,
                    married    BOOLEAN,
                    vehicles   VehicleSet     OPTIONAL,
                    occupation OccupationType OPTIONAL
                    }

VehicleSet     ::= SET OF VehicleType

VehicleType    ::= INTEGER
                    {
                    volvo         (0),
                    generalMotors (1),
                    ford          (2),
                    chrysler      (3),
                    nissan        (4),
                    toyota        (5),
                    honda         (6),
                    bmw           (7),
                    mercedes      (8),
                    other         (9)
                    }
```

```
NameType        ::= OCTET STRING

AddressType     ::= OCTET STRING

OccupationType ::= OCTET STRING

END
```

This ASN.1 source describes a person. The person may be male or female. The information for males and females contain some information specific to their type (maidenName and husbandsName or wifesName), as well as some information applicable to both males and females (genericInfo). The name of the file containing this ASN.1 source is *sample*.

This source is compiled using the ASN.1 compiler using the following command:

```
CASN1 sample
```

The result of this compilation is the creation of several files. These files are all placed in the directory ./out. If no such directory exists, one is created. If the directory does exist, its contents are removed before compilation begins. All of the files of importance created by this run of CASN1 begin with the module name of the ASN.1 module (Sample). The files produced are Sample.defs.c, Sample.encode.c and Sample.decode.c.

The file containing C definitions corresponding to the ASN.1 types in the specification is Sample.defs.c. The listing for this file is as follows:

```
#include "/casn1/defs.h"
#include "/casn1/element.h"

extern bool     ASN1_ERROR_FLAG;
/* a person may be a male or female */
typedef OCTS    *OccupationType;        /* OCTET STRING */
IDX             *Encode_OccupationType();
OccupationType  Decode_OccupationType();
```

```
typedef OCTS    *AddressType;        /* OCTET STRING */
IDX             *Encode_AddressType();
AddressType      Decode_AddressType();

typedef OCTS    *NameType;        /* OCTET STRING */
IDX             *Encode_NameType();
NameType         Decode_NameType();

typedef int      VehicleType;        /* INTEGER */
#define volvo 0
#define generalMotors 1
#define ford 2
#define chrysler 3
#define nissan 4
#define toyota 5
#define honda 6
#define bmw 7
#define mercedes 8
#define other 9
IDX             *Encode_VehicleType();
VehicleType      Decode_VehicleType();

typedef Set
Of(VehicleType) VehicleSet;        /* SET OF */
        IDX             *Encode_VehicleSet();
VehicleSet       Decode_VehicleSet();

typedef struct GenericInfoType {/* SET/SEQ */
        AddressType     address;/* AddressType */
        int             age;        /* INTEGER */
        NameType        name;        /* NameType */
        bool            married;/* BOOLEAN */
        VehicleSet      vehicles;        /* VehicleSet */
        OccupationType  occupation;        /* OccupationType */
}               GenericInfoType;
IDX             *Encode_GenericInfoType();
GenericInfoType *Decode_GenericInfoType();
```

```
typedef struct MaleInfo {        /* SET/SEQ */
        NameType        wifesName;        /* NameType */
        GenericInfoType *genericInfo;        /* GenericInfoType(constructor) */
}               MaleInfo;
IDX             *Encode_MaleInfo();
MaleInfo        *Decode_MaleInfo();

typedef struct FemaleInfo {        /* SET/SEQ */
        NameType        maidenName;        /* NameType */
        NameType        husbandsName;        /* NameType */
        GenericInfoType *genericInfo;        /* GenericInfoType(constructor) */
}               FemaleInfo;
IDX             *Encode_FemaleInfo();
FemaleInfo      *Decode_FemaleInfo();

typedef struct PERSON {                /* CHOICE */
        int             choice;        /* indicate the choice of data */
        union {                        /* choices */
                FemaleInfo      *female;        /* CONTEXT 1 FemaleInfo(constructor) */
#define PERSON_female_tag 0xa0000001
                MaleInfo        *male;        /* CONTEXT 2 MaleInfo(constructor) */
#define PERSON_male_tag 0xa0000002
        }               data;
}               PERSON;
IDX             *Encode_PERSON();
PERSON          *Decode_PERSON();
```

Notice that every ASN.1 type created in the specification has a corresponding type in the file Sample.defs.c. Note also that the definitions in this file are built upon the basic (built-in) types described earlier in this document.

As an example, take the ASN.1 type (in module *Sample*) *FemaleInfo*. The corresponding C structure has a field for each of the fields defined in the ASN.1 specification. The last field, *genericInfo* refers to another constructed type, as it does in the ASN.1 specification. Note also that the field *vehicles* of structure *GenericInfoType* is defined as a "Set Of VehicleSet". Remember that as described above, *Set* is defined as LIST *, and OF(X) is defined to nothing. This means that the definition in Sample.defs.c "typedef Set Of(VehicleType)

VehicleSet" means that the variable *VehicleSet* is simply a pointer to a LIST.

For every type defined in the file Sample.defs.c, both encode and decode routines are created. Generally, however, it is only the top-level encode and decode routines which are called by the application. For this example, an application would only call the routine to encode an instance of the structure *PERSON*, and never explicitly call the routine to encode, for example, a *VehicleSet*. The top-level routines to encode and decode the *PERSON* structure will call the lower level routines as part of their function.

The listings for the files Sample.encode.c and Sample.decode.c are as follows with the encode file first:

```
/* file Sample.encode.c */

#include "Sample.defs.c"
/* a person may be a male or female */

IDX             *Encode_OccupationType (class, form, code, OccupationType_VP)
short            class;
short            form;
long             code;
OccupationType   OccupationType_VP;
{
        IDX             *p;
        p = (IDX *) Encode_octs (class, form, code, OccupationType_VP);

        return (p);
}

IDX             *Encode_AddressType (class, form, code, AddressType_VP)
short            class;
short            form;
long             code;
AddressType      AddressType_VP;
{
        IDX *p;
        p = (IDX *) Encode_octs (class, form, code, AddressType_VP);
```

```
        return (p);
}

IDX             *Encode_NameType (class, form, code, NameType_VP)
short            class;
short            form;
long             code;
NameType         NameType_VP;
{
        IDX *p;
        p = (IDX *) Encode_octs (class, form, code, NameType_VP);

        return (p);
}

IDX             *Encode_VehicleType (class, form, code, VehicleType_VP)
short            class;
short            form;
long             code;
VehicleType      VehicleType_VP;
{
        IDX *p;
        p = (IDX *) Encode_int (class, form, code, VehicleType_VP);

        return (p);
}

IDX             *Encode_VehicleSet (class, form, code, VehicleSet_VP)
short            class;
short            form;
long             code;
VehicleSet       VehicleSet_VP;
{
        IDX *p;
        p = (IDX *) Encode_struct_beg (class, form, code);
        ListFor (VehicleSet_VP)
                Encode_VehicleType (UNIVERSAL, PRIMITIVE, 2,
                    (VehicleSet_VP)->next->item);
```

```
        Encode_struct_end (DEFINITE);
        return (p);
}

IDX             *Encode_GenericInfoType (class, form, code, GenericInfoType_VP)
short            class;
short            form;
long             code;
GenericInfoType *GenericInfoType_VP;
{
        IDX *p;
        p = (IDX *) Encode_struct_beg (class, form, code);
        Encode_AddressType (UNIVERSAL, PRIMITIVE, 4,
            GenericInfoType_VP->address);
        Encode_int (UNIVERSAL, PRIMITIVE, CODE_INTEGER,
            GenericInfoType_VP->age);
        Encode_NameType (UNIVERSAL, PRIMITIVE, 4,
            GenericInfoType_VP->name);
        Encode_bool (UNIVERSAL, PRIMITIVE, CODE_BOOLEAN,
            GenericInfoType_VP->married);
        if (GenericInfoType_VP->vehicles != NULL)
        {
                Encode_VehicleSet (UNIVERSAL, CONSTRUCTOR, 17,
                    GenericInfoType_VP->vehicles);
        }
        if (GenericInfoType_VP->occupation != NULL)
        {
                Encode_OccupationType (UNIVERSAL, PRIMITIVE, 4,
                    GenericInfoType_VP->occupation);
        }
        Encode_struct_end (DEFINITE);
        return (p);
}

IDX             *Encode_MaleInfo (class, form, code, MaleInfo_VP)
short            class;
short            form;
long             code;
MaleInfo        *MaleInfo_VP;
```

```
{
        IDX *p;
        p = (IDX *) Encode_struct_beg (class, form, code);
        if (MaleInfo_VP->wifesName != NULL)
        {
                Encode_NameType (UNIVERSAL, PRIMITIVE, 4,
                    MaleInfo_VP->wifesName);
        }
        Encode_GenericInfoType (UNIVERSAL, CONSTRUCTOR, 16,
            MaleInfo_VP->genericInfo);
        Encode_struct_end (DEFINITE);
        return (p);
}

IDX             *Encode_FemaleInfo (class, form, code, FemaleInfo_VP)
short            class;
short            form;
long             code;
FemaleInfo      *FemaleInfo_VP;
{
        IDX             *p;
        p = (IDX *) Encode_struct_beg (class, form, code);
        if (FemaleInfo_VP->maidenName != NULL)
        {
                Encode_NameType (UNIVERSAL, PRIMITIVE, 4,
                    FemaleInfo_VP->maidenName);
        }
        if (FemaleInfo_VP->husbandsName != NULL)
        {
                Encode_NameType (UNIVERSAL, PRIMITIVE, 4,
                    FemaleInfo_VP->husbandsName);
        }
        Encode_GenericInfoType (UNIVERSAL, CONSTRUCTOR, 16,
            FemaleInfo_VP->genericInfo);
        Encode_struct_end (DEFINITE);
        return (p);
}

IDX             *Encode_PERSON (class, form, code, PERSON_VP)
```

```
short           class;
short           form;
long            code;
PERSON         *PERSON_VP;
{
        IDX *p;
        switch (PERSON_VP->choice)
        {
        case PERSON_female_tag:
                p = (IDX *) Encode_struct_beg (CONTEXT, CONSTRUCTOR, 1);
                Encode_FemaleInfo (UNIVERSAL, CONSTRUCTOR, 16,
                    PERSON_VP->data.female);
                Encode_struct_end (DEFINITE);
                break;
        case PERSON_male_tag:
                p = (IDX *) Encode_struct_beg (CONTEXT, CONSTRUCTOR, 2);
                Encode_MaleInfo (UNIVERSAL, CONSTRUCTOR, 16,
                    PERSON_VP->data.male);
                Encode_struct_end (DEFINITE);
                break;
        default:
                error ("Encode_PERSON : No such choice\n");
        };
        return (p);
}
/* file Sample.decode.c */

#include "Sample.defs.c"
/* a person may be a male or female */

OccupationType  Decode_OccupationType (class, form, code, B,
            OccupationType_VP)
short           class;
short           form;
long            code;
byte          **B;
OccupationType  OccupationType_VP;
{
```

```
        if (ASN1_ERROR_FLAG == TRUE)
                return (FALSE);
        OccupationType_VP = (OCTS *) Decode_octs (class, form, code,
            B, OccupationType_VP);

        return (OccupationType_VP);
}

AddressType     Decode_AddressType (class, form, code, B, AddressType_VP)
short           class;
short           form;
long            code;
byte          **B;
AddressType     AddressType_VP;
{

        if (ASN1_ERROR_FLAG == TRUE)
                return (FALSE);
        AddressType_VP = (OCTS *) Decode_octs (class, form, code,
            B, AddressType_VP);

        return (AddressType_VP);
}

NameType        Decode_NameType (class, form, code, B, NameType_VP)
short           class;
short           form;
long            code;
byte          **B;
NameType        NameType_VP;
{

        if (ASN1_ERROR_FLAG == TRUE)
                return (FALSE);
        NameType_VP = (OCTS *) Decode_octs (class, form, code, B,
            NameType_VP);

        return (NameType_VP);
}
```

```
VehicleType     Decode_VehicleType (class, form, code, B, VehicleType_VP)
short           class;
short           form;
long            code;
byte          **B;
VehicleType     VehicleType_VP;
{

        if (ASN1_ERROR_FLAG == TRUE)
                return (FALSE);
        VehicleType_VP = (int) Decode_int (class, form, code, B,
            VehicleType_VP);

        return (VehicleType_VP);
}

VehicleSet      Decode_VehicleSet (class, form, code, B, VehicleSet_VP)
short           class;
short           form;
long            code;
byte          **B;
VehicleSet      VehicleSet_VP;
{

        if (ASN1_ERROR_FLAG == TRUE)
                return (FALSE);
        Decode_struct_beg (class, form, code, B);
        VehicleSet_VP = ListCreate ();
        while (ListEnd (*B))
                ListAppend (&(VehicleSet_VP), (UNIV)
                    Decode_VehicleType (UNIVERSAL, PRIMITIVE, 2, B, 0));
        Decode_struct_end (B);
        return (VehicleSet_VP);
}

GenericInfoType *Decode_GenericInfoType (class, form, code, B,
        GenericInfoType_VP)
short           class;
```

```
short           form;
long            code;
byte          **B;
GenericInfoType *GenericInfoType_VP;
{
        if (ASN1_ERROR_FLAG == TRUE)
                return (FALSE);
        if (GenericInfoType_VP == NULL)
                GenericInfoType_VP = (GenericInfoType *)
                    GETBUF (sizeof (GenericInfoType));
        Decode_struct_beg (class, form, code, B);
        GenericInfoType_VP->address = (AddressType)
            Decode_AddressType (UNIVERSAL, PRIMITIVE, 4, B,
            GenericInfoType_VP->address);
        GenericInfoType_VP->age = (int)
            Decode_int (UNIVERSAL, PRIMITIVE, CODE_INTEGER, B,
            GenericInfoType_VP->age);
        GenericInfoType_VP->name = (NameType)
            Decode_NameType (UNIVERSAL, PRIMITIVE, 4, B,
            GenericInfoType_VP->name);
        GenericInfoType_VP->married = (bool)
            Decode_bool (UNIVERSAL, PRIMITIVE, CODE_BOOLEAN, B,
            GenericInfoType_VP->married);
        if (TestTag (UNIVERSAL, CONSTRUCTOR, 17, *B) != FALSE)
        {
                GenericInfoType_VP->vehicles = (VehicleSet)
                    Decode_VehicleSet (UNIVERSAL, CONSTRUCTOR, 17, B,
                    GenericInfoType_VP->vehicles);
        }
        if (TestTag (UNIVERSAL, PRIMITIVE, 4, *B) != FALSE)
        {
                GenericInfoType_VP->occupation = (OccupationType)
                    Decode_OccupationType (UNIVERSAL, PRIMITIVE, 4, B,
                    GenericInfoType_VP->occupation);
        }
        Decode_struct_end (B);
        return (Ge.     InfoType_VP);
}
```

```
MaleInfo        *Decode_MaleInfo (class, form, code, B, MaleInfo_VP)
short            class;
short            form;
long             code;
byte           **B;
MaleInfo         *MaleInfo_VP;
{
        if (ASN1_ERROR_FLAG == TRUE)
                return (FALSE);
        if (MaleInfo_VP == NULL)
                MaleInfo_VP = (MaleInfo *) GETBUF (sizeof (MaleInfo));
        Decode_struct_beg (class, form, code, B);
        if (TestTag (UNIVERSAL, PRIMITIVE, 4, *B) != FALSE)
        {
                MaleInfo_VP->wifesName = (NameType)
                    Decode_NameType (UNIVERSAL, PRIMITIVE, 4, B,
                    MaleInfo_VP->wifesName);
        }
        MaleInfo_VP->genericInfo = (GenericInfoType *)
            Decode_GenericInfoType (UNIVERSAL, CONSTRUCTOR, 16, B,
            MaleInfo_VP->genericInfo);
        Decode_struct_end (B);
        return (MaleInfo_VP);
}

FemaleInfo      *Decode_FemaleInfo (class, form, code, B, FemaleInfo_VP)
short            class;
short            form;
long             code;
byte           **B;
FemaleInfo       *FemaleInfo_VP;
{
        if (ASN1_ERROR_FLAG == TRUE)
                return (FALSE);
        if (FemaleInfo_VP == NULL)
                FemaleInfo_VP = (FemaleInfo *) GETBUF (sizeof (FemaleInfo));
        Decode_struct_beg (class, form, code, B);
        if (TestTag (UNIVERSAL, PRIMITIVE, 4, *B) != FALSE)
        {
```

```
                FemaleInfo_VP->maidenName = (NameType)
                    Decode_NameType (UNIVERSAL, PRIMITIVE, 4, B,
                    FemaleInfo_VP->maidenName);
        }
        if (TestTag (UNIVERSAL, PRIMITIVE, 4, *B) != FALSE)
        {
                FemaleInfo_VP->husbandsName = (NameType)
                    Decode_NameType (UNIVERSAL, PRIMITIVE, 4, B,
                    FemaleInfo_VP->husbandsName);
        }
        FemaleInfo_VP->genericInfo = (GenericInfoType *)
            Decode_GenericInfoType (UNIVERSAL, CONSTRUCTOR, 16, B,
            FemaleInfo_VP->genericInfo);
        Decode_struct_end (B);
        return (FemaleInfo_VP);
}

PERSON          *Decode_PERSON (class, form, code, B, PERSON_VP)
short            class;
short            form;
long             code;
byte           **B;
PERSON           *PERSON_VP;
{
        if (ASN1_ERROR_FLAG == TRUE)
                return (FALSE);
        if (PERSON_VP == NULL)
                PERSON_VP = (PERSON *) GETBUF (sizeof (PERSON));
        PERSON_VP->choice = GetTag (*B);
        switch (PERSON_VP->choice)
        {
        case PERSON_female_tag:
                Decode_struct_beg (CONTEXT, CONSTRUCTOR, 1, B);
                PERSON_VP->data.female = (FemaleInfo *)
                    Decode_FemaleInfo (UNIVERSAL, CONSTRUCTOR, 16, B,
                    PERSON_VP->data.female);
                Decode_struct_end (B);
                break;
        case PERSON_male_tag:
```

```
                Decode_struct_beg (CONTEXT, CONSTRUCTOR, 2, B);
                PERSON_VP->data.male = (MaleInfo *)
                    Decode_MaleInfo (UNIVERSAL, CONSTRUCTOR, 16, B,
                    PERSON_VP->data.male);
                Decode_struct_end (B);
                break;
        default:
                (int) (*B) += (*(LENS - 1));
                error ("Decode_PERSON : No such choice\n");
        };
        return (PERSON_VP);
}
```

An examination of these three included files will show each of the C types defined for their corresponding ASN.1 types, and an encode and decode routine for each. An example of code using these types and routines follows:

```
#include "standards.h"
#include "os.h"
#include "Sample.defs.c"

bool     ASN1_ERROR_FLAG;

PERSON *makePerson()
{
    PERSON          *tempPerson;
    FemaleInfo      *female;
    GenericInfoType *otherInfo;
    LIST            *tempList;

    /* allocate storage (on top memory frame) for PERSON structure */
    tempPerson = (PERSON *) TempMalloc( sizeof( PERSON ) );

    /* make this person a female */
    tempPerson->choice = PERSON_female_tag; /* defined in Sample.defs.c */

    /* allocate storage (on top memory frame) for FemaleInfo structure */
```

```
    female = (FemaleInfo *) TempMalloc( sizeof( FemaleInfo ) );
    tempPerson->data.female = female;

    /* make her maiden name and husbands name */
    female->maidenName   = OCTSBld( "smith" );
    female->husbandsName = OCTSBld( "Jones" );

    /* now make the generic information */
    otherInfo = (GenericInfoType *) TempMalloc( sizeof(GenericInfoType) );
    female->genericInfo = otherInfo;

    otherInfo->address    = OCTSBld( "2170 Wilshire Way" );
    otherInfo->age        = 27;
    otherInfo->name       = OCTSBld( "Mary Jones" );
    otherInfo->married    = TRUE;

    /* now make a list of vehicle types owned... */
    tempList = NULL;    /* initialize to empty */
    ListAppend( &tempList, bmw );
    ListAppend( &tempList, mercedes );
    ListAppend( &tempList, other );  /* SAAB - yuppies you know */

    otherInfo->vehicles = tempList;

    /* occupation is an optional field - lets leave it out for this record */
    otherInfo->occupation = NULL;

    return( tempPerson );
}

/* this routine constitutes a Threads process to encode and decode the  */
/* PERSON structure */
PROCESS encode_decode()
{
    IDX     *idxList;
    PERSON  *decodedPerson;
    int     len;
    char    *buf;
```

```
    ASN1_ERROR_FLAG = 0;

    /* make a PERSON and encode her using BER */
    idxList = Encode_PERSON( 0, 0, 0, makePerson() );

    /* serialize the encoded buffers into one contiguous beffer for decoding */
    serIDX( idxList, &buf, &len );

    /* decode the buffer which was perviously encoded */
    decodedPerson = Decode_PERSON( 0, 0, 0, &buf, 0 );

    /* the memory for the decodedPerson is allocated from the top memory */
    /* frame of the calling process.
    /* the decodedPerson should be identical to the structure created by */
    /* makePerson(). The application may now examine this structure for  */
    /* printing - etc.                                                   */
}

mainp()
{
    Create( encode_decode, 6000, "encode_decode", 0, NORM );
}
```

Part III

Persistent Storage

Chapter 5

Persistent Object Store

5.1 Introduction

Operations to store, access, and manage arbitrary *objects* on disk are provided by the Object Store. As far as the object store is concerned, an object is simply zero or more contiguous bytes. In addition to basic object I/O, the Object Store provides inverted indexes and atomic transactions.

The Object Store uses a database similar to Unix's dbm and can typically locate an object without any disk accesses. An object that is smaller than *KBUFFER_SIZE* – 1 bytes is stored directly in the database. A bigger object, however, is stored in its own Unix file, with its entry in the database containing the name of the file.

5.1.1 Objects

Each object is associated with an *object type* and each object type is uniquely identified by a `type/managerNo` pair:

```
typedef struct
    {
    uint_8      type;
    uint_8      managerNo;
    } TM;
```

Each object is named by a unique identifier called a UID:

```
typedef struct
    {
    uint_32     timeOfDay;
    uint_16     serial;
    TM          type;
    } UID;
```

Certain information about an object type must be passed to the Object Store. This information is stored in a structure that contains the object type identifier, the name of the database file to use for all objects of this type, and pointers to functions for serializing (encoding), deserializing (decoding), and indexing objects of this type.

```
typedef struct ObjTypeDefn
    {
    TM          typeManager;
    char        *volume;
    void        (*encode)(void *object, char **buffer, long *length);
    void        (*decode)(void **object, char *buffer, long length);
    Set Of(OCTS) (*index)(void *object);
    } ObjTypeDefn;
```

5.1.2 Interfaces

Many Object Store functions return one of the following result codes:

```
typedef enum
    {
    ObjOk, ObjUpdateError, ObjBindError, ObjOpenError,
    ObjAILFull, ObjNotFound, ObjExists
    } ObjRc;
```

The following functions are provided by the Object Store:

```
ObjRc ObjInit(table, create)
```

```
ObjTypeDefn *table;
bool create;

void ObjTerm(typeManager)
TM *typeManager;

UID *ObjNew(typeManager)
TM *typeManager;

ObjRc ObjDispose(uid)
UID *uid;

ObjRc ObjMap(uid, data)
UID *uid;
char **data;

ObjRc ObjSecure(instance, uid)
UNIV *instance;
UID *uid;

Set Of(UID) ObjLookup(typeManager, keyValue)
TM *typeManager;
OCTS *keyValue;

bool ObjRPbegin(typeManager, mode)
TM *typeManager;
int mode;

bool ObjRPcommit(typeManager, mode)
TM *typeManager;
int mode;

bool ObjRPabort(typeManager, mode)
TM *typeManager;
int mode;
```

ObjInit() must be called before an object type is used. If *create* is non-zero, the database is created if it does not exist.

ObjTerm() closes the database associated with the object type. This should not be called if there is a transaction in progress.

ObjNew() creates a new zero-length object and returns its UID.

ObjDispose() destroys an object given its UID.

ObjMap() reads an object given its UID and sets *data* to point to the object.

ObjSecure() updates an object given its UID and new instance data. The object must be in memory, as the result of an ObjNew() or ObjMap(), before ObjSecure() can be called. ObjSecure() calls the encoding and indexing routines for the object type.

ObjLookup() takes a key and returns a (possibly empty) set of corresponding UIDs.

The Object Store provides atomic transactions. ObjRPbegin() starts a transaction, ObjRPcommit() commits the transaction, and ObjRPabort() aborts the transaction. These functions return TRUE on success, FALSE on failure. If a crash occurs before ObjRPcommit() commits the updates or if ObjRPabort() is called, all modifications to the database since ObjRPbegin() are undone. The *mode* parameter must be 1 (*OBJ_UPDATE*) if the transaction calls ObjNew() or ObjSecure(); any other value implies a read-only transaction.